PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970 c. C-30, et ses amendements subséquents.

An Architecture for Extending Constraint Satisfaction

by

Richard G. Brown, B.Eng., S.M.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

Sept. 13, 1991

© Richard Gregory Brown 1991

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70957-X

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies and
Research acceptance of the thesis

An Architecture for Extending Constraint Satisfaction

submitted by Richard Gregory Brown, B.Eng., S.M., in partial fulfilment
of the requirements for the degree of Doctor of Philosophy.

Chair,

Department of Systems and Computer Engineering

Thesis Supervisors

External Examiner

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Sept. 13, 1991

Abstract

Current constraint programming languages are deficient in the areas of extensibility and expressiveness. While their built-in constraint satisfaction methods offer some relief from the problems of solving arithmetic relations in Prolog, their common “black-box” approach is overly restrictive. As well, the languages make no attempt to utilize any metalevel data that users might have which could aid the satisfaction process.

An architecture for a general constraint satisfaction mechanism is presented. It is extensible in that new satisfaction procedures may be added to solve new classes of constraints, as well as specialized methods whose abilities may overlap those of other procedures. The solver uses a hierarchical procedure organization and a Prolog-like search technique to identify the most efficient ones available, yet it minimizes the amount of data that is sent to inappropriate procedures. New procedures are created by adding three components to a generic processing skeleton and placing the procedure in the hierarchy. It is shown that it is not possible to determine optimal procedure configurations without a priori knowledge of program contents and run-time behaviour.

By making use of three types of metalevel data, it is shown that the satisfaction process can be made much more efficient. Language constructs are presented which allow such data to be expressed in a declarative fashion, and the solver is modified to make use of its procedural interpretation. If no metalevel data is provided with a constraint, the solver is still able to satisfy it but at the expense of (1) increased processing times, and (2) the possible generation of redundant solutions and solutions that can neither be confirmed nor dismissed. The use of metalevel data has pitfalls similar to those seen with Prolog’s cut operator; i.e., if the data is not used correctly, valid solutions to a program may be lost.

Acknowledgements

I will always be grateful to my co-supervisors, Drs. Gerald Karam and John Chinneck, for their advice and countless words of technical wisdom, but most of all, I would like to thank them for their patience and understanding as my extra-curricular life went on a rather wild ride about one year into the fray. As well, I would like to give special thanks to John for helping me defeat my Nemesis at least this one time (i.e., writing in some sort of organized fashion), and Gerald for his boundless enthusiasm and those gosh-darn, keep-me-on-my-toes questions.

I would also like to thank Dr. Bernie Pagurek for introducing me to the joys of logic programming and answering many questions over these past few years. On the other side of the academic fence, I would like to thank Mr. Bill Older at Bell-Northern Research for his insightful comments and alternative views, and Drs. Aly Nawwar and Carlos Ferregut for inspiration and general words of advice.

Closer to home, for those much appreciated words of encouragement and occasional kicks in the behind, I am forever grateful to Christine LeDuc, Joan Green, my parents Freda and Len Brown, Bubba (for I know you would have if you could have), and * (that's a wildcard for anyone I missed).

Finally, I would like to thank NSERC, Fleet Technology Ltd., Carleton University, and my parents for considerable financial support over the years. I am forever in your debt (pun intended).

Contents

1	Introduction	1
1.1	Logic Programming and Mathematics	1
1.2	Thesis Objective	4
1.3	Thesis Outline	5
2	Constraint Programming Languages	7
2.1	Background: Prolog Basics	7
2.2	The History of Constraints	11
2.3	Methods of Constraint Satisfaction	14
2.3.1	Local Propagation or Method of Isolation	14
2.3.2	Term Rewriting	15
2.3.3	A Modified Version of the Simplex Algorithm	16
2.3.4	Integer and Rational Variable Decision Procedures	17
2.3.5	Interval Arithmetic	18
2.4	Recent Constraint Programming Languages	18
2.4.1	CLP($\mathcal{R}$)	19
2.4.2	Trilogy	20
2.4.3	CHIP	20
2.4.4	BNR Prolog	21
2.4.5	Prolog III	22

3	Design Objectives and Considerations	23
3.1	Limitations of Existing Approaches	23
3.2	Design Objectives	30
3.3	Design Considerations	32
4	The Extensible Architecture	36
4.1	An Overview of the Solver Architecture	36
4.1.1	Basic Processes	37
4.1.2	Procedure Organization	41
4.2	A Detailed Description	42
4.2.1	Definitions	42
4.2.2	Process Interfaces	44
4.2.3	MP Description	50
4.2.4	Procedure Components	56
4.2.5	Procedure Description	62
4.2.6	Propagating BACKTRACK Messages	66
4.2.7	Comments on the Search Strategy	67
4.3	A Simple Example	69
4.4	Procedure Restrictions	74
4.5	Incorporating Type 2 Variables	82
5	Configurations and Satisfaction Failure	85
5.1	Organizing Procedures	85
5.1.1	Procedure Placement Requirements	86
5.1.2	Adding Procedures with No Overlap	87
5.1.3	Adding Procedures with Overlapping Capabilities	89
5.1.4	Improving Acceptable Configurations	98
5.2	Identifying Satisfaction Failure	100
5.2.1	Identifying Global Satisfaction Failure	101

5.2.2	Identifying Local Satisfaction Failure	102
6	Solver Properties	104
6.1	General Properties	106
6.2	Responses Occur only if Conditions Met	111
6.3	Responses Occur if Conditions Met	117
7	Expressiveness	131
7.1	Data Typing	132
7.1.1	Specifying the Data Type of a Variable	132
7.1.2	Making Use of Variable Data Types	133
7.2	Constraint Algebraic Classifications	133
7.2.1	Specifying Classification Data	135
7.2.2	Classification Data Considerations	137
7.3	Grouping Constraints	141
7.3.1	Specifying the Group Data	141
7.4	Combining Classification and Group Data	143
7.4.1	Modifying the Solver for Classification and Group Data	144
7.4.2	Modifying Procedures for Metalevel Data	149
7.5	Effects on Solver Properties	150
8	Examples	153
8.1	Creating Procedure Components	154
8.1.1	Recognition Rules	154
8.1.2	Sufficiency Predicate	155
8.1.3	Function Calls	156
8.2	Example 1	159
8.2.1	Solving without Metalevel Data	159
8.2.2	Solving with Metalevel Data	170

8.2.3	Performance Comparison	174
8.3	Example 2	179
8.3.1	Solving without Metalevel Data	181
8.3.2	Solving with Metalevel Data	192
8.3.3	Performance Comparison	206
9	Conclusions	207
9.1	A Research Summary	207
9.2	Conclusions	208
9.3	Contributions	209
9.4	Further Research	211
	References	215
A	Handling Simple YES Responses	220
B	Constraint Substitutions	223
C	Proofs of Lemmas	225
D	Proof of Properties	277
D.1	Proofs of General Properties	277
D.2	Responses Occur only if Conditions Met	290
D.3	Responses Occur if Conditions Met	305
E	Proof of Backtrack Properties	340
F	Combining Metalevel Data	371

...I submit to the public a small machine by my invention, by means of which you alone may, without any effort, perform all the operations of arithmetic, and may be relieved of the work which has oftentimes fatigued your spirit ...

Blaise Pascal

List of Tables

4.1	PE to MP Interface	45
4.2	Possible Reply Sequences	47
4.3	MP to Procedure <i>Pi</i> Interface	48
4.4	Procedure to Procedure Interface	49
4.5	MP Functions	56
4.6	Function Results	60
7.1	An example CP Table	136
8.1	Procedures for Example 1	160
8.2	Comparison of the number of messages sent	177
8.3	Comparison of the solver's reply sequences	178
8.4	Procedures for Example 2	181
8.5	Comparison of the solver's reply sequences	193
8.6	Comparison of the solver's reply sequences	194
8.7	Comparison of the number of messages sent	206

List of Figures

4.1	Constraint Solver Processes.	38
4.2	Conceptual model of recursive solver.	40
4.3	Constraint management data structures.	51
4.4	MP state transition diagram.	53
4.5	Typical Procedure P_i	57
4.6	P_i data structures.	61
4.7	P_i state transition diagram.	63
4.8	Solver states.	70
4.9	Solver states (continued).	72
4.10	Solver states (continued).	73
5.1	Capabilities of P_i relative to P_j	88
5.2	Adding procedures with no partially overlapping capabilities	90
5.3	Adding a procedure with partially overlapping capabilities	92
5.4	Multiple possible parents	94
5.5	Adding P_i when multiple parents are allowed	97
7.1	CGCP graph	146
7.2	Modified MP state transition diagram	147
8.1	Example 1	161
8.2	Example 1 (continued)	164

8.3	Example 1 (continued)	166
8.4	Example 1 (continued)	168
8.5	Example 1 (continued)	169
8.6	Example 1 with metalevel data	172
8.7	Example 1 with metalevel data (continued)	173
8.8	Example 1 with metalevel data (continued)	175
8.9	Example 1 with metalevel data (continued)	176
8.10	Example 1 with metalevel data (continued)	177
8.11	Example 2	180
8.12	Example 2 (continued)	184
8.13	Example 2 (continued)	186
8.14	Example 2 (continued)	188
8.15	Example 2 (continued)	189
8.16	Example 2 (continued)	191
8.17	Example 2 with metalevel data	196
8.18	Example 2 with metalevel data (continued)	198
8.19	Example 2 with metalevel data (continued)	200
8.20	Example 2 with metalevel data (continued)	201
8.21	Example 2 with metalevel data (continued)	202
8.22	Example 2 with metalevel data (continued)	203
8.23	Example 2 with metalevel data (continued)	205

Nomenclature

- Pi^+ — Any procedure in the solver, including MP
- LS — Last (previous) state of any procedure
- MP(k) — Procedure MP at the k^{th} level of recursion
- $Pi(k)$ — Procedure Pi at the k^{th} level of recursion
- $Pi^+(k)$ — Pi^+ at the k^{th} level of recursion
- CM — Constraint Message
- AVR — Alternate Value Request
- L_{M_i} — The list of constraints that is sent in a CM M_i
- A_{M_i} — The argument of CM M_i , which if set to 1, indicates that L_{M_i} contains every constraint in PS^4
- PS^4 — MP(k)'s problem set when it reaches state 4
- $PS_{M_i}^-$ — MP(k)'s problem set immediately before it receives a CM M_i
- $PS_{M_i}^+$ — $PS_{M_i}^- \cup L_{M_i}$; that is, MP(k)'s problem set immediately it has been updated with the constraints in M_i
- $PS_{M_i}^*$ — $PS_{M_i}^+$ after MP(k) has satisfied zero or more MP-solvable constraints in $PS_{M_i}^+$ (using values determined with transitions 24 and 3)
- $PS_{M_i}^4$ — $PS_{M_i}^*$ when MP(k) has reached state 4

Chapter 1

Introduction

1.1 Logic Programming and Mathematics

Computer programming languages try to provide a convenient interface between users of the machine and the underlying hardware. The language may provide easy access to machine-specific details or it may completely obscure the hardware, the most useful depending on the problem at hand. At one extreme, assembly languages may be necessary to access microprocessor registers while at the other, very high level languages allow users to concentrate on the description of problems which are not related to computers at all, much less the host machine.

While Prolog [37], the most common of the so-called logic based languages, has proven to be quite useful in some application areas, it has not fared well when applied to problems involving significant amounts of “number crunching.” This stems from some inherent limitations in Prolog’s underlying inference engine. Prolog determines the truth of a statement by comparing its syntax with other program data, not on the semantics a programmer may have had in mind. For instance, the comparison $1+1 = 2?$ fails because the left hand side contains two more characters than the right. This is not necessarily wrong depending on what one wanted to compare

(the character sequence or the arithmetic values), but it does demonstrate that the inference engine may apply a different interpretation to programs than what was intended by the programmer.

Even if Prolog interprets numeric values correctly, other problems arise. While Prolog has no difficulty searching through lists of symbolic data looking for solutions to subgoals, it is rarely practical to contain lists of numeric solutions to equations in programs. Some relief is provided by system predicates such as *is/2*¹, as in *X is Y+1*, as they signal the inference engine to provide special case handling so that arguments are treated as numerical values or variables representing such. However, it will be seen that common system predicates have serious limitations and they do not provide adequate solutions.

To overcome the inherent limitations of the built-in inference engine and the system predicates, a new class of languages known as constraint programming languages has been developed [3, 26]; most are based on Prolog. In general, a constraint programming language provides one or more specialized algorithms for solving expressions that relate members of some computational domain; the expressions are referred to as constraints. All of the new languages have at least some ability to solve arithmetic constraints automatically, hence, for the purposes of this thesis, constraints are considered to be the predicates that describe arithmetic equalities, disequations, and inequalities in a logic program.

A constraint programming language has more power than a conventional logic programming language because it can solve arithmetic relationships automatically. This provides the following benefits:

- improved programmer productivity,
- programs that are easier to understand,
- programs that are easier to modify, and
- possibly comparable efficiency to programs written in imperative languages.

¹“/n” after a predicate name indicates the number of predicate arguments (the predicate’s *arity*).

Programmer productivity is improved because arithmetic relations may be stated (to a certain extent) without concern for their syntax or which variables are bound when the expressions are encountered. Thus, programmers are freed from having to study the order of evaluation of a program, a time consuming and error prone task.

Programs should be easier to understand because problems can be stated more compactly and in a purely declarative fashion. It will be seen that one constraint may do the work of several clauses using *is* predicates, therefore less code is required and procedural considerations are lessened.

Constraints make programs easier to modify because (1) the effects of changes may be fully understood in the declarative interpretation of a program, and (2) (ideally) constraints may be changed without having to consider the order of evaluation of programs. Thus, arithmetic relations can be added, changed, or deleted with much more freedom than is possible in languages that do not use constraints.

Not only do constraint programming languages promote the development of programs that are impractical in “standard” Prolog, their efficiency may be comparable to specialized programs written in imperative languages. One example of this is an Operations Research cutting-stock problem described by Dincbas et al. [12].

Other application areas that have been explored using constraint programming languages are:

- simulation, fault diagnosis, and test signal generation for digital circuit design [35],
- lumped circuits, digital filtering, and electromagnetic fields in electrical engineering [19],
- option trading analysis (a business finance application) [23],
- assembly-line sequencing, scheduling, and warehouse location [13, 14].

The methods used to process arithmetic constraints are referred to as constraint satisfaction procedures. In any one implementation, they usually only cover a very limited class of problems such as real-valued variables in linear equations. The reasons for this are (1) practicality, since it would be impossible to include every known

satisfaction procedure, and (2) efficiency, since searching for a suitable method could become expensive if very many were available. Consequently, all existing constraint programming languages have limited application areas, although some are much more restricted than others.

Even if a constraint programming language has the ability to solve a very wide class of problems, the efficiency of the satisfaction procedure may become a concern. Whenever a wide class of problems can be solved, a general purpose solution procedure is often used, the efficiency of which is a function of the constraints at hand. Even if more specialized procedures can be built on top of any built-in mechanisms, it is unlikely they would be as efficient as possible and a confusing mixture of user-supplied and built-in methods can result. An overhead penalty also results if unnecessary built-in methods cannot be disabled.

1.2 Thesis Objective

At some point in any constraint programming language, an algorithm must be used to solve the constraints even though users may not be aware of it. It is recognized that as long as (1) some mathematical problems are undecidable, (2) no single method can solve all constraints efficiently, and (3) it is not practical to include every conceivable method, then compromises will always have to be made in language implementations. Thus, if a set of constraints cannot be solved in a suitable fashion because of language limitations, the following options exist:

1. the problem can be restated or simplified so that it becomes solvable,
2. Prolog procedures can be used to augment the constraint solver's abilities,
3. another constraint programming language can be used, or
4. the problem can be solved using an imperative language.

It will be shown that all of these approaches are deficient in one or more ways and so the main objective of this thesis becomes:

- to present a new approach to constraint satisfaction that will give one language the ability to solve virtually any constraint, provided that all variables are defined in terms of the language's primitive data types.

The new approach should support the use of multiple satisfaction procedures since programs may contain many different types of constraints, and it should allow the procedures to be tailored to the problem at hand so that any overhead associated with the satisfaction process is minimized. However, as in existing languages, the constraint solver should remain in the background, values being determined whenever sufficient data becomes available.

As stated in the previous section, another consideration in constraint satisfaction is how efficiently the expressions can be solved. If the solver is to have more than one satisfaction process available, a second objective becomes:

- to explore means of making the satisfaction process as efficient as possible.

Two areas need to be investigated, the first covering the solver itself. Assuming that multiple satisfaction procedures are available, a means of efficiently locating suitable ones at run-time becomes necessary.

The second area of investigation lies beyond the solver and in the users' programs. While the use of explicit calls to invoke satisfaction procedures is considered undesirable (because it introduces procedural, extra-logical considerations into programs), users may have metalevel data regarding constraints and appropriate solution methods that could aid the satisfaction process. Users should be able to express such data in a declarative fashion and the constraint solver should have the ability to make use of it. However, if no metalevel data is provided, the solver should still be able to satisfy the constraints but at the expense of increased overhead.

1.3 Thesis Outline

The organization of this thesis is as follows: Chapter 2 presents background information on logic programming, limitations of common system predicates, and

the need for constraints. Common methods of constraint satisfaction and current Prolog-based constraint programming languages are then reviewed.

Chapter 3 identifies the limitations of current constraint programming languages. The design objectives for a new constraint solving architecture are then presented.

Chapter 4 presents the new constraint solving architecture and a simple example to demonstrate the built-in search mechanism. The requirements of components for new procedures are presented.

Chapter 5 discusses methods for organizing procedures into a suitable hierarchy and means of recognizing when the limits of the solver have been exceeded.

Chapter 6 presents properties which the solver has been proven to possess (the proofs appearing in Appendices C, D, and E).

Chapter 7 presents extensions to a generic Prolog language which allow metalevel data to be expressed, and means of interpreting the data so as to make the satisfaction process much more efficient. The effects on the properties of Chapter 6 are also discussed.

Chapter 8 demonstrates (1) how a new satisfaction procedure can be created, and (2) how the solver would process two sets of constraints, both with and without the use of metalevel data.

Chapter 9 presents the conclusions of this thesis and recommendations for further research.

Chapter 2

Constraint Programming Languages

2.1 Background: Prolog Basics

In their simplest form, Prolog programs consist of a list of facts embodied in predicate definitions. For instance, the following is a valid Prolog program:

```
object( apple, round ).  
object( computer, square ).
```

It consists of the definition of the predicate *object/2*. Its first argument is the name of an object and its second describes the object's shape. The program states that an apple is round and that a computer is square; all other statements are assumed to be false.

To make use of the program, it can be queried using the following:

```
object( X, square )?
```

Since Prolog conventions use lower and upper case letters to represent constants and variables, respectively, the query effectively asks the interpreter to prove that there is an object *X* whose shape is square. It would do so by syntactically matching predicate names and their arguments with what is known to be true according to

the program. Since X is a variable, the interpreter can match the query with the second fact and so it would return:

YES: $X=computer$

Queries do not need to include any variables, and so the following could also be made:

$object(ball, round)?$

Since there is no fact describing balls, the interpreter would return NO.

Prolog programs can include rules as well as facts. The following states that the color of X is red if X is a round object.

$color(X, red) :- object(X, round).$

The consequence (or head) of the rule is $color(X, red)$, and its body consists of the single antecedent $object(X, round)$ (although multiple antecedents are possible). If the rule was included in the above program, the following query could be used to determine if any object Y is red in color:

$color(Y, red)?$

Since there are no facts regarding colors, the rule must be used. The rule states that an object is red if it is round and so it must be determined if a round object exists. The interpreter matches the query with the head of the rule and in so doing it binds Y in the query to X in the rule. The antecedent remains to be proven and so the interpreter matches $object(X, round)$ with the first fact in the program. Since Y was bound to X and X was bound to *apple*, the interpreter replies:

YES: $Y=apple$

If another solution was requested, the interpreter would try to find one to its last subgoal, that is, $object(X, round)$. There is none and since there are no other ways of proving $color(Y, red)$, the interpreter returns NO.

One of the most interesting features of Prolog is its ability to run programs "in reverse" When programs are written and queries are made, predicate arguments may be inputs or outputs as demonstrated by the following:

color(apple, Y)?

The query would be matched with the head of the rule in which case X would be bound to *apple* and Y would be bound to *red*. As before, the antecedent of the rule must still be proven true and so the interpreter tries to prove *object(apple, round)*. It does so by matching this new subgoal with the first fact in the program, and since Y was initially unknown, the interpreter replies:

YES: Y=red

When predicates make use of arithmetic values and expressions, it is usually not practical to prove their truth by syntactically matching pairs of numeric arguments. Consider the predicate *sum/3* whose third argument is the sum of the first two (as in *sum(1,1,2)*); such predicates are of little use because there are too many combinations of values to list for non-trivial programs. Another problem arises when arithmetic operators are involved. For instance, the predicates *number(1+1)* and *number(2)* will not match because the argument of the first contains two more characters than the argument of the second.

To help avoid such unexpected results, system predicates (ones provided with a language) are used to evaluate expressions according to arithmetic conventions. For instance, *1+1 =:= 2?* returns true because the infix *=:=* predicate causes both sides to be reduced (if they are not already) and the two sides compared arithmetically. Unfortunately, such predicates may only be used under limited conditions. If X was unknown, the query *1+1 =:= X?* would raise a system error because *=:=* requires that all of its arguments be bound.

The need for more powerful system predicates resulted in a now standard method for handling arithmetic relations. It is henceforth referred to as the *is* approach, named for the infix predicate used to assert arithmetic equality, as in *X is Y+1*. Provided it has not previously been assigned a value, a single variable to the left of an *is* will be assigned the value of the expression on the right. This is very similar to the method used in imperative languages. In logic programming, however, if the left hand variable already has a value, it is simply compared to the value of the right hand side. If the two are equal, the assertion is accepted, otherwise logical failure

results which causes the search mechanism to backtrack over the equation. In no case is the value of the left hand variable overwritten.

Unfortunately, the *is* approach has serious limitations that may prevent the program from finding all of the possible solutions to a problem. The *is* predicate violates the declarative model of logic programs in several ways:

- It can cause unnecessary backtracking to occur because a variable to the right of the *is* was not bound, even though a solution for the expression existed. For instance, *4 is X* will fail because *X* is unbound.
- It cannot process simultaneous dependent expressions. Using *A is B+1* and *A is 2×B* will not determine any values because whichever predicate is encountered first, an unbound variable appears on the right hand side.
- An *is* predicate cannot generate alternate solutions to equations. Asserting *X is arcsin(0)* will not generate an infinite number of solutions for *X*.
- *is* predicates are not always commutative with other subgoals. For instance, asking *Y is 5, X is Y-1?* will produce *X=4* as a solution, but *X is Y-1, Y is 5?* will not. Again this is a result of the unbound variable on the right hand side.
- Arithmetic expressions involving *is* cannot be used for determining symbolic solutions.

These problems result in users having to consider the evaluation order of a program to ensure that answers are not lost; that is, they must ensure that none of the above situations can arise. As well, the ability to run programs "in reverse" is reduced since the above limitations may prevent solutions from being found for some combinations of inputs.

The extra-logical problems resulting from *is* are inconsistent with some of the more desirable aspects of logic programming. In general, they violate the high-level declarative interpretation of a logic program in which the predicates are seen as a description of a problem. Under this idealistic view, neither the order of subgoals nor

the arrangement of symbols within a subgoal should affect the results of a program, provided that the relationships have not changed.

Since arithmetic expressions occur so frequently in every application area and the algorithms for solving them are difficult to implement using *is* and logic procedures (that is, the facts and rules which define the truth of a predicate), another approach has received much attention in the last few years. It provides the language with the power to solve at least some expressions automatically without any assistance from the user. As well, the relationships can (ideally) be asserted without concern for their syntax. For instance, $X = Y \times 5$, $0 = X - Y \times 5$, and $X/Y = 5$ should all produce identical results. Languages with such capabilities are known as constraint programming languages.

2.2 The History of Constraints

Constraints first appeared in Ivan Sutherland's Sketchpad system [39]. It introduced the idea that the declarative specification of a problem could contain enough information to directly guide its solution procedure. It was not until the early 1980's that the constraint paradigm was extended significantly in the Ph.D. theses of Borning [4] and Steele [36].

Borning's interactive graphics system could determine a general solution procedure for a set of constraints when specific input values were not yet known. Since problems for the targeted application area would typically have many solutions, it would generate and compile just one solution procedure. Specific input values would come from the position of a pointing device such as a mouse, solutions being generated in realtime as the pointing device was moved.

Steele's system provided software "objects" that mimicked physical circuit components. After connecting several objects together so that various inputs and outputs had common values, levels in the network could be specified. As soon as one value was set, it could result in others being determined and so on, values propagating through the network of objects. For instance, setting the output of an AND gate to true would result in all of its inputs being set to true as well. If the inputs were

connected to other components, other values in the network could become known. The direction of value propagation was not predetermined in any way, unlike real world components. Unfortunately, Steele's system was computationally very weak as it relied solely on the technique of local propagation (to be discussed). This meant that it would not always determine values even when there was sufficient information available to do so.

The methods used for constraint satisfaction in these early systems were not very sophisticated, although symbolic processing was tried in at least one experimental system. While some problems were readily solved using the simple techniques, no single method emerged as the best for solving "real-world" problems.

Most of these early constraint systems revolved about the concept of interconnected objects or constraint networks [44]; there was some other work, however. In the Ph.D. thesis of G. Huet [22], the problem of solving equality constraints was investigated as a basis for a higher order logic. Ten years later, Colmerauer [8] would base Prolog II, and eventually Prolog III, on the determination of whether or not a system of equations over infinite trees had at least one solution. Research areas that were eventually affected by the use of constraints included logic, automatic theorem proving, natural language processing, and truth maintenance systems [44].

Constraint Programming Languages

The use of constraints in logic programming languages has only appeared since the mid 1980's as more powerful logic languages became necessary and the theoretical implications could be justified [25, 26, 43]. Even though new methods of improving Prolog with constraints continue to be proposed [3], many of the original concepts and methods continue to be used.

A constraint programming language might be thought of as a traditional logic programming language with a built-in automatic equation solving system. However, constraint solvers differ significantly from commercially available systems such as Macsyma, Maple, etc. Prolog tries to solve subgoals one at a time and so the arithmetic expressions in a program are not available all at once. As well, backtracking may require data to be retracted. Consequently, a constraint solver must

often solve expressions incrementally, adding new constraints to an ongoing problem set.

Another feature that differentiates constraint programming languages from traditional logic programming languages is that constraints are processed using a delay mechanism. Thus, logical failure does not result if a solution cannot be found immediately, provided that the possibility of finding one still remains. As more expressions are encountered, enough information may be collected to determine unique values or to decide that no solution can exist. The delay mechanism is necessary for handling simultaneous equations, disequations and inequalities since they are encountered one at a time and they need to be processed as a set. Note that whether values are determined as soon as possible or not until they are actually needed is another issue. While a delay mechanism is necessary to gather constraints for simultaneous processing, it is not necessarily needed to postpone the determination of values as long as possible. The latter technique, known as lazy evaluation, is known to have some efficiency advantages since it can help to avoid finding values that are never used [31].

Because of the difficulty in solving arbitrary expressions, most constraint programming languages can only solve a very limited class of problems requiring one solution procedure. Typical problem domains are integer- or real-valued variables in linear equations. Only BNR Prolog [2] has the ability to solve much more difficult expressions, but it has just one general purpose method for solving them. A second method that relies on the first can also be invoked using a special system predicate.

Some new languages (Trilogy [10] and BNR Prolog) also provide for user-defined routines written in other languages. Although these could be used to solve expressions beyond the domain of the constraint system, such routines must be called explicitly in the program. At run-time, routine inputs must be bound and outputs unbound. Consequently, a user would have to consider carefully where to place a call in the program. In Trilogy, another limitation is that backtracking across the routine does not undo any bindings resulting from the call. While such user-defined routines may be of use, they do not improve the power of the constraint system.

2.3 Methods of Constraint Satisfaction

Potentially any algorithm could be incorporated to solve constraints, however there are a few that have been particularly successful. These are discussed briefly below.

2.3.1 Local Propagation or Method of Isolation

Local propagation is the simplest and fastest method for constraint satisfaction [36]; it is also referred to as *value inference* by Van Hentenryck [41]. Local propagation can solve one equation with a single unknown as long as the variable appears just once, as in:

$$\log(X-3) = 2$$

Local propagation is based on the method of isolation which searches for an equivalent expression that isolates the unknown, producing an expression that may be directly evaluated. For instance, an equivalent expression to the above may be found by repeatedly applying the inverse of the outermost operation on the left hand side to the expression the right hand side, as shown below:

$$\log(X-3) = 2$$

$$X-3 = 10^2$$

$$X-3 = 100$$

$$X = 100+3$$

$$X = 103$$

Once a value has been determined, it is propagated to other constraints that may then become solvable.

Although local propagation succeeds quickly because of the very limited search space, it is computationally very weak. It cannot solve simultaneous equations as long as the equations contain multiple variables, nor can it solve an equation in one unknown if the variable appears more than once. Consider the following code segment in which *A* and *B* are uninstantiated:

$$\dots, A+B = 2, A-B = 0, \dots$$

These cannot be solved since rearranging either always specifies one variable in terms of another.

Local propagation is also unable to combine variable instances in an expression. Thus, it cannot even solve $A+A=2$ because whichever way it is rearranged, one variable instance is specified in terms of the other, just as if the equation had two distinct unknowns.

2.3.2 Term Rewriting

Term rewriting is a symbolic processing procedure that rearranges and substitutes expressions in an attempt to reduce them to a readily evaluable form [7]. The process is guided by a list of rewrite rules that transforms constraints from one form to another. Such rules may be provided by the user in Bertrand, a constraint programming language developed by Leler [29]. It is not reviewed with the other languages in § 2.4 because it is not logic based.

In his Ph.D. thesis, Leler developed augmented term rewriting, a technique that uses a table of bindings containing variables and their equivalent symbolic expressions. For instance, given the equation $X = 2 - Y$, the table would contain the binding $2 - Y$ for variable X . Whenever a bound variable is encountered in another constraint, it is replaced with its equivalent expression. Given another constraint $Z + (X) = Y$, it would be rewritten as $Z + (2 - Y) = Y$. Depending on the available rules, other rewritings might then be possible. Typically, the objective is to reduce the number of occurrences of variables, possibly by putting them in some order and combining their coefficients. If one of the remaining variables can be isolated, as can Z in $Z + 2 - 2 \times Y = 0$, it is entered into the state table along with its equivalent expression. The whole process is repeated until only primitive terms remain (numeric or symbolic constants), or until no more rewritings are possible.

Term rewriting also appears in “equational” and “logic plus functional” programming languages. Such languages try to marry the benefits of logic languages and functional languages such as LISP. Proposed examples include EQLOG [17] and FUNLOG [38]. In general, languages based on term rewriting are limited in

practice by the inherent difficulties of equational resolution and equational unification [46, 40]. Simply put, trying to solve equations by rewriting them involves a huge search space; consequently, the process may be so inefficient as to make it impractical. There are also significant restrictions on what rules may be used to ensure that their application will eventually terminate. Another problem is that of overcomputation whereby attempts to find alternate solutions may result in the same solution being generated many times over. These difficulties have restricted “equational” and “logic plus functional” languages to the academic world.

A certain amount of term rewriting will be found in most constraint systems since constraints start as symbolic expressions. Typically, constraint programming languages perform a minimal amount of symbolic processing to check that the constraints are solvable using the available methods, to rewrite expressions into a canonical form, and to extract the variable coefficients. The results of these efforts are then used as inputs to standard numeric algorithms.

2.3.3 A Modified Version of the Simplex Algorithm

Three constraint programming languages, CLP($\mathcal{R}$), CHIP, and Prolog III, use a simplex-like algorithm to process linear equations, disequations, and inequalities. It is a technique borrowed from the field of Operations Research [18]. In the two phase simplex algorithm, the first phase is normally used to determine a feasible solution to a set of linear constraints, the optimum being found by the second phase. In CLP($\mathcal{R}$) and Prolog III, an incremental version of only the first phase is implemented. As each new constraint is encountered, it is added to a set of those previously processed. If there is no solution to the set with the new constraint, it is deemed inconsistent and logical failure occurs. If the new constraint causes unique values to exist for one or more variables, they are taken as the solutions. No values are determined unless the solutions are unique.

The version of the algorithm in CHIP is also capable of finding optimum values; that is, it can determine the variable bindings that will maximize an objective function.

2.3.4 Integer and Rational Variable Decision Procedures

Some researchers consider problems involving constraints on real-valued variables to be intractable because of finite word length limitations [15]. Computations on such variables can result in small round-off errors which may cause programs to behave incorrectly. Integer- and rational-valued variables are not subject to such errors and so they do not introduce extralogical problems into a program. Thus, constraints on these variables may be considered to be better suited to a logic programming environment.

Efficient procedures for solving such constraints are the subject of current research. Some methods subdivide the possible range of values a variable may have either by the use of *decision procedures* [10] or various *look-ahead inference rules* [41]. For instance, consider the following relations where all of the variables are assumed to be integer-valued:

$$X+Y+Z \leq 10$$

$$Y \geq 3$$

$$X, Z \geq 0$$

If we note that Y must be at least three, making the substitution in the first constraint reduces it to:

$$X+Z \leq 7$$

If X is not less than zero, then Z can be no more than seven. Thus the range for Z may be reduced to:

$$0 \leq Z \leq 7$$

Similar reasoning may be used to narrow the other ranges to $3 \leq Y \leq 10$ and $0 \leq X \leq 7$.

Brute force searching need not be used to restrict the ranges of values although it could be used as a last resort. Note that such a search would be impossible if the variables were real-valued (and the reals could be computed [11]), or impractical if conventional floating point numbers were used.

2.3.5 Interval Arithmetic

As discussed in § 2.3.4, methods for solving constraints on real-valued variables are subject to finite word length limitations and the use of conventional floating point numbers can introduce extra-logical problems into a program. A relatively new area of mathematics known as interval arithmetic overcomes the finite word length problems and offers a new approach for handling constraints on real-valued variables [32, 34, 33].

Interval arithmetic uses a pair of conventional floating-point numbers to act as lower and upper limits for real-valued variables. By conservatively controlling the rounding directions when any values are computed (rounding up or down for the upper and lower limits respectively), the results are intervals that are guaranteed to contain the actual value of interest. When an interval's lower limit equals its upper limit, an exact value known as a point interval has been found. If an interval's lower limit ever exceeds its upper limit, then no solution exists for that variable.

Initially, a variable whose value is not known may be assigned a very wide interval range. As constraints are encountered, the range of the interval may be narrowed but always conservatively so that the solution (if any) is guaranteed to fall within the new limits.

The use of intervals to represent real-valued variables is not a method for solving constraints per se. However, their use does allow real-valued variables to be represented with no adverse effects and potentially any method for solving expressions involving interval-valued variables could be used as a constraint satisfaction procedure. While there are many specialized techniques for such expressions, the only language to provide intervals, namely BNR Prolog, relies on a form of local propagation.

2.4 Recent Constraint Programming Languages

There is a growing number of constraint programming languages available both commercially and for research purposes. The remainder of this section reviews five

of them briefly.

2.4.1 CLP($\mathcal{R}$)

The developers of CLP($\mathcal{R}$) claim it to be the first implementation of their general Constraint Logic Programming scheme, the real numbers $\mathcal{R}$ being the (theoretical) domain of application [24, 26, 20]. CLP($\mathcal{R}$) automatically processes linear equations and inequalities using the first phase of the two phase simplex method (described in § 2.3.3). Linear expressions are added one at a time to a set that is determined to have at least one solution. If a new constraint is found to be inconsistent with the others, logical failure occurs and the constraint is rejected. If the constraint results in a unique solution for some variables, the values are assigned.

When a variable is used to represent a real number, it is subject to the finite word length limitations of floating point representations. Consequently, strict inequalities ($<$, $>$) are converted to regular inequalities ($\leq$, $\geq$) based on the offset of ϵ , the smallest floating point number that can be represented on the host computer. For instance,

$$X > 0$$

is converted to

$$X \geq \epsilon$$

Nonlinear constraints are delayed until they become linear after other values are determined. For instance the constraint $Y \times X + Z = 1$ is nonlinear because it contains a product of variables, thus no attempt is made to verify that it is consistent with any others previously processed. If it is later found that Y equals 5, the constraint $5 \times X + Z = 1$ is added to the processed set. A “quasi-linear” constraint such as $X \times Y/Y = 1$ is also delayed because the equivalent linear equation of $X = 1$ cannot be detected.

CLP($\mathcal{R}$) replies to queries with YES if it can determine that at least one solution exists to a problem and it will list any unique solutions found. If no solution is possible, it returns NO. If there are nonlinear constraints yet to be processed when

it reaches the end of the search tree, it will return **MAYBE** since it has not been determined that a solution exists. $\text{CLP}(\mathcal{R})$ does not have any built-in optimization capability, nor does it allow other satisfaction methods to be added to the system.

Limitations of $\text{CLP}(\mathcal{R})$ for metaprogramming are discussed by Heintze et al. [21]. Techniques are described for manipulating symbolic expressions containing arithmetic function symbols, that is, symbols that are normally interpreted by the constraint system.

2.4.2 Trilogy

Trilogy is a logic programming language whose constraint system can process linear constraints containing integer variables only [10]. It supports equations and inequalities containing integer or rational coefficients. Trilogy is more limited than $\text{CLP}(\mathcal{R})$ in that it does not have a delay mechanism for non-linear constraints; that is, constraints must be linear at compile time, otherwise they are rejected. For instance, the above example of $Y \times X + Z = 1$ would not be accepted even though at run-time it would be linearized if Y were known.

Trilogy provides for the inclusion of compiled functions written in other languages, allowing other arithmetic solution procedures to be used. However, explicit calls must be made in the program to invoke the functions and they must only be made when all of the inputs are known and outputs are unbound. This is unlike other satisfaction methods which wait in the background, determining values whenever possible. Although such functions may extend the usefulness of the language, they do not extend the power of its constraint system. Such functions would also appear to introduce extra-logical problems into the language, but Voda [43] claims otherwise.

2.4.3 CHIP

CHIP, Constraint Handling in Prolog, is one of the most powerful languages available, having constraint processing capabilities in three domains: finite domain restrictive terms, boolean terms, and linear rational forms [15]. It is the only language

that provides some form of user-defined constraints, but only for finite domain terms. It has no capability for extending the satisfaction methods in the other two areas.

Because of the difficulty in reliably solving constraints on real-valued variables, methods for solving arithmetic rational terms were implemented instead. These are fully embedded in the language for efficiency reasons, preventing the addition of new methods. It does provide an optimization capability for integer-valued variables using a branch-and-bound algorithm [18]. For instance, a set of inequality constraints could be declared involving some variable X . Its maximum or minimum value could then be determined using the predicates *maximum*(X) or *minimum*(X).

2.4.4 BNR Prolog

BNR Prolog is a logic programming language that provides a constraint processing capability for expressions involving interval-valued variables [2]. Variables assigned the primitive interval data type may be initially assigned a range as wide as $\pm 3.4 \times 10^{38}$. As constraints are asserted, the intervals are automatically narrowed using a form of local propagation. BNR Prolog uses a set of arithmetic primitives that conservatively control the rounding directions, enabling it to give results that are guaranteed to be correct. This approach results in a constraint processing capability that is free of extra-logical problems, unlike those that use conventional floating point representations for real-valued variables.

Often the narrowing method used does not narrow the intervals enough to give a useful answer; for instance, intervals may not be narrowed at all when simultaneous equations are entered. However, a call can be made to a special *solve* predicate that recursively subdivides the range of a specified variable. The new range propagates to other variables and so their intervals may be narrowed. If any interval's lower limit exceeds its upper limit, this contradiction signals that there is no solution in the asserted range and so it is rejected. *Solve* can narrow intervals to point intervals when a variable has a unique solution, and it can generate a series of solution intervals when more than one answer exists.

Unfortunately, BNR Prolog's constraint system can be quite slow, at times tak-

ing several minutes to solve some relatively small problems. This inefficiency is to be expected since the *solve* predicate uses a bisection search. Fortunately, the processing times (with and without *solve*) can be reduced if the initial ranges of some intervals can be narrowed using a priori knowledge of the answers. For instance, a solution to $X^6 = 111$ could be found much more quickly if X was initially assigned the range 1 to 10, rather than the default $\pm 3.4 \times 10^{38}$.

BNR Prolog qualifies all interval solutions with "If there is an answer, it is within the indicated range." This is an important consideration when answers are being interpreted because it casts some doubt on their validity. For instance, it is possible for solution intervals to be quite wide even though the true solution consists of unique values, and variables can be assigned ranges of values even though the constraints have no solution.

BNR Prolog's interval constraint system is believed to be the most general available. It is capable of solving nonlinear problems that include transcendental functions, something that no other language can do. Some of its abilities and limitations for optimization purposes are discussed by Brown et al. [6]. No provision has been made to allow the built-in satisfaction methods to be replaced or augmented with other procedures.

2.4.5 Prolog III

Prolog III is being developed by the originators of Prolog [9]. Its capabilities are expected to be similar to some of those of CHIP and Trilogy; that is, it will have some capability for solving linear expressions involving integer and rational-valued variables. Until it is released, its full potential and limitations will not be known.

Chapter 3

Design Objectives and Considerations

3.1 Limitations of Existing Approaches

All of the new languages have been written with some problem domain in mind, the domain being defined by the type of values that may be assigned to variables and the algebraic class of the constraints. All but one provide efficient methods to satisfy constraints in very limited domains; only BNR Prolog can solve constraints for a much wider class of problems, but only for interval-valued variables and at the expense of requiring a very general satisfaction procedure. Other, more specific, limitations include:

- Satisfaction methods for constraints involving integer-valued variables cannot be added to CLP($\mathcal{R}$) and BNR Prolog in such a way that the solver remains in the background, nor can methods for real-valued variables be added to Trilogy and CHIP.
- Except for BNR Prolog, none of the languages can be made to solve even simple nonlinear expressions without resorting to Prolog procedures and the use of *is* predicates.

- It is not possible to replace $\text{CLP}(\mathcal{R})$'s simplex algorithm or BNR Prolog's local propagation system with more efficient methods for special case problems.
- Since the languages can only process constraints involving one data type, it follows that there are restrictions on the types of values that can be assigned to variables. For instance, only integer values may be assigned to constrained variables in Trilogy.
- The qualifications assigned to the results of the constraint satisfaction process are at times confusing. For instance, there is always some doubt about whether or not any solution exists between the limits of a BNR Prolog interval.
- The solutions that are returned may be incorrect because of errors introduced by the use of conventional floating point numbers.

Because of the approaches taken in their design, all of the languages have limited capabilities, if any, outside of the targetted domains. Consequently, when constraints cannot be solved by a language, the following options exist to ensure that valid solutions are not lost:

- the constraints can be restated or simplified so that they become solvable,
- Prolog procedures can be used to augment the constraint solver's abilities, or
- another programming language can be used.

Each option is considered below.

Restating or Simplifying the Problem

At times it may be very easy to restate a constraint so that it becomes solvable. For instance, $1/(X+1) = 2$ can easily be rewritten so as to become linear in the eyes of $\text{CLP}(\mathcal{R})$. However, if the available primitive data types are not appropriate, it may be virtually impossible to restate the constraints. For instance, it would be very difficult to rewrite expressions involving real-valued variables if the constraint system could only handle integers. Thus, because of the limited capabilities of most

current languages and the arbitrary complexity of constraints, it is frequently not possible to rewrite expressions so that they become solvable.

Simplifying problems is undesirable as well because it creates more work for the user, problem statements become more complicated, and arithmetic approximations are introduced. Thus, restating or simplifying constraints in order to overcome limitations of the constraint solver is not, in general, an acceptable solution.

Using Prolog Procedures

The option of augmenting the built-in methods with Prolog procedures is not a very good solution because it results in difficulties similar to those which are encountered when a language has no constraint system at all. Every arithmetic relation must be checked against the abilities of the solver and a Prolog procedure must be used whenever an expression falls outside of the solvable class. Since the purpose of the procedure is to solve an arithmetic relation, it must implement a numeric algorithm in Prolog code. This is usually difficult to do, although the abilities of the solver may be of some help.

Consider the quadratic $A \times X^2 + B \times X + C = 0$ when only linear constraints can be solved. It would have to be replaced with a subgoal, say $q(X, A, B, C)$, and a Prolog procedure such as the following:

$q(X, A, B, C) :- X \text{ is } (-B + \text{sqrt}(B*B - 4*A*C)) / (2*A).$

$q(X, A, B, C) :- X \text{ is } (-B - \text{sqrt}(B*B - 4*A*C)) / (2*A).$

$q(X, A, B, C) :- X2 \text{ is } X*X, A*X2 + B*X + C = 0.$

The first two clauses would determine the two solutions for X assuming A , B , and C were known, while the third could determine A , B , or C given X . The latter is possible because the "quadratic" becomes a linear equation in A , B , and C when X is known, and the solver was stated to be capable of solving linear relations. Note, however, that if two values were unknown, say X and A , no solution would be found even if another relation involving X and A was available. There are two reasons for this:

1. all of the above clauses would fail because there are unbound variables to the right of each *is* predicate, and

2. it would require a means of solving simultaneous, nonlinear equations, something that neither the constraint solver nor the above procedure can do.

Unless the user was sure that such a combination of bindings could not occur, more code would have to be written to handle all of the possible cases.

In general, having to augment the constraint solver with Prolog procedures is not desirable because:

- Identifying all of the combinations of values that could result in unsolvable constraints is time consuming, complicated, and error prone.
- Prolog is not a language well suited to implementing numeric algorithms, thus writing Prolog procedures to augment the abilities of the constraint solver can be difficult.
- Inconsistencies arise, some relations being satisfied by the constraint solver, some by user-supplied Prolog procedures, and others by a combination of the two, as in the third clause above.

Using Another Language

If another constraint programming language was used, it might be able to solve all of the expressions but first the program would have to be converted and tested. Besides being time consuming, this approach might only offer a temporary solution because all languages have their limitations and if the program was ever modified, its constraints could become unsolvable once more.

If an imperative language was used, the entire program would have to be reformulated. This would be a major effort for all but the most trivial of programs and all of the benefits of using a constraint programming language would be lost. Thus, using another language is often a poor solution.

The above options indicate that current constraint programming languages are not particularly good tools when they are faced with constraints that they were not designed to solve. Thus, the wider the range of constraints they can solve, the

better. However, when a language has a general purpose constraint solver (as in BNR Prolog), problems may remain as indicated by the answers to the following questions:

Is the solver capable of solving every arithmetic relation that will ever be encountered?

It is very unlikely that any solver will ever be able to satisfy every possible combination of mathematical functions. If the limits of the solver are exceeded, the user is once more faced with the options described above.

Is the solver efficient enough for all of the constraints that it will ever have to solve?

The wider the range of solvable constraints, the more general the satisfaction mechanisms need to be. Since general purpose methods tend to be slower than specialized ones, general purpose solvers are likely to be much less efficient. For instance, BNR Prolog is capable of solving a wide range of constraints using a recursive subdivision algorithm embodied in its *solve* predicate, but it is highly unlikely that such a method is always as efficient as more specialized techniques.

Can the solver generate alternate solutions to problems? If so, can it generate every solution to every combination of constraints?

Usually, general purpose methods do not have information regarding specific problem classes. Thus, they have no efficient way of generating all of the solutions to a set of constraints; in fact, they do not even know how many exist. The only way that alternate solutions can be found is to search, possibly using an iterative procedure such as a bisection search [18] in combination with a series of starting guesses. While such an approach may be acceptable at times, it can be very inefficient. For example, much time could be spent looking for alternate solutions even though none existed.

Can it handle constraints involving various data types, such as floating point, integer, and complex numbers? Can it handle constraints involving more than one type at a time?

As long as problems can involve multiple data types, a constraint system will be required to handle them. No such system exists, however, because of the difficulties involved. In fact, solving problems with only one data type may be very difficult, as demonstrated by current research into efficient methods for solving constraints on integer-valued variables [41]. To solve relations with multiple data types, Prolog's untyped variables would need to be declared as integers, rational, etc. This information could then be used to ensure that variables were bound only to values of the appropriate data type.

Is the solver sensitive to numeric instabilities, causing errors such as indicating a solution exists when there is none or failing to recognize the existence of a solution?

Any constraint solver that makes use of conventional integer or floating point number representations has some limitations. Since computers normally use a fixed number of bytes to represent each value, there are limits to the ranges of integer and floating point values, and limits to which real numbers can be represented at all [11]. Such problems result in $CLP(\mathcal{R})$ having to approximate strict inequalities such as $X > 0$ with $X \geq \epsilon$. Such approximations may prevent acceptable solutions from being found or indicating the presence of a solution when none exists.

Other sources of numeric instability, such as roundoff errors, may also invalidate solutions. Only systems using the interval data type seem to be immune although they have problems of their own. In particular, interval widths may become so wide that the answers become unusable [28, 34]. This is a consequence of the conservative, outward rounding of interval limits whenever arithmetic operations are performed. It may be concluded that regardless of the methods used, there will always be limits to constraint satisfaction techniques.

Are the answers qualified, so as to indicate the possibility of error? If so, are the qualifications unavoidable because of the mathematics or an indication of weaknesses in the satisfaction mechanisms?

It is preferable that answers not be qualified with statements that are confusing or hide useful information. For instance, in BNR Prolog, it is possible for an interval-valued variable to be assigned a range of values even though no solution exists. This is justified by the qualification that applies to all answers, namely "If there is an answer, then it is between the indicated limits." While this is technically correct, unskilled users could easily misinterpret program results.

Since some problems are undecidable, the above qualification is acceptable at times. However, since it also applies to problems for which solutions are known to exist, one can never be sure whether or not indicated intervals are definite solutions. It would be much more preferable if, whenever possible, BNR Prolog qualified its answers with "There is an answer between the indicated interval limits."

While BNR Prolog is conservative with its qualifications, those of $\text{CLP}(\mathcal{R})$ may be incorrect. If $\text{CLP}(\mathcal{R})$ returns YES and a set of values, it is implicit that the values are only accurate to within the limits set by the floating point number representations. Thus, the solutions are potentially wrong and so it may be argued that $\text{CLP}(\mathcal{R})$ should qualify every answer with MAYBE.

Whether they incorporate constraint solvers for very limited domains or for much more general problem areas, all of the new constraint programming languages have some limitations. Each language was implemented with a single constraint satisfaction mechanism so that it would be easier to develop and the solution overhead would be limited. While it may not have been practical to include a vast collection of satisfaction methods in each language, it is felt that providing a very limited capability overly restricts the number of application areas for a language. On the other hand, if a very general satisfaction method is provided, it is unlikely that it can solve every conceivable constraint, and even less likely that it can do so efficiently. Thus, current constraint programming languages are limited by deficiencies in the areas of extensibility and expressiveness.

Extensibility

A constraint programming language is considered to be extensible when it can be made to solve new classes of constraints. Current languages fail in this regard because of the following:

1. None permit the addition of new satisfaction procedures for solving new classes of constraints, nor is it possible to disable the built-in methods when they are not required.
2. None permit the addition of new procedures for solving special case problems more efficiently than the built-in methods.

These two issues are very important because they are a direct cause of the language limitations listed at the beginning of § 3.1. For instance, if new procedures could be added to the constraint solvers of Trilogy and CHIP, the languages could satisfy polynomials involving real-valued variables, and if special case methods could be added to CLP($\mathcal{R}$), the language could solve tridiagonal sets of equations [16] without having to use its much more general simplex algorithm.

Expressiveness

None of the current languages provide a means of allowing users to express metalevel data which could aid the satisfaction process by helping to locate appropriate satisfaction methods (although metalevel data has been used to narrow the search space for a single method [42]). Since current languages do not permit the use of an arbitrary number of satisfaction procedures, the use of such data is admittedly limited. However, it will be seen in Chapter 7 that when multiple methods are available, metalevel data can be of great benefit.

3.2 Design Objectives

There will always be limitations to what can be expected of a constraint satisfaction mechanism. No single procedure has ever been found that efficiently solves all problems, thus more than one method must be available. On the other hand, it is

not practical to incorporate every conceivable procedure because of the huge memory requirements and impractical search times. A more realistic approach is:

- to allow the capabilities of the constraint solver to be extended so that it incorporates all procedures expected to be necessary,
- to provide a means of indicating when the abilities of the solver have been exceeded so that corrective action may be taken, and
- to allow the expression of as much metalevel information as possible, and to utilize it to the fullest in order to aid the satisfaction process.

These broad requirements are dealt with in more detail below.

An Extensible Solver

A new constraint solving architecture is required, one in which the user has direct control over the satisfaction mechanisms. It needs to be extensible so that it can be made to satisfy arbitrary constraints using consistent syntax and semantics. It must allow new satisfaction methods to be incorporated yet not require users to call them explicitly at some point in the program (as in the case of Trilogy, for instance). Instead, the solver needs to remain in the background, solving arithmetic expressions whenever possible.

Handling Satisfaction Failure

Another issue related to extensibility is how well a language handles instances of satisfaction failure, that is, cases in which the system failed to solve a set of constraints or to determine that no solution existed. Whenever the limits of the solver are exceeded, a description of any unsatisfied constraints should be available so that corrective action may be taken, such as adding a new procedure. It is interesting to note that one of the new ideas in the constraint system of Steele [36] was its ability to explain how values were determined; now a facility for explaining why values were not found is required.

Note that cases involving unsatisfiable constraints do not need to be explained since proving a constraint has no solution does not mean that the solver has failed

in any way. Only those cases which could result in valid solutions being lost need to be identified.

Guiding the Solver with Metalevel Data

A more expressive programming language is also needed so that metalevel data regarding constraints may be specified. As is the case with Prolog programs, there are two interpretations of such data. From the declarative viewpoint, it can provide additional information regarding constraints and how they are related to one another; its procedural interpretation can be used to guide the satisfaction process.

Although metalevel data may be contained implicitly in the syntax of constraints and the structure of the program, its explicit declaration should help the solver to find the most efficient satisfaction methods as quickly as possible. Thus, one design objective is to create language constructs that will allow metalevel data to be expressed, and another is to create a solver architecture that can make use of it. As well, the solver should be robust so that imprecise or non-existent data will only slow the satisfaction process.

3.3 Design Considerations

Assuming that the solver can contain an arbitrary number of satisfaction procedures, it must have a means of matching constraints and procedures. Clearly, trying to apply Gaussian elimination to a quadratic equation will not accomplish anything useful. It is recognized that identifying candidate constraints for a particular method may not be easy because even simple relationships may be totally obscured by complicated syntax. For instance, solving the expression:

$$X \times ((2 \times X)/X + (3 \times X))/(2 + 3 \times X) - 1 = 0$$

is trivial, but only if the equivalent expression $X - 1 = 0$ can be detected. To further complicate matters, the form of an expression can change as values are determined.

Matching constraints and procedures results in another design consideration, that is, the desire to minimize the amount of duplicate testing. For instance, if a single check could determine that a constraint was nonlinear, there would be

no point in trying to solve it using Gaussian elimination or a specialized version for tridiagonal equations. Thus, each test for matching constraints and procedures should serve more than one constraint-procedure pair.

Built-in Search Mechanism

If the constraint solver is to support multiple satisfaction procedures, some form of search mechanism is required to find an appropriate method at run-time. As well, a means of giving preference to the most efficient method is needed. This will help to minimize the use of more general satisfaction methods that are slower, are less robust, and may have difficulty generating alternate solutions.

Retaining Data for Future Use

Unless the solver can determine that a constraint is unsatisfiable, it must be able to retain the data for future use. Ideally, when the solver attempts to solve a constraint with one unknown, it should be able to determine a solution immediately or prove that no solution exists. However, this may not be possible because the limitations of the available methods may be exceeded; e.g., an iterative procedure may fail to converge. If a constraint cannot be satisfied or rejected, it needs to be delayed because sometime later, the solver may be able to satisfy it using solutions resulting from forthcoming data.

When a constraint contains more than one variable, it may not contain enough data for the solver to generate a unique solution or to determine that none exists. Therefore, the solver must be able to retain the data for use with forthcoming constraints. Even if a delayed constraint cannot be satisfied as one of a solvable set, the solver may be able to simplify it to the point that it becomes solvable if it has variables in common with other constraints.

There is yet another reason for storing constraints for future use even if the solver can satisfy them as soon as they are encountered. The Prolog inference engine may backtrack into constraints from time to time and so the solver may be requested to generate alternate solutions.

Solver Replies

The solver needs to return some sort of result whenever it is sent a constraint or a request for an alternate solution. Its results should be as meaningful as possible and contain a minimum number of duplicate answers. At a minimum, the solver should be able to indicate the following after being sent a constraint *c* or a request for an alternate solution:

1. *c* has no solution or none other than those returned already,
2. *c* might have a solution,
3. *c* has a solution,
4. *c* might have a solution and a set of values that might satisfy it,
5. *c* has a solution and a set of values that might satisfy it, or
6. *c* has a solution and a set of values that will satisfy it.

The first response is needed to instruct the inference engine to backtrack over an unsolvable constraint; the second and third are required to indicate that a constraint has, or may have, a solution but it is not possible to generate specific values (or there are none to be determined); the fourth response is useful for cases in which the solver is able to satisfy a subset of constraints but it cannot determine if the solution makes other (delayed) constraints unsolvable; the fifth provides an intermediate response between the fourth and the sixth; the sixth response is required for cases in which a solution is known to satisfy a constraint and all those previously encountered.

When requested for alternate solutions, the solver should also be able to return an indication that it abandoned the search for a solution because it ran out of search paths to explore.

Reply Sequences

Another design consideration is the potential sequence of replies that the solver can produce. Some sequences are considered to be undesirable because they could cause valid solutions to be lost or they may result in multiple identical responses and an excessive number of backtrack points. The following cases need to be considered:

- The solver should not be able to backtrack over a constraint until every possible search path (within the solver) has been explored.
- If the solver contained several procedures that could determine that a constraint has (or might have) a solution, it could potentially return as many responses to that effect as there were procedures. Designs that could allow this to happen should be avoided since only one such response is useful.
- The solver should not be able to indicate that a constraint might have a solution before it indicates that a constraint has one for sure. The latter is a stronger response and if the solver can determine such, there is no need for the weaker to ever be returned.

Making Maximum Use of the Data

When a constraint is sent to the solver, all possible search paths must be explored in an attempt to satisfy it. If a set of assignments is generated, they should be applied to every other constraint sharing the variables. This will allow outstanding constraints to be reduced, possibly to the point that they become solvable, and it provides a check to ensure that the generated values do not violate any other constraint.

Chapter 4

The Extensible Architecture

This chapter is divided into five sections. The first presents an overview of the solver architecture, and then in the second, some definitions are given and a detailed description of the solver's processes and their interfaces. The third section presents a short example to demonstrate how the solver operates, the fourth specifies the requirements for a new satisfaction procedure, and the fifth describes how the solver can be modified to handle Type 2 variables.

Type 2 variables are those that are always bound to some (possibly discontinuous) range of values, whereas Type 1 variables are those that are either unbound or are bound to specific values. The latter are by far the most common in both logic and conventional programming languages, but for the purposes of this thesis, Type 1 variables may be considered to be the untyped variables found in all Prolog-based languages. Examples of Type 2 variables are BNR Prolog's interval-valued variables and CHIP's domain variables.

4.1 An Overview of the Solver Architecture

This section presents an overview of the solver architecture using Type 1 variables only. These variables can be used to represent integers, rational and floating point numbers, symbolic values, etc. It is also assumed when a satisfaction method indicates that a solution will satisfy a constraint, the solution will do so when the values

are substituted; that is, extra-logical problems introduced by finite word length limitations will not invalidate the solution.

4.1.1 Basic Processes

The new solver architecture consists of a collection of independent processes, as depicted in Figure 4.1. There is one coordinating process, MP, and a collection of *satisfaction procedures*; the latter are denoted Pa , Pb , Pc , Henceforth, the term *procedure* will refer to MP or a satisfaction procedure unless the context is clear otherwise.

In general, MP acts as an interface between the Prolog inference engine (PE) and the other procedures, and it controls some aspects of the solver's search process. More specifically, its functions include:

- adding new constraints sent from PE and other procedures to its constraint management data structures,
- determining solutions to simple constraints,
- detecting contradictions in simple constraints,
- passing constraints to procedures in the solver,
- returning results determined by itself or other procedures, and
- forwarding requests for alternate solutions to other procedures.

Procedures are the processes of the solver which have the ability to solve non-trivial constraints. Although every procedure shares some common processing characteristics, each one also contains some unique constraint satisfaction capabilities. The functions of a procedure include:

- receiving constraints that are sent from another procedure and determining whether or not use can ever be made of the data,

Figure 4.1: Constraint Solver Processes.

- sending constraints to other procedures and forwarding any useful results that are returned,
- after determining that enough data is available, executing a function to solve for some values,
- requesting MP to verify any solutions that its functions generated,
- returning results to the procedure that sent it the constraint data, and
- determining alternate solutions upon request.

The system is recursive in nature and it may be thought of as the one shown in Figure 4.2. Conceptually, the solver may be considered to contain an infinite number of procedures.¹ Besides two global constraint management data structures which are used by MP, every recursive instance of MP and the other procedures contains some local data for keeping track of what search paths are left to explore at each level of recursion.

Calls to the constraint solver occur whenever PE encounters a constraint. PE sends the constraint in a list to MP, causing MP to update two global data structures which may already contain unsolved constraints. If MP cannot solve the constraint or its solution does not allow every other outstanding constraint to be solved, it sends the updated constraints to the other procedures. If a procedure can generate any solutions, it recursively sends them to MP in the form of a list of *assignment constraints*, that is, expressions of the form *Variable = Value*, and then the whole process repeats. MP updates any unsolved constraints using the new solutions and if any constraints are left unsolved, it calls the other procedures. This process repeats until all constraints are satisfied or no more values can be generated. Finally, each

¹Although the solver's conceptual model contains an infinite number of procedures, an actual implementation would not require an infinite amount of memory. Whenever PE sends a constraint to MP, if the total number of uninstantiated variables in the constraints is n , the solver will recurse k times, $k \leq n$, before a solution is returned (the proof appears in Appendix C). Data does not need to be stored for procedure instances beyond the k^{th} level of recursion and so the memory requirements are limited.

Figure 4.2: Conceptual model of recursive solver.

procedure replies to the call it received and in so doing, the solutions are returned to PE.

4.1.2 Procedure Organization

One of the objectives of the satisfaction process is to solve the constraints as efficiently as possible. Since the solver could contain many applicable procedures, a search for the best becomes necessary. If the procedures were arranged into a list, a search through them might take more time than that required to execute any applicable solution method. A hierarchical organization is much better, procedures becoming more specialized with depth into the tree. The intention is that a constraint solvable by a procedure at level i of the tree should also be solvable by ancestor procedures at levels 1 to $i-1$, the former containing the children of MP.

The solver's search mechanism for locating satisfaction methods is based on the hierarchical organization and is embodied in the characteristics of its procedures. When a procedure P_i is sent a list of constraints, it filters out any that it has been sent before and any that it and its children cannot satisfy. P_i then passes the remaining data to each of its children in turn, giving each one the opportunity to solve the constraints. If any values are found, the solutions are returned back through the hierarchy, otherwise P_i tries to solve the constraints using its own satisfaction methods.

The hierarchical organization combined with this search procedure has three important advantages.

1. Procedures are sent a minimum number of constraints that they cannot process since their ancestors filter out much of the unusable data. This helps to minimize any duplication of effort.
2. Procedures that are deepest in the tree are given the first opportunity to solve the constraints, and, as will be shown, their actions may prevent slower, general purpose procedures from ever being used. Since specialized procedures will be at the leaves of the hierarchy, the solver architecture helps to ensure that constraints are satisfied with the most efficient methods available.

3. The search process built into every procedure results in a generic procedure skeleton. This helps to make the solver extensible in that only a few pieces need to be added to complete a new procedure.

A specific procedure organization is henceforth referred to as a solver *configuration*. Many different configurations are possible for the same set of procedures. For now, it is assumed that every configuration consists of an n -way tree in which MP is at the root, each procedure has at most one parent, and each procedure has MP as one of its ancestors. Other configuration possibilities are discussed in Chapter 5.

4.2 A Detailed Description

The following notation appears in the definitions and later subsections.

- C^- denotes the set of constraints sent to the solver prior to constraint c , and C denotes the set $C^- \cup \{c\}$.
- P_i denotes any procedure in the solver other than MP.
- P_i^+ denotes any procedure in the solver, including MP.
- $MP(k)$, $k \geq 0$, denotes procedure MP at the k^{th} level of recursion.
- $P_i(k)$, $k \geq 0$, denotes a procedure P_i at the k^{th} level of recursion.
- $P_i^+(k)$, $k \geq 0$, denotes a procedure P_i^+ at the k^{th} level of recursion.

4.2.1 Definitions

Constraint Definitions

Definition 1

A *reduced constraint* is a constraint that has had (1) at least one of its primitive mathematical functions executed (such as addition or multiplication), or (2) at least one of its variables replaced with a numeric or symbolic value.

Definition 2

A *totally reduced constraint* is a constraint which has had all of its primitive mathematical functions executed and all of its variables replaced with numeric or symbolic values. Symbolic solutions would involve the assignment of character strings to variables, the strings possibly including symbols representing arithmetic operators. For instance, constraints of the form $12 = 12$, $x0 \leq 3$, and $4 \times y0 + 1 = z0$ are all totally reduced² while $X + y0 = 1$ is not.

Definition 3

A constraint c is *feasible* if every constraint in C has at least one common solution. If no such solution exists, c is said to be *nonfeasible*. Whether or not the solver can determine either case is irrelevant.

Definition 4

A constraint c is *verifiable* when the solver can determine that c is feasible, regardless of whether or not the solver can determine substitutions for any of its variables.

*Solution Definitions***Definition 5**

A list S of variable/value substitutions is known as a *full solution* (or just a *solution*) to a constraint c when S contains substitutions for every variable in c ; S may also contain substitutions for other variables in C^- .

Definition 6

A list S of variable/value substitutions is known as a *partial solution* to a constraint c if S contains substitutions for some but not all of the variables in c ; S may also contain substitutions for other variables in C^- .

Definition 7

If a partial or full solution S to a constraint c makes it verifiable, then S is known as a *definite solution*.

²Note that $x0$ and $y0$ are considered to be symbolic values rather than variables because they start with lower case letters, a Prolog convention.

Definition 8

If a constraint c is not verifiable with a partial or full solution S applied, then S is known as a *tentative* partial or full solution.

Definition 9

Two solutions $S1$ and $S2$ are considered to be different if (1) either solution contains a substitution that the other does not, or (2) $S1$ and $S2$ were derived using a different combination of procedures even if they contain the same substitutions. (The latter case is necessary for some of the properties presented in Chapter 6.)

*Function, Procedure, and Solver Replies***Definition 10**

Functions, procedures, and the solver are said to have *verified* a list L of constraints when they have determined that the constraints in L are feasible.

Definition 11

Functions, procedures, and the solver are said to have *rejected* a list L of constraints when they are passed the list and they return NO.

4.2.2 Process Interfaces

There are only four interfaces that need to be defined because each procedure generates and responds to calls in a predetermined and limited number of ways. The interfaces are as follows:

1. the PE to MP interface,
2. the MP to procedure interface,
3. the procedure to procedure interface, and
4. the procedure to MP interface.

The PE to MP Interface

Table 4.1 defines the PE to MP interface. There are only two types of messages that PE can send to MP. PE will send a *Constraint Message* (CM) whenever it encounters a new constraint, and it will send an *Alternate Value Request* (AVR) when it backtracks into one. CM's have two arguments, the first being a list of one constraint. The second argument is undefined; it is only included to be consistent with CM's sent between other procedures. AVR messages have no arguments because it is implicit that they refer to the data contained in the previous CM.

PE to MP Interface		
Message	MP Reply	Interpretation
CM($L,-$) or AVR	MAYBE	The constraints might be feasible.
CM($L,-$) or AVR	YES	The constraints are feasible.
CM($L,-$) or AVR	MAYBE(S)	S is a full or partial, tentative solution.
CM($L,-$) or AVR	YES+MAYBE(S)	Constraints are feasible and S is a full or partial, tentative sol'n.
CM($L,-$) or AVR	YES(S)	S is a full or partial, definite solution.
CM($L,-$)	NO	L contains nonfeasible constraints.
AVR	NO	No solution is possible (other than those returned already).
AVR	ABANDON	The solver has exhausted all possible search paths.

Table 4.1: PE to MP Interface. L contains a single constraint.

Table 4.2 lists the possible reply sequences that the solver can produce with regard to a constraint c . The following observations are made:

1. After returning MAYBE or YES, a subsequent AVR will cause MP to return NO or ABANDON. This indicates that MAYBE and YES can be returned once only and so the number of duplicate answers is minimized. As well, the solver cannot indicate that a constraint might have a solution after indicating that it has one for sure, and vice versa.
2. When MAYBE or YES is returned, it is known that the solver has exhausted all of the possible search paths looking for a solution. It is hoped that PE can continue without values for any uninstantiated variables appearing in the constraints and that appropriate solutions will be found (using data yet to arrive) before PE must abandon a search path.
3. Once YES(S) or YES+MAYBE(S) are returned, MAYBE and MAYBE(S) cannot be returned.
4. MP cannot return any other solution regarding c after it has returned NO or ABANDON. This ensures that (1) MP will not return any solutions to c after it has told PE to backtrack over it, and (2) MP cannot tell PE to backtrack over c more than once.

The MP to Procedure Interface

Table 4.3 describes the MP to procedure interface. MP can send CM's and AVR's to its children just as PE sent it CM's and AVR's. The reply interpretations are the same as those of MP, however, the following differences exist:

- a procedure cannot return ABANDON in response to an AVR,
- the first argument of a CM now contains a list of one or more constraints,
- the second argument of a CM is always set to 1, and
- MP may send BACKTRACK messages to its children.

The need for the second argument in a CM is explained in the description of the procedure to procedure interface. As with the PE to MP interface, it is used here

Possible Reply Sequences
MAYBE $\rightarrow$ ABANDON
YES $\rightarrow$ ABANDON YES $\rightarrow$ NO
MAYBE(S) $\rightarrow$ NO MAYBE(S) $\rightarrow$ MAYBE MAYBE(S) $\rightarrow$ YES MAYBE(S) $\rightarrow$ MAYBE(S') MAYBE(S) $\rightarrow$ YES+MAYBE(S') MAYBE(S) $\rightarrow$ YES(S')
YES+MAYBE(S) $\rightarrow$ NO YES+MAYBE(S) $\rightarrow$ YES YES+MAYBE(S) $\rightarrow$ YES+MAYBE(S') YES+MAYBE(S) $\rightarrow$ YES(S')
YES(S) $\rightarrow$ NO YES(S) $\rightarrow$ YES YES(S) $\rightarrow$ YES+MAYBE(S') YES(S) $\rightarrow$ YES(S')

Table 4.2: Possible Reply Sequences. Each sequence is denoted $X \rightarrow Y$ where Y is the solver's response to an AVR after previously returning X .

only to be consistent with CM's sent between other procedures in the solver. The BACKTRACK message is used by MP to tell its children to discard any data which was sent in the previous CM; procedures have no meaningful reply to these messages.

MP to Procedure Interface		
Message	P_i Reply	Interpretation
CM($L,1$) or AVR	MAYBE	The constraints might be feasible.
CM($L,1$) or AVR	YES	The constraints are feasible.
CM($L,1$) or AVR	MAYBE(S)	S is a full or partial, tentative solution.
CM($L,1$) or AVR	YES+MAYBE(S)	Constraints are feasible and S is a full or partial, tentative sol'n.
CM($L,1$) or AVR	YES(S)	S is a full or partial, definite solution.
CM($L,1$)	NO	L contains nonfeasible constraints
AVR	NO	No solution is possible (other than those returned already).
BACKTRACK	-	

Table 4.3: MP to Procedure P_i Interface. L contains a non-empty list of constraints.

The Procedure to Procedure Interface

The procedure to procedure interface is described in Table 4.4 and it is very similar to the MP to procedure interface. All of the calls and responses are the same with the exception that the second argument of a CM may now be 0 or 1.

A procedure can return YES, YES+MAYBE(S), or YES(S) only if it or one of its descendants can determine that there is at least one solution to every constraint that has been sent to the solver. A procedure can determine such only if it has

access to every constraint and since a CM list may only contain a subset of the total data, the second argument becomes necessary. Since MP has access to every constraint, it always sets the second argument to 1 as indicated in Table 4.3. Other calling procedures will set the argument to 1 only if they were sent every constraint and they accepted them all as being potentially useful.

Procedure to Procedure Interface		
Message	Procedure Reply	Interpretation
CM(L,V) or AVR	MAYBE	The constraints might be feasible.
CM(L,V) or AVR	YES	The constraints are feasible.
CM(L,V) or AVR	MAYBE(S)	S is a full or partial, tentative solution.
CM(L,V) or AVR	YES+MAYBE(S)	Constraints are feasible and S is a full or partial, tentative sol'n.
CM(L,V) or AVR	YES(S)	S is a full or partial, definite solution.
CM(L,V)	NO	L contains nonfeasible constraints.
AVR	NO	No solution is possible (other than those returned already).
BACKTRACK	–	–

Table 4.4: Procedure to Procedure Interface. L contains a non-empty list of constraints, and V may be set to 0 or 1.

The Procedure to MP Interface

The procedure to MP interface is almost identical to that of the PE to MP interface and so Table 4.1 may be used. The only difference is that unlike PE, procedures may send more than one constraint in a CM list.

4.2.3 MP Description

The two global constraint management data structures used by MP are VV and SL. See Figure 4.3. VV is a table of all of the constraint variables and their values, NULL being assigned if a value is unknown. SL is a list of stacks, one for every constraint that is sent to MP. The top of each stack contains the most recent version of the constraint as each goes through a series of simplifications before it is totally reduced. The set of constraints consisting of the top constraint of each stack is known as MP's *problem set*.

As previously discussed, the solver may be thought of as having an infinite number of MP procedures, one for each level of recursion. Each instance of MP, denoted $MP(k)$, contains a private stack, each entry of which contains a four-field record. The fields contain:

1. the state $MP(k)$ was in when it received the CM (denoted LS for last state).
2. V , a flag indicating whether or not any child verified that the constraints were feasible.
3. the last child that it called before it returned a solution, and
4. the changes that $MP(k)$ made to VV and SL between the time that it received the CM and the time that it replied.

$MP(k)$ also uses a list $D(k)$ for accumulating solutions to be returned to its caller. Although its use is somewhat redundant considering the fourth record field, explanations are simplified if the two are kept separate.

The state transition diagram of $MP(k)$ is shown in Figure 4.4. $MP(k)$ is initially in state 0 but in general it can receive a message while waiting in state 0, 1, 2, or 3. When it receives a CM containing list L of constraints, it creates a new stack record, enters the state that it is leaving in the first field, initializes the second field to 0, and then makes transition 1, 17, 18, or 19, depending on the value of LS. $MP(k)$

(a) Constraint $c(X)$ sent to MP

(b) VV and SL updated

(c) $d(X, Y)$ sent to MP, VV and SL updated

(d) Value for X sent to MP, VV and SL updated

Figure 4.3: Constraint management data structures.

creates a new stack in SL for each constraint in L ,³ and if the constraints contain any new variables, they are added to VV. Next, MP examines its problem set; the following outcomes are possible:

1. If an *MP-rejectable* constraint is detected, $MP(k)$ discards any changes it made to VV and SL (recorded in the fourth field of its stack), returns to the state indicated in the first field, pops the stack, and returns NO. Consequently, it has discarded any changes resulting from the CM. (MP-rejectable constraints are discussed later in this section; for now it suffices to say that they are nonfeasible constraints, such as $1 = 2$).
2. If $MP(k)$'s problem set is totally reduced and satisfied, it will make transition 15 to state 1 and return YES to its caller.
3. If $MP(k)$ detects an *MP-solvable* constraint in its problem set when it reaches state 7, it will make transition 24 to state 6. (MP-solvable constraints are discussed later in this section; for now it suffices to say that they are constraints which are solvable using the method of isolation. An assignment constraint is a trivial example.)
4. Otherwise, $MP(k)$ will make transition 2 to state 4.

If the third case arises, $MP(k)$ will make transition 24 and solve the constraint that caused the transition. It does so by using the method described below, and then it:

- adds the solution to $D(k)$ and VV,
- substitutes the result into any constraint sharing the variable,
- reduces any constraints that it can by performing primitive arithmetic operations, and

³From a practical standpoint, any assignment constraints in L do not need to be entered into new stacks because they are satisfied already; thus, the solutions they contain could be added to VV directly. However, for the sake of consistency, such constraints will be treated as if they were not yet solved (unless noted otherwise).

Figure 4.4: MP state transition diagram.

- rechecks its problem set.

If another MP-solvable constraint arises, $MP(k)$ will make transition 3, solve the constraint, return to state 6, and so on. If an MP-rejectable constraint ever arises, $MP(k)$ will behave as described in Item 1 above; that is, it will discard any changes resulting from the CM and return NO. If, on the other hand, $MP(k)$ totally reduces and satisfies its problem set, it will make transition 25 to state 1 and return $YES(D(k))$ to its caller, $D(k)$ containing all of the solutions found since $MP(k)$ received the CM.

If the fourth case arises or $MP(k)$ makes transition 4 after exhausting the MP-solvable constraints, it then goes into a loop. $MP(k)$ passes its problem set to its first child by making transition 7 and then it waits for a reply. If MAYBE is returned, it makes transition 8 and the process repeats. If a child returns YES, $MP(k)$ makes transition 9 and sets V equal to 1 (to remember that a solution does exist), and it continues to call its children.

If a child P_i of $MP(k)$ returns a response other than MAYBE or YES, $MP(k)$ makes transition 10, 11, 12, 13, or 21. If P_i returns NO, $MP(k)$:

1. discards the changes to VV and SL (recorded in the fourth stack field),
2. sends BACKTRACK messages to any children which were called before P_i .
3. returns NO to its caller,
4. returns to state LS, and
5. pops its stack.

If a child returns $YES(S)$ or $YES+MAYBE(S)$, $MP(k)$ returns $YES(S')$ or $YES+MAYBE(S')$ where $S' = S \cup D(k)$. As well, $MP(k)$ sets V to 1 and records which child returned the solution in the third field of the stack. If a child returns $MAYBE(S)$, $MP(k)$ returns $MAYBE(S')$ if V is set to 0, or $YES+MAYBE(S')$ if V is set to 1; it also records which child returned the solution.

If $MP(k)$ exhausts all of its children before a reply other than MAYBE or YES is returned (or if $MP(k)$ has no children at all), it makes transition 5, 6, 27, or 28, depending on the value of V and the contents of $D(k)$.

MP(k) and AVR's

If $MP(k)$ is left in state 1, 2, or 3 after replying to its caller, it may receive an AVR regarding the constraints in the last CM. If $MP(k)$ is in state 1, its problem set was totally reduced and satisfied without any choices being made and so no alternate solutions exist. Consequently, $MP(k)$ will undo changes to VV and SL, reply NO, return to LS, and pop its stack.

If $MP(k)$ receives an AVR while in state 2, it had previously replied YES or MAYBE after all of its children were exhausted. Since there are no search paths left to explore, $MP(k)$ discards any changes to VV and SL, sends BACKTRACK messages to all of its children, replies ABANDON, returns to LS, and pops its stack.

If $MP(k)$ is in state 3 when it receives an AVR, it had previously returned a solution forwarded by one of its children. Thus, $MP(k)$ will (1) send an AVR to that child to see if it can generate another solution, and (2) return to state 5, prepared to continue the search from where it was left off.

MP-Rejectable and MP-Solvable Constraints

An MP-rejectable constraint is a constraint that MP may find to have no solution. Candidate expressions are (1) constraints which cannot be solved using the method of isolation because they violate some arithmetic axiom, such as $X=3/0$, and (2) totally reduced expressions which are false, such as $1=2$.

An MP-solvable constraint is a constraint which MP can convert into an assignment constraint by applying the method of isolation and the functions listed in Table 4.5. Candidate expressions are equations which contain a single occurrence of just one variable. A trivial example of an MP-solvable constraint is an assignment constraint; a more difficult example can be found in § 2.3.1.

Note that none of the functions in Table 4.5 involve multiple solutions; for instance, cosine and sine functions are supported but not arccosine and arcsine. This prevents significant limitations from being built into the framework of the solver.

For instance, if a function has an infinite number of solutions and the system could generate only a few of them, possible solutions could be lost. On the other hand, the alternative of generating every solution could make the reduction process non-terminating. By not dedicating the solver to one approach or the other, it can be configured however one wishes.

MP Functions
addition, subtraction
multiplication, division
exponentiation
sine, cosine, tangent
natural exponent and logarithm
base 10 exponent and logarithm

Table 4.5: MP Functions.

4.2.4 Procedure Components

A typical procedure P_i is shown in Figure 4.5. Besides containing a built-in search mechanism that is common to all procedures (other than MP), P_i contains:

- *recognition rules*,
- *a problem stack*,
- *a data stack*,
- *a set of satisfaction functions*,
- *a sufficiency predicate*, and
- *a function call list*.

Figure 4.5: Typical Procedure P_i .

Recognition Rules

A procedure's recognition rules, defined as a Prolog procedure, select the constraints that the procedure's functions are capable of solving; they do so by comparing the syntax of constraints passed in CM's with a previously specified form. This approach results in a double-edged sword because it requires constraints to be syntactically recognizable, yet it allows any syntax to be used.

The rules of every procedure P_i must be defined so that:

- P_i can recognize any constraint that its satisfaction functions are capable of processing, either individually or in conjunction with other data.
- P_i must recognize any constraint that its children can recognize, otherwise the children will not be passed every constraint that their functions can solve.
- P_i can recognize any constraint that MP has found to be satisfied. This is required to ensure that P_i eliminates satisfied constraints from its problem set (for the reasons given in Restriction SF4, § 4.4). Appendix B discusses the use of *constraint substitutions* to improve efficiency and to help avoid the problems of recognizing totally reduced and satisfied constraints.

A constraint is *recognizable* if there is at least one procedure in the solver which can match the constraint's syntax with a form defined in its recognition rules; otherwise the constraint is *unrecognizable*.

Problem Stack

Each entry in P_i 's problem stack contains a list of constraints that it has accepted. The data is private in the sense that it can only be accessed by P_i , but it is global in the sense that it can be accessed by each recursive instance of $P_i(k)$. The topmost stack entry is known as P_i 's *problem set*.

Data Stack

As in the case of $MP(k)$, procedures may be thought of as having individual data stacks, one for each level of recursion. Each stack entry is a six-field record with the fields containing:

1. the state $Pi(k)$ was in when it received the CM (again denoted LS for last state),
2. V , a flag indicating whether or not the constraints in Pi 's problem set are feasible,
3. the last child that $Pi(k)$ called before it returned a solution,
4. a list of changes to Pi 's problem set,
5. A , a flag indicating whether or not Pi 's problem set contains every available constraint, and
6. a pointer to the next function call to be made.

$Pi(k)$ uses the first three fields in the same way that $MP(k)$ uses the first three fields of its local stack. The fourth is used to keep track of what changes have been made to the problem set so that they may be undone if necessary, and the fifth is passed as an argument in function calls. The sixth field enables $Pi(k)$ to keep track of what search paths are left to explore.

Satisfaction Functions

Satisfaction functions are the part of a procedure that actually solve the constraints. Whenever a function is called, it is passed the procedure's problem set and the fifth field of the data stack record; i.e., an indication of whether or not the set contains every available constraint. Eventually, functions must return one of the responses listed in Table 4.6. They are not expected to retain any state information and procedures will not backtrack into them in an attempt to find alternate solutions.

Other restrictions that govern function behaviour are discussed in § 4.4. Appendix A discusses the problems that occur when a function returns YES; it also helps to explain the procedures' search mechanism.

The purpose of a procedure having multiple functions is two-fold. First, it allows a separate function to be used for each alternate solution, and second, it allows alternate solution methods to be used in case of numerical stability problems. Further

Function Results	
Reply	Interpretation
MAYBE	The problem set constraints might be feasible.
YES	The problem set constraints are feasible.
MAYBE(<i>S</i>)	The problem set constraints might be feasible and <i>S</i> is a tentative, full or partial solution.
YES(<i>S</i>)	The problem set constraints are feasible and <i>S</i> is a definite, full or partial solution.
NO	The problem set is nonfeasible, or there are no solutions other than those returned already.

Table 4.6: Function Results.

details of their use appear below under the description of a procedure's function call list.

At times it is very advantageous if the functions of a procedure can process constraints incrementally. This prevents the entire set of constraints in a procedure's problem set from having to be resolved from scratch whenever the procedure's problem set is updated; i.e., the new data can be added to a set that is already partially solved. While provisions for doing so are not included in the following descriptions, it could be accomplished by using two more pointers in the stack field record. As indicated in Figure 4.6, a seventh field would access intermediate data structures created by satisfaction function calls, and the eighth would point to the seventh field of the previous stack entry. This would provide the necessary link for incremental processing.

Sufficiency Predicate

The sufficiency predicate is a Prolog procedure that examines the procedure's problem set and determines if there is enough data available to warrant calling

Figure 4.6: P_i data structures.

the satisfaction functions. The predicate succeeds when there is enough data and fails otherwise. A classic example involves a procedure that can apply Gaussian elimination to a set of linear equations. In such a case, the sufficiency predicate would test to see if the number of equations and variables were equal.

There may be instances when it is desirable to set the sufficiency predicate to always succeed. Equations in one unknown do not need to be combined with other constraints and so the satisfaction functions should be called whenever new data arrives.

Function Call List

A procedure's function call list indicates the order in which its functions will be called and the arguments that will be passed. There can be a one-to-one correspondence between the listed calls and the available functions, or various functions could be called multiple times with different input arguments. For instance, if a procedure were to solve quadratics, it could have one function for each solution and a third to indicate that there are no others; this would require the use of three calls to the three different functions. Another possibility is to use three calls to a single function, each call containing different input flags. Thus, the use of a separate function call list provides more flexibility than is otherwise available.

4.2.5 Procedure Description

The state diagram of a typical procedure $P_i(k)$ is shown in Figure 4.7. $P_i(k)$ is initially in state 0 but in general it will be waiting for a CM in state LS=0, 2, 5, 8, or 11. If it receives a CM containing list L of constraints and flag $A0$, $P_i(k)$ creates a new stack record, stores the state it was leaving in the first record field, and makes transition 1, 37, 16, 22, or 36 to state 1. In making the transition, it tests the constraints in L that are not already in its problem set with its recognition rules. New or previously unrecognized constraints that pass the rules are added to the problem set; those that are reduced versions of previously recognized constraints are used to replace the older versions. Any changes to the problem set are recorded in the fourth field of the new stack record.

Figure 4.7: *Pi* state transition diagram.

If the problem set is not updated because no new data can pass the recognition rules, $P_i(k)$ makes transition 2 to state 2 and returns MAYBE. If $P_i(k)$ accepts every constraint in L and $A\theta$ is 1, it makes transition 3 and initializes A to 1. Otherwise, it makes transition 4 and sets A to 0.⁴ In making transition 3 or 4, $P_i(k)$ also initializes (1) V to 0 (since it has not been determined that its problem set is feasible) and (2) the sixth field to the first function call.

Both transitions 3 and 4 result in $P_i(k)$ reaching state 3. If it has any children, it sends a CM to the first with its problem set and the value of A as arguments. If MAYBE is returned, $P_i(k)$ makes transition 6 back to state 3, it calls its next child, and the process repeats. If a child returns YES, $P_i(k)$ sets V equal to 1 (to remember that the constraints are feasible) and it makes transition 7 back to state 3 to continue the search. Eventually, $P_i(k)$ will exhaust all of its children or a child will return a response other than MAYBE or YES.

If a child returns NO, $P_i(k)$ makes transition 12 and in so doing it:

- sends a BACKTRACK message to any child that was sent a CM.
- discards any changes made to its problem set (recorded in the fourth data record field),
- returns NO to its caller.
- returns to state LS (indicated in the first data record field), and
- pops its data stack.

If a child returns YES(S) or YES+MAYBE(S), $P_i(k)$ makes transition 11 or 41 respectively to state 5, sets V to 1, records the child that returned the solution in the third record field, and returns YES(S) or YES+MAYBE(S) to its caller. If a child returns MAYBE(S) and V is 0, $P_i(k)$ makes transition 9, records the child in the third field, and returns MAYBE(S). If a child returns MAYBE(S) and V is 1, $P_i(k)$ makes transition 10, records the child, and returns YES+MAYBE(S).

⁴A value of 1 for A indicates that $P_i(k)$'s problem set contains every available constraint, but this requires (1) $P_i(k)$ to have been sent every constraint (as indicated by $A\theta = 1$), and (2) $P_i(k)$ to have accepted every one of them

If $Pi(k)$ has no children or they all return MAYBE or YES, $Pi(k)$ makes transition 5 and tests its problem set with its sufficiency predicate. If the predicate fails, $Pi(k)$ makes transition 18 or 39 and returns MAYBE or YES, depending on the value of V . If the sufficiency predicate succeeds, $Pi(k)$ makes transition 17 to state 9.

$Pi(k)$ now starts to test its satisfaction functions using transition 23. It makes the function call pointed to by the sixth record field and it passes its problem set and the value of A as arguments. It also increments the sixth field record (so that it points to the next function call), and it waits in state 10 for the function to reply. If MAYBE is returned, $Pi(k)$ makes transition 25 back to state 9 and the process repeats. If the function returns YES, $Pi(k)$ sets V equal to 1 (if it was not already) and it continues with the next function call.

If a function returns NO, $Pi(k)$ makes transition 29 and in so doing it:

- sends BACKTRACK messages to all of its children.
- discards any changes made to its problem set,
- returns NO to its caller.
- returns to state LS. and
- pops its data stack.

If a function returns YES(S) or MAYBE(S), S being a list of solutions in the form of assignment constraints. $Pi(k)$ makes transition 26 or 27 and it sends S in a CM to $MP(k+1)$. $Pi(k)$ also sets V to be 1 if transition 27 is made.

If $MP(k+1)$ returns NO, $Pi(k)$ makes transition 28 in order to continue the search with the next available function call. If there are none left, $Pi(k)$ makes transition 19 or 20 and returns MAYBE or YES, depending on the value of V . If $MP(k+1)$ returns YES(S') or YES+MAYBE(S'), $Pi(k)$ returns the same to its caller by making transition 33 or 42, and sets V to 1. If MP returns MAYBE(S') and V is 0, $Pi(k)$ returns MAYBE(S') using transition 32; if V is 1, $Pi(k)$ returns YES+MAYBE(S') using transition 31.

There are no transitions to handle the cases of $MP(k+1)$ returning MAYBE or YES because $MP(k+1)$ cannot return these responses (although $MP(0)$ can return

them to PE). Whenever $P_i(k)$ sends $MP(k+1)$ a CM, it will send it a list S of assignment constraints and so $MP(k+1)$ will either reject the values or it will return them as part of a list of substitutions.

$P_i(k)$ and AVR's

Besides CM's, $P_i(k)$ may be sent AVR's if it is waiting in state 5 or 11. If it is in state 5, $P_i(k)$ had previously returned a solution generated by the child indicated in the third record field and so it forwards the AVR to that child. $P_i(k)$ then moves back to state 4 to continue the search from where it was left off. $P_i(k)$ may return to (1) state 5 if the child returns another solution, (2) state LS if the child returns NO, or (3) state 3 if the child can return only MAYBE or YES. The latter causes $P_i(k)$ to make transition 6 or 7 back to state 3 and so the search continues with its next child.

If $P_i(k)$ receives an AVR while in state 11, it had previously returned a solution after calling one of its functions and making a recursive call to $MP(k+1)$. The AVR causes $P_i(k)$ to make transition 34 to state 6 and send an AVR to $MP(k+1)$. Depending on $MP(k+1)$'s reply, $P_i(k)$ may return to state 9 in order to continue the search with the next available function call (indicated in the sixth record field), or it may make transition 51, 52, 53, or 54 and reply to its caller.

State 6 is necessary for one simple reason: that is, $MP(k+1)$ can return NO but not ABANDON if it is sent a CM, but it can return either if it is sent an AVR. Thus, transition pairs 28 and 58, 31 and 51, 32 and 52, 33 and 53, and 42 and 54 perform the same operations, that is, they either allow another function to be called or they forward a solution returned by $MP(k+1)$.

4.2.6 Propagating BACKTRACK Messages

BACKTRACK messages are used to force procedures to discard data that has been found to be nonfeasible or data for which all possible solutions have already been returned. BACKTRACK messages originate when (1) a procedure $P_i(k)$ executes one of its satisfaction functions and NO is returned, or (2) when $MP(k)$ is asked to generate an alternate solution when it is in state 2; i.e., after all of solver search

paths at the k^{th} level of recursion have been exhausted.

If a function of $P_i(k)$ returns NO, $P_i(k)$ sends BACKTRACK messages to all of its children as part of transition 29. These children are in state 2 or 8 since they must have returned MAYBE or YES for $P_i(k)$ to have made transition 5, 17, and 23. If a procedure is in state 2, it did not accept any of the constraint data and so it did not send CM's to any of its children. Thus, it does not need to propagate the BACKTRACK messages as part of transition 38. However, if a procedure is in state 8, it must tell all of its children to forget the data and so it issues BACKTRACK messages as part of transition 21.

When $P_i(k)$ returns NO, its parent must tell all of the children it called before $P_i(k)$ to forget the data and so more BACKTRACK messages are issued. If $MP(k)$ is the parent of $P_i(k)$, it sends the messages as part of transition 21; if a procedure $P_j(k)$ is the parent of $P_i(k)$, it sends the messages as part of transition 12. $P_j(k)$ returns NO to its parent and the parent behaves in an identical manner to $P_j(k)$. Eventually NO is returned to $MP(k)$ and so all procedures that were sent the data are told to discard it.

If $MP(k)$ is sent an AVR while in state 2, it issues BACKTRACK messages to all of its children as part of transition 20. Each procedure in the solver tells its children to discard the data (unless the procedure did not accept any of it), and so all traces of the data are lost.

4.2.7 Comments on the Search Strategy

The search strategy is quite similar to that used by Prolog interpreters which is desirable from a consistency standpoint. There is a depth-first aspect since a procedure's children are given the first opportunity to solve a set of constraints, and the ordered search through a procedure's children is analogous to the left-to-right search through the subgoals in a clause. Consequently, the search strategy lets the leftmost applicable procedure that is deepest in the tree execute a satisfaction function first.

This approach is believed to be reasonably efficient but it is not perfect. On one hand, the most specialized function is being sought, the search being performed

from general purpose through to specialized, top down through the tree. This is very efficient for rejecting entire sets of procedures that cannot solve a constraint, and it identifies in reverse order the procedures that will be given the opportunity to call their satisfaction functions.

Unfortunately, the hierarchy does not express the relative degree of specialization of procedures in different branches. For instance, two siblings might both be capable of solving some constraints but any differences in their efficiency is not reflected in the hierarchy. The only means of controlling which gets used first is by altering the left-to-right order in which their parent will call them. This is similar to the technique used by Prolog programmers to control which subgoals in a clause are proven first.

It is recognized that there is a trade-off between the time spent searching for an efficient method and the time that could be saved by just using the first capable one encountered. However, specialized procedures offer other important benefits that makes locating them worthwhile. General purpose routines cannot generate alternate solutions to a problem without searching since they do not even know if alternate solutions exist. This could result in very inefficient and unproductive processing and the unnecessary generation of tentative solutions. Besides, unless general purpose procedures were capable of generating all of the possible solutions and knowing that there were no others, the specialized procedures would still have to be used since other solutions might exist. As well, specialized procedures are more likely to be able to determine the exact number of solutions to a set of constraints and this information can be used to halt the search for alternates at the earliest opportunity. This is demonstrated by the third function returning NO in the example of § 4.3. As well, specialized procedures may even be able to simplify the problem so that alternates become progressively easier to find. For instance, each solution to a polynomial can be used to reduce its order. Thus, the use of specialized procedures can repay the effort taken to find them in several ways.

4.3 A Simple Example

This example demonstrates several of the basic solver operations, including how it delays unrecognizable constraints, how it can generate alternate solutions, and how it may be able to satisfy a constraint even though no procedure can recognize it. The sample equations are:

$$c1: X^X = 4$$

$$c2: X^2 - 3 \times X + 2 = 0$$

The tree consists of a single procedure Q that can solve quadratics (Fig. 4.8(a)). It has two functions for calculating the two roots, and a third that indicates there are no other solutions if one is requested.

When $c1$ arrives, $MP(0)$ creates a new stack record, it adds a new stack to SL , and it enters the variable X into VV (Fig. 4.8(b)). $MP(0)$ is unable to satisfy or reject c and so it makes transition 2 to state 4. It then sends $c1$ in a list to $Q(0)$. $Q(0)$ creates a new stack record and tries to recognize the constraint in L . It fails to recognize $c1$ and so it makes transition 2 to state 2 and returns MAYBE (Fig. 4.8(c)). In turn, $MP(0)$ makes transition 8 back to state 4. Since it has run out of children to test and no values are available, $MP(0)$ makes transition 5 and returns MAYBE to PE.

$MP(0)$ handles the arrival of $c2$ in the same way it handled $c1$ (Fig. 4.8(d)). It creates a new stack record, adds a new stack to SL , and examines its problem set. It cannot reject or satisfy $c2$ and so it sends its problem set to $Q(0)$. $Q(0)$ creates a new stack record, accepts the second constraint, and makes transition 4 to state 3. It has no children to call and so it tries to solve $c2$ itself. Its sufficiency predicate succeeds when tested and so $Q(0)$ makes transition 17 to state 9. $Q(0)$ calls its first function and $YES([X=1])$ is returned. This causes $Q(0)$ to recursively call $MP(1)$ and send it the assignment constraint (Fig. 4.9(a)). $Q(0)$ is left waiting in state 12 for a reply.

When $MP(1)$ receives the message, it adds a third stack to SL and it moves to state 7. It detects that the new constraint is MP-solvable and so it makes transition

24. $MP(k)$ enters the value for X into VV , it substitutes the value into every constraint in SL , and it reduces them. Upon re-examining its problem set, $MP(k)$ determines that $c1$ is nonfeasible and so it (1) discards the changes to SL and VV , (2) it makes transition 26, and (3) it returns NO to $Q(0)$. This causes $Q(0)$ to make transition 28 back to state 9 and to make its next function call. $YES([X=2])$ is returned and so $Q(0)$ recursively calls $MP(1)$ and it waits in state 12 for a reply (Fig. 4.9(b)).

When $MP(1)$ receives the CM, it once more adds a third stack to SL and moves to state 7. It detects that the new constraint is MP-solvable and so it makes transition 24. It enters the value for X into VV , it substitutes the value into every constraint in SL , and it reduces them. Unlike the previous time, every constraint is now totally reduced and satisfied, thus $MP(k)$ makes transition 25 to state 1, returning $YES([X=2])$ to $Q(0)$. This causes $Q(0)$ to make transition 33 and returns $YES([X=2])$ to $MP(0)$ (Fig. 4.9(c)). Finally, $MP(0)$ makes transition 13 and returns $YES([X=2])$ to PE.

If PE backtracks into $c2$, it sends an AVR to $MP(0)$ (Fig. 4.9(d)). Since $MP(0)$ was left in state 3, it sends an AVR to $Q(0)$ before returning to state 5. $Q(0)$ was left in state 11 and so it sends an AVR to $MP(1)$ before moving to state 6. $MP(1)$ was left in state 1 and so it makes transition 16. It discards the last changes it made to VV and SL , and it returns NO to $Q(0)$ (Fig. 4.10(a)). $Q(0)$ makes transition 58 back to state 9 and it calls its last function (Fig. 4.10(b)). The function returns NO and so $Q(0)$ makes transition 29. In so doing, it restores its problem set, returns to state 2, and returns NO to $MP(0)$. This causes $MP(0)$ to make transition 21, discard any traces of $c2$ in SL , and return NO to PE (Fig. 4.10(c)). This leaves both $MP(0)$ and $Q(0)$ as they were before $c2$ was sent to the solver.

If PE backtracks into $c1$, it sends another AVR to $MP(0)$. $MP(0)$ was left in state 2 and since it has no search paths left to explore, it makes transition 20. It discards the remaining changes to SL and VV , sends a BACKTRACK message to $Q(0)$, and returns NO to PE (Fig. 4.10(d)). $Q(0)$ receives the message while waiting in state 2 and so it pops its stack and returns to state 0. Thus, the status of every procedure is restored to what it was before PE sent the first CM to the solver.

Figure 4.9: Solver states (continued).

Figure 4.10: Solver states (continued).

4.4 Procedure Restrictions

To complete the description of the solver architecture, some restrictions need to be placed on constraints which are sent to the solver, the various procedure components, and procedure configurations. These restrictions are also required in the proofs of Chapter 6.

Restrictions on Constraints

Restriction C1

Each constraint will contain at most a finite number of uninstantiated variables.

This is used with Restriction C2 to ensure that MP never has more than a finite number of uninstantiated variables in its problem set. For all practical purposes, Restriction C1 does not limit the types of constraints that users would ever include in a program.

Restriction C2

The solver will be sent at most a finite number of constraints.

If the solver can be sent at most a finite number of constraints and each constraint can contain at most a finite number of variables, MP can never have an infinite number of uninstantiated variables in its problem set. As in the case of Restriction C1, for all practical purposes, Restriction C2 does not limit the number of constraints that would ever appear in a (properly working) program.

Restriction C3

If the solver returns a variable value to PE and PE encounters the variable again in another constraint, the value will be substituted before the constraint is sent to the solver.

This is a minor point which helps to limit the amount of work that MP must do. It is realistic in that in a typical language implementation, all variable instances would be bound to a common table entry.

*Restrictions on Recognition Rules and Sufficiency Predicates***Restriction RS1**

If any function of a procedure P_i or its descendents can (1) verify, (2) partially or fully satisfy, or (3) reject a constraint c , the recognition rules of P_i will succeed when they test c , regardless of whether or not necessary additional data is immediately available.

This restriction is required to ensure that each procedure will accept those constraints that its functions, and those of its descendents, are capable of processing; otherwise, their abilities will be underutilized. Note that the restriction does not say that the recognition rules will be able to identify every possible syntactic variation of the constraint. The rules need only match the abilities of the functions and so Restriction RS1 is not seen as a major limitation.

Restriction RS2

Given any amount of additional data, if no function of a procedure P_i or its descendents can (1) verify, (2) partially or fully satisfy, or (3) reject a constraint c , the recognition rules of P_i will fail when they test c .

This restriction is required to ensure that procedures will not pass constraints to their functions that the functions are not capable of processing; otherwise, the efficiency of the satisfaction process will suffer as functions will be passed useless data, or worse, it will cause system errors. Such a restriction does not limit the solver's potential abilities.

Restriction RS3

If any function of a procedure P_i can (1) verify, (2) partially or fully satisfy, or (3) reject the problem set of P_i , the sufficiency predicate of P_i will succeed when tested.

This is required to ensure that functions are called when enough data is available. Otherwise, solutions may not be generated as soon as possible, and this could cause (1) PE to abandon a search path prematurely, and (2) valid solutions to be lost.

Sufficiency predicates must analyze the data contained in a procedure's problem set. When an equality constraint contains a single variable, no more data is required to solve it and so the sufficiency predicate can succeed immediately. When multiple variables are involved, the problem set will typically need to contain the same number of equations. Thus, sufficiency predicates need to be able to determine the number of equations and unique variables in a procedure's problem set. Writing such a predicate would be straightforward as long as metalevel predicates for analyzing variables are available.

Other problem classes may have different requirements. For instance, many optimization problems require the presence of an objective function and one or more equality and inequality constraints. Since variables appearing in a logic program can be assigned a value only once (unless backtracking has occurred), an objective function could not be optimized more than one time. Thus, programs would need to be written so that the objective function was the last constraint added to a procedure's problem set. A sufficiency predicate could be used to search for an objective function, its presence triggering a call to a satisfaction function containing an optimization algorithm. An objective function would be straightforward to identify if it was written so as to be syntactically recognizable, as in $X = \text{maximum}(Y1 + 5 \times Y2)$.

Restriction RS4

If no function of a procedure P_i can (1) verify, (2) partially or fully satisfy, or (3) reject the problem set of P_i , its sufficiency predicate will fail when tested.

This is required to ensure that the functions are not called when insufficient data is available; otherwise, the efficiency of the satisfaction process will suffer.

Restriction RS5

Recognition rules and sufficiency predicates must terminate when tested.

This is required to ensure that the satisfaction process will terminate. Besides the usual problems of ensuring termination, a potential problem lies in any constraints that the rules and predicates might contain. However, such "metalevel" constraints

would most likely be very simple, involving little more than comparing the number of constraints and variables in a procedure's problem set.

Restriction RS6

If a constraint c is recognized by a set of recognition rules, then these same rules must also be able to recognize simplified versions of c .

This restriction is necessary to ensure that (1) procedures will always have the most up to date versions of constraints in their problem sets and (2) satisfaction functions are only passed constraints which contain uninstantiated variables; otherwise, procedures could generate values for variables that have already been bound.

Consider the following scenario. A procedure $P_i(k)$ could add a constraint c to its problem set, c containing variables X , Y , and Z . If some other procedure were to determine a substitution for X which simplified c to c' , $P_i(k+1)$ could be sent a CM containing c' as well as some other newly reduced constraints. If $P_i(k+1)$ did not replace every constraint containing X with the newest version, it might call one of its functions, generate a value for X , and send the solution to $MP(k+2)$ as an assignment constraint. Unless the value that was generated by $P_i(k+1)$ was identical to the first solution, $MP(k+2)$ would have to reject it and $P_i(k+1)$ would go into a loop testing all of its satisfaction functions.

The above scenario indicates that without Restriction RS6, procedures could make needless satisfaction function calls because their problem sets did not contain the most up to date versions of constraints. Unfortunately, this restriction could make a procedure's recognition rules much more difficult to implement.

Another approach is possible which does not require Restriction RS6; it is as follows:

If a procedure was sent a reduced version of a constraint, it would be tested against the procedure's recognition rules. If it passed, it would replace the old version; if it failed, it would not be added and the old version would not be used in the new problem set.

This technique has the advantage that it does not complicate the recognition rules,

but it complicates the common search mechanism of the procedures. Thus, the proofs of Chapter 6 rely on Restriction RS6.

Restrictions on Satisfaction Functions

It is assumed in the following restrictions that a function can determine that (1) a set of constraints is feasible only if they are feasible, or (2) a set is nonfeasible only if they are nonfeasible.

Restriction SF1

Each procedure can only make a finite number of function calls.

This restriction is required to ensure that the satisfaction process terminates. Since (1) limiting the number of function calls limits the number of solutions that may be returned, and (2) some functions have an infinite number of solutions, the solver will not be able to return every possible solution. On the other hand, it is unlikely that a user would wish to try every possible solution since it would take an infinite length of time.

It is important to note that this restriction is a result of the mathematics involved and not because of the solver architecture. Procedures could make an infinite number of function calls by turning the function call list into a loop; however, this could result in a procedure going into an infinite loop, generating every possible solution to a set of constraints only to have each one rejected by some other procedure in the solver. Restriction SF1 prevents such scenarios, albeit at the risk of lost solutions.

Restriction SF2

Satisfaction functions must terminate when called.

This is required to ensure that the satisfaction process will terminate. The restriction requires functions to be able to detect when they are caught in infinite loops, a common problem when iterative satisfaction methods are being used.

Restriction SF3

A function F will return NO when passed a set S of constraints only

if (1) S contains one or more nonfeasible constraints, or (2) it can be guaranteed that all possible solutions to S have already been returned, including YES when unique values cannot be determined.

This restriction is required to ensure that the solver will generate every solution that it is capable of producing. While the number of solutions to a procedure's problem set must be determined before a function can return NO, any other constraints which were not recognized by the procedure do not need to be considered. Typically, only procedures with quite specialized satisfaction methods can have functions which can return NO.

Restriction SF3 must be considered when dealing with functions that may not return specific values to a set of constraints. For instance, if a function was passed a set of inequality constraints that had multiple solutions, it might return a simple YES response. This would cause the host procedure to continue the search for a list of substitutions and so it would make its next function call. If the next function returned NO, the procedure would make transition 29 and immediately return NO without indicating that the constraints were feasible. Thus, functions that can return YES must not be followed by a function that can return NO. Appendix B describes a method which permits procedures to return a YES-NO reply sequence.

Restriction SF4

If a satisfaction function is passed a set of constraints that contains no unbound variables, it will return NO, MAYBE, or YES; i.e., it will not return YES(S) or MAYBE(S).

This is required to ensure that (1) the solver does not try to assign more than one value to a variable and (2) the satisfaction process will terminate. It does not limit the types of constraints that could be solved, although it must be relaxed if the solver is to process Type 2 variables (as discussed in the next section).

Restriction SF5

A function F will return YES if and only if (1) it is passed every available

constraint, (2) it can determine that the constraints are feasible, and (3) it cannot determine any substitutions.

This restriction does not limit the abilities of the solver; it only ensures that its results are correct and as much use as possible is made of the abilities of each function. It prevents the solver from returning YES or YES(S) when the constraints may not be feasible with the substitutions applied, and it ensures that the solver will return an indication that the constraints encountered to date are feasible when a function can determine such.

Whenever a procedure makes a function call, the function is passed a flag indicating whether or not it is being passed every available constraint. Thus, the flag determines in part whether or not YES or YES(S) can be returned.

Restriction SF6

A function F will return YES(S) if and only if (1) it is passed every available constraint, (2) it can determine a set of substitutions S for one or more of the variables in the constraints, and (3) it can determine that the constraints are feasible with the substitutions in S applied.

This restriction does not limit the abilities of the solver; it only ensures that its results are correct and as much use as possible is made of the abilities of each function. It prevents the solver from returning YES(S) when the constraints may not be feasible with the substitutions applied, and it ensures that the solver will return an indication that any definite solutions are returned as such.

Restriction SF7

If a function could be passed a problem set containing multiple constraints, each constraint has multiple solutions, and the constraints can be solved independently of each other, then the function should satisfy one constraint at a time.

This restriction is actually a guideline that should help in the design of satisfaction functions. A problem can arise when a procedure is passed a set of constraints,

each with multiple solutions, and the constraints are normally solved one at a time. For instance, consider the case of a procedure for solving quadratics when it is passed several all at once. If the procedure solved them all together, it would also have to be able to generate all of the possible solution combinations. It is believed that trying to write functions to behave in such a fashion would be very difficult and so a simpler approach is desirable.

It would be much easier to solve only one constraint and then return the solution. This would force the solver to recurse and eliminate one of the constraints. On the next recursion, the procedure would update its problem set. It could then solve the next constraint, return the solution, let the solver recurse, and so on, until all of the constraints were satisfied. If an alternate solution was requested, an AVR would be passed to the solver's deepest level of recursion and the procedure would generate a second solution to just the last constraint.

A second AVR would cause the solver to reject the last constraint at the deepest level of recursion and then an alternate solution to the second last constraint would be sought. Once a second solution for it became available, the solver would recurse and resolve the last constraint starting with its first solution. Thus, by solving only one constraint at a time, the satisfaction process can proceed in an orderly manner.

Restrictions on Procedure Organization

Restriction PO1

Each procedure will have at most a finite number of descendents.

This restriction is used to ensure that the satisfaction process will terminate, among other properties. It prevents the solver from containing more than a finite number of procedures. For all practical purposes, Restriction PO1 does not limit the number of procedures that users would likely include in a program.

Restriction PO2

No procedure shall have a descendent that is also one of its ancestors.

This restriction is used to ensure that the satisfaction process will terminate. Without it, the solver could become caught in a infinite loop of procedure calls.

Restriction PO2 does not limit the organization of procedures that users are likely to require because procedures should become more specialized with depth into the tree. Consequently, if a procedure had a descendent that was also an ancestor, the hierarchical organization would be violated.

Note that this restriction does not prevent multiple procedures in any one configuration from having identical satisfaction capabilities, nor does it prevent a procedure from having more than one parent. Such possibilities are explored in Chapter 5.

4.5 Incorporating Type 2 Variables

So far, the solver description has been based on the processing of constraints with Type 1 variables only. While these are essential for handling many of the common data types, Type 2 variables are better suited for representing interval- or domain-valued variables as used in BNR Prolog and CHIP. Nothing has been built in to the generic procedure skeleton that prevents the use of Type 2 variables and so only the satisfaction functions and the abilities of MP need to be modified. Since the necessary changes are a function of the specific data type being added and the use of a very general approach for Type 2 variables would likely be inefficient, the ability to modify MP will be demonstrated by way of an example. In particular, the remainder of this section will describe how the solver can be modified to handle interval-valued variables.

The following changes need to be made:

- Restriction SF4 needs to be relaxed so that when satisfaction functions are passed constraints in which all variables are bound, they can return YES(*S*) or MAYBE(*S*).
- MP must be able to assign multiple values to all Type 2 variables and maintain a history of previous assignments for each.
- Whenever new interval values become available, MP must be able to replace any previous assignments in constraints with the new values.

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

The first change allows satisfaction functions to generate new interval limits for a variable that is already bound to a range of values. Restriction SF4 needs to be replaced with the following:

Restriction SF4'

If a satisfaction function returns $YES(S)$ or $MAYBE(S)$, S may not contain substitutions for Type 1 variables that are already bound, and it may not contain any new interval limits for any Type 2 variable X such that (1) the new lower limit is less than the previous lower limit, (2) the new upper limit is greater than the previous upper limit, and (3) the new lower limit is greater than the new upper limit.

As with Restriction SF4, Restriction SF4' is required to ensure that the satisfaction process does not become nonterminating. If Type 1 variables can be assigned a value only once and the range of each Type 2 variable can only be narrowed as a result of a recursive call, the solver cannot recurse indefinitely.

The second change is required so that new intervals may be assigned without losing data that may be necessary for backtracking. A stack becomes necessary for each Type 2 variable in VV . If MP determined new interval limits for a variable, it would push them onto the appropriate stack, allowing the old values to be restored if the new ones led to a contradiction.

The third change is required so that new limit data can be distributed to the constraints. Unlike Type 1 variables which can be eliminated from constraints once they are bound to specific values, Type 2 variables must remain so that MP can assign new values as they become available.

While the above changes are essential to implementing the interval data type, other changes are necessary to make the constraint processing more efficient. These are:

- MP should be able to perform basic arithmetic operations on Type 2 variables and combinations of Type 1 and Type 2 variables.

- MP should be able to detect (1) constraints which are satisfied as a result of intervals shrinking to point intervals, and (2) nonfeasible constraints arising from Type 2 variables containing contradictory interval limits.

If basic arithmetic operations were not performed by MP, some other procedure in the solver would have to perform them. This would involve some searching even when simple calculations were being made. Since any one constraint could include both Type 1 and Type 2 variables and the result of combining an interval with other numeric data results in another interval, MP would need to be able to perform basic operations on all of the numeric data types and combinations of them. Local propagation techniques could also be used to narrow intervals before any constraints were sent to the procedures.

MP should also have the ability to detect satisfied and nonfeasible constraints. This would allow the solver to return substitutions or abandon search paths at the earliest opportunity. Otherwise, a procedure would have to be found to determine the same results.

Chapter 5

Configurations and Satisfaction Failure

5.1 Organizing Procedures

In Chapter 4, it was assumed that the procedures in the solver are organized into an n-way tree and that new procedures may be added into the hierarchy as necessary. The problems of where to add new procedures must now be addressed. It will be seen that:

1. Creating an acceptable configuration is not difficult since many are possible.
2. The amount of searching that the solver must do is affected by the procedure organization in much the same way that Prolog's search mechanism is affected by the top-to-bottom, left-to-right ordering of clauses and subgoals.
3. Whoever is in charge of creating an acceptable configuration must be aware of the relative capabilities of the available procedures; that is, whether or not multiple procedures can satisfy the same set of constraints. Note, however, that it is not assumed that the people organizing the procedures will have been involved in their implementation.

The best solver configurations may be difficult, if not impossible, to determine because critical data is not likely to be available when the procedures are being or-

ganized; that is, data describing the order in which constraints will appear and the expected variable bindings. This is not considered to be a major problem because:

1. One of the main advantages of the solver architecture is its ability to efficiently find a suitable procedure.
2. It will be seen in Chapter 7 that the use of metalevel data may be used to effectively change the solver configuration using data that is available only at run-time.
3. § 5.2 presents techniques for determining when the capabilities of the solver have been exceeded, permitting corrective action to be taken.

Consequently, creating an optimal procedure organization is not critical to the constraint satisfaction process.

5.1.1 Procedure Placement Requirements

Since a procedure can only solve constraints if it has access to them and it can only access those that are sent in CM's, the first requirement for an acceptable procedure configuration becomes:

Requirement 1:

Procedures must be placed so that they will be sent any constraints that their satisfaction functions can process.

This requirement could be satisfied by making every procedure a child of MP since MP will send every constraint that it cannot solve to its children. However, such a configuration would be very inefficient because many procedures would perform the same syntactic analysis of constraints. A hierarchical configuration is much better because it can satisfy both Requirement 1 and the design objective, therefore another requirement becomes:

Requirement 2:

A procedure must be made a child of another procedure whenever possible.

The conditions that determine when a procedure can be made a child of another are a function of which constraints the procedure can recognize (and presumably solve). For instance, a procedure that solves quadratics cannot be made a child of a procedure that does not recognize them, but it could be made a child of a procedure that solves polynomials of arbitrary order.

Before the algorithms for organizing procedures can be presented, some nomenclature for comparing procedure capabilities is required. The capabilities of procedure P_i relative to another P_j are depicted in Figure 5.1 and are defined as follows:

1. Disjoint: The sets of constraints that P_i and P_j can solve do not intersect.
2. Superset: The set of constraints that P_i can solve includes every constraint that P_j can solve. It is assumed that a set of constraints which is solvable by both can be solved most efficiently by P_j ; otherwise there is no need for P_j in the solver.
3. Subset: The set of constraints that P_i can solve is a subset of the constraints that P_j can solve. It is assumed that a set of constraints which is solvable by both can be solved most efficiently by P_i ; otherwise there is no need for P_i in the solver.
4. Partially Overlapping: Some constraints are solvable by both P_i and P_j while others are solvable by just one or the other.

5.1.2 Adding Procedures with No Overlap

When a new procedure P_i is to be added to a set S of procedures and the capabilities of no two procedures partially overlap, determining a position for P_i is a straightforward, two step process. First, a suitable parent is found and then its children are identified. To find the parent, the following algorithm may be used. It is assumed that all existing procedures were added using the same algorithm:

Figure 5.1: Capabilities of P_i relative to P_j

Step 1. Initialize procedure Px^+ to MP.

Step 2. If the capabilities of Pi are a subset of those of some child Py of Px^+ , set Px^+ to Py and repeat step 2, otherwise continue.

Step 3. Make Pi a child of Px^+ .

Note that in Step 2, the capabilities of Pi can be a subset of those of only one child of Px^+ because the partial overlap case is assumed not to arise.

Once Pi has been made the child of some procedure Px^+ , its children may be identified using the following algorithm:

Step 4. For every child Py of Px^+ other than Pi , if the capabilities of Py are a subset of those of Pi , make Py a child of Pi , otherwise leave it as a child of Px^+ .

Consider the tree depicted in Figure 5.2. If Pi can solve polynomials in one variable, its capabilities are a subset of those of Pa , therefore Px^+ is set to Pa in Step 2. Since Pi 's capabilities are not a subset of Pb 's, Pi is made a child of Pa in Step 3. Finally, Pb is made a child of Pi using Step 4.

5.1.3 Adding Procedures with Overlapping Capabilities

When adding a procedure whose capabilities partially overlap those of existing procedures, it may be placed so that (1) the final configuration is a strict n-way tree, or (2) it has multiple parents. The second option has one advantage over the first: it allows a procedure to be placed deeper in the hierarchy so that it will be sent a minimum number of constraints that it cannot recognize. While this feature would appear to improve the solver's efficiency, the use of multiple parents allows a procedure to be sent the same data several times (once from each parent), therefore some of the efficiency advantages are lost. As well, the use of multiple parents disrupts the property that procedures will be passed constraints in top-to-bottom, left-to-right order. Consequently, a procedure could be sent a set of constraints after it has already solved them for another parent.

a) Relative procedure capabilities

b) Adding Procedure Pi

Figure 5.2: Adding procedures with no partially overlapping capabilities

It is not known which type of configuration would be the best on average; examples for either case can be created. The best configurations are program dependent and so algorithms for adding procedures in either way are presented below.

The Strict N-Way Tree Case

When a new procedure's capabilities partially overlap those of existing procedures, all of the options for placing it may not be obvious. For instance, consider the case depicted in Figure 5.3 in which procedure P_i can solve some constraints that no other procedure can, and some others more efficiently than an existing procedure P_a . The options include (1) making P_i a child of P_a , (2) making it a child of MP, and (3) using two versions of the same procedure, one with P_a as its parent and the other with MP.

By Requirement 1, P_i could be made a child of P_a only if P_a 's recognition rules were modified so that they would recognize any constraint that was processable by P_i . However, this would cause constraints beyond P_a 's capabilities to appear in its problem set and so its sufficiency predicate and satisfaction functions would have to be modified as well. In general, modifying existing procedures to accomodate others is a poor approach since it complicates them, causes unnecessary processing, and increases the chances of errors in the solver.

If two versions of P_i were used, one with P_a as its parent and the other with MP, denoted P_{iP_a} and P_{iMP} respectively, several problems could arise. Unless the recognition rules of P_a were modified, P_{iP_a} would be sent only those constraints that P_a could recognize. This would cause the problems just mentioned to occur again and so P_{iP_a} would not solve as many constraints as it should. Its efforts would also be duplicated by P_{iMP} whenever the two were sent the same constraints. Another minor consideration is that P_{iP_a} would have to be in a subtree to the left of P_{iMP} , otherwise P_{iMP} would always process the constraints before P_{iP_a} had a chance.

In general, another problem arises if two versions of P_i are made children of some procedures P_j and P_k when P_j and P_k cannot recognize every constraint that P_i can process. It is possible that at some point, the two versions could have different problem sets, neither of which was solvable even though a single instance of P_i could

a) Relative Procedure Capabilities

b) Adding P_i

Figure 5.3: Adding a procedure with partially overlapping capabilities

solve all of the data combined. Thus, a single instance of the procedure is preferred as long as it is placed so that it has access to all of the data that it can recognize. This indicates that P_i should be made a child of MP.

This conclusion results in the same algorithms that appeared in the § 5.1.2. As before, a suitable parent for P_i is found first and then its children are identified.

Step 1. Initialize procedure P_{x^+} to MP.

Step 2. If the capabilities of P_i are a subset of those of some child P_y of P_{x^+} , set P_{x^+} to P_y and repeat step 2, otherwise continue.

Step 3. Make P_i a child of P_{x^+} and exit.

Note that in Step 2, the capabilities of P_i could be a subset of those of several children of P_{x^+} ; an example appears in Figure 5.4. If the final configuration is to be an n-way tree, then one child of P_{x^+} must be chosen as the new P_{x^+} . As long as the capabilities of P_i remain a subset of the new P_{x^+} , any child may be chosen. However, to make the solver as efficient as possible, the following guideline should be used:

Guideline 1:

If S denotes the set of all possible parents of P_i in Step 2, P_i should be made a child of the procedure $P_j \in S$ which will send it the least number of constraints.

This will help to minimize the number of constraints that are needlessly sent to P_i , that is, those constraints which it cannot recognize. To determine which procedures would send it the least amount of data, it should be remembered that the more constraints that a procedure can recognize, the more that it will send to its children. As well, one must consider the constraints that are likely to be sent to the solver, and these are a function of user's programs and run-time bindings.

Once P_i has been made the child of some procedure P_{x^+} , its children may be identified using the following:

a) Relative Procedure Capabilities

b) Adding Pi

Figure 5.4: Multiple possible parents

Step 4. For every child P_y of P_{x^+} other than P_i , if the capabilities of P_y are a subset of those of P_i , make P_y a child of P_i , otherwise leave it as a child of P_{x^+} .

As in § 5.1.2, this process makes existing children of P_{x^+} children of P_i whenever possible.

The Multiple-Parent Case

The restrictions on the tree configuration will now be relaxed so that a procedure may have multiple parents. The resulting hierarchies will be deeper than those restricted to strict n-way trees, but procedures may be sent the same constraints more than once. To add a procedure P_i , a suitable parent is found first and then its children are identified. The parent may be identified using the following algorithm:

Step 1. Initialize procedure P_{x^+} to MP.

Step 2. If the capabilities of P_i are a subset of those of some child P_y of P_{x^+} , set P_{x^+} to P_y and repeat step 2, otherwise continue.

Step 3. If the capabilities of P_i overlap those of the union of all children of P_{x^+} or they are disjoint, make P_i a child of P_{x^+} and exit.

Step 4. Identify a set Q of children of P_{x^+} such that the union of their capabilities is a superset of the capabilities of P_i .

Step 5. For each procedure $P_k \in Q$, identify the smallest possible set Q_k of its children such that the union of their capabilities is a superset of the capabilities of P_i intersected with those of P_k . If no such set exists for any $P_k \in Q$, go to Step 6; otherwise, replace P_k with the members of Q_k and repeat Step 5.

Step 6. Make P_i a child of each member of Q and exit.

This algorithm will make P_i a child of every procedure in a set Q such that the combined capabilities of the procedures in Q are a superset of those of P_i . Step 2 is used to force P_i as deep into the hierarchy as possible with only one procedure

as its parent. As in the previous algorithm, the capabilities of P_i could be a subset of those of several children of P_{x^+} and so one child needs to be chosen as the new P_{x^+} . As before, Guideline 1 may be used to help make the selection.

Once Step 2 can no longer be used, Step 3 is used to see if P_i can solve any constraints that none of the children of the current P_{x^+} can solve. If so, P_i must be made a child of P_{x^+} ; otherwise, P_i can still be placed deeper in the hierarchy. Step 4 finds a set Q of children which can act collectively as a parent to P_i but since there may be more than one set to choose from, the following guideline may be used:

Guideline 2:

Let Q^* denote the set of sets of all possible parents of P_i in Step 4.

Choose $Q_k \in Q^*$ as Q if Q_k will result in P_i being sent the least number of constraints.

As with Guideline 1, this helps to minimize the number of constraints that are needlessly sent to P_i , that is, those that it cannot recognize.

Each procedure $P_k \in Q$ may have a set of children such that their capabilities combined with those of the other procedures in Q are still a superset of the capabilities of P_i . If so, Step 5 replaces P_k with an appropriate set of children and then it repeats the process as many times as possible. This identifies a set of procedures as deep in the hierarchy as possible that can act as a parent to P_i . Finally, in Step 6, P_i is made a child of the remaining procedures.

Once a set Q of parents for P_i has been found, its children may be identified using the following:

Step 7. For every $P_{x^+} \in Q$, for every child P_y of P_{x^+} other than P_i , if the capabilities of P_y are a subset of those of P_i , make P_y a child of P_i , otherwise leave it as a child of P_{x^+} .

An example of incorporating a procedure using the above method appears in Figure 5.5.

a) Relative Procedure Capabilities

b) Adding P_i

Figure 5.5: Adding P_i when multiple parents are allowed

5.1.4 Improving Acceptable Configurations

The previous sections presented algorithms for determining which procedures should be made children of other procedures. However, no discussion was made of how best to order sibling procedures from left to right.

Experience has shown in Borning's Thinglab and $\text{CLP}(\mathcal{R})$, among other constraint solving programs and languages, that it is best if simple constraints are solved first. The idea is that the solutions to the simple ones can make more difficult expressions easier to solve. Thus, another guideline results:

Guideline 3:

Procedures for solving simple constraints should be to the left of any others.

Unfortunately, it may be difficult to determine which constraints are "simpler" than others. In general, the degree of difficulty in satisfying constraints is a function of the number of variables they contain and the number of steps in the satisfaction process. For instance, it is easier to solve a linear constraint in one unknown than to solve a set with one hundred. However, it is difficult to say if solving a high degree polynomial in one unknown is harder to solve than a large set of linear inequalities.

Another means of reducing the amount of searching for a procedure stems from how often a procedure is used in a program. For instance, if a program contained many polynomials and the solver contained one procedure Pa for solving polynomials and another Pb for solving linear equations, it would be more efficient to have Pa to the left of Pb because it would minimize the number of polynomials needlessly sent to Pb . This leads to another guideline:

Guideline 4:

Procedures that are frequently used should be to the left of any others.

At times, this guideline can conflict with Guideline 3. For example, if a program contained many nonlinear constraints and a few trivial equations, Guideline 4 contradicts Guideline 3 in suggesting that a procedure for solving polynomials should be to the left of a procedure for solving simple equations.

Another guideline may be found in the desire to minimize the number of tentatively qualified answers that the solver can produce. For instance, consider the following reply sequence that the solver could produce in response to a CM and four AVR's:

$$\text{MAYBE}(S1) \rightarrow \text{MAYBE}(S2) \rightarrow \text{YES}(S3) \rightarrow \text{YES}(S4) \rightarrow \text{NO}$$

Note that only the last two solutions are qualified as being definite. If some procedure *Pa* had generated the first two replies and another *Pb* had generated the last three, rearranging the procedures so that *Pb* was tested first could have simplified the series to:

$$\text{YES}(S3) \rightarrow \text{YES}(S4) \rightarrow \text{NO}$$

Thus, besides affecting the efficiency of finding a satisfaction procedure, the left-to-right procedure order can affect the "quality" of answers that the solver can produce. Consequently, another guideline results:

Guideline 5:

Procedures that produce definite solutions to constraints should be to the left of those that produce tentative results.

As before, this guideline may conflict with the others. For instance, if a frequently used procedure returns tentative results, Guideline 5 indicates that it should be to the left of any others, while Guideline 4 suggests that it should be to the right.

The basis for each guideline is easily understood but collectively, the guidelines may not be easy to satisfy in practice. They can be contradictory regardless of whether or not procedures have overlapping capabilities, and so the best configurations cannot be determined without a priori knowledge of the contents of users' programs. This includes what constraints will be used, in what order they will be encountered, and what the variable bindings will be. If such data is not available, the solver will have to perform more searching than it would otherwise. Fortunately, metalevel data can make use of run-time data and make up for any loss in efficiency.

5.2 Identifying Satisfaction Failure

The need to identify instances of satisfaction failure was discussed in § 3.2. It is required to ensure that when the limits of the solver are exceeded, corrective action may be taken. This could involve adding a new procedure or modifying an existing one. Satisfaction failure does not refer to instances of constraint failure; that is, occasions when the solver determines that a set of constraints is nonfeasible.

The intent of this section is to show how the solver architecture can be exploited to detect problem situations. It is assumed that whenever they arise, the relevant data can be saved in a file for further analysis. The problems of explaining the data using some form of natural language processing are not addressed since to do so would introduce problems well beyond the subject of this thesis.

The following sections discuss methods for identifying instances of satisfaction failure on two levels. Failure can be recognized on a global scale when the solver abandons a set of constraints because (1) no procedure can determine if a subset of the constraints has a solution, or (2) no procedure can generate any more alternate solutions or determine that there are no more to return. Satisfaction failure can also be identified on a local level when (1) procedures cannot determine if all solutions to their problem sets have been returned, or (2) their satisfaction functions experience numeric instabilities.

Reports of local level failure might appear to be more useful than global ones because they identify problems in more detail. However, global failure does not necessarily result from instances of local level failure because the solver's limitations may be exceeded even though its procedures all behave satisfactorily. Consequently, the problems of global and local failure are treated separately.

Unfortunately, all of the following methods may have false-positive outcomes; that is, they may indicate a problem situation when none exists. This limitation is considered to be acceptable for the following reason: In order to identify those cases in which solutions have been lost and never any others, the solver would have to be able to determine when an arbitrary set of constraints had a solution. Since no method exists for doing so, the best approach is to be conservative and report all

cases that might reveal solver limitations.

5.2.1 Identifying Global Satisfaction Failure

Instances of global failure occur when $MP(k)$ rejects a list of constraints before all possible solutions have been returned, and $MP(k)$ can reject a list only when it returns NO or ABANDON. By Property B6.1 (in Chapter 6), if $MP(k)$ returns NO in response to a CM, then its problem set is nonfeasible, and by Property B6.2, if $MP(k)$ returns NO in response to an AVR, then all possible solutions to the constraints in its problem set have already been returned. Thus, global failure can occur only when $MP(k)$ returns ABANDON.

$MP(k)$ can return ABANDON only by making transition 20 after receiving an AVR while in state 2. It could reach state 2 only by making transition 5, 6, 27, or 28 after all of its children returned MAYBE or YES, that is, after the abilities of every procedure were exhausted. Since the possibility remains that there may be solutions other than any returned already, $MP(k)$'s problem set should be saved for analysis.

If $MP(k)$'s problem set is saved whenever it makes transition 20, some data may be saved unnecessarily. For instance, consider the following code segment when the solver can process simultaneous linear equations:

$$\dots, A+B = 1, A-B = 1, \dots$$

Assuming that variables A and B are uninstantiated, $MP(0)$ would end up in state 2 after it was sent the first equation. When the second arrives, the solver could then return the only solution to the pair. If enough AVR's were issued, $MP(0)$ would eventually reject the first constraint using transition 20. This would cause the first equation to be saved as a potential source of lost solutions even though the only solution to the pair was returned. It is as if the solver was saying "It has not been determined that all solutions to $A+B = 1$ have been returned and so the constraint should be saved." Treating each constraint individually, this is true since the equation does have an infinite number of solutions.

Such false-positive reports of potential problems can also be caused by procedures which fail to return NO when it is possible to determine that all possible solutions to a set of constraints have been returned. Consider what could happen if an AVR was issued after the solver returned the only solution to the above pair of equations. An AVR would be sent to the procedure which generated the solution but if it did not return NO, the solver would continue searching. Thus, MP(0) could eventually return MAYBE or YES using transition 5. Another AVR would cause it to backtrack over the second constraint using transition 20 and so the pair of equations would be saved. Thus, false-positive reports may occur as a result of procedures failing to return NO when they should.

Another cause of false-positive reports is the fact that the constraints are sent to the solver one at a time, rather than as a set. This is a consequence of the manner in which typical Prolog inference engines process the subgoals in clauses. However, it will be seen in Chapter 7 that metalevel data can be used to collect constraints so that they are sent to the solver as a set. Thus, the number of false-positive reports of global satisfaction failure can be minimized.

5.2.2 Identifying Local Satisfaction Failure

The technique described for recognizing global satisfaction failure does not help to identify the limitations of specific procedures or their satisfaction functions. For instance, it cannot determine which procedures could not generate alternate solutions to their problem sets or which satisfaction functions did not execute as intended. Fortunately, the solver architecture can also support a means of identifying instances of local satisfaction failure.

Once a procedure has started to use its satisfaction functions, as long as it does not make transition 19, 20, or 29, there is a chance that it will be able to generate all of the possible solutions. A procedure can make transition 29 only if one of its functions returns NO but according to Restriction SF3, a function can return NO only if all solutions to the procedure's problem set have been returned already, or the set is nonfeasible. Thus, local failure cannot occur when a procedure makes

transition 29.

If a procedure makes transition 19 or 20, it cannot make any more function calls (with the same problem set), and since its last function call could not have returned NO, it could not have determined that all possible solutions were returned. This indicates that when a procedure makes transitions 19 and 20, an instance of local satisfaction failure could have occurred. Thus, the name of the procedure and its problem set should be saved for future analysis. As with the technique for global failure, false-positive outcomes are possible.

Even more detailed reporting of local satisfaction failure is possible if a function can determine that a solution was lost because its limits were exceeded. For instance, it might not be able to solve a set of constraints even though a solution was known to exist. In order to provide users with as much information as possible, the function would need to have a means of indicating that potential solutions were lost. Since returning MAYBE is not very specific, one solution is to provide each function call with an error flag. Whenever an error condition was indicated, the solver would save the relevant data and report the incident. Such reports of local satisfaction failure would be very useful because they could not be false-positive.

Chapter 6

Solver Properties

The purpose of this chapter is to review the properties of the solver. While it may be obvious why some of them are necessary (such as the termination property), it may not be clear why others are desirable and/or necessary. As well, it may not be apparent how the property could be undermined by procedure components which failed to follow the restrictions of § 4.4. Thus, after each property is stated, it is explained why the property is significant and how it may depend on the restrictions. Note that although a restriction may not appear explicitly in a proof, it may still play a key role because it appears in a supporting lemma. Most of the property proofs may be found in Appendix D; those related to backtracking appear in Appendix E as noted.

An Overview of the Properties

The properties are divided into the following categories:

- A: general properties,
- B: properties which prove that various results will be returned only if certain conditions are met, and
- C: properties which prove that various results will or will not be returned if certain conditions are met.

The general properties are quite straightforward. They show that the satisfaction process will terminate and the solver will backtrack over data in an appropriate

fashion. As well, they ensure that the solver will search for and return solutions as described in Chapter 4. The properties of categories B and C are used to show that the solver will produce various results if and only if it is appropriate to do so. The “conditions” referred to are a function of what constraints have been sent to the solver, the abilities of the available procedures, and what results have already been returned.

Because of the recursive nature of the solver, most properties depend on the result of a procedure at the k^{th} level of recursion making a recursive call to $\text{MP}(k+1)$. Thus, to ensure that the properties are as general as possible, CM’s are normally sent at an arbitrary level of recursion, that is, to $\text{MP}(k)$ or $Pi(k)$. However, for those properties which are meaningful only at the base level of recursion (such as when PE sends a CM to the $\text{MP}(0)$), the 0^{th} level may be referred to explicitly or the substitution $k=0$ can be made.

For the sake of brevity, the following notation is used:

- L_{M_i} denotes the list of constraints that is sent in a CM M_i .
- PS^4 denotes $\text{MP}(k)$ ’s problem set when it reaches state 4.
- A_{M_i} denotes the argument of CM M_i which if set to 1, indicates that L_{M_i} contains every constraint in PS^4 .
- $PS_{M_i}^-$ denotes $\text{MP}(k)$ ’s problem set immediately before it receives a CM M_i .
- $PS_{M_i}^+$ denotes $PS_{M_i}^- \cup L_i$; i.e., $\text{MP}(k)$ ’s problem set immediately after it has been updated with the constraints in M_i .
- $PS_{M_i}^*$ denotes $PS_{M_i}^+$ after $\text{MP}(k)$ has satisfied zero or more MP-solvable constraints in $PS_{M_i}^+$ (using values determined with transitions 24 and 3).
- $PS_{M_i}^4$ denotes $PS_{M_i}^*$ when $\text{MP}(k)$ has reached state 4.

As well, the *status* of a procedure $Pi^+(k)$ collectively refers to (1) its state as indicated in its state transition diagram, and (2) the contents of its data structures.

6.1 General Properties

Property A1.1

Given the following conditions:

1. a procedure $Pi(k)$ is sent a CM,
 2. every function call made by $Pi(k)$ or one of its descendents terminates,
 3. every recognition rule and sufficiency predicate that is tested by $Pi(k)$ or one of its descendents terminates, and
 4. if any recursive calls are made, $MP(k+1)$ responds in a finite length of time,
- then $Pi(k)$ will respond in a finite length of time.

Property A1.2

Given the following conditions:

1. $MP(k)$ is sent a CM.
 2. every function call that is subsequently made terminates. and
 3. every recognition rule and sufficiency predicate that is subsequently tested terminates.
- then $MP(k)$ will respond in a finite length of time.

Comments

These two properties are required to ensure that the satisfaction process will terminate. Although the second is of most interest since PE interacts with $MP(0)$, termination can be guaranteed only if every procedure $Pi^+(k)$ responds to a CM in a finite length of time. Thus, both proofs are necessary and they must hold for an arbitrary level of recursion.

Either property could be undermined if Restriction SF1, SF2, or RS5 were violated because a procedure $Pi(k)$ might not respond if it could make an infinite number of calls to satisfaction functions, or a recognition rule, sufficiency predicate, or satisfaction function did not terminate. The property also relies on:

- Restriction PO1 to ensure that there is not an infinite number of procedures to call,
- Restriction PO2 to prevent a set of procedures from being caught in an infinite loop of inter-procedures calls, and
- Restrictions C1 and C2 to ensure that the number of constrained variables is finite (in which case the solver cannot recurse indefinitely).

Property A2

Given the following conditions:

1. $MP(k)$ is sent a CM M_a and zero or more AVR's,
2. some procedure $Pi(k)$ can recognize at least one constraint in $PS_{M_a}^4$,
3. no procedure $Ph^+(k)$ can determine that $PS_{M_a}^*$ or $PS_{M_a}^4$ contains nonfeasible constraints, and
4. based on the order in which procedures can call their satisfaction functions, no procedure $Pq^+(k)$ which is before $Pi(k)$ can determine that all possible solutions to the constraints in L_{M_a} have been returned already,

then:

1. $Pi(k)$ will be sent a CM before $MP(k)$ returns NO or ABANDON, and
2. its updated problem set will contain every constraint $c \in PS_{M_a}^4$ that $Pi(k)$ can recognize.

Comments

This property ensures that before the solver can backtrack over data contained in a CM, any procedure $Pi(k)$ which could recognize any data in $MP(k)$'s problem set will be sent a CM, and its updated problem set will contain every constraint in $MP(k)$'s problem set that it can recognize. The only exception is $Pi(k)$ may not be sent a CM if some other procedure determines that all solutions to the constraints

have already been returned (in which case there is no point in sending the data to $P_i(k)$).

This property relies on Restriction RS1 to ensure that every ancestor of $P_i(k)$ will recognize the data that $P_i(k)$ could make use of, and Restriction SF3 to ensure that if the solver does backtrack over the data before it is sent to $P_i(k)$ (in which case some satisfaction function must have returned NO), no more solutions are possible.

Property A3.1

Given the following conditions:

1. a procedure $P_i(k)$ is sent a CM containing constraint c ,
2. $P_i(k)$ has a function F_i that can generate the assignment constraint $X = x_i$ for some variable X in c ,
3. $P_i(k)$ has a descendent $P_j(k)$ with a function F_j that can generate the assignment constraint $X = x_j$ given c , and
4. $P_i(k)$ will return both solutions if sent a sufficient number of AVR's, that is, neither will be rejected if sent to $MP(k+1)$,

then $P_i(k)$ will return the solution $X = x_j$ before returning $X = x_i$.

Property A3.2

Given the following conditions:

1. a procedure $P_a(k)$ is sent a CM containing constraint c ,
2. $P_a(k)$ has a descendent $P_i(k)$ with a function F_i that, if given c , can generate the assignment constraint $X = x_i$ for some variable X in c ,
3. $P_a(k)$ has another descendent $P_j(k)$ with a function F_j that, if given c , can generate the assignment constraint $X = x_j$,
4. $P_i(k)$ is in a subtree to the left of the subtree containing $P_j(k)$, and
5. $P_a(k)$ will return both solutions if sent a sufficient number of AVR's, that is, neither will be rejected if sent to $MP(k+1)$,

then $Pa(k)$ will return the solution $X = x_i$ before returning $X = x_j$.

Comments

Properties A3.1 and A3.2 are used to show the order in which procedures will be called and the solver will return solutions. They indicate that the solver does use the depth-first, left-to-right search procedure described in Chapter 4.

Both of the properties rely upon Restrictions RS1 and RS3 to ensure that the necessary recognition rules and sufficiency predicates will succeed in order for the constraint data to reach procedures' functions.

Property A4.1

Given the following conditions:

1. $MP(0)$ receives a CM,
2. it returns $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$ using transition 10, 11, 12, or 13 after S has been forwarded back through $n > 0$ levels of recursion, and
3. $MP(0)$ subsequently receives an AVR,

then:

1. the status of every procedure $Pi^+(k)$, $k = 0, \dots, n-1$, will be restored to what it was before $MP(n)$ replied, with the exception that any procedure $Pa(j)$, $j = 0, \dots, n-1$ that called $MP(j+1)$ will return to state 6 rather than 12,
2. any status changes associated with the n^{th} level of recursion will be discarded.
3. NO or ABANDON will be returned to the procedure that had previously called $MP(n)$, and
4. the procedure that called $MP(n)$ will return to state 9 in order to test its next satisfaction function.

Comments

Property A4.1 is important because it permits properties which describe how the solver responds to CM's to apply to how it responds to AVR's; its proof may be found in Appendix E.

The property indicates that if the solver has returned a set of substitutions back through a series of recursive calls and it is then sent an AVR, then every procedure that was involved with forwarding the solution back to $MP(0)$ will be restored so that the search will be continued from the last choicepoint. The reasons for restoring procedures to state 6 rather than 12 is explained in Property A4.2.

Property A4.2

Given the following conditions:

1. a procedure $Pi(k)$ sends $MP(k+1)$ a CM,
2. $MP(k+1)$ replies so that $Pi(k)$ returns $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$ using transition 31, 32, 33, or 42, and
3. $Pi(k)$ subsequently forwards an AVR to $MP(k+1)$,

then:

1. $Pi(k)$ will treat ABANDON results returned from $MP(k+1)$ as if they were NO results returned while it was waiting in state 12, and
2. $Pi(k)$ will treat all other responses from $MP(k+1)$ as if it received them while waiting in state 12.

Comments

This property is proven in Appendix E. It is used to show that there is nothing very significant about a procedure being restored to state 6 rather than 12 when it receives an AVR. If a procedure $Pi(k)$ sends a CM to $MP(k+1)$, $MP(k+1)$ cannot return ABANDON; it can do so only if it is sent an AVR. Thus, if $Pi(k)$ receives an AVR while in state 11, it moves to state 6 so that it may receive an ABANDON reply as well as the others that it could receive while in state 12.

Property A4.3

If $MP(\theta)$ returns NO in response to a CM M_i , then:

1. the status of every procedure $Pi^+(0)$ in the solver will be returned to what it was before $MP(\theta)$ received the CM, and
2. any status changes resulting from recursive calls which were made since M_i was received will be discarded.

Property A4.4

If $MP(\theta)$ returns NO or ABANDON in response to an AVR, then:

1. the status of every procedure $Pi^+(0)$ in the solver will be restored to what it was before $MP(\theta)$ received the most recent CM M_i , and
2. any status changes at deeper levels of recursion which were made since M_i was received will be discarded.

Comments

The proofs of Properties A4.3 and A4.4 may be found in Appendix E. They indicate that when the solver returns NO or ABANDON in response to a CM or AVR, it has been restored to the state it was in before it received the CM, or the CM to which the AVR referred.

6.2 Responses Occur only if Conditions Met

The following proofs indicate that the solver will return various results only if certain conditions are met.

Property B1

If $MP(k)$ returns MAYBE in response to a CM M_i , then:

1. $MP(k)$ could not determine if the constraints in L_{M_i} were feasible or nonfeasible,

2. $MP(k)$ could not generate any solutions to any subset of constraints in $PS_{M_i}^+$,
3. no procedure $Pa(k)$ had a satisfaction function which could determine if the constraints in L_{M_i} were feasible or nonfeasible, and
4. no procedure $Pa(k)$ had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Comments

Property B1 describes the conditions that must have been present for $MP(k)$ to return MAYBE. It indicates that the capabilities of all its applicable procedures have been tested but without success. If any substitutions could be generated, they led to contradictions but no procedure could determine that constraints were nonfeasible.

The property relies upon (1) Restrictions RS1 and RS3 to ensure that if any procedure had a function which could have generated any substitutions or verified or rejected the constraints, its recognition rules and sufficiency predicate would have succeeded, and (2) Restriction SF5 to ensure that if any function could have returned YES when it was called, it would have done so (in which case $MP(k)$ would not return MAYBE).

In practice, this proof is only applicable to the base level of recursion (that is, for $k=0$) because whenever a procedure $Pi(k-1)$, $k \geq 1$ makes a recursive call, it sends $MP(k)$ a set of assignment constraints. Since these are trivial to solve, $MP(k)$ would either return at least as many substitutions as assignment constraints using $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$, or NO to indicate that the assignments led to a contradiction.

Property B2

If $MP(k)$ returns YES in response to a CM M_i , then:

1. the constraints in L_{M_i} are feasible,

2. $MP(k)$ could not generate any definite or tentative solutions to a subset of the constraints in $PS_{M_i}^+$, and
3. no procedure $Pa(k)$ could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Comments

This property describes the conditions that must have been present for $MP(k)$ to return YES. It indicates that the capabilities of all its applicable procedures have been tested, $MP(k)$ or at least one other procedure could determine that the constraints were feasible, and no procedure could generate any substitutions that did not lead to contradictions.

Property B2 relies upon many of the restrictions by way of the lemmas used in its proof. The ones of most interest are:

- Restriction SF5; it ensures that if any function returns YES, it was passed every constraint and the constraints were feasible,
- Restriction SF6; it ensures that if any function could have generated substitutions, it would have done so, and
- Restrictions RS1, RS3, and RS6 to ensure the success of recognition rules and sufficiency predicates of procedures with functions that could process the constraints.

In practice, this proof is only applicable for the base level of recursion (i.e., for $k=0$) because whenever a procedure $P_i(k-1)$, $k \geq 1$ makes a recursive call, it sends $MP(k)$ a set of assignment constraints. Since these are trivial to solve, $MP(k)$ would either return at least as many substitutions as assignment constraints using $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$, or NO to indicate that the assignments led to a contradiction.

Property B3

If $MP(k)$ returns $MAYBE(S)$ in response to a CM M_a , then:

1. $MP(k)$ could not determine if the constraints in $PS_{M_i}^+$ were feasible or nonfeasible,
2. no procedure $Pa(k)$ had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible,
3. if the solution was forwarded back through n levels of recursion, $n \geq 1$, no $MP(k+i)$, $i=1, \dots, n$, could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, and
4. if the solution was forwarded back through n levels of recursion, $n \geq 1$, no procedure $Pj(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Comments

This property describes the conditions that must have been present for $MP(k)$ to return $MAYBE(S)$. $MP(k)$ can return such a response either by making transition 10 after some child returns the same, or by making transition 27 after it has exhausted all of its children. In the first case, the substitutions would have been forwarded back through at least one level of recursion and so conclusions (3) and (4) apply. However, in the second case, conclusions (3) and (4) do not apply and all search paths at level k (and deeper) would have been explored.

Property B3 relies upon (1) Restrictions RS1 and RS3 to ensure that if any procedure with a function on an explored search path could have verified or rejected the constraints, the procedure's recognition rules and sufficiency predicate would have succeeded (in which case $YES(S)$, $YES+MAYBE(S)$, or NO would have been returned), and (2) Restrictions SF5 and SF6 to ensure that if any function could have returned YES or $YES(S)$, it would have done so (in which case $MP(k)$ would not return $MAYBE(S)$).

Property B4

If $MP(k)$ returns $YES+MAYBE(S)$ in response to a CM M_i , then:

1. the constraints in L_{M_a} are feasible,
2. if $MP(k)$ returned YES+MAYBE(S) after the substitutions in S were forwarded back through $n \geq 1$ levels of recursion, then no $MP(k+i)$, $i=1, \dots, n$, could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, and
3. if $MP(k)$ returned YES+MAYBE(S) after the substitutions in S were forwarded back through n levels of recursion, $n \geq 1$, then no procedure $P_j(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Comments

This property describes the conditions that must have been present for $MP(k)$ to return YES+MAYBE(S). $MP(k)$ can return such a response either by making transition 12 after some child returns the same, or by making transition 11 after one child returns YES and another returns MAYBE(S). Since the substitutions in S must have been forwarded back through at least one level of recursion, conclusions (2) and (3) apply regardless of whether $MP(k)$ made transition 11 or 12.

Like Property B2, this property relies upon many of the restrictions by way of the lemmas used in its proof. The ones of most interest are:

- Restriction SF5; it ensures that if any function returns YES, it was passed every constraint and the constraints were feasible,
- Restriction SF6; it ensures that if any function could have generated definite substitutions, it would have done so, and
- Restrictions RS1, RS3, and RS6 to ensure the success of recognition rules and sufficiency predicates of procedures with functions that could process the constraints.

Property B5

If $MP(k)$ returns $YES(S)$ in response to a CM M_a , then (1) the constraints in L_{M_a} are feasible, and (2) S contains a definite partial or full solution to the constraints in L_{M_a} .

Comments

This property describes the conditions that must have been present for $MP(k)$ to return $YES(S)$. $MP(k)$ can return such a response either by making transition 28 (in which case it must have generated some substitutions itself and at least one child returned YES), or by making transition 13 after some child returned the same.

The dependence of Property B5 on the procedure restrictions is the same as that of Property B4.

Property B6.1

If $MP(k)$ returns NO in response to a CM M_i , then $PS_{M_i}^+$ contained nonfeasible constraints.

Comments

Property B6.1 describes the conditions that must have been present for $MP(k)$ to return NO in response to a CM. It indicates that regardless of whether $MP(k)$ returned NO by making transition 14, 26, or 21, the constraints in L_{M_i} and $MP(k)$'s problem set had no solution in common.

This property relies upon Restrictions SF3 to ensure that if a child of $MP(k)$ returns NO after it or one of its descendents calls a function and the function returns NO, then the constraints in $PS_{M_i}^+$ had no solution.

Property B6.2

If $MP(k)$ returns NO in response to an AVR regarding some CM M_i , then all possible solutions to the constraints in L_{M_i} have been returned already.

Comments

Property B6.2 is similar to that of B6.1 but it provides for the case when constraints are feasible but the number of solutions is finite. It also reveals an interesting limitation of the solver that needs some explanation.

If $MP(k)$ returns NO in response to an AVR after having previously returned a series of MAYBE(S) replies, one of the following two cases has occurred:

1. A function has determined that all possible solutions to the constraints in $PS_{M_i}^+$ have been returned already, but nothing has been determined regarding the feasibility of the constraints in $PS_{M_i}^+$ (and the returned solutions).
2. A function has determined that there are no solutions to the constraints in $PS_{M_i}^+$, in which case all previously returned solutions must be invalid.

Since case (2) provides data that (1) does not, it would be useful to know which one has occurred. Unfortunately, this is not possible from the reply sequence alone and so a limitation of the solver is revealed. However, in order to generate such a reply sequence, the solver would have to abandon search paths on occasion and so the techniques discussed in § 5.2 could be of use in differentiating the two cases. Another possibility would be to modify the requirements of satisfaction functions so that two different "NO" responses were possible, and to incorporate additional transitions into the state transition diagrams of $MP(k)$ and the other procedures.

As in the case of Property B6.1, this property relies upon Restrictions SF3 to ensure that if any function returns NO, then all possible solutions have been returned already.

6.3 Responses Occur if Conditions Met

The following proofs indicate that the solver will or will not return various results if certain conditions are met.

Property C1.1

For any $k \geq 1$, if $MP(k)$ is sent some CM M_a , then $MP(k)$ will not return MAYBE or YES.

Comments

This property is used to explain some "missing" transitions from the state transition diagram of Figure 4.7. Whenever some procedure $P_i(k-1)$, $k \geq 1$, sends a

CM to $MP(k)$, it waits in state 12 or 14 for a reply. In the state transition diagram of Figure 4.4, $MP(k)$ could return MAYBE or YES but Figure 4.7 does not provide transitions for these replies. However, by Property C1.1, $MP(k)$ cannot return MAYBE or YES to some procedure $P_i(k-1)$ and so the diagram of Figure 4.7 is acceptable. Note that this property does not apply to $MP(0)$ and so MAYBE or YES can be returned to PE.

Property C1.2

If $MP(k)$ receives a CM M_a and the constraints in L_{M_a} are feasible, then $MP(k)$ will not return NO.

Comments

This property is used to ensure that the solver cannot reject a feasible set of constraints at any level of recursion. It is also used in the proofs of several other properties.

Property C1.2 relies upon Restriction SF3 to prevent any function from returning NO before $MP(k)$ can return all possible solutions.

Property C1.3

If $MP(k)$ receives a CM M_a containing a nonfeasible constraint c , then YES, YES(S), and YES+MAYBE(S) will not be returned.

Comments

This property is used to ensure that the solver cannot indicate that a nonfeasible set of constraints is feasible. It is also used in the proofs of several other properties.

Property C1.3 is based on Properties B2, B4, and B5 and so it indirectly depends on most of the restrictions of § 4.4. The ones of most interest are Restrictions SF5 and SF6 because they prevent any function from returning YES or YES(S) if it is passed a nonfeasible set of constraints.

Property C2.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,

2. no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one is not satisfied,
3. no procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^+$,
4. no procedure has a satisfaction function which can determine that a subset of constraints in $PS_{M_a}^+$ is nonfeasible,
5. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, and
6. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$,

then $MP(k)$ will return MAYBE.

Comments

Property C2.1 describes a set of conditions which will result in $MP(k)$ returning MAYBE in response to a CM. It indicates that procedures could recognize data in $MP(k)$'s problem set but substitutions may not be returned. This property depends on many restrictions by way of the lemmas in its proof.

In practice, Property C2.1 is only applicable for the case of $k=0$ because whenever a procedure $Pi(k-1)$, $k \geq 1$ makes a recursive call, it sends $MP(k)$ a set of assignment constraints. Since these are trivial to solve, condition (2) would not hold and so $MP(k)$, $k > 0$, could not return MAYBE.

Property C2.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one is not satisfied, and

3. no constraint in $PS_{M_a}^+$ is recognizable by any procedure $P_i(k)$.

then $MP(k)$ will return MAYBE.

Comments

Property C2.2 describes a second set of conditions that would result in $MP(k)$ returning MAYBE. It indicates that if $MP(k)$ does not generate any substitutions using local propagation and no procedure can recognize data in $MP(k)$'s problem set, no substitutions will be returned and the constraints can neither be verified or rejected.

This property is based on Property C2.1 and so it indirectly depends on many of the restrictions of § 4.4. As well, it is only applicable in practice for the case of $k=0$ for the reasons given with Property C2.1.

Property C2.3

If $MP(k)$ is sent an AVR after having returned MAYBE, it will return ABANDON.

Comments

This property ensures that after the solver has returned MAYBE, any request for an alternate solution will result in an ABANDON reply. This verifies that one of the reply sequences listed in Table 4.2 will hold. Property C2.3 does not depend on any procedure restrictions.

Property C3.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$, and
 2. $MP(k)$ can determine that every constraint in $PS_{M_a}^+$ is satisfied,
- then $MP(k)$ will return YES.

Comments

This property describes one set of conditions which will result in $MP(k)$ returning YES, and it will do so by making transition 15. It requires $MP(k)$ to determine that every constraint in its problem set is satisfied.

In practice, this property is only applicable for the case of $k=0$ because whenever a procedure $P_i(k-1)$, $k \geq 1$, makes a recursive call, it sends $MP(k)$ a set of assignment constraints. Thus, $PS_{M_a}^+$ must contain unsatisfied constraints if $k > 0$.

Property C3.1 does not depend on any procedure restrictions.

Property C3.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraints in L_{M_a} are MP-solvable and at least one is not satisfied,
3. at least one procedure $P_i(k)$ has a function F which can verify the constraints in L_{M_a} .
4. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of $PS_{M_a}^+$, and
5. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$.

then $MP(k)$ will return YES.

Comments

This property describes a second set of conditions which will result in $MP(k)$ returning YES, and it will do so by making transition 6. It requires $MP(k)$ to not generate any substitutions and one of its descendents to determine that the constraints in its problem set are feasible.

This property relies upon many of the restrictions by way of the lemmas and Property C1.2 which appear in its proof. The ones of most interest are:

- Restriction SF3; it prevents any function from returning NO before YES can be returned.
- Restriction SF5; it ensures that if any function returns YES, it was passed every constraint and the constraints were feasible,

- Restriction SF6; it ensures that if any function could have generated definite substitutions, it would have done so, and
- Restrictions RS1, RS3, and RS6 to ensure the success of recognition rules and sufficiency predicates of procedures with functions that can process the constraints.

In practice, this property is only applicable for the case of $k=0$ because whenever a procedure $P_i(k-1)$, $k \geq 1$, makes a recursive call, it sends $MP(k)$ a set of assignment constraints and since these are trivial to satisfy, condition (2) would not hold.

Property C3.3

If $MP(k)$ is sent an AVR after having returned YES, it will return NO or ABANDON.

Comments

This property ensures that after the solver has returned YES, any request for an alternate solution will result in the solver returning NO or ABANDON. This verifies that one of the reply sequence listed in Table 4.2 will hold. Property C3.3 does not depend on any procedure restrictions.

Property C4.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains one or more MP-solvable constraints,
3. $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints in L_{M_a} ,
4. $MP(k)$ cannot detect any nonfeasible constraints in any instance of $PS_{M_a}^*$,
5. no procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^*$.

6. no procedure has a satisfaction function which can determine that a subset of constraints in $PS_{M_a}^4$ is nonfeasible,
7. if any procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$, then $MP(k+1)$ will return $MAYBE(S)$ when sent the substitutions, and
8. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of $PS_{M_a}^4$,

then $MP(k)$ will return $MAYBE(S)$.

Comments

Property C4.1 describes a set of conditions which will result in $MP(k)$ returning $MAYBE(S)$; it could do so by making either transition 27 or 10, depending on the outcome of condition (7). The key point is that $MP(k)$ could return $MAYBE(S)$ without any substitutions being forwarded back through recursive calls. However, $MP(k)$ must satisfy at least one constraint itself because of condition (2).

Property C4.1 relies upon (1) Restriction SF3 to ensure that no function will return NO as long as the possibility of additional solutions remain, and (2) Restriction SF5 to ensure that if any function (on an explored search path) could return YES, it would do so (in which case $MP(k)$ would not return $MAYBE(S)$).

Property C4.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraint in L_{M_a} is MP-rejectable and at least one is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints, and $MP(k)$ cannot detect any nonfeasible constraints in any instance of $PS_{M_a}^*$,

4. at least one procedure $P_i(k)$ has a function F which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ returns $MAYBE(S)$ when sent the substitutions,
5. if any procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^4$, it does not call the function before $P_i(k)$ calls F ,
6. if any procedure has a function which can determine that a subset of the constraints in $PS_{M_a}^4$ is nonfeasible, it does not call the function before $P_i(k)$ calls F ,
7. if any procedure has a function which can generate a definite partial or full solution to a subset of $PS_{M_a}^4$, it does not call the function before $P_i(k)$ calls F , and
8. if any procedure has a function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return $YES+MAYBE(S)$ or $YES(S)$ if sent the solution, it does not call the function before $P_i(k)$ calls F ,

then $MP(k)$ will return $MAYBE(S)$.

Comments

Like Property C4.1, C4.2 describes a set of conditions which will result in $MP(k)$ returning $MAYBE(S)$; however, it must do so by making transition 10. Note that condition (4) requires some procedure to make a recursive call and $MP(k+1)$ to return $MAYBE(S)$. Property C4.1 describes a set of conditions which would result in $MP(k+1)$ returning such a response¹ and so the two properties together show how a $MAYBE(S)$ response can be generated and then forwarded back to PE through multiple levels of recursion.

Property C4.2 indirectly relies upon virtually all of the restrictions of § 4.4.

Property C5.1

Given the following conditions:

¹Note that the k in Property C4.1 needs to be replaced by $k+1$.

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains at least one constraint which is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints,
4. at least one procedure $Pa(k)$ has a satisfaction function F_a which can verify the constraints in $PS_{M_a}^4$,
5. at least one procedure $Pb(k)$ has a function F_b which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return $MAYBE(S)$ if sent the substitutions,
6. $Pa(k)$ calls F_a before $Pb(k)$ calls F_b , and
7. if any procedure has a satisfaction function which can generate a definite or tentative solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return $YES(S)$ or $YES+MAYBE(S)$ if sent the solution, then the procedure does not call the function before $Pb(k)$ calls F_b ,

then $MP(k)$ will return $YES+MAYBE(S)$.

Comments

Property C5.1 describes a set of conditions which will result in $MP(k)$ returning $YES+MAYBE(S)$ by making transition 11. Condition (4) requires some procedure to verify that $MP(k)$'s problem set is feasible, and condition (5) requires some procedure to make a recursive call and $MP(k+1)$ to return $MAYBE(S)$. Thus, two separate pieces of information are combined to create the $YES+MAYBE(S)$ response.

Property C5.1 indirectly relies upon virtually all of the restrictions of § 4.4.

Property C5.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains at least one constraint which is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints,
4. at least one procedure $Pb(k)$ has a function F_b which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ returns YES+MAYBE(S) when sent the substitutions, and
5. if any procedure has a satisfaction function which can generate a definite or tentative solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return MAYBE(S) or YES(S) if sent the substitutions, then the procedure does not call the function before $Pb(k)$ calls F_b ,

then $MP(k)$ will return YES+MAYBE(S).

Comments

Like Property C5.1, C5.2 describes a set of conditions which will result in $MP(k)$ returning YES+MAYBE(S); however, in this case it will do so by making transition 12. Note that condition (4) requires some procedure to make a recursive call and $MP(k+1)$ to return YES+MAYBE(S). Property C5.1 describes a set of conditions which would result in $MP(k+1)$ returning such a response ² and so the two properties together show how a YES+MAYBE(S) response can be generated and then forwarded back to PE through multiple levels of recursion.

Property C5.2 indirectly relies upon virtually all of the restrictions of § 4.4.

Property C6.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$.

²Note that the k in Property C5.1 needs to be replaced by $k+1$.

2. L_{M_a} contains one or more MP-solvable constraints, and
3. $MP(k)$ can totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints in L_{M_a} ,

then $MP(k)$ will return $YES(S)$.

Comments

Property C6.1 describes a set of conditions which will result in $MP(k)$ returning $YES(S)$ in response to a CM. It requires $MP(k)$ to satisfy every constraint without calling any other procedure and so it will return $YES(S)$ by making transition 25.

Property C6.2 indirectly relies upon virtually all of the restrictions of § 4.4.

Property C6.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains one or more MP-solvable constraints,
3. at least one procedure $P_i(k)$ has a satisfaction function which can verify the constraints in $PS_{M_a}^+$,
4. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, and
5. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$.

then $MP(k)$ will return $YES(S)$.

Comments

Property C6.2 describes another set of conditions which will result in $MP(k)$ returning $YES(S)$, and it will do so by making transition 28. The property requires (1) $MP(k)$ to have generated at least one substitution and (2) one of its descendants to have verified the results.

Property C6.2 indirectly relies upon virtually all of the restrictions of § 4.4.

Property C6.3

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. there is at least one procedure $Pi(k)$ with a function F which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ will return $YES(S)$ when sent the substitutions, and
3. if any procedure sends a CM to $MP(k+1)$ before $Pi(k)$ calls F , it will receive a NO reply,

then $MP(k)$ will return $YES(S)$.

Comments

Property C6.3 describes a third set of conditions which will result in $MP(k)$ returning $YES(S)$, and it will do so by making transition 13. The property requires some procedure to have made a recursive call and $MP(k+1)$ to have returned $YES(S)$. Properties C6.1 and C6.2 describe two sets of conditions which would result in $MP(k+1)$ returning such a response ³ and so the three properties together show how a $YES(S)$ response can be generated and then forwarded back to PE through multiple levels of recursion.

This property indirectly relies upon virtually all of the restrictions of § 4.4.

Property C7.1

If (1) a CM M_a is sent to $MP(k)$ and (2) there exists a constraint $c \in L_{M_a}$ such that c is MP-rejectable, then $MP(k)$ will return NO.

Comments

Property C7.1 describes a set of conditions which will cause $MP(k)$ to return NO by making transition 14. No other procedures are called between the time that

³Note that the k in Properties C6.1 and C6.2 needs to be replaced by $k+1$.

$MP(k)$ receives the CM and when it replies. This property does not rely on any of the restrictions of § 4.4.

Property C7.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains one or more MP-solvable constraints, and
3. the solution to some MP-solvable constraint $c \in L_{M_a}$ results in some other constraint $d \in PS_{M_a}^*$ becoming MP-rejectable,

then $MP(k)$ will return NO.

Comments

Property C7.2 describes a second set of conditions which will cause $MP(k)$ to return NO, and it will do so by making transition 26. No other procedures are called between the time the $MP(k)$ receives the CM and when it replies. This property does not rely on any of the restrictions of § 4.4.

Property C7.3

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. at least one procedure $Pi(k)$ has a function F which can determine that $PS_{M_a}^4$ contains one or more nonfeasible constraints, and
3. if any procedure has a satisfaction function which could generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would not reject the solution, then the procedure does not call the function before $Pi(k)$ calls F ,

then $MP(k)$ will return NO.

Comments

Property C7.3 describes a third set of conditions which will result in $MP(k)$ returning NO, and it will do so by making transition 21. It requires some procedure $Pi(k)$ to have a function F which can find a subset of $MP(k)$'s problem set to be nonfeasible, and $Pi(k)$ must call F before some other response can be returned to $MP(k)$.

This property is based in part on Property C1.3, and C1.3 uses Properties B2, B4, and B5 in its proof. Thus, C7.3 depends on most of the restrictions of § 4.4.

Chapter 7

Expressiveness

Although metalevel data has been used in logic programming for many years [5, 1, 21], its application to constraint satisfaction problems is quite new. One approach has been to transform metalevel data into additional constraints which narrow the search space for a generate-and-test satisfaction method [42]. However, the use of metalevel data to help identify appropriate satisfaction methods is believed to be new to this thesis. In this chapter, it is shown how such data can be specified in a declarative fashion and how its procedural interpretation can be used to:

- locate suitable procedures in the hierarchy as quickly as possible,
- reduce the amount of data that is needlessly passed to procedures,
- overcome minor limitations of available satisfaction functions, and
- minimize the number of duplicate and tentative solutions that the solver may produce.

Each of the first three sections of this chapter presents a different type of metalevel data; the fourth section looks at the potential problems of using different types simultaneously; the fifth examines the effects on solver properties.

7.1 Data Typing

The simplest form of metalevel data is that describing the arithmetic data type of variables. In Trilogy and CLP($\mathcal{R}$), the arithmetic type of a variable is implicitly declared by the presence of the (uninstantiated) variable in a constraint. In BNR Prolog, an explicit declaration is required for any variable which is to be processed using its interval arithmetic methods. However, in no case does the arithmetic data type *guide* the satisfaction process. To do so with the new solver architecture, two issues must be addressed:

- the data type of a variable must be specified, and
- procedures must be able to make use of the data.

7.1.1 Specifying the Data Type of a Variable

An example of specifying the data type of a variable may be found in BNR Prolog. To specify that a variable X is interval-valued with (initial) lower and upper limits of X_L and X_U respectively, the following metalevel predicate is used:

```
interval( X, [ XL , XU ] )
```

Here, the functor name indicates the arithmetic class and so if this approach were extended, a separate predicate would be required for each data type.

Another possibility is to use a more general predicate which takes the data type as an argument, as in the following:

```
of_class( X, Type )
of_class_and_value( X, Type, X0 )
```

The first would be read as "Variable X is of the *Type* arithmetic class", and the second as "Variable X is of the *Type* arithmetic class with a value of X_0 ", where X_0 indicates a specific value or range of values.

The *of_class* technique will be used in the remainder of this thesis because it is consistent with other metalevel predicates to be discussed.

7.1.2 Making Use of Variable Data Types

As discussed by Van Hentenryck [41], on average, it is better to spend a little time determining that values do not need to be generated than to generate them and then test to see if they are appropriate. Thus, although the *of-class* predicates could be used to check for appropriate values after a variable was instantiated, it is better to (1) use the predicate to declare the type of a variable before it appears in a constraint, and (2) let the solver make use of the data to avoid calling inappropriate satisfaction functions. This can be accomplished by using *of-class* predicates in a procedure's recognition rules as follows:

1. Whenever a procedure tests a constraint to see if it is recognizable, it isolates each variable and checks to see if it is of a particular arithmetic type.
2. If the type of each variable is either unspecified or it is consistent with the types that the procedure's satisfaction functions can assign, then the remainder of the constraint is checked; otherwise, the constraint is considered to be unrecognizable.

The most significant advantages offered by this technique are (1) it helps the solver avoid the generation of inappropriate values, and (2) it helps to minimize the amount of data that is needlessly passed to procedures. For instance, if only rational-valued solutions to a constraint are of interest, once one procedure uses the data type information to determine that it cannot generate appropriate values, that is, it finds the constraint to be unrecognizable, the constraint is not passed to any of its descendants.

7.2 Constraint Algebraic Classifications

The algebraic classification of an arithmetic expression is a very useful piece of information. It may be used to:

- determine whether or not an expression can be solved without the use of any other data,

- identify a solution procedure, and
- determine the number of solutions.

Thus, the algebraic classifications of constraints in combination with a priori knowledge of how to solve expressions of various classes may be used to guide the satisfaction process.

The syntax of constraints implicitly describes the classification of a constraint but unfortunately, trying to make use of this data can be time consuming. Whenever a procedure compares the syntax of a constraint with the patterns defined in its recognition rules, it is testing to see if the constraint falls into a class that it can solve. Depending on how many procedures are sent a constraint and how many rules they contain, many comparisons might have to be made before a function could be called.

A better approach is to provide the user with:

- a means of explicitly stating metalevel data about the algebraic class of a constraint, and
- a means of mapping algebraic classifications to procedures in the solver.

The metalevel data is in the form of Prolog clauses. They will state that a constraint is of a particular class if certain conditions are met, and the classification will be mapped via a table to one or more procedures in the tree. Thus, the metalevel data will indirectly identify a suitable procedure or, as will be shown, at least narrow the search space for finding one. The classification data will not, however, improve the abilities of the procedures. Asserting that a constraint is of a particular class will not give the solver any new means of satisfying it.

This section does not present an explanation of how the solver could make use of the classification data because any required modifications would not be able to accommodate the metalevel data of the next section. Consequently, means of utilizing both types of data are not presented until the end of § 7.3.

7.2.1 Specifying Classification Data

The metalevel data will be specified using a label and a list of clauses. The label provides a convenient means of referring to the constraint as a unit while classification data is indicated in the clauses. The head of each clause uses the functor name *isa* with arity of two. The first argument refers to the constraint's label, while the second indicates its algebraic class.

To ensure that a parser can be written for constraints with classification data, the clause list is preceded by an “#” symbol, an operator that indicates the presence of metalevel data. For instance, the following describes constraint *c1* as a linear expression:

c1: $X + Y + Z = 5$ *#*[*isa*(*c1*,linear)].

Since constraints may fall into more than one class, several *isa* clauses may be used, each separated by a comma as in:

c2: $X^3 + X^2 + X - 100 = 0$ *#*[*isa*(*c2*,polynomial), *isa*(*c2*,nonlinear)].

The class of a constraint may depend on which variables are bound and the values of the bindings, therefore rules may be used to indicate the class as in the following:

c3: $X^N - 5X + 2N = 0$
#[*isa*(*c3*(*X*,*N*), nonlinear) :- [*var*(*X*), *var*(*N*)],
isa(*c3*(*X*,*N*), quadratic) :- [*var*(*X*), *N*=2],
isa(*c3*(*X*,*N*), poly) :-[*var*(*X*), *of_class*(*N*,integer), *N*>2]].

These rules state that:

- if neither *X* nor *N* are bound, *c3* is nonlinear.
- if *X* is not bound and *N* is bound to the value 2, *c3* is a quadratic, and
- if *X* is not bound and *N* is an integer greater than 2, *c3* is a polynomial.

Note that in the above rules, (1) the antecedents are enclosed in brackets so that they do not run into the beginning of the next clause, and (2) an *isa* predicate's first argument must contain references to any variables that appear in the rule's antecedents; otherwise, the rules would not be consistent with Prolog conventions.

The CP Table

To map the classification data to a procedure, a *Class-Procedure* (CP) table is used; a typical example is shown in Table 7.1. Beside the name of each class are the names of one or more procedures which can solve constraints of that class. Thus, given the classification of a constraint, one or more suitable procedures can be identified immediately. Also note that procedure names can appear beside more than one classification. The use of the table is explained in more detail in the next section.

CP Table		
Classification	Procedure Name	Procedure can Satisfy
linear_inequality	Pa	Linear equations and inequalities
linear	Pa	Linear equations and inequalities
	Pb	Linear equations
tridiagonal	Pa	Linear equations and inequalities
	Pb	Linear equations
	Pc	Tridiagonal linear equations
quadratic	Pd	Quadratic equations
	Pe	Quadratic equations symbolically

Table 7.1: An example CP Table

7.2.2 Classification Data Considerations

Before the solver's search mechanism is modified to make use of the classification data, the following questions are considered.

Question 1: If no metalevel data is provided with a constraint, should it be assumed that nothing has been determined about the algebraic classification of the constraint, in which case the solver should process it using the methods described in Chapter 4?

If one were to answer "Yes" to this question, it could be argued that the closed-world assumption of Prolog would be violated.¹ Under the closed-world assumption, if no metalevel predicate could prove that a constraint c was of some class Q , then it would be assumed that c did not fall into that class. Consequently, there would be no point in sending it to a procedure that could solve constraints of class Q . Taking this argument further, if no metalevel predicates were provided with constraint c , it could be assumed that c did not fall into any class that the solver could satisfy and so there would be no point in sending it to the procedures.

This line of reasoning is tantamount to requiring users to classify every constraint but to do so would violate the purpose of the solver, namely to provide a means of automatically solving arithmetic expressions without requiring any assistance from the user. Therefore, it will be assumed that if no metalevel data is provided, nothing has been determined regarding the classification of a constraint.

Question 2: If metalevel data is provided with a constraint but every *isa* predicate fails, what does this mean? It could be interpreted in several ways:

1. It has been determined that the constraint does not belong to any class appearing in the *isa* predicate list, and if it does not fall into one of these classes, then the solver cannot process it in its current form.

¹Under Prolog's closed world assumption, a predicate is assumed to be false if it cannot be proven true using the available program data.

2. It has been determined that the constraint is not of any class appearing in the *isa* predicate list, but it could be of some other class that the solver can process.
3. Nothing has been determined about the classification of the constraint and so it could be of any class, including one or more of those listed in the failed *isa* predicates.

When a constraint contains variables, it may fall into many different classes, depending upon which variables are bound and the specific bindings. Since the use of metalevel data is intended to be reasonably convenient, users are not expected to provide an exhaustive list of potential classifications. However, if the first interpretation was adopted and users provided little metalevel data, the solver would be likely to abandon the search process prematurely. Thus, the first interpretation will not be used.

Under the second interpretation, the failure of an *isa* predicate contains useful information which could be used to narrow the search space. For instance, if a constraint *c* was assumed not to be of some class *Q* because *Q* appeared in a failed *isa* predicate, then the solver could avoid sending *c* to procedures that satisfied constraints of class *Q*. However, consider the following implications:

- In general, procedures which are to be avoided may be at arbitrary depths in the hierarchy and so every procedure would have to be able to determine which of its children should be sent CM's and which should not.
- In a single CM, a procedure could be sent two new constraints, one that was found to be of some class *Q* and another that was found not to be of that class (because of a failed *isa* predicate). Thus, procedures would have to be able to send selected parts of their problem set to different children.

Both of these indicate that the satisfaction process could become much more complicated and so the second interpretation will not be used.

The only remaining possibility is the third interpretation. Fortunately, it is consistent with that of the total absence of metalevel data as discussed in Question (1).

Thus, if all *isa* predicates which are provided with a constraint fail, the solver will process the constraint using the methods described in Chapter 4.

Question 3: When all of the search paths indicated by successful metalevel clauses have been exhausted, should any others be explored and if so, which ones?

Consider the following example. Assume that one metalevel predicate was provided with a constraint *c* and it indicated that *c* should be sent to some procedure *Pa*. If MP sent it a CM and *Pa* returned MAYBE, then *Pa* must have lacked sufficient data to generate a more useful response. While the chance does exist that some other procedure might be able to satisfy *c* in combination with other data, a problem arises if the solver were to continue searching.

In general, constraints may be satisfied in isolation only if they are equations in one unknown. Thus, most can be satisfied only in combination with other data. Since PE encounters constraints one at a time, the solver will often have to receive additional data before it is able to satisfy a constraint, even if metalevel data is provided with it. Thus, if the solver were to continue searching just because some procedure returned MAYBE

- it would end up exploring additional search paths after receiving most CM's, and
- in general, it would have to be able to explore every possible search path because it would have no way of knowing which ones to explore.

Clearly, this approach defeats the intended purpose of metalevel data and so if any metalevel predicate succeeds, the solver will explore only the indicated search paths.

It should be noted that this interpretation places a modest burden on the user to ensure that if any algebraic classifications are provided, they map to procedures that are capable of generating every solution to the constraint. Otherwise, the possibility exists that all solutions may not be found even though the solver could

generate them if the metalevel data was not provided. Thus, the use of metalevel data becomes similar to that of Prolog's infamous cut ("!") operator: depending on how the *isa* clauses are used, they may or may not affect what may be computed from a program.

Question 4: If the class of a constraint is indicated by the success of an *isa* clause, could the constraint also fall into a more restrictive subclass? For instance, does specifying that a constraint is a polynomial preclude the possibility that it is a quadratic?

An affirmative reply indicates that the search for a procedure should not be confined to just the procedures indicated in the CP table, but should also include all of their descendents. This solution tactic will be used because it makes the solver more robust, that is, it lets the system search for the best available method but only among those subclasses of the indicated class. Another benefit is users do not have to provide very precise classifications since any data at all would help to narrow the search space.

In conclusion, the above discussions indicate the following:

- In the total absence of metalevel data or in the absence of metalevel data that can be proven true, the solver should behave as described in Chapter 4.
- If any metalevel data which is provided with a constraint is proven to be true, the solver should use it in order to direct the constraint to appropriate procedures. However, if the procedures are unable to return specific values or reject the constraint, the solver should not send the constraint to procedures which have not been identified using metalevel data.
- A procedure which receives a CM should give its children the first opportunity to satisfy the constraints in its problem set, even if the CM was caused by the success of a metalevel clause.

7.3 Grouping Constraints

Another useful piece of information is which constraints in a group can be, or need to be, solved as a simultaneous set. Consider the following:

$$X + Y = 1$$

$$X \times Z = 3$$

$$X - Y = 2$$

If only an iterative satisfaction method was available, one approach would be to solve the three equations in three unknowns. A better way would be to solve the first and third and then the second would become trivial to satisfy (since Z would be the only unknown). Also note that if the second equation was changed to an inequality, the three might not appear to have a unique solution even though the solutions for X and Y have not changed.

The above example indicates that if a set of constraints can be partitioned into subsets, the satisfaction process may be simplified. Thus, if users have metalevel data regarding how constraints should be grouped, they should be able to express it and the solver should be able to make use of it.

7.3.1 Specifying the Group Data

Constraints will be partitioned into (possibly overlapping) subsets called *groups*. By default, if no metalevel data is provided with a constraint, it will be assumed to belong to a *global* group. Users must declare groups at the beginning of a program and they need to give each one a unique name. For instance, the following code segment declares two groups with the names *set1* and *set2*.

```
group: set1, set2.
```

A special kind of group known as a *unit group* is declared in a similar manner; these are discussed below.

To associate a constraint with one or more groups, a metalevel *belongs_to* clause is used in a similar fashion to the *isa* predicates of the previous section. The constraint

needs to be labelled and it is followed by an '#' operator and a list of *belongs_to* clauses. Consider the following example:

group: set1.

a(X,Y,Z):- c1: $Z^*(X-Y) = 1$ #[belongs_to(c1(Z), set1) :- [not var(Z)]],
 c2: $Z^*(X+Y) = 2$ #[belongs_to(c2(Z), set1) :- [not var(Z)]],
 c3: $Z^2 = 100$.

The first and second equations are declared to be a part of group *set1* as long as *Z* is ground, and *c3* is declared (by default) to belong to the global group.

Unit Groups

At times, it may be known that no constraint in a group can be satisfied until one in particular is encountered. For instance, in the above example, if *Z* is ground but *X* and *Y* are not, values for *X* and *Y* cannot be determined unless constraints *c1* and *c2* are combined. Thus, there is no point in sending *c1* to the solver as soon as it is encountered.²

If a constraint is associated with a unit group, PE will not send it to the solver as soon as it is encountered. Instead, it will delayed until a constraint which is tagged with a *last_in_group* predicate is available; only then will every constraint in the group be sent to MP, and they will be sent in a single CM. For instance, the above example can be modified as follows:

unitgroup: set1.

a(X,Y,Z):- c1: $Z^*(X-Y) = 1$ #[belongs_to(c1(Z), set1) :- [not var(Z)]],
 c2: $Z^*(X+Y) = 2$ #[last_in_group(c2(Z), set1) :- [not var(Z)]],
 c3: $Z^2 = 100$.

²Other examples may be found in (1) boundary value problems in which a set of equations depends on a fixed number of boundary conditions, and (2) optimization problems in which no values can be generated until all *constraints* (in the Operations Research sense of the word) and an objective function are available.

Assuming that Z is instantiated when predicate a is tested (so that the *belongs.to* and *last.in.group* clauses will succeed), the declaration of *set1* as a unit group prevents $c1$ from being sent to the solver when it is first encountered. Once PE detects the *last.in.group* predicate, it sends both $c1$ and $c2$ to $MP(\theta)$ in a single CM.

If (1) PE was to send an AVR to MP after backtracking into $c2$, and (2) MP was to return NO, then PE would backtrack over both $c2$ and $c1$, that is, PE would not send an AVR to MP with regard to $c1$ because $c1$ was not sent to the solver in a separate CM. Thus, the use of unit groups can reduce both the number of CM's that are sent to $MP(\theta)$ and the number of search paths that are created.

7.4 Combining Classification and Group Data

One might expect that the algebraic classification of a constraint should not affect the way that it could be grouped, nor should the groups influence how it could be classified. Unfortunately, trying to accomodate such an approach results in the problems described in Appendix F. In general, if the two types of data are used independently, it is very easy for related constraints to be split into groups such that no procedure has sufficient data to generate any substitutions (unless much more complicated search methods are used).

To avoid these problems, algebraic classifications and group data are combined so that groups may be classified as well as individual constraints. This allows grouped constraints to be indirectly classified. It is believed that this is a reasonable approach because constraints that are grouped together are likely to require the same satisfaction mechanisms. Besides avoiding significant changes to the existing solver, this approach also has the following advantages:

- it is as easy to understand as when the two forms of metalevel data are expressed independently,
- it is more convenient to use, and
- it will ensure that all of the constraints in a group have at least one common classification.

Assuming that Z is instantiated when predicate a is tested (so that the *belongs_to* and *last_in_group* clauses will succeed), the declaration of *set1* as a unit group prevents $c1$ from being sent to the solver when it is first encountered. Once PE detects the *last_in_group* predicate, it sends both $c1$ and $c2$ to $MP(0)$ in a single CM.

If (1) PE was to send an AVR to MP after backtracking into $c2$, and (2) MP was to return NO, then PE would backtrack over both $c2$ and $c1$, that is, PE would not send an AVR to MP with regard to $c1$ because $c1$ was not sent to the solver in a separate CM. Thus, the use of unit groups can reduce both the number of CM's that are sent to $MP(0)$ and the number of search paths that are created.

7.4 Combining Classification and Group Data

One might expect that the algebraic classification of a constraint should not affect the way that it could be grouped, nor should the groups influence how it could be classified. Unfortunately, trying to accomodate such an approach results in the problems described in Appendix F. In general, if the two types of data are used independently, it is very easy for related constraints to be split into groups such that no procedure has sufficient data to generate any substitutions (unless much more complicated search methods are used).

To avoid these problems, algebraic classifications and group data are combined so that groups may be classified as well as individual constraints. This allows grouped constraints to be indirectly classified. It is believed that this is a reasonable approach because constraints that are grouped together are likely to require the same satisfaction mechanisms. Besides avoiding significant changes to the existing solver, this approach also has the following advantages:

- it is as easy to understand as when the two forms of metalevel data are expressed independently,
- it is more convenient to use, and
- it will ensure that all of the constraints in a group have at least one common classification.

The metalevel data is expressed in much the same manner as before. However, group names may now be followed by a list of *isa* facts; there is no need for variables. For instance, two groups could be declared as follows:

```
group: simple_eqn #[ isa( simple_eqn, linear), isa( simple_eqn, quadratic)],
      nonlin_eqn #[ isa( nonlin_eqn, nonlinear) ].
```

This declares *simple_eqn* to be a group of linear and quadratic constraints, and *nonlin_eqn* to be a group of nonlinear expressions.

As described in § 7.3, constraints are associated with the groups by the use of *belongs_to* clauses as demonstrated below:

```
c: A^N+A = 4 #[belongs_to( c(N), simple_eqn) :- [ N=1 ],
               belongs_to( c(N), simple_eqn) :- [ N=2 ],
               belongs_to( c(N), nonlin_eqn) :- [ N/=1, N/=2 ] ].
```

Note that “ $\neq$ ” represents “ $\neq$ ” using standard keyboard symbols. The bracketed names indicate that *c* is a member of the groups *simple_eqn* and *nonlin_eqn*, depending on the value of *N*. Thus, if *N* has a value other than 1 or 2, *c* becomes a member of the *nonlin_eqn* group.

7.4.1 Modifying the Solver for Classification and Group Data

The solver will now be modified to make use of both algebraic classification and group metalevel data. The modifications are based on the discussions of § 7.2.2 and the following design decision:

Metalevel clauses are to be evaluated only once, that is, they will not be re-evaluated every time that new data becomes available.

It is possible that whenever new variable bindings are generated by the solver, the values could result in the successful evaluation of a metalevel clause. Thus, one design approach would be to re-evaluate metalevel data which had previously

failed (and possibly data which had previously succeeded to see if it was still true). However, this approach is very complicated, it introduces more search overhead, and if all of the metalevel clauses provided with a constraint had previously failed, then the solver would have searched for an appropriate satisfaction procedure already. Thus, there is no guarantee that an outstanding clause could ever provide useful information.

Modifications to MP

To keep track of the metalevel data, MP will make use of a data structure called a *Constraint-Group-Classification-Procedure* (CGCP) graph; an example is depicted in Figure 7.1. Constraints, groups, classifications, and procedures are represented as nodes in the graph, while edges associate:

- constraints with groups in the C-G layer,
- groups with classifications in the G-C layer, and
- classifications with procedures in the C-P layer.

The edges in the C-G layer are derived from a constraint's *belongs_to* clauses; those in the G-C layer are provided by the list of *isa* clauses that follows a group declaration; the C-P edges are derived from the CP table. Edges can also link constraint and classification nodes when a constraint is explicitly classified using an *isa* clause.

The new state diagram for $MP(k)$ is shown in Figure 7.2. Note that the top and bottom portions of the diagram are almost identical to the transitions of Figure 4.4: the new transitions in the middle will make use of the metalevel data.

When PE encounters a constraint, it immediately evaluates any associated metalevel clauses (which may involve sending constraints in clause antecedents to the solver). Once all of the clauses have been evaluated, PE sends the constraint with a list of any successful data to $MP(0)$.

When $MP(k)$ receives a CM M_i , it initially behaves as it did before, that is, it adds any new variables to VV and any new constraints to SL. It then looks at its

Interpretation:

c1 is linear.

c2 and c3 belong to group s1.

c3 and c4 belong to group s2.

c4 is a quadratic.

s1 contains linear constraints.

s2 contains nonlinear constraints.

P1, P2, and P3 can satisfy linear constraints.

P4 can satisfy nonlinear constraints.

P5 can satisfy quadratic constraints.

Figure 7.1: CGCP graph

Figure 7.2: Modified MP state transition diagram

new problem set and if all constraints are satisfied or an MP-rejectable constraint is detected, it returns YES or NO using transitions 14 or 15. If one or more constraints can be satisfied using the method of isolation, it makes transition 24 and possibly 3 (one or more times). Depending on the results, $MP(k)$ may return YES(S) or NO using transition 25 or 26.

If $MP(k)$ does not return YES, YES(S), or NO using one of the above transitions, it makes transition 2 or 4 to state 8. In so doing, it updates the CGCP graph by adding a new node for each constraint, and adding edges as indicated by any accompanying metalevel data.

Once $MP(k)$ reaches state 8, it searches the graph for a path from a constraint node to a procedure node (to identify some procedure P_i which should be sent new data). Since as much data as possible should be sent in each CM, $MP(k)$ searches the graph for any other data that belongs to the same group, or in case the original path did not pass through a group node, any other constraints that have been directly classified and linked by paths to the same procedure (i.e., constraints which belong to the global group). $MP(k)$ then sends a CM to P_i .

If P_i returns MAYBE or YES, $MP(k)$ makes transition 30 or 31 and continues the search by looking for another set of (grouped) constraints. If all paths leading to P_i have been used, the search continues with the next procedure and so on. Eventually, a procedure will either return a response other than MAYBE or YES, or all paths will be exhausted.

If a procedure returns NO in response to a CM, $MP(k)$ makes transition 36 and in so doing it:

- discards any changes made to its data structures since the last CM was received.
- sends BACKTRACK messages to any procedures it has called (one for each CM that it sent), and
- returns NO to its caller.

If a procedure returns MAYBE(S), YES+MAYBE(S), or YES(S), $MP(k)$ makes transition 32, 33, 34, or 35, depending on the reply, and it returns a list of substi-

tutions to its caller. It then waits for another message. If it receives another CM, it makes transition 37 and the above process repeats; if it receives an AVR, it makes transition 38 and sends an AVR to the last procedure to return a list of substitutions. Depending on its reply, another list of substitutions may be returned or $MP(k)$ will return to state 4 in order to continue the search for paths from constraint to procedure nodes.

It is possible that $MP(k)$ will exhaust the data contained in the CGCP graph before some procedure returns NO (either in response to a CM or an AVR). If so, the condition on transition 29 will become false and that on 39 will become true. This will cause $MP(k)$ to continue with the same search mechanism that it used in Chapter 4, but it will use only those constraints that have no metalevel data.

It may seem unnecessary to make transition 39 and to use the original search mechanism if every constraint that was affected by the data in a CM had metalevel data associated with it. However, consider the following scenario. Assume that at some previous level of recursion, say $k-2$, some procedure $P_i(k-2)$ generated some substitutions which affected a constraint c that had no metalevel data as well as some others that did. $P_i(k-2)$ would make a recursive call and $MP(k-1)$ would use the values to reduce every constraint that it could, including c . If no contradictions arose and some constraints remained unsatisfied, then $MP(k-1)$ would make transition 29 and send the constraints with metalevel data back to the procedures. If some procedure $P_j(k-1)$ generated more substitutions, it would send them to $MP(k)$ using a recursive call. However, even if it were to send substitutions that affected constraints with metalevel data only (as originally assumed), then $MP(k)$ would still need to use its original search mechanism on the remaining constraints (i.e., those without metalevel data) because the simplified form of c would not have been sent to the procedures.

7.4.2 Modifying Procedures for Metalevel Data

In order to make use of the metalevel data, procedure's require minimal modifications. Each procedure's problem stack is replaced with a list of stacks, one for

each group. Whenever a procedure P_i receives a CM M_i , the message contains the name of the group to which the data belongs, or no group name to indicate that the data belongs to the global group. P_i processes the data in L_{M_i} in the same manner that was described in Chapter 4 but whenever it makes a transition that involves its problem set, it accesses the problem set corresponding to the group name in the CM. For instance, if P_i receives a CM M_i , updates one of its stacks, sends a CM to some child P_j and P_j returns NO, then P_i only pops the stack that was affected by the data in L_{M_i} .

7.5 Effects on Solver Properties

If no metalevel data is provided with the constraints in a program, the solver retains the properties described in Chapter 6. This is apparent from the following observations:

- If MP can never make transition 29 in Figure 7.2, it will move to state 4 using transition 39 whenever it reaches state 8. Since the remaining transitions are the same as those in the original state diagram, MP will behave as described in Chapter 4.
- As described in the previous section, procedures other than MP make use of metalevel data by adding data which is tagged with group names to corresponding problem stacks. If no such data is ever received and other constraints are added to default stacks for ungrouped data, procedures will behave as described in Chapter 4.

If metalevel data is used, however, several of the solver's properties could be undermined if the data prevents the solver from exploring search paths which could lead to useful responses. Most of the properties which are affected are those which ensure that the solver will return various responses if certain conditions are met. This includes Properties C3.2, C4.2, C5.1, C5.2, C6.2, C6.3, and C7.3 because they all rely on some procedure being passed a set of constraints when it has the capability of generating substitutions or determining that constraints are nonfeasible. If such

a procedure is not sent a CM because of metalevel data, there is no guarantee that some other procedure will be able to produce the same results.

Other affected properties include Properties A2, A3.1, A3.2, B1, and B2. Property A2 (which ensures that procedures will be sent data that they can recognize) could be undermined because a procedure may not be sent a CM; Properties A3.1 and A3.2 (which indicate substitutions will be returned based on the top-to-bottom, left-to-right ordering of procedures) may no longer hold because ancestors and siblings of a procedure may not be sent CM's; Properties B1 and B2 (both of which make deductions regarding procedure capabilities after the solver returns MAYBE or YES) may no longer hold because both are based on the assumption that the solver explores all search paths before replying.

The use of metalevel data will not, in general, affect properties which prove that various results will be returned only if certain conditions are met. This includes Properties B3, B4, B5, B6.1, and B6.2. These properties still hold because they are based on the results of the most recently explored search path; that is, the results do not depend on what search paths the solver might not have been able to explore either before or after the most recent path. For instance, if the solver returns YES(*S*) after being sent a CM, then the solver must have been able to satisfy all constraints, or if the solver returns NO, then MP or some other procedure must have determined that some constraints were nonfeasible.

Other properties not affected by the use of metalevel data include the following:

- Properties A1.1 and A1.2 (the termination properties) because the use of metalevel data can only reduce the number of search paths to be explored,
- Properties A4.1 to A4.4 (the backtracking properties),
- Properties C1.1, C2.3, C3.1, C3.3, C6.1, C7.1, and C7.2 because they are concerned with the behaviour of MP only,
- Properties C1.2 and C1.3 (which indicate that certain responses will not be returned given certain conditions) because metalevel data will not result in a procedure determining that feasible constraints are nonfeasible and vice versa,

- Property C2.2 (which indicates a set of conditions that will cause MAYBE to be returned) because metalevel data will not enable procedures to recognize constraints that they could not recognize before, and
- Properties C2.1 and C4.1 (which indicate sets of conditions which will cause MAYBE and MAYBE(*S*) to be returned) because metalevel data will not enable a procedure to determine that constraints are feasible or nonfeasible if it could not do so already.

It may be concluded that the use of inappropriate metalevel data could undermine the solver's abilities to generate useful responses but it would not result in the return of misleading results. However, if metalevel data does not prevent the solver from returning all of the unique solutions that it is capable of generating, then the solver properties will still hold.

Chapter 8

Examples

This chapter demonstrates (1) the creation of a typical recognition rule and sufficiency predicate, and (2) the high-level requirements for a satisfaction function (the lower-level details being dependent on the data structures used in the host language). As well, the satisfaction process is demonstrated using two examples. In Example 1, all possible solutions to a set of nonlinear equations are generated with and without the use of metalevel data. In Example 2, a set of recursively generated equations and an inequality involving integrals and a trigonometric term are processed; however, because of an instance of satisfaction failure, the constraints cannot be satisfied. It is shown that under such circumstances, the use of metalevel data still offers performance improvements.

The examples make use of Figures 8.1 to 8.23 in which some liberties have been taken with the labelling of CM arguments in order to minimize diagram clutter. Constraint arguments are not enclosed in square brackets (common list notation) even though they are passed in a list, and parameter *A* (the second CM argument) has been omitted altogether. As well, CM's may appear without any arguments when CM's are sent between procedures at the same level of recursion (in which case the CM's must contain the constraints in the calling procedure's problem set). However, all examples are based on the description of CM's as they are specified in Chapter 4.

8.1 Creating Procedure Components

Recognition rules and sufficiency predicates are defined as Prolog procedures. Each clause in the procedure is defined in terms of system predicates and predicates which are defined by the person creating the procedure. In the following sections, it is assumed that the following system predicates are available:

- *var/1* which succeeds only if its single argument is not bound,
- *number/1* which succeeds only if its single argument is bound to a number,
- *constraint_var/2* which succeeds only if its first argument is an unbound variable in its second argument (a constraint), and
- *set_of/3* which succeeds if its third argument is the set of instances of its first argument for which the second argument is true.

The latter is used with *constraint_var* in order to find the set of variables appearing in a constraint. Although the use of *set_of* is consistent with its use in [37], it must be capable of handling variables at the metalevel, that is, it must consider (1) variables which can be bound to different values as separate unknowns, and (2) variables that are bound to the same value (regardless of whether or not it has been determined) as a single variable.

8.1.1 Recognition Rules

A recognition rule consists of the clauses defining the predicate *recognized/1*. By definition, *recognized* takes a single input argument which is always bound to a constraint that has been sent to the procedure. Thus, a constraint *c* will be recognized only if *recognized(c)* can be proven true.

A typical recognition rule is as follows:

```
recognized( C ) :- lin_expression( C )
```

```
lin_expression( S1=S2 ) :- lin_sequence( S1 ), lin_sequence( S2 ).
```



```

lin_sequence( T )      :- lin_term( T ).
lin_sequence( T1+T2 ) :- lin_sequence( T1 ), lin_sequence( T2 ).
lin_sequence( T1-T2 ) :- lin_sequence( T1 ), lin_sequence( T2 ).

```

```

lin_term( A )      :- number( A ).
lin_term( A )      :- var( A ).
lin_term( A*B )    :- number( A ), var( B ).
lin_term( A*B )    :- var( A ), number( B ).

```

Only one clause is used to define *recognized* and it has a single antecedent, *lin_expression/1*. Thus, *recognized(C)* will succeed only if *lin_expression(C)* succeeds. This requires *C* to contain an equal sign and the expressions on either side to be linear sequences as defined by *lin_sequence/1*. A subgoal *lin_sequence(S)* requires *S* to be either a linear term as defined by *lin_term/1* or the sum or difference of two linear subsequences. A subgoal *lin_term(T)* will succeed only if its input argument is either a constant, a variable, or a product of the two.

Note that the above rule can be made to recognize linear inequalities by simply adding more *lin_expression* clauses with *=* replaced by *>*, *<*, etc.

8.1.2 Sufficiency Predicate

A procedure's sufficiency predicate consists of the clauses defining the predicate *sufficient/1*. By definition, *sufficient* has the procedure's problem set as its input argument, i.e., a list of constraints.

A typical sufficiency predicate is as follows:

```

sufficient( P ) :- distinct_vars( P, [], V ),
                  list_length( V, X ),
                  list_length( P, Y ),
                  X = Y.

```

```

distinct_vars( [], V, V ).
distinct_vars( [C|L], V0, V ) :- set_of( V, constraint_var( V, C ), V1 )
                                append( V0, V1, V2 ),
                                no_doubles( V2, Vunique ),
                                distinct_vars( L, Vunique, V ).

```

The predicate *sufficient* is defined in terms of *distinct_vars/3* and *list_length/2*, the latter being a common system predicate.¹ *Distinct_vars* makes use of system predicates *set_of/3* and *constraint_var/2*, plus *append/3* and *no_doubles/2*, common user-defined predicates which may be found in [37].

The sufficiency predicate succeeds only if the number of distinct variables in the constraints is equal to the number of constraints in the problem set. *Distinct_vars* generates a list *V1* of unique variables in constraint *C* by using *set_of* which in turn uses *constraint_var*. *Append* is used to create list *V2*, the result of appending *V1* to list *V0* (which contains the variables from constraints which *distinct_vars* identified in previous recursions), and then *no_doubles* is used to eliminate any duplicates. Finally, *distinct_vars* recursively tests the constraints remaining in list *L*.

Once *distinct_vars* has exhausted the list and succeeded, the next subgoals in *sufficient* are to determine the lengths of lists *P* and *V* (using *list_length/2*) and to see if the lengths are equal.

8.1.3 Function Calls

A user creates the functions for a procedure by defining them in a programming language with calling conventions that are consistent with those of the system. For the sake of argument, assume that the C programming language is used [27] and that variables of type *ConstraintPtr* point to a constraint. Functions must be able to accept a list of pointers to the constraints in a procedure's problem set and so the following data structure may be used:

¹The definition of *length/2*, a predicate equivalent to *list_length/2*, may be found in [37]

```
struct ConstraintNode {  
    ConstraintPtr c;  
    struct ConstraintNode *next;  
};
```

Note that the structure contains a pointer *c* to a constraint and a pointer *next* to subsequent structures.

In order to return results, functions must be able to return a list of assignment constraints that is qualified with either MAYBE, YES, or NO. To do so, the following data structure could be used:

```
struct ResultsNode {  
    union {  
        int qualifier;  
        ConstraintPtr c;  
    } results;  
    struct ResultsNode *next;  
};
```

Note that the structure's *results* field is a union which can contain *qualifier*, an integer whose value indicates MAYBE, YES, or NO, and *c*, a pointer to a constraint. Such an arrangement allows the first node in the list to return NO or qualify constraints in subsequent nodes with YES or MAYBE.

Once the functions have been defined, the system must have some means of calling them. It would not know their names beforehand and so the following may be used to create a list of pointers to the functions:

```
struct FunctionNode {  
    struct ResultsNode *(*f)(ConstraintNode *, int );  
    struct FunctionNode *next;  
};
```

The first structure field is declared to be a pointer to a function which is passed a pointer to a *ConstraintNode* and an integer (to be described), and the function

returns a pointer to a `ResultsNode`. The second structure field contains *next*, a pointer to a subsequent `FunctionNode` structure, and so a list of `FunctionNode`'s can be used to store a series of satisfaction function calls.

Given the above data structures, satisfaction functions are defined as follows:

```
struct ResultsNode *f1(struct ConstraintNode *nextc, int A)
{
    struct ResultsNode *r;
    Satisfaction function code
    return r;
}
```

As described above, the function is passed a pointer to a list of pointers to constraints and an integer *A*, the latter containing the value of the fifth field in the procedure's data stack (which indicates whether or not the function is being passed every available constraint). The function returns a pointer *r* to a list of one or more `ResultsNode`'s.

Since a user may wish to call the same function more than once but with different input arguments, using an intermediate level of function calls may be useful. Consider the following code segment:

```
struct ResultsNode *f1(struct ConstraintNode *nextc, int A)
{
    return g(nextc, A, 0);
}

struct ResultsNode *f2(struct ConstraintNode *nextc, int A)
{
    return g(nextc, A, 1);
}
```

```

struct ResultsNode *g(struct ConstraintNode *nextc, int A, int i)
{
    struct ResultsNode *r;
    Satisfaction function code
    return r;
}

```

Note that if the constraint solver accessed a list of FunctionNode's containing pointers to $f1$ and $f2$, g could be called twice. Since $f1$ and $f2$ pass different values for i to g , it is possible to keep track of how many times g has been called and to control its behaviour accordingly.

8.2 Example 1

The first example consists of the following four equations:

$$\begin{aligned}
 c1: W \times (X + Y) &= 2 \\
 c2: W + X + Y + Z &= 4 \\
 c3: W^2 &= 1 \\
 c4: 4 \times W - Y + Z &= 4
 \end{aligned}$$

Although the number of unknowns and equations is equal, there are two sets of solutions because $c3$ is a quadratic.

The procedures to be used are described in Table 8.1 and depicted graphically in Fig. 8.1(a). Note that the procedures have been labelled so that the types of constraints they can satisfy are apparent.

8.2.1 Solving without Metalevel Data

Constraints $c1 - c4$ could be incorporated into a Prolog procedure as shown below:

Available Procedures	
Name	Capability
Nonlin	Has one function which can solve sets of nonlinear equations when the number of variables equals the number of equations.
Poly	Solves polynomial equations in one unknown.
Quad	Has two functions for returning the two solutions to a quadratic, and a third which always returns NO (to indicate that there are no solutions besides the ones returned by the first two functions).
Lin	Solves linear equations and inequalities. It generates values only when a variable has a unique solution.
GElim	Solves sets of linear equations when the number of variables equals the number of equations.
Tri	Has one function which can solve sets of tridiagonal equations when the number of variables equals the number of equations, and a second which always returns NO (to indicate that there are no solutions besides the ones returned by the first function).

Table 8.1: Procedures for Example 1

(a)

(b)

(c)

(d)

Figure 8.1: Example 1

$$\begin{aligned}
 a(W, X, Y, Z) :- W * (X + Y) &= 2, \\
 W + X + Y + Z &= 4, \\
 W^2 &= 1, \\
 4*W - Y + Z &= 4.
 \end{aligned}$$

The constraints are not labelled because no metalevel data is provided; however, they will still be referred to as *c1* - *c4*.

If the query $a(W, X, Y, Z)?$ is made, the following events take place:

(1) A CM containing *c1* is sent to $MP(0)$ (Fig. 8.1(b)). $MP(0)$ adds variables X, Y, and W to VV and adds a new stack to SL. Since *c1* contains multiple unknowns, $MP(0)$ cannot generate any values and so it makes transition 2 to state 4. Since no metalevel data is available, $MP(0)$ makes transition 39 to state 6 and it calls its first child.

(2) Nonlin recognizes *c1* and so it adds it to its problem set. Its only child Poly cannot recognize *c1* and so Poly returns MAYBE. Nonlin makes transition 5 and tests its sufficiency predicate. The predicate fails because its problem set contains more unknowns than equations and so Nonlin returns MAYBE to $MP(0)$. $MP(0)$ continues the search by sending a CM to Lin but Lin cannot recognize *c1*. Thus, Lin returns MAYBE and since all search paths have been explored, $MP(0)$ makes transition 5 and returns MAYBE to PE.

(3) When $MP(0)$ receives *c2*, Z is added to VV and another stack is added to SL. $MP(0)$ cannot satisfy *c2* and so it makes transitions 2 and 39; it then sends a CM to Nonlin (Fig. 8.1(c)). Nonlin recognizes *c2* and so it sends a CM to Poly. Poly cannot recognize *c2* and so it returns MAYBE. Nonlin test its sufficiency predicate but the predicate fails and so Nonlin returns MAYBE. $MP(0)$ sends a CM to Lin and Lin recognizes *c2*. This causes it to send a CM to Gelim and Gelim recognizes *c2*. Gelim sends a CM to Tri but Tri cannot recognize *c2* because it contains too many unknowns. Tri returns MAYBE and then Gelim tests its sufficiency predicate. The predicate fails because there are more variables than equations in its problem set and so Gelim returns MAYBE. Lin tests its sufficiency predicate but the predicate

fails for the same reasons. Thus, Lin returns MAYBE and $MP(0)$ returns the same to PE.

(4) When $MP(0)$ receives $c3$, no new variables are added to VV but a third stack is added to SL. $MP(0)$ sends a CM to Nonlin and Nonlin adds $c3$ to its problem set (Fig. 8.1(d)). Similarly, Nonlin sends a CM to Poly and Poly adds $c3$ to its empty problem set. Poly sends a CM to Quad and Quad adds $c3$ to its problem set. Since Quad has no children, it tests its sufficiency predicate and the predicate succeeds. It then calls its first satisfaction function and the solution $W=1$ is returned. Quad makes transition 26 and sends a CM to $MP(1)$ (Fig. 8.2(a)).

(5) $MP(1)$ substitutes the value for W into constraints $c1$, $c2$, and $c3$, reducing them to:

$$\begin{array}{ll} c1': 1 \times (X + Y) & = 2 \\ c2': 1 + X + Y + Z & = 4 \\ c3': 1 & = 1 \end{array}$$

Since $c1'$ and $c2'$ are not satisfied, $MP(1)$ sends a CM to Nonlin. Nonlin updates its problem set with all three and sends a CM to Poly. Poly replaces $c3$ with $c3'$, and so does Quad. None of these procedures can generate any substitutions and so all return MAYBE to their caller. $MP(1)$ then sends a CM to Lin and Lin updates its problem set. Similarly, Gelim and Tri update their problem sets. Again, all return MAYBE because no values can be determined and so $MP(1)$ returns $MAYBE([W=1])$ to Quad. The solution is then forwarded back to $MP(0)$ and to PE (Fig. 8.2(b)).

(6) After receiving $c4$, $MP(0)$ sends a CM to Nonlin (Fig. 8.2(c)). Nonlin sends a CM to Poly but Poly cannot recognize anything new in Nonlin's problem set. Nonlin tests its sufficiency predicate and the predicate succeeds, causing Nonlin to call its first satisfaction function. The substitutions $X=1$, $Y=1$, and $Z=1$ are generated and so Nonlin sends a CM to $MP(1)$. $MP(1)$ uses the substitutions to totally reduce and satisfy the remaining constraints and so it returns $YES(S)$ using transition 25. This causes Nonlin to make transition 33 and in so doing, it sets V to 1 and returns $YES(S)$ to $MP(0)$. $MP(0)$ in turn makes transition 13 and returns $YES(S)$ to PE.

Figure 8.2: Example 1 (continued)

(7) If PE backtracks into c_4 , it sends an AVR to $MP(0)$ and $MP(0)$ sends one to Nonlin (Fig. 8.2(d)). In turn, Nonlin sends an AVR to $MP(1)$. $MP(1)$ makes transition 16 and returns NO and so Nonlin tries to call another satisfaction function. Since it has none, it returns YES to $MP(0)$ using transition 19. $MP(0)$ continues the search by sending a CM to Lin and Lin adds c_4 to its problem set (Fig. 8.3(a)). It then sends a CM to Gelim and Gelim sends a CM to Tri. Tri recognizes c_4 and so its sufficiency predicate succeeds when tested. Tri calls its first function and the function returns the same values that Nonlin produced. Tri sends a CM to $MP(1)$ and $MP(1)$ returns YES(S) after making transition 25. The results are returned to Tri and so Tri make transition 33. Both Gelim and Lin make transition 11 and so the results are forwarded back to $MP(0)$ and PE.

(8) If PE backtracks into c_4 again, it sends another AVR to $MP(0)$ and $MP(0)$ sends an AVR to Lin (Fig. 8.3(b)). The AVR is forwarded to Tri and Tri sends an AVR to $MP(1)$. $MP(1)$ returns NO using transition 16 and so Tri calls its next function. The function returns NO and so Tri makes transition 29 and returns NO to Gelim. Gelim and Lin both make transition 12 and so NO is returned to $MP(0)$. This results in $MP(0)$ making transition 21 and in so doing, it sends a BACKTRACK message to Nonlin. Nonlin does likewise and so any procedure which had recognized c_4 backtracks over the data. Thus, all traces of c_4 are removed from the solver.

(9) If PE backtracks into c_3 , it sends another AVR to $MP(0)$ (Fig. 8.3(c)). This time, $MP(0)$ sends an AVR to Nonlin, Nonlin sends one to Poly, and Poly sends one to Quad. Quad sends an AVR to $MP(1)$ and $MP(1)$ makes transition 20. In so doing, it:

- sends BACKTRACK messages to its children (and the children do likewise),
- discards c_1 , c_2 , and c_3 , and
- returns ABANDON.

(10) The ABANDON response causes Quad to call its next satisfaction function and so the substitution $W=-1$ is returned (Fig. 8.3(d)). Quad makes a recursive call

(a)

(b)

(c)

(d)

Figure 8.3: Example 1 (continued)

and $MP(1)$ uses the value to reduce $c1$, $c2$, and $c3$ to the following:

$$c1': -1 \times (X + Y) = 2$$

$$c2': -1 + X + Y + Z = 4$$

$$c3': 1 = 1$$

As before, $c1'$ and $c2'$ are not satisfied and so $MP(1)$ sends a CM to Nonlin. Nonlin, Poly, and Quad update their problem sets and all return MAYBE. $MP(1)$ sends a CM to Lin and Lin, Gelim, and Tri update their problem sets. Again, all return MAYBE and so $MP(1)$ returns $MAYBE([W=-1])$ to Quad. Quad returns the same to Poly and so the solution is forwarded back to $MP(0)$ and PE (Fig. 8.4(a)).

(11) When PE encounters $c4$ again, the description of step (6) applies once more except the values for X , Y , and Z have changed (Fig. 8.4(b)). Nonlin is sent a CM and it generates the substitutions $X=-1$, $Y=-1$, and $Z=7$. It sends a CM to $MP(1)$ and $MP(1)$ uses the the values to totally reduce and satisfy its problem set. As before, the solutions are forwarded back to $MP(0)$ and PE.

(12) If PE backtracks into $c4$, the description of step (7) applies again and so an AVR is sent to Nonlin (Fig. 8.4(c)). Nonlin sends an AVR to $MP(1)$, $MP(1)$ returns NO, and Nonlin returns YES. $MP(0)$ sends a CM to Lin and eventually Tri calls its first function (Fig. 8.4(d)). It generates the same values that Nonlin just produced and $MP(1)$ uses them to satisfy its problem set. $MP(1)$ returns $YES(S)$ to Tri and the results are forwarded back to PE.

(13) As in step (8), if PE sends an AVR regarding $c4$ to $MP(0)$, the AVR is forwarded to Tri and to $MP(1)$ (Fig. 8.5(a)). $MP(1)$ returns NO and so Tri calls its next function. The function returns NO and so NO is returned to $MP(0)$. $MP(0)$ sends a BACKTRACK message to Nonlin and so all traces of $c4$ are removed from the solver.

(14) If PE backtracks into $c3$ a second time, another AVR is sent to Quad and $MP(1)$ returns ABANDON again (Fig. 8.5(b)). The ABANDON response causes Quad to call its third satisfaction function but this time NO is produced. This

(a)

(b)

(c)

(d)

Figure 8.4: Example 1 (continued)

Figure 8.5: Example 1 (continued)

causes Quad to make transition 29 and so NO is returned to MP(0). This results in all traces of $c3$ being removed from the solver.

(15) If PE backtracks into $c2$, an AVR is sent to MP(0) and MP(0) makes transition 16. It sends BACKTRACK messages to its children and its children do likewise (Fig. 8.5(c)). Thus, all traces of $c2$ are removed from the solver. If PE backtracks into $c1$, the above steps repeat (Fig. 8.5(d)) and so the status of the solver is restored to what it was before it was sent any CM's.

8.2.2 Solving with Metalevel Data

The equations of the previous section will now be satisfied with the help of metalevel data. The data arises from the observations that:

1. $c3$ is a quadratic in W , and
2. given a value for W , $c1$, $c2$, and $c4$ become a tridiagonal set of equations.

Thus, metalevel data is required which would (1) allow $c3$ to be solved as a quadratic, and (2) prevent $c1$ and $c2$ from being sent to the solver before $c3$. Thus, the Prolog procedure of Example 1 is modified as follows:

```
unitgroup: s1 #[ isa( s1, tridiagonal ) ].
```

```
a( W, X, Y, Z ) :- c1: W*(X + Y)      = 2 #[ belongs_to( c1, s1 ) ],
                  c2: W + X + Y + Z = 4 #[ belongs_to( c2, s1 ) ],
                  c3: W^2              = 1 #[ isa( c3, quadratic ) ],
                  c4: 4*W - Y + Z      = 4 #[ last_in_group( c4, s1 ) ].
```

Note that a unit group has been declared and constraints $c1$, $c2$, and $c4$ have been associated with it. As well, $c3$ has been declared to be a quadratic.

When the query $a(W, X, Y, Z)?$ is made, the following events take place:

(1) When PE encounters $c1$ and $c2$, it does not send them to the solver because of their association with the unit group; however, it does send $c3$ (Fig. 8.6(a)). Since

$MP(0)$ cannot satisfy $c3$, it makes transition 2 and updates the CGCP graph using the metalevel data provided. After identifying the path between the constraint and procedure nodes, $MP(0)$ makes transition 29 and sends $c3$ in a CM to Quad. Quad updates its problem set and since it has no children, it tests its sufficiency predicate. The predicate succeeds and so Quad makes its first function call. The function returns $YES([W=1])$ and so Quad sends a CM to $MP(1)$ (Fig. 8.6(b)). $MP(1)$ uses the value to totally reduce $c3$ and then it returns $YES([S])$ to Quad using transition 25. Quad makes transition 33 and in so doing returns the same result to $MP(0)$. Finally, $MP(0)$ returns $YES(S)$ to PE.

(2) When PE encounters $c4$, it detects that it is the last constraint in group $s1$ and so it sends all constraints in the group to the solver (Fig. 8.7(a)). Note that the substitution $W=1$ has already been made in $c1$ and $c2$. Since $MP(0)$ cannot satisfy any of the constraints, it makes transition 2 and updates the CGCP graph. It adds three new constraint nodes and three paths to group $s1$. $MP(0)$ detects the paths to procedure Tri and so it sends Tri a CM using transition 29. Tri adds all three constraints to a new problem stack created for group $s1$ and then it tests its sufficiency predicate. The predicate succeeds and so Tri makes its first function call. The function returns $MAYBE([X=1, Y=1, Z=1])$ causing Tri to send a CM to $MP(1)$. $MP(1)$ uses the values to satisfy the outstanding constraints in its problem set and then it returns $YES(S)$ using transition 25. Tri returns the same to its caller ($MP(0)$) and $MP(0)$ returns $YES(S)$ to PE using transition 35.

(3) If PE backtracks into $c4$, it sends an AVR to $MP(0)$ (Fig. 8.7(b)). $MP(0)$ makes transition 38 and sends an AVR to Tri, and Tri sends the same to $MP(1)$. $MP(1)$ makes transition 16 and returns NO, causing Tri to call its next function. The function returns NO and so Tri makes transition 29 and returns NO to $MP(0)$. $MP(0)$ makes transition 36 and in so doing, it discards all traces of $c1$, $c2$, and $c4$ and returns NO to PE. As a result, PE backtracks over $c4$.

(4) When PE backtracks into $c3$, it sends an AVR to $MP(0)$ and $MP(0)$ sends the same to Quad using transition 22 (Fig. 8.8(a)). Quad sends an AVR to $MP(1)$ and

(a)

(b)

Figure 8.6: Example 1 with metalevel data

MP(1) returns NO using transition 16. This causes Quad to call its next function, generate the assignment $W=-1$, and send a CM to MP(1) (Fig. 8.8(b)). As described in (1), MP(1) uses the value to totally reduce its problem set and so YES(S) is returned to Quad. Quad returns the same to MP(0) using transition 53 and so MP(0) returns YES(S) to PE.

(5) When PE encounters $c4$ the second time, step (2) repeats but with different values (Fig. 8.9(a)). PE sends all three constraints in the group to MP(0) and MP(0) updates the CGCP graph. MP(0) sends a CM to Tri and Tri generates the substitutions $X=-1$, $Y=-1$, and $Z=7$. These are sent to MP(1) and MP(1) returns YES(S). Tri returns the values to its caller and so YES(S) is returned to MP(0) and to PE.

(6) As described in (3), if PE backtracks into $c4$, it sends an AVR to MP(0). MP(0) sends an AVR to Tri. Tri sends an AVR to MP(1), and MP(1) returns NO using transition 16 (Fig. 8.9(b)). Tri calls its second function and the function returns NO. Tri makes transition 29, backtracks over group $s1$, and returns NO to MP(0). This causes MP(0) to make transition 36 and return NO to PE.

(7) When PE backtracks into $c3$ for the second time, it sends an AVR to MP(0) and MP(0) sends the same to Quad (Fig. 8.10). Quad sends an AVR to MP(1) and MP(1) returns NO. This causes Quad to call its third function but this time the function returns NO. Quad returns NO and so it and MP(0) backtrack over the data from the first CM. Thus, the status of all procedures is restored to what it was before any CM's were sent to the solver.

8.2.3 Performance Comparison

The use of metalevel data provided significant performance improvements in two ways. As seen in Table 8.2, it greatly reduced the number of messages required to generate all possible solutions to the constraints. Only 18% as many CM's were necessary, 40% as many AVR's, plus BACKTRACK messages were eliminated entirely.

(a)

(b)

Figure 8.8: Example 1 with metalevel data (continued)

(a)

(b)

Figure 8.9: Example 1 with metalevel data (continued)

Figure 8.10: Example 1 with metalevel data (continued)

Note that the table only includes messages sent between solver procedures, that is, those sent between PE and MP(0) have not been included.

Inter-Procedure Message Calls		
Message	Without Metalevel Data	With Metalevel Data
CM	39	7
AVR	20	8
BACKTRACK	24	0

Table 8.2: Comparison of the number of messages sent.

Table 8.3 indicates that the number of CM's and AVR's sent between PE and MP(0) have also been reduced, both by approximately one half. Thus, the use of metalevel data can reduce the number of messages sent between the solver's procedures as well as between PE and MP(0).

The other significant performance improvement was the elimination of all tentative solutions and redundant replies. As seen in Table 8.3, without the metalevel

Solver Reply Sequences			
Without Metalevel Data		With Metalevel Data	
PE Message	Solver Reply	PE Message	Solver Reply
CM(<i>c1</i>)	MAYBE	CM(<i>c3</i>)	YES([<i>W</i> =1])
CM(<i>c2</i>)	MAYBE	CM(<i>c1</i> , <i>c2</i> , <i>c4</i>)	YES([<i>X</i> =1,...])
CM(<i>c3</i>)	MAYBE([<i>W</i> =1])	AVR(<i>c1</i> , <i>c2</i> , <i>c4</i>)	NO
CM(<i>c4</i>)	YES([<i>X</i> =1,...])	AVR(<i>c3</i>)	YES([<i>W</i> =-1])
AVR(<i>c4</i>)	YES([<i>X</i> =1,...])	CM(<i>c1</i> , <i>c2</i> , <i>c4</i>)	YES([<i>X</i> =-1,...])
AVR(<i>c4</i>)	NO	AVR(<i>c1</i> , <i>c2</i> , <i>c4</i>)	NO
AVR(<i>c3</i>)	MAYBE([<i>W</i> =-1])	AVR(<i>c3</i>)	NO
CM(<i>c4</i>)	YES([<i>X</i> =-1,...])	-	-
AVR(<i>c4</i>)	YES([<i>X</i> =-1,...])	-	-
AVR(<i>c4</i>)	NO	-	-
AVR(<i>c3</i>)	NO	-	-
AVR(<i>c2</i>)	ABANDON	-	-
AVR(<i>c1</i>)	ABANDON	-	-

Table 8.3: Comparison of the solver's reply sequences.

data, the solver produced two MAYBE responses before returning the first value for W as a tentative solution. As well, the CM's and AVR's regarding $c4$ produced identical sets of values on two separate occasions (after each of the solutions for W was returned). Finally, the solver produced two ABANDON responses when it was sent AVR's regarding the first two constraints.

On the other hand, when the metalevel data was used, the solver returned nothing but definite solutions or NO replies, and it did not generate any duplicate solutions.

8.3 Example 2

The following equations and inequality are used:

$$c1 : X_1 + X_2 = 2$$

$$c2 : X_1 + 2X_2 + X_3 = 4$$

$$c3 : X_2 + 2X_3 + X_4 = 4$$

$$c4 : X_3 + 2X_4 + X_5 = 4$$

$$c5 : X_4 + 2X_5 + X_6 = 4$$

$$c6 : X_5 + 2X_6 + X_7 = 4$$

$$c7 : X_6 + X_7 = 2$$

$$d1 : \int_{X_1}^{X_2} Y^2 dY + \int_{X_1}^{X_2} Y^3 dY \geq \cos^{-1}(0.5)$$

Note that the first seven constraints form a set of tridiagonal equations which has a unique solution ($X_i = 1$ for $i=1, \dots, 7$), and the last constraint involves an inverse trigonometric function. Since an arccosine function has an infinite number of solutions, a procedure other than MP must generate values for it.

The procedures to be used are described in Table 8.4 and depicted in Fig. 8.11(a). Unfortunately, since no procedure can process an expression as complicated as $d1$, the solver would return MAYBE when sent the last constraint. This example demonstrates what would happen if a user modified $d1$ to better match the available procedures.

(a)

(b)

(c)

(d)

Figure 8.11: Example 2

Given that procedures *Int* and *Acos* can only recognize very simple constraints, the user might replace *d1* with the following expressions:

$$c8 : Y1 = \int_{X_1}^{X_2} Y^2 dY$$

$$c9 : Y2 = \int_{X_1}^{X_2} Y^3 dY$$

$$c10 : Y3 = \cos^{-1}(0.5)$$

$$c11 : Y1 + Y2 \geq Y3$$

Available Procedures	
Name	Capability
GElim	Solves sets of linear equations when the number of variables equals the number of equations.
Tri	Has one function which can solve sets of tridiagonal equations when the number of variables equals the number of equations, and a second which always returns NO (to indicate that there are no solutions other than the ones returned by the first function).
Int	Has one function which can satisfy constraints of the form $Z = \int_{X_1}^{X_2} Y^N dY$ where X_1 , X_2 , and N are given, and a second which always returns NO.
Acos	Has three functions for satisfying constraints of the form $Z = \cos^{-1}(X)$ where X is given. The functions generate the first three positive arccosine values.

Table 8.4: Procedures for Example 2

8.3.1 Solving without Metalevel Data

The above constraints may be generated with the following Prolog procedure:

```

a0( X1, X2 ) :- X1 - X2 = 2,
                a1( X1, X2, 2 ),
                Y1 = int( Y^2, X1, X2 ),
                Y2 = int( Y^3, X1, X2 ),
                Y3 = acos( 0.5 ),
                Y1 + Y2 >= Y3.

```

```

a1( X1, X2, N ) :- N < 7,
                  X1 + 2*X2 + X3 = 4.
                  N1 = N + 1,
                  a1( X2, X3, N1 ).
a1( X1, X2, N ) :- N = 7,
                  X1 + X2 = 2.

```

Note the function notation used to represent the integrals. The first argument specifies the expression to be integrated, and the second and third indicate the lower and upper integration limits, respectively. Similar notation is used to represent the arccosine term of constraint *c10*. Unfortunately, MP will not be able to determine that these constraints are satisfied using its substitute-and-reduce mechanism because it has no way of eliminating the function notation. Consequently, constraint substitutions are used as described in Appendix B. This will allow MP to tag the constraints as satisfied without having to reduce the expressions.

In the following description, the first clause of *a1* is used several times because of recursive calls, therefore variables in each invocation will be differentiated by a suffix. For instance, variable *X1* in the third invocation will be denoted *X1₃*.

If the query *a0(X1, X2)?* is made, the following events take place:

(1) PE sends constraint *c1* to MP(0) (Fig. 8.11(b)). Since MP(0) cannot satisfy it and no metalevel data was provided, MP(0) sends it to GElim using transition 7. GElim recognizes it and sends a CM to Tri using transition 8. Tri adds *c1* to its problem set and tests its sufficiency predicate. The predicate fails and so Tri returns MAYBE using transition 18. Similarly, GElim tests its sufficiency predicate and

returns MAYBE. $MP(0)$ makes transition 8 and sends CM's to its other children but none can recognize the constraint. Thus, all return MAYBE to $MP(0)$ and $MP(0)$ returns the same to PE using transition 5.

(2) PE encounters subgoal $a1$ and so it unifies 2 and variable N_1 in the first clause of $a1$, and variables $X1$ and $X2$ of $a0$ with $X1_1$ and $X2_1$ of $a1$, respectively. Since $X1$ and $X1_1$ cannot be bound to different values, PE substitutes $X1$ for $X1_1$ in any subsequent constraints. The same holds true for variables $X2$ and $X2_1$.

(3) PE sends the constraint $2 < 7$ to $MP(0)$ (Fig. 8.11(c)). $MP(0)$ determines that the constraint is satisfied but since its problem set contains an unsatisfied constraint, it calls its children. All return MAYBE because they cannot recognize any new data and so $MP(0)$ returns MAYBE to PE.

(4) PE encounters the equivalent of constraint $c2$ and sends it in a CM to $MP(0)$ (Fig. 8.11(d)). $MP(0)$ cannot satisfy it and so it makes transitions 2 and 39. It sends a CM to Gelim and as in the case of $c1$, Gelim and Tri both update their problem sets and return MAYBE. $MP(0)$ sends CM's to its other children but they too return MAYBE and so $MP(0)$ returns the same to PE using transition 5.

(5) PE encounters the constraint $N1_1 = N_1 + 1$ where N_1 has already been bound to the value 2. Thus, PE sends the constraint $N1_1 = 2 + 1$ to $MP(0)$ and $MP(0)$ assigns $N1_1$ the value 3 (Fig. 8.12(a)). Since its problem set contains unsatisfied constraints, $MP(0)$ calls its children and all return MAYBE. Consequently, $MP(0)$ returns $MAYBE([N1_1=3])$ using transition 27.

(6) The next subgoal encountered by PE is predicate $a1$ and so the first recursive call is made. N_2 is bound to the value 3, and $X2_1$ and $X3_1$ are unified with $X1_2$ and $X2_2$, respectively. Since $X2_1$ was unified with $X2$ of $a0$, PE substitutes $X2$ for $X1_2$ in any subsequent constraints. Similarly, PE substitutes $X3_1$ for $X2_2$.

(7) Steps (3) to (6) now repeat, each time the value of N_i increasing by one and the last two variables of constraint c_i being bound to the first two variables of c_{i+1} . This process repeats until N_6 is bound to the value 7 in which case $MP(0)$ returns

(a)

(b)

(c)

(d)

Figure 8.12: Example 2 (continued)

NO in response to the constraint $7 < 7$ (Fig. 8.12(b)). This causes PE to search for an alternate solution to $a1$ and so it uses the second clause. This results in PE sending the constraint $7 = 7$ to $MP(0)$ and as in step (3), $MP(0)$ returns MAYBE using transition 5 (Fig. 8.12(c)).

(8) The next subgoal encountered by PE is the equivalent of constraint $c7$ and so CM's are sent from PE to $MP(0)$, from $MP(0)$ to GElim, and from GElim to Tri (Fig. 8.12(d)). When Tri tests its sufficiency predicate, the predicate succeeds (because its problem set contains an equal number of constraints and variables) and so Tri makes its first function call. The function returns $YES(S)$ where S contains the substitutions $X1=1, X2=1, X3_1=1, \dots, X3_5=1$. Note that the latter five variables were created when the first clause of $a1$ was invoked five times.

(9) Tri makes a recursive call to $MP(1)$ and $MP(1)$ uses the values to totally reduce and satisfy all outstanding constraints. Note that those involving various versions of N are already satisfied. $MP(1)$ returns $YES(S)$ using transition 25 and so Tri, GElim, and $MP(0)$ return the same to their callers. Thus, subgoal $a1$ of $a0$ has been satisfied.

(10) PE sends constraint $c8$ to $MP(0)$ (Fig. 8.13(a)). $MP(0)$ sends CM's to its children but only Int recognizes $c8$. It calls a function and the function returns the solution $Y1 = 0$. A recursive call is made containing the value for $Y1$ and a constraint substitution for $c8$ (denoted $c8=*$ in Fig. 8.13(a)). This permits $MP(1)$ to determine that all constraints are satisfied and so it returns $YES([Y1=0])$. Int does likewise and so $MP(0)$ returns $YES(S)$ to PE.

(11) PE encounters constraint $c9$ and so the steps described in step (10) repeat with $c9$ and $Y2$ replacing $c8$ and $Y1$, respectively. Thus, $MP(0)$ returns $YES([Y2=0])$ to PE.

(12) PE sends constraint $c10$ to $MP(0)$ and $MP(0)$ in turn sends CM's to its children (Fig. 8.13(b)). Acos is the only procedure to recognize $c10$ and it calls its first function. The function returns the solution $Y3=1.047198$ and so Acos makes

(a)

(b)

(c)

(d)

Figure 8.13: Example 2 (continued)

a recursive call containing the value for $Y3$ and a constraint substitution for $c10$. This permits $MP(1)$ to determine that all constraints are satisfied and so it returns $YES(S)$. $Acos$ does likewise and so $MP(0)$ returns $YES(S)$ to PE . Note that both $Acos$ and $MP(0)$ set their V flags to 1.

(13) When PE encounters the last constraint, all variables are already bound to values and so the constraint $0 + 0 \geq 1.047198$ is sent to $MP(0)$ (Fig. 8.13(c)). $MP(0)$ returns NO using transition 14 and so PE backtracks into $c10$. AVR's are sent from PE to $MP(0)$, from $MP(0)$ to $Acos$, and from $Acos$ to $MP(1)$ (Fig. 8.13(d)). $MP(1)$ returns NO using transition 16 and so $Acos$ makes another function call. The function returns the substitution $Y3 = 7.330383$ (the previous solution plus 2π) and so $Acos$ sends $MP(1)$ a CM containing the value for $Y3$ and a constraint substitution for $c10$ (Fig. 8.14(a)). As described in step (12), this results in $MP(0)$ returning $YES(S)$ to PE .

(14) PE encounters the last constraint again and as described in step (13), $MP(0)$ returns NO and PE backtracks into $c10$ (Fig. 8.14(b)). This causes another AVR to be sent to the solver and $Acos$ to make another function call (Figs. 8.14(c)). Eventually, $YES([Y3=13.613568])$ is returned to PE (Fig. 8.14(d)).

(15) When PE sends $c11$ to the solver for the third time, $MP(0)$ returns NO as it did before and so PE backtracks into $c10$ again (Fig. 8.15(a)). PE sends a third AVR to the solver regarding $c10$ and the AVR is forwarded to $MP(1)$ (Fig. 8.15(b)). $MP(1)$ returns NO to $Acos$ and so $Acos$ tries to make another function call. However, $Acos$ has exhausted its list of satisfaction functions even though it has only generated three solutions of the infinite number possible. Since its V flag was set to 1 in step (12), $Acos$ returns YES using transition 19 and since $MP(0)$ has no search paths left to explore, it returns YES to PE using transition 6.

(16) When PE encounters constraint $c11$ again, no value is available for $Y3$ and so it sends the constraint $0 + 0 \geq Y3$ to $MP(0)$ (Fig. 8.15(c)). No procedure in the solver can generate any values that satisfy both $c10$ and $c11$ and so $MAYBE$ is

(a)

(b)

(c)

(d)

Figure 8.14: Example 2 (continued)

(a)

(b)

(c)

(d)

Figure 8.15: Example 2 (continued)

returned to PE. Note that this is the first time that the solver did not return NO when sent *c11*.

(17) When PE backtracks into *c11*, $MP(0)$ returns ABANDON using transition 20 and it sends BACKTRACK messages to all of its children (Fig. 8.15(d)). This removes all traces of *c11* from the solver. Similarly, when PE backtracks into *c10*, $MP(0)$ returns ABANDON and all traces of *c10* are discarded.

(18) When PE backtracks into *c9*, AVR's are sent from $MP(0)$ to Int and from Int to $MP(1)$ (Fig. 8.16(a)). $MP(1)$ returns NO using transition 16 and so Int calls its next function. The function returns NO because no solution is possible other than the one that was returned already and so Int returns NO using transition 29. Thus, $MP(0)$ returns NO to PE using transition 21.

(19) When PE backtracks into *c8*, the description of step (18) applies once more and so $MP(0)$ returns NO to PE. Similarly, if PE backtracks into *c7*, Tri returns NO after calling its second function and $MP(0)$ returns NO to PE (Fig. 8.16(b)).

(20) As PE continues to backtrack across constraints *c6* – *c1* and those involving *N* and *N1*, it sends more AVR's to the solver (Fig. 8.16(c)). Each time that it does, $MP(0)$ returns ABANDON using transition 20 and the solver discards all traces of another constraint. Note that $MP(0)$ will return ABANDON because it returned MAYBE or MAYBE(S) in response to every constraint which was sent before *c7* (as described in steps (1) to (7)).

It has just been shown that the solver was able to satisfy every constraint up to and including *c10*, but it returned NO or MAYBE every time that it was sent *c11*. This might lead us to believe that all eleven constraints might not be feasible but if we use the techniques described in § 5.2.1, we can determine otherwise.

Consider the sequence of messages and replies shown in Table 8.6 and $MP(0)$'s problem set at the time of each reply. When $MP(0)$ returned the first ABANDON reply (in response to the one and only AVR regarding *c11*), the only unsatisfied constraints in its problem set were $Y3 = \text{acos}(0.5)$ and $0 + 0 \geq Y3$. If we remember

(a)

(b)

(c)

Figure 8.16: Example 2 (continued)

that the former has an infinite number of positive and negative solutions, then we know that the constraints were feasible. Since the solver was unable to determine this, we can conclude that its limits have been exceeded.

We can determine the cause of the limitation by considering the reply sequences of Tables 8.5 and 8.6. By the alternating sequence of AVR's regarding *c10* and CM's containing *c11*, we can see that the solver generated values for *Y3* that were initially acceptable but led to a contradiction in *c11*. Eventually, the solver returned MAYBE in response to an AVR when it could not generate any more values for *Y3* and the next CM resulted in the feasible constraints (to which $MP(\theta)$ returned MAYBE). We may conclude that the solver failed because procedure *Acos* was unable to generate any of the infinite number of negative solutions and so the procedure needs to be modified.

8.3.2 Solving with Metalevel Data

The Prolog procedures of the previous section will now be modified with metalevel data. The data arises from the observations that:

- constraints *c1* – *c7* form a tridiagonal set and none can be satisfied until all are available, and
- constraints *c8* and *c9* can be satisfied only by procedure *Int*, and constraint *c10* can only be satisfied by *Acos*.

Thus, procedures *a0* and *a1* are modified as shown below:

```
unitgroup: s1 #[ isa( s1, tridiagonal ) ].
```

```
a0( X1, X2 ) :- c1: X1 - X2 = 2 #[ belongs_to( c1, s1 ) ],
               a1( X1, X2, 2 ),
               c8: Y1 = int( Y^2, X1, X2 ) #[ isa( c8, integral ) ],
               c9: Y2 = int( Y^3, X1, X2 ) #[ isa( c9, integral ) ],
               c10: Y3 = acos( 0.5 ), #[ isa( c10, trigonometric ) ],
               Y1 + Y2 >= Y3.
```

Solver Reply Sequences			
Without Metalevel Data		With Metalevel Data	
PE Message	Solver Reply	PE Message	Solver Reply
CM(<i>c1</i>)	MAYBE	CM(<i>c1</i> , ..., <i>c7</i>)	YES([<i>X1</i> =1, ...])
CM(<i>c2</i>)	MAYBE	CM(<i>c8</i>)	YES([<i>Y1</i> =0])
CM(<i>c3</i>)	MAYBE	CM(<i>c9</i>)	YES([<i>Y2</i> =0])
CM(<i>c4</i>)	MAYBE	CM(<i>c10</i>)	YES([<i>Y3</i> =1.0471])
CM(<i>c5</i>)	MAYBE	CM(<i>c11</i>)	NO
CM(<i>c6</i>)	MAYBE	AVR(<i>c10</i>)	YES([<i>Y3</i> =7.330])
CM(<i>c7</i>)	YES([<i>X1</i> =1, ...,])	CM(<i>c11</i>)	NO
CM(<i>c8</i>)	YES([<i>Y1</i> =0])	AVR(<i>c10</i>)	YES([<i>Y3</i> =13.614])
CM(<i>c9</i>)	YES([<i>Y2</i> =0])	CM(<i>c11</i>)	NO
CM(<i>c10</i>)	YES([<i>Y3</i> =1.0471])	AVR(<i>c10</i>)	MAYBE
CM(<i>c11</i>)	NO	CM(<i>c11</i>)	MAYBE
AVR(<i>c10</i>)	YES([<i>Y3</i> =7.330])	AVR(<i>c11</i>)	ABANDON
CM(<i>c11</i>)	NO	AVR(<i>c10</i>)	ABANDON
AVR(<i>c10</i>)	YES([<i>Y3</i> =13.614])	AVR(<i>c9</i>)	NO
CM(<i>c11</i>)	NO	AVR(<i>c8</i>)	NO
AVR(<i>c10</i>)	MAYBE	AVR(<i>c1</i> , ..., <i>c7</i>)	NO
CM(<i>c11</i>)	MAYBE	–	–

Table 8.5: Comparison of the solver's reply sequences (for constraints *c1* - *c11* only).

Solver Reply Sequences			
Without Metalevel Data		With Metalevel Data	
PE Message	Solver Reply	PE Message	Solver Reply
AVR(<i>c11</i>)	ABANDON	–	–
AVR(<i>c10</i>)	ABANDON	–	–
AVR(<i>c9</i>)	NO	–	–
AVR(<i>c8</i>)	NO	–	–
AVR(<i>c7</i>)	NO	–	–
AVR(<i>c6</i>)	ABANDON	–	–
AVR(<i>c5</i>)	ABANDON	–	–
AVR(<i>c4</i>)	ABANDON	–	–
AVR(<i>c3</i>)	ABANDON	–	–
AVR(<i>c2</i>)	ABANDON	–	–
AVR(<i>c1</i>)	ABANDON	–	–

Table 8.6: Comparison of the solver's reply sequences (for constraints *c1* – *c11* only).

```

a1( X1, X2, N ) :- N < 7,
                    c: X1 + 2*X2 + X3 = 4 #[ belongs_to( c, s1 ) ],
                    N1 = N + 1,
                    a1( X2, X3, N1 ).
a1( X1, X2, N ) :- N = 7,
                    c7: X1 + X2 = 2 #[ last_in_group( c7, s1 ) ].

```

Note that constraints $c1 - c7$ are now associated with a unit group $s1$, $c8$ and $c9$ are declared to be of the *integral* algebraic class, and $c10$ is declared to be of the *trigonometric* class. As well, the CGCP graph of Fig. 8.17 contains a node for group $s1$ and edges join the *integral* and *tridiagonal* nodes to the Int and Tri nodes, respectively.

If the query $a0(X1, X2)?$ is made, the following events take place:

(1) PE encounters constraint $c1$ but it does not send it to $MP(\theta)$ because of its association with unit group $s1$.

(2) PE encounters subgoal $a1$ and so it unifies 2 and variable N_1 in the first clause of $a1$, and variables $X1$ and $X2$ are unified with $X1_1$ and $X2_1$, respectively. PE sends the constraint $2 < 7$ to $MP(\theta)$ (Fig. 8.17(a)). $MP(\theta)$ determines that the constraint is satisfied and since there are no values to be determined, it returns YES using transition 15.

(3) PE encounters the equivalent of constraint $c2$ but it does not send it to $MP(\theta)$ because of its association with unit group $s1$.

(4) PE encounters the constraint $N1_1 = N_1 + 1$ where N_1 has already been bound to the value 2. Thus, PE sends the constraint $N1_1 = 2 + 1$ to $MP(\theta)$ (Fig. 8.17(b)). $MP(\theta)$ assigns $N1_1$ the value 3 and since every constraint in its problem set is satisfied, it returns $YES([N1_1=3])$.

(5) PE encounters subgoal $a1$ for the second time and so the first recursive call is made. In a manner similar to that described in step (2), variables $X2_1$ and $X3_1$ are

Figure 8.17: Example 2 with metalevel data

unified with $X1_2$ and $X2_2$, respectively, and N_2 is bound to the value 3. PE sends the constraint $3 < 7$ to $MP(0)$ and $MP(0)$ returns YES.

(6) Steps (3) to (5) now repeat, each time the value of N_i increasing by one and the last two variables of constraint c_i being unified with the first two variables of c_{i+1} . This process repeats until N_6 is bound to the value 7 and so $MP(0)$ returns NO in response to the constraint $7 < 7$ (Fig. 8.17(c)).

(7) PE searches for an alternate solution to $a1$ and so it uses the second clause. It sends the constraint $7 = 7$ to $MP(0)$ and since it is satisfied, $MP(0)$ returns YES using transition 15 (Fig. 8.18(a)).

(8) When PE encounters the *last_in_group* predicate with $c7$, it sends every constraint in group $s1$ to $MP(0)$ (Fig. 8.18(b)). $MP(0)$ cannot satisfy any of the constraints and so it makes transition 2 to state 4. Since metalevel data is available with the constraints, $MP(0)$ adds the appropriate edges to the CGCP graph and determines that procedure *Tri* should be sent all seven constraints. $MP(0)$ sends a CM to *Tri* and *Tri* updates its problem set with all seven constraints. It tests its sufficiency predicate and the predicate succeeds. *Tri* calls its first function and the function returns the solutions $X1 = 1, X2 = 1, X3_1 = 1, \dots, X3_5 = 1$. *Tri* makes a recursive call and $MP(1)$ uses the values to totally reduce and satisfy all outstanding constraints. Thus, $MP(1)$ returns $YES(S)$ to *Tri* and *Tri* returns the same to $MP(0)$. $MP(0)$ makes transition 35 and returns the same to PE.

(9) Since subgoal $a1$ of $a0$ has been satisfied, PE sends constraint $c8$ to $MP(0)$ (Fig. 8.18(c)). $MP(0)$ cannot satisfy it and so it updates the graph and finds the path from $c8$ to *Int*. $MP(0)$ sends a CM to *Int*, *Int* recognizes $c8$, and it tests its sufficiency predicate. The predicate succeeds and *Int* calls its first function. The function returns the solution $Y1 = 0$ and so *Int* makes a recursive call containing the value for $Y1$ and a constraint substitution for $c8$. $MP(1)$ determines that all constraints are satisfied and so it returns $YES([Y1=0])$ using transition 25. *Int* returns the same as does $MP(0)$ using transition 35.

Figure 8.18: Example 2 with metalevel data (continued)

(10) PE encounters $c9$ and the description of step (8) applies once more (with $c9$ and $Y2$ replacing $c8$ and $Y1$, respectively) (Fig. 8.19(a)). Thus, $\text{YES}([Y2=0])$ is returned to PE.

(11) PE sends constraint $c10$ to $\text{MP}(0)$ and $\text{MP}(0)$ uses the metalevel data to update the graph (Fig. 8.19(b)). This results in $\text{MP}(0)$ sending a CM to Acos and since Acos can satisfy $c10$, it calls its first function. The substitution $Y3 = 1.047198$ is returned and Acos makes a recursive call. $\text{MP}(1)$ determines that all constraints are satisfied and so it returns $\text{YES}(S)$. Acos does likewise and so $\text{MP}(0)$ returns $\text{YES}([Y3=1.047198])$ using transition 35.

(12) When PE encounters the last constraint, all variables are bound to values and so the constraint $0 + 0 \geq 1.047198$ is sent to $\text{MP}(0)$ (Fig. 8.19(c)). $\text{MP}(0)$ returns NO using transition 14 and so PE backtracks into $c10$. PE sends an AVR to $\text{MP}(0)$ and $\text{MP}(0)$ sends an AVR to Acos using transition 38 (Fig. 8.20(a)). Acos sends the same to $\text{MP}(1)$ and $\text{MP}(1)$ returns NO using transition 16. Acos makes another function call and the function returns the next substitution for $Y3$ (Fig. 8.20(b)). As described in step (11), this results in another recursive call and $\text{MP}(0)$ returning $\text{YES}([Y3=7.330383])$ to PE.

(13) PE encounters the last constraint again and as described in step (12), $\text{MP}(0)$ returns NO and PE backtracks into $c10$ (Fig. 8.20(c)). This causes another AVR to be sent to the solver and Acos to make another function call (Fig. 8.21(a)). Eventually, $\text{YES}([Y3=13.613568])$ is returned to PE (Fig. 8.21(b)).

(14) When PE sends $c11$ to the solver for the third time, $\text{MP}(0)$ returns NO as it did before and PE backtracks into $c10$ (Fig. 8.21(c)). PE sends a third AVR to the solver regarding $c10$ and the AVR is forwarded to $\text{MP}(1)$ (Fig. 8.22(a)). $\text{MP}(1)$ returns NO and Acos tries to make another function call. However, since Acos has exhausted its list of satisfaction functions, it returns YES using transition 19 (Fig. 8.22(b)). Since $\text{MP}(0)$ has no search paths left to explore, it returns YES to PE.

(a)

(b)

(c)

Figure 8.19: Example 2 with metalevel data (continued)

(a)

(b)

(c)

Figure 8.20: Example 2 with metalevel data (continued)

(a)

(b)

(c)

Figure 8.21: Example 2 with metalevel data (continued)

(a)

(b)

(c)

Figure 8.22: Example 2 with metalevel data (continued)

(15) When PE encounters constraint $c11$ again, no value is available for $Y3$ and so it sends the constraint $0 + 0 \geq Y3$ to $MP(0)$ (Fig. 8.22(c)). No procedure in the solver can generate any values that satisfy both $c10$ and $c11$ and so $MP(0)$ returns MAYBE using transition 6.

(16) When PE backtracks into $c11$, it sends an AVR to $MP(0)$. $MP(0)$ returns ABANDON using transition 20 and sends BACKTRACK messages to all of its children (Fig. 8.23(a)). This removes all traces of $c11$ from the solver. Similarly, when PE backtracks into $c10$, $MP(0)$ returns ABANDON and all traces of $c10$ are discarded.

(17) When PE backtracks into $c9$, it sends an AVR to the solver, $MP(0)$ makes transition 38, and an AVR is sent to Int (Fig. 8.23(b)). Int sends the same to $MP(1)$ and $MP(1)$ returns NO using transition 16. Int calls its next function but the function returns NO because the only solution was returned already. Thus, Int returns NO to $MP(0)$ using transition 29 and $MP(0)$ returns NO using transition 36.

(18) When PE backtracks into $c8$, the description of step (17) applies once more and so $MP(0)$ will return NO to PE. Similarly, if PE backtracks into the sending of constraints $c1 - c7$, an AVR will be sent to Tri and Tri will return NO after calling its second function. Thus, $MP(0)$ returns NO using transition 36 and the solver discards all traces of $c1 - c7$ (Fig. 8.23(c)).

(19) As PE continues to backtrack, it sends AVR's regarding constraints involving N and $N1$ (as described in steps (2) – (7)). Since each of these resulted in $MP(0)$ returning YES or YES(S) using transitions 15 or 25, $MP(0)$ returns NO using transition 16 in response to each AVR. Consequently, all traces of every constraint are eventually removed from the solver.

It has just been shown that the solver was unable to satisfy every constraint as was the case when no metalevel data was provided. However, this was to be expected because the metalevel data could not compensate for the underlying cause

Figure 8.23: Example 2 with metalevel data (continued)

of the solver's inability to satisfy the constraints, that is, the fact that Acos could not generate negative substitution values.

8.3.3 Performance Comparison

Even though the metalevel data did not result in the solver being able to satisfy the constraints, its use still provided significant performance improvements. As seen in Table 8.7, it greatly reduced the number of CM's and BACKTRACK messages which were sent between the solver's procedures. Only 38% as many CM's were necessary, 92% as many AVR's, and 19% as many BACKTRACK messages. Note that as in Example 1, only messages sent between MP(0) and other procedures have been included in the table.

Inter-Procedure Message Calls		
Message	Without Metalevel Data	With Metalevel Data
CM	42	16
AVR	13	12
BACKTRACK	32	6

Table 8.7: Comparison of the number of messages sent.

As well, the use of metalevel data reduced the number of CM's and AVR's which were sent to the solver by 43%, and the number of ABANDON replies which were returned by 75%. Since the latter indicates potentially lost solutions and may be used to help diagnose instances of satisfaction failure (as demonstrated at the end of § 8.3.1), it follows that the use of metalevel data may help to identify limitations of available procedures.

Chapter 9

Conclusions

9.1 A Research Summary

Current constraint programming languages are deficient in the areas of extensibility and expressiveness. While their built-in constraint satisfaction methods offer some relief from the problems of solving arithmetic relations in Prolog, their common “black-box” approach is overly restrictive.

An architecture for a general constraint satisfaction mechanism has been presented. It is extensible in that new satisfaction methods may be added to cover new classes of constraints. Specialized methods with overlapping capabilities can be incorporated and the solver’s built-in search mechanisms will identify and make use of the most efficient ones available. The hierarchical organization of procedures helps to identify appropriate satisfaction methods as quickly as possible, and at the same time, it helps to minimize the amount of data which is sent to procedures when there is no need to do so.

The new architecture retains the benefits of current constraint solvers in that its inner workings remain hidden from the users of the language in which it is embedded. As long as the necessary procedures are present, the solver will satisfy the constraints whenever it is possible to do so. Thus, the solver may be thought of as a “black box” at one level, yet it is still extensible.

The problems of how to organize the procedures into a hierarchy have been in-

vestigated. Algorithms have been presented which will create acceptable procedure configurations and the implications of different orderings have been investigated. Guidelines have been presented for organizing procedures when multiple configurations are possible.

Means of determining when the run-time abilities of the solver have been exceeded have also been identified. The possibilities of failure at the global and local levels were discussed. Recognizing instances of the former could indicate that new procedures might need to be added, while the latter could indicate that existing procedures might need to be modified.

Three types of metalevel data which could aid the satisfaction process have been presented. It was shown how the solver could make use of one type by the use of metalevel predicates in procedures' recognition rules, and the other two types by making minimal changes to the state transition diagrams of the solver's procedures.

9.2 Conclusions

The following conclusions have been reached:

- Constraint solvers need not be restricted to a single satisfaction mechanism. An extensible constraint solver is possible that can make use of an arbitrary number of satisfaction procedures.
- If an extensible solver is to be created (not just the one presented), the following must be provided:
 1. a means of selecting which constraints will be passed as inputs to the various methods,
 2. a means of collecting related constraints for simultaneous processing,
 3. a means of determining when satisfaction methods should be invoked, and
 4. a means of verifying that the values generated by a procedure to satisfy one set of constraints do not raise contradictions in any other set.

- A hierarchical organization of satisfaction procedures offers many benefits but it is not ideal. For instance, it offers an easy to understand framework for incorporating new procedures, but adding procedures with capabilities that overlap those of existing methods may result in redundant processing.
- It is not difficult to organize procedures into an acceptable hierarchy, that is, one which makes full use of procedure capabilities. However, it is virtually impossible to determine optimal configurations without a priori knowledge of what constraints will appear in a program, and in what order they will be encountered.
- It is possible to determine when the limits of a constraint solver have been exceeded by identifying when the solver could backtrack over data for which all possible solutions might not have been returned.
- False-positive reports of satisfaction failure at both the global and local levels are likely to occur because there is no (known) technique which can determine the number of solutions to an arbitrary set of constraints. However, this is a short-coming of our knowledge of mathematics, not of the solver architecture.
- The use of metalevel data can make constraint satisfaction much more efficient by helping to keep the solver focused on search paths that are likely to be productive. As well, metalevel data can be used to reduce the generation of both duplicate sets of substitutions and tentative replies.
- The use of metalevel data has pitfalls similar to those seen with Prolog's cut operator, that is, if it is not used correctly, valid solutions to a program may be lost.

9.3 Contributions

The following contributions have been made:

1. It has been shown that (1) current constraint programming languages are deficient in the areas of extensibility and expressiveness, and (2) the languages are

subject to the same limitations as their predecessors because their constraint solvers are not extensible.

2. A new approach to constraint satisfaction has been presented. It addresses the problems of extensibility and provides a framework for implementing a much more powerful constraint solver than was previously possible.
3. The solver's requirements for new satisfaction procedures have been presented. It was shown how a new procedure could be created by adding three components to a generic procedure skeleton. Restrictions for creating new procedure components were also specified.
4. Algorithms and guidelines for organizing satisfaction procedures into a hierarchy have been presented. It has also been shown that optimal configurations cannot be created without a priori knowledge of the contents of users' programs.
5. It has been shown that it is possible to determine when the abilities of a constraint solver have been exceeded. Such data could be invaluable in identifying when valid solutions to a program have been lost, and it would provide guidelines for taking corrective measures.
6. The use of metalevel data to aid the constraint satisfaction process has been investigated. Language constructs permitting the expression of several types of metalevel data have been presented, and it has been shown that such data may be expressed in a declarative fashion.
7. Methods of utilizing the metalevel data have been presented. It has been shown that its procedural interpretation can be used to:
 - (a) help locate suitable satisfaction methods at run-time, and in so doing, make the satisfaction process much more efficient,
 - (b) overcome minor limitations of solver procedures by partitioning constraints into smaller, more easily solved subsets, and

- (c) help suppress the generation of (1) duplicate solutions and (2) solutions that can neither be confirmed nor dismissed.

9.4 Further Research

There are many issues that remain to be investigated. These include implementation issues, extending the use of metalevel data, the assignment of functions and compound values to variables, optimal procedure configurations, and making the system concurrent. Each is considered in more detail below.

Implementation Issues

The problems associated with implementing the solver can be divided into two areas. One involves those portions of the solver which are independent of any particular solver configuration; these will be referred to as “system implementation” problems. The other involves the problems of how to build and incorporate new procedures into the solver; these are referred to as “procedure definition” problems. These areas are not independent because the manner in which the language makes use of the components will affect how they may be written.

System implementation problems include the following:

- defining how the solver will work with a Prolog interpreter, that is, should MP and the other procedures be implemented as independent modules or should they be embedded into the interpreter for the sake of efficiency,
- implementing an efficient method for MP to search for MP-solvable and MP-rejectable constraints in its problem set (a form of topological sorting [45] might be suitable),
- creating data structures so as to minimize copying and maximize sharing of data between procedures,
- defining how constraints will be represented and passed as arguments to procedures and their components,

- determining how the components of a procedure may be specified, e.g., should they be required to accept some predefined list of arguments,
- implementing system metalevel predicates for examining variables in constraints, and
- creating a means of specifying where procedures should fit into the solver hierarchy.

Needless to say, implementing the solver would require a considerable amount of programming. It might be possible to build a prototype entirely in Prolog although some problems might arise when implementing satisfaction functions and appropriate metalevel predicates might not be available.

Procedure definition problems include:

- how to specify efficient recognition rules and sufficiency predicates for a particular class of constraints,
- deciding on how many satisfaction functions and function calls to use, and
- how to write efficient satisfaction functions, especially those which can benefit from incremental processing.

The latter of these is likely to borrow heavily from techniques used in numerical analysis. Examples of this were encountered in Chapter 3 where current constraint programming languages were seen to make use of branch and bound optimization, bisection searching, and parts of the well-known simplex algorithm.

Extending the Use of Metalevel Data

The three types of metalevel data discussed in Chapter 7 were presented to demonstrate that metalevel data could be used to make the satisfaction process much more efficient. Although the types presented were thought to be the most likely to be useful, others are certainly possible. As well, the manner in which different combinations of data may be interpreted offers some interesting challenges. In Chapter 7, such problems resulted in the classification of groups of constraints rather than the independent use of classification and group data.

One of the problems encountered when trying to make use of the metalevel data was that the truth of the data could easily change as values were determined. For instance, metalevel data might declare that a constraint was of a particular class even though the statement might not be true after some values were substituted. This resulted in the problem of how to ensure that procedures were always dealing with the most up-to-date versions of constraints without using an exhaustive search procedure (which would defeat the purpose of using metalevel data). Several schemes were devised but they all became quite complicated and wandered well away from any original use of the metalevel data.

It is possible that some use could be made of metalevel data that initially failed by incorporating some form of a truth-maintenance system [44]. This might require a constraint solver for boolean satisfaction problems and the use of temporal logic since the state of a constraint may change with time. Thus, the best methods for using metalevel data remain unclear.

Assigning Functions and Compound Values

There are times when it is necessary and/or convenient to be able to assign functions to variables rather than fixed values. Although this ability was not discussed or investigated in depth, it is believed that the solver could support such assignments. When procedures make recursive calls, they could pass a pointer to a function as the variable "value", and specific values would only be generated (and returned to PE) if every argument of the function was bound.

Another approach would be to allow procedures to generate constraints as variable assignments since functions could always be expressed as a series of arithmetic relationships. However, this could affect the termination characteristics of the solver because it could cause additional variables to be generated every time the solver recursed. (In the proof of termination, the solver could not recurse indefinitely because each recursion eliminated at least one variable.)

The ability to assign compound values to variables would also be useful for problems involving matrices, vectors, and complex numbers. Again, these problems were not discussed or investigated in depth but it is believed that the solver could

support such assignments if minor changes were made to MP and the remaining language supported such data types.

Procedure Configurations

Methods of organizing procedures to make the satisfaction process as efficient as possible are another area worthy of research. While static configurations were the only ones discussed, another possibility involves dynamically reconfiguring the hierarchy based on the number of times procedures call their functions. Such an approach might require alternate configurations to be known before run-time.

Concurrency

It may be possible to modify the solver so that its procedures behave as independent processing entities. One can imagine each procedure broadcasting CM's to all of its children simultaneously rather than having to wait for one child to reply before another can be sent a message. However, it is believed that considerable effort would be required to make the system concurrent. For instance, if multiple procedures could simultaneously generate solutions for the same variable, a new means of handling conflicting answers becomes necessary.

References

- [1] Aiello, L., C. Cecchi, and D. Sartini (1986), "Representation and Use of Meta-knowledge," *Proceedings of the IEEE*, Vol. 74, No. 10, pp. 1304-1321.
- [2] Bell-Northern Research (1988), *Prolog Language Description, Version 1.0*, Ottawa, Ontario.
- [3] Beringer, H. and F. Porcher (1989), "A Relevant Scheme for Prolog Extensions: CLP (Conceptual Theory)", *Logic Programming: Proceedings of the Sixth International Conference*, MIT Press, Cambridge, Mass, pp. 131-148.
- [4] Borning, A. (1985), "Thinglab - A Constraint Oriented Simulation Laboratory", Ph.D. Thesis, Stanford University, Palo Alto, Calif.
- [5] Bowen, K., and R. Kowalski (1982), "Amalgamating Language and Metalanguage in Logic Programming", *Logic Programming*, Academic Press, London, pp. 153-172.
- [6] Brown, R., J. Chinneck, and G. Karam (1989), "Optimization with Constraint Programming Systems", *Impacts of Recent Computer Advances on Operations Research*, North-Holland, New York, pp. 463-473.
- [7] Bundy, A. (1983), *The Computer Model of Mathematical Reasoning*, Academic Press, London.
- [8] Colmerauer, A. (1982), "Prolog and Infinite Trees", *Logic Programming*, Academic Press, New York, pp. 231-251.

- [9] Colmerauer, A. (1987), "Opening the Prolog-III Universe", *Byte Magazine*, August 1987, pp. 177-182.
- [10] Complete Logic Systems (1987), *Trilogy User's Guide*, North Vancouver, British Columbia.
- [11] Cutland, N.J. (1980). *Computability: an Introduction to Recursive Function Theory*. Cambridge University Press, New York, New York.
- [12] Dincbas, M., H. Simonis, and P. Van Hentenryck (1988a), "Solving a Cutting-Stock Problem in Constraint Logic Programming", *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass, pp. 42-58.
- [13] Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf (1988b), "Applications of CHIP to Industrial and Engineering Problems", *First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, U.S.A.
- [14] Dincbas, M., H. Simonis, and P. Van Hentenryck (1988c), "Solving the Car-Sequencing Problem in Constraint Logic Programming", *European Conference on Artificial Intelligence*, Munich, F.R.G.
- [15] Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier (1988d), "The Constraint Programming Language CHIP", Technical Report TR-LP-37, European Computer-Industry Research Centre, Muenchen, F.R.G.
- [16] Gerald, C.F. (1980), *Applied Numerical Analysis*, Second Edition, Addison-Wesley, Reading, Mass, p. 75.
- [17] Goguen, J. and J. Meseguer (1986), "EQLOG: Equality, Types, and Generic Modules for Logic Programming", *Logic Programming. Functions, Relations, and Equations*, Prentice Hall, Englewood Cliffs, New Jersey, pp. 295-355.

- [18] Gottfried, B.S., and J. Weisman (1973), *Introduction to Optimization Theory*, Prentice-Hall, Englewood Cliffs, New Jersey.
- [19] Heintze, N., S. Michaylov, and P. Stuckey (1987a) "CLP($\mathcal{R}$) and Some Electrical Engineering Problems", *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, Cambridge, Mass, pp. 675-703.
- [20] Heintze, N., J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap (1987b) *The CLP($\mathcal{R}$) Programmer's Manual*, Version 2.0. Dept. of Computer Science, Monash University, Victoria, Australia.
- [21] Heintze, N., S. Michaylov, P. Stuckey, and R. Yap (1989) "On Meta-Programming in CLP($\mathcal{R}$)", *Logic Programming: Proceedings of the North American Conference 1989*, MIT Press, Cambridge, Mass, pp. 52-66.
- [22] Huet, G. (1972), "Constrained Resolution: A Complete Method for Higher Order Logic", Ph.D. Thesis, Case Western Reserve University.
- [23] Huynh, T. and C. Lassez (1988), "CLP($\mathcal{R}$) Options Trading Analysis System", *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass, pp. 59-69.
- [24] Jaffar, J. and J.-L. Lassez (1987), "From Unification to Constraints", *Logic Programming '87, Proceedings of the Sixth Conference*, Tokyo, Springer-Verlag.
- [25] Jaffar, J. and P. Stuckey (1986), "Logic Program Semantics for Programming with Equations", *Third International Conference on Logic Programming*, Springer-Verlag, pp. 313-326.
- [26] Jaffar, J. and S. Michaylov (1987), "Methodology and Implementation of a CLP System", *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, Cambridge, Mass, pp. 196-218.
- [27] Kernighan, B., and D. Ritchie (1988), *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey.

- [28] Kulisch, U., and W. Miranker (1981), *Computer Arithmetic in Theory and in Practice*, Academic Press.
- [29] Leler, Wm. (1988), *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, Reading, Mass.
- [30] Logicware Inc. (1985), *MProlog Language Reference Manual*, Release 2.1, Toronto, Ontario.
- [31] Malachi, Y. and R. Waldinger (1986), "TABLOG: A New Approach to Logic Programming", *Logic Programming. Functions, Relations, and Equations*, Prentice Hall, Englewood Cliffs, New Jersey, pp. 365-394.
- [32] Moore. R. (1966), *Interval Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.
- [33] Older, W. (1989), "Interval Arithmetic Specification", Computing Research Laboratory Ref. 89032, Bell-Northern Research, Ottawa, Ontario.
- [34] Ratschek, H. and J. Rokne (1984), *Computer Methods for the Ranges of Functions*, Ellis Horwood Ltd., Chichester.
- [35] Simonis, H. and M. Dincbas (1987), "Using an Extended Prolog for Digital Circuit Design", *Proceedings of the IEEE International Workshop on AI Applications to CAD Systems for Electronics*, Munich, F.R.G.
- [36] Steele, G.L. (1980), "The Definition and Implementation of a Computer Programming Language based on Constraints", Ph.D. Thesis, MIT, Cambridge, Mass.
- [37] Sterling, L. and E. Shapiro (1986), *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, Cambridge, Mass.
- [38] Subrahmanyam, P. and J.-H. You (1984), "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming", *Proceedings of the IEEE International Symposium on Logic Programming*, Atlantic City, New Jersey.

- [39] Sutherland, I.E. (1963), "Sketchpad: A Man-Machine Graphical Communication Interface", Ph.D. Thesis, MIT, Cambridge, Mass.
- [40] van Emden, M. and K. Yukawa (1987), "Logic Programming with Equations", *Journal of Logic Programming* 1987, 4:265-288.
- [41] Van Hentenryck, P. (1989), *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, Mass.
- [42] Van Hentenryck, P. and Y. Deville (1991), "The Cardinality Operator: A New Logical Connective for Constraint Logic Programming", *Logic Programming: Proceedings of the Eighth International Conference*, Paris, June 1991.
- [43] Voda, P. (1988), "Types of Trilogy", *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass, pp. 580-589.
- [44] Winston, P.H. (1984), *Artificial Intelligence*, Second Edition, Addison-Wesley, Reading, Mass, pp. 237-247.
- [45] Wirth, N. (1976), *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, p. 182.
- [46] Zachary, J. (1988), "A Pragmatic Approach to Equational Logic Programming", *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass, pp. 295-310.

Appendix A

Handling Simple YES Responses

A major design decision had to be made regarding how the solver should handle function results that do not fully satisfy a set of constraints C . When C is sent to the procedures, every function that is called will return one of the following:

1. MAYBE: The set may have a solution but no substitutions can be found.
2. MAYBE(S): The set may have a solution and S may contain some or all of it.
3. YES: The set does have a solution but it cannot be found.
4. YES(S): The set does have a solution and S contains some or all of it.
5. NO: The set does not have a solution.

When case (1) results, the search continues with the next function since nothing useful has been determined. In cases (2) and (4), the substitutions in list S are verified recursively before any results are returned, while case (5) causes the set of constraints to be rejected. The simple YES of case (3) causes an interesting problem to occur; it results in the following choices:

- (1) Should the procedure return YES immediately, and if so, should all of its ancestors do likewise until YES is returned to MP?

If this approach was taken, consider what would happen if a procedure returned YES before any values were determined, that is, the solver had not yet recursed. The inference engine would try to continue with no new values even though some other function in the solver might have been able to generate a solution. Such an approach could also create a large number of backtrack points. For instance, if there were N procedures that could verify that a constraint c was feasible, it would take N AVR's before c was passed to all of the procedures.

- (2) Should the YES be remembered but the search continued with the next function and possibly the next procedure, the YES being returned only if specific values cannot be found, and then as a separate solution?

If the solver were to continue searching and the next function returned NO (to indicate that there were no solutions other than the ones that were returned already), then clearly, backtracking should not occur before the YES response was returned. Thus, the solver would have to remember the NO and return it only after having received a subsequent AVR. Even if continued searching produced a MAYBE(S) response, the solver would have to (1) return the YES and remember the MAYBE(S) in case a subsequent AVR was received, or (2) return the MAYBE(S) and remember the YES. Consequently, the solver would have to be able to remember various solutions for future use.

This approach could also be taken to the extreme, the solver generating every solution that it was capable of producing before returning any values were returned. Besides complicating the solution process and greatly increasing the time to return the first solution, this tactic could easily result in an overwhelming amount of data. Thus, taking this approach to the extreme is dismissed as being impractical.

- (3) Should the YES be remembered but the search continued with the next function and possibly the next procedure, the YES being combined with any specific values that are generated?

This requires each procedure to be able to remember a simple YES and to continue the forward search as long as the possibility of a solution remains. If any tentative values are generated, that is, those qualified with MAYBE, a YES+MAYBE(S)

solution can be used to indicate that there is a solution, but it cannot be guaranteed that the values in S satisfy all of the constraints.

While such an approach helps to prevent the generation of many backtrack points (as could occur with choice (1)), it still has the problem of what to do if a subsequent function returns NO. This is not a big problem if the following restriction is placed on when a function can return NO (it appears as part of Restriction SF3 in Section 4.4):

A function F will return NO when passed a set S of constraints only if it can be guaranteed that all possible solutions to S have already been returned, including YES when unique values cannot be determined.

The third approach is considered to be the best because (1) it does not result in an excessive number of choicepoints, and (2) it does not overly complicate the solution process.

Appendix B

Constraint Substitutions

Whenever a function F generates some substitutions, they are sent to MP as a list S of assignment constraints. MP extracts the values, substitutes them wherever the variables appear in its problem set, and then simplifies the constraints. If F had generated values for every variable in a constraint c , c will be totally reduced after MP simplifies it. However, this work is redundant if F used c in its solution process. To avoid the unnecessary work, F could return “constraint substitutions” along with the variable substitutions. The constraint substitutions would flag those constraints that are known to have been totally reduced and satisfied, allowing MP to immediately tag them as such.

Besides the efficiency improvement, this approach can be used to take care of situations in which:

1. there are constraints that are satisfied regardless of what values are assigned to their variables, as in $Y = Y$, and
2. there are constraints with all of their variables bound but only a function is capable of determining that they are satisfied

If a function F was passed such a set of constraints, it would need to be able to return YES without generating any values. However, this would let the forward search continue without letting MP and the other procedures know that the constraints no longer need to be considered. As well, if the next function call resulted in

a NO reply to indicate that there were no alternate solutions, this would cause the procedure to immediately return NO and so the affirmative solution would be lost. This situation is avoided by having F return YES(S) where S contains constraint substitutions, and letting the solver recurse.

Appendix C

Proofs of Lemmas

Many of the lemmas are worded along the lines of:

If a procedure $P_i(k)$ returns reply X in response to a CM, then a function F of $P_i(k)$ or one of its descendants returned reply Y .

It is implicit that F returned Y between the time that $P_i(k)$ was sent the CM and when it returned X .

The lemmas are grouped into seven sets from “A” to “G”, those in each set being loosely related. They are also ordered so that no lemma depends upon any other which appears later in the series. This prevents circular arguments from occurring.

Lemma A1

If a procedure $Pa(k)$ receives a CM from its parent and it updates its problem set, the updated set will be a subset of its parent’s problem set.

Proof:

Assume that a procedure $Pa(k)$ is sent a CM M , but its updated problem set contains a constraint c which is not in the problem set of its parent, $Pb^+(k)$. If c is in the problem set of $Pa(k)$, it must have been sent in CM M , or it must have been present when the CM was received. Each case is considered below.

Case 1: c was sent in CM M_i

If this were true, c must have been in the problem set of $Pb^+(k)$. This contradicts the assumption and so c could not have been sent in CM M_i .

Case 2: c was present when M_i was received

If c was already in the problem set of $Pa(k)$, it must have been sent to $Pa(k)$ in an earlier CM. This indicates that $Pb^+(k)$ must still have c in its own problem set or a reduced version of it. If $Pb^+(k)$ still has c in its problem set, the assumption is contradicted and so the problem set must contain a reduced version of c ; call it c' . However, if this were true, $Pb^+(k)$ would have sent c' to $Pa(k)$ in M_i and by Restriction RS6, $Pa(k)$ would have recognized it and replaced c . This raises another contradiction and so c could not have been present when CM M_i was received.

If c could not have been sent in CM M_i or present when M_i was received, a contradiction arises and so the original assumption must be wrong. Therefore, when a procedure $Pa(k)$ receives a CM and it updates its problem set, the updated set will be a subset of its parent's problem set.

Q.E.D.

Corollary A1.1

When a procedure $Pa(k)$ receives a CM M_i from its parent and it updates its problem set, every constraint in the updated set is a member of L_{M_i} .

Proof:

By Lemma A1, when $Pa(k)$ receives a CM and it updates its problem set, the updated set is a subset of its parent's problem set. Since L_{M_i} and the parent's problem set are identical, $Pa(k)$'s updated set contains a subset of the constraints in L_{M_i} .

Q.E.D.

Corollary A1.2

When a procedure $Pa(k)$ is sent a CM from its parent and it updates its problem set, its updated set is a subset of each of its ancestors' problem sets.

Proof:

Assume that a procedure $Pa(k)$ is sent a CM M_i but its updated problem set contains a constraint c which is not in the problem set of some ancestor, $Px(k)$. By Lemma A1, $Pa(k)$'s problem set is a subset of its parent's problem set and so $Px(k)$ cannot be the parent of $Pa(k)$. Thus, $Px(k)$ must be an ancestor of the parent of $Pa(k)$.

Let $Pb(k)$ denote the parent of $Pa(k)$. It was just shown that c must be a member of $Pb(k)$'s problem set, but now the above arguments repeat with $Pb(k)$ replacing $Pa(k)$; that is, $Pb(k)$'s problem set is a subset of its parent's problem set. Thus, $Px(k)$ cannot be the parent of $Pb(k)$, and so it must be an ancestor of the parent of $Pb(k)$.

These arguments repeat indefinitely and every time that they are used, $Px(k)$ must be another higher in the hierarchy. If repeated enough times, it follows that $Px(k)$ cannot be the parent of a child of $MP(k)$, i.e., $Px(k)$ cannot be $MP(k)$, and so it must be an ancestor of $MP(k)$. Since $MP(k)$ has no ancestors, a contradiction arises and so the original assumption must be wrong. Therefore, when a procedure $Pa(k)$ is sent a CM and it updates its problem set, the updated set is a subset of each of its ancestors' problem sets.

Q.E.D.

Corollary A1.3

If a procedure $Pa(k)$ is sent a CM M_i and it subsequently calls a satisfaction function F , its problem set when it calls F is a subset of each of its ancestors' problem sets.

Proof:

Assume that when $Pa(k)$ calls F , it has a constraint c in its problem set which is not in the problem set of one of its ancestors. By Corollary A1.2, when $Pa(k)$ receives a CM M_i and it updates its problem set, its updated set is a subset of each of its ancestors' problem sets. Therefore, c must be added to the set after $Pa(k)$ updates its problem set but before $Pa(k)$ calls F . Since there are no transitions which allow this to happen, a contradiction arises and so the original assumption

must be wrong. Thus, if a procedure $Pa(k)$ is sent a CM and it subsequently calls a satisfaction function F , its problem set when it calls F is a subset of each of its ancestors' problem sets.

Q.E.D.

Lemma A2

If (1) a procedure $Pa(k)$ is sent a CM M_i , and (2) $Pa(k)$ or a descendent calls a satisfaction function F before $Pa(k)$ replies, then F will be passed a subset of the constraints in L_{M_i} .

Proof:

Assume that a procedure $Pj(k)$ in the same subtree as $Pa(k)$ calls a function F after receiving a CM but it has a constraint c in its problem set such that $c \ni L_{M_i}$. The following cases are possible.

Case 1: $Pj(k)$ is $Pa(k)$

By Corolary A1.1, $Pa(k)$'s updated problem set would consist of a subset of the constraints in L_{M_i} . Therefore, F would be passed a subset of the constraints in L_{M_i} , and so F could not be passed c . This raises a contradiction and so $Pj(k)$ cannot be $Pa(k)$.

Case 2: $Pj(k)$ is a descendent of $Pa(k)$

By Corolary A1.3, if a procedure $Pa(k)$ is sent a CM M_i and it subsequently calls a satisfaction function F , its problem set when it calls F would be a subset of each of its ancestors' problem sets. Thus, $Pj(k)$'s problem set would be a subset of $Pa(k)$'s, and by Corolary A1.1, $Pa(k)$'s updated problem set would be a subset of the constraints in L_{M_i} . Thus, F would be passed a subset of the constraints in L_{M_i} , and so F could not be passed c . This raises a contradiction and so $Pj(k)$ cannot be a descendent of $Pa(k)$.

If $Pj(k)$ cannot be $Pa(k)$ or one of its descendents, a contradiction arises and so the original assumption must be wrong. Therefore, if a procedure $Pa(k)$ is sent a CM M_i and $Pa(k)$ or a descendent calls a satisfaction function F before $Pa(k)$

replies, then F will be passed a subset of the constraints in L_{M_i} .

Q.E.D.

Lemma A3

For any CM M_a which is sent to $MP(k)$, if a procedure $Pi(k)$ is subsequently sent a CM M_i , then its updated problem set will contain every constraint $c \in PS_{M_a}^4$ that it can recognize.

Proof:

Assume that $Pi(k)$ is sent a CM M_i but its updated problem set does not contain some constraint $c \in PS_{M_a}^4$ which it can recognize. This indicates that $Pi(k)$ could not have been sent c by its parent; otherwise, $Pi(k)$ would have to fail to recognize c but then Restriction RS1 would be violated. Let $Pm(k)$ be the child of $MP(k)$ at the root of the subtree containing $Pi(k)$. $Pi(k)$ could be $Pm(k)$ or one of its descendants; each case is considered below.

Case 1: $Pi(k)$ is $Pm(k)$

If $MP(k)$ is the parent of $Pi(k)$, $MP(k)$ would send it a CM containing $PS_{M_a}^4$ and so $Pi(k)$ would be sent c . This raises a contradiction and so $Pi(k)$ cannot be $Pm(k)$.

Case 2: $Pi(k)$ is a descendent of $Pm(k)$

If c was not sent to $Pi(k)$, either (1) c was sent to its parent $Pz(k)$ and $Pz(k)$ failed to recognize it, or (2) c was not sent to $Pz(k)$. The first case could not occur without violating Restriction RS1 and so c could not have been sent to $Pz(k)$.

Let $Py(k)$ denote the parent of $Pz(k)$. The above arguments now repeat with $Pz(k)$ and $Py(k)$ replacing $Pi(k)$ and $Pz(k)$, respectively. If c was not sent to $Pz(k)$, Restriction RS1 would be violated and so c could not have been sent to $Py(k)$. These arguments repeat for every ancestor of $Pi(k)$, up to and including $Pm(k)$, the child of $MP(k)$. However, $PS_{M_a}^4$ is sent to $Pm(k)$ and $c \in PS_{M_a}^4$. Thus, a contradiction arises and so $Pi(k)$ cannot be a descendent of $Pm(k)$.

If $Pi(k)$ cannot be $Pm(k)$ or one of its descendants, a contradiction arises and so the original assumption must be wrong. Thus, given the above conditions, $Pi(k)$'s

updated problem set will contain every constraint $c \in PS_{M_a}^4$ that $Pi(k)$ can recognize.

Q.E.D.

Lemma A4

Given the following conditions:

1. $MP(k)$ is sent a CM M_a ,
2. a procedure $Pj(k)$ is sent a CM M_j before $MP(k)$ replies,
3. $Pj(k)$ has a function F which can (1) determine that the constraints in $PS_{M_a}^4$ are feasible or (2) generate substitutions for one or more variables in $PS_{M_a}^4$ and determine that the constraints are feasible with the substitutions applied, and
4. F is called before $Pj(k)$ replies,

then:

1. F will be passed every constraint in $PS_{M_a}^4$, and
2. $A_{Pj(k)}$ will be set to 1.

Proof:

Each result is considered below.

Result 1: F will be passed $PS_{M_a}^4$

Given condition (3) and Restriction RS1, the recognition rules of $Pj(k)$ must be able to recognize every constraint in $PS_{M_a}^4$, and by Lemma A3, the updated problem set of $Pj(k)$ would contain every constraint $c \in PS_{M_a}^4$. Thus F will be passed every constraint in $PS_{M_a}^4$.

Result 2: $A_{Pj(k)}$ will be set to 1

Assume that $A_{Pj(k)}$ might not be set to 1 when F is called. It could be set to 0 only if $Pj(k)$ failed to recognize every constraint in L_{M_j} , or A_{M_j} was 0. In Result 1,

it was shown that the first possibility could not be true and so A_{M_j} would have to be 0.

If A_{M_j} was 0, then the parent $Pm(k)$ of $Pj(k)$ was sent a CM M_b and it either failed to recognize every constraint in L_{M_b} or A_{M_b} was set to 0. The first case could not occur because of Restriction RS1 and so A_{M_b} must have been 0.

These arguments repeat for every ancestor of $Pj(k)$, up to and including some child of $MP(k)$. However, when $MP(k)$ sends a CM M_c to one of its children, A_{M_c} is set to 1. This raises a contradiction and so the assumption must be wrong. Thus, given the above conditions, $A_{Pj(k)}$ will be set to 1.

Q.E.D.

Corolary A4.1

Given the following conditions:

1. a procedure $Pi(k)$ is sent a CM M_i ,
2. it has a function F which can determine that the constraints in L_{M_i} are feasible.
and
3. F is called before $Pi(k)$ replies.

then F will return YES.

Proof:

F can determine the feasibility of the constraints in L_{M_i} only if L_{M_i} contains every constraint in $PS_{M_i}^4$. Lemma A4 indicates that F would be passed this set and $A_{Pi(k)}$ would be set to 1, and so by Restriction SF5, F would return YES.

Q.E.D.

Corolary A4.2

Given the following conditions:

1. a procedure $Pi(k)$ is sent a CM M_i ,
2. it has a function F which can generate substitutions that when applied to the constraints in L_{M_i} will result in a feasible set of constraints.

3. F can determine that the constraints in L_{M_i} are feasible with the substitutions applied, and

4. F is called before $Pi(k)$ replies,

then F will return YES(S).

Proof:

F can determine the feasibility of the constraints in L_{M_i} only if it is passed $PS_{M_i}^4$. Lemma A4 indicates that F would be passed this set and $A_{Pi(k)}$ would be set to 1, and so by Restriction SF6, F would return YES(S).

Q.E.D.

Lemma A5

For any CM M_a , if the constraints in $PS_{M_a}^4$ are feasible, then the constraints in L_{M_a} are feasible.

Proof:

Assume that the constraints in $PS_{M_a}^4$ are feasible but the constraints in L_{M_a} are not. By definition, if the constraints in $PS_{M_a}^4$ are feasible, then either the constraints are already satisfied, or there is at least one set of substitutions which can satisfy every constraint $c \in PS_{M_a}^4$. The following two cases must be considered.

Case 1: $PS_{M_a}^4$ is identical to $PS_{M_a}^+$

These sets are identical only if MP(k) does not make transitions 24 and 3, in which case $PS_{M_a}^4$ consists of the set $PS_{M_a}^- \cup L_{M_a}$. However, if L_{M_a} contains nonfeasible constraints, then $PS_{M_a}^4$ does as well. This raises a contradiction and so $PS_{M_a}^4$ cannot be identical to $PS_{M_a}^+$ if L_{M_a} contains nonfeasible constraints.

Case 2: $PS_{M_a}^4$ is not identical to $PS_{M_a}^+$

The sets are not identical if MP(k) has generated one or more substitutions using transitions 24 and 3. If L_{M_a} contains some nonfeasible constraint c , then one of the following two cases must hold.

Case 2.1: c is variable free and nonfeasible

If c is variable free and nonfeasible, then it must be contradictory. However, if $c \in L_{M_a}$, then $c \in PS_{M_a}^+$ and $c \in PS_{M_a}^4$ (because $MP(k)$ could not generate any substitutions which would affect c). However, if $PS_{M_a}^4$ contains c , then the set cannot be feasible.

Case 2.2: c contains one or more variables

If c contained one or more variables and it is nonfeasible, then there is no set of substitutions which could make it feasible. If $MP(k)$ determined substitutions for one or more of its variables, reducing it to some constraint $c' \in PS_{M_a}^4$, then c' would still be nonfeasible and so would $PS_{M_a}^4$. If $MP(k)$ determined substitutions for variables in $PS_{M_a}^*$ such that c was unaffected, then $PS_{M_a}^4$ would contain c and the set would still be nonfeasible.

Whether c contains variables or not, $PS_{M_a}^4$ would be nonfeasible if c is nonfeasible. It follows that whether or not $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical, a contradiction is raised and so the original assumption must be wrong. Thus, for any CM M_a , if the constraints in $PS_{M_a}^4$ are feasible, then the constraints in L_{M_a} are feasible.

Q.E.D.

Lemma B1

$MP(k)$'s problem set, $PS_{M_i}^+$, can never contain more than a finite number of uninstantiated variables.

Proof:

The number of uninstantiated variables in the problem set of $MP(k)$ is the sum of the numbers of uninstantiated variables in the constraints that are sent to $MP(k)$. By Restriction C2, the solver would be sent only a finite number of constraints, and by Restriction C1, each constraint contains at most a finite number of uninstantiated variables. Let n be the number of constraints that are sent and m be the maximum number of variables in any constraint. Therefore, the number of uninstantiated variables must be less than or equal to the product of n and m , and if both n and m are finite, so is their product. Hence, the problem set of $MP(k)$ can never contain

more than a finite number of uninstantiated variables.

Q.E.D.

Lemma B2

If there is a set of procedures $\{ Pa(k), Pb(k+1), Pc(k+2), \dots \}$ such that each procedure $Pi(k+m)$, $m \geq 0$, has made a recursive call to $MP(k+m+1)$ and $MP(k+m+1)$ cannot reply because a descendent $Pj(k+m+1)$ has yet to receive a reply from $MP(k+m+2)$, then there exists a value of n , $n \geq 1$, such that no procedure $Pl(k+n)$ will make a recursive call; i.e., the solver can recurse only a finite number of times before no procedure $Pl(k+n)$ can make a recursive call.

Proof:

Let n_v denote the number of variables in MP's problem set when $MP(k+1)$ receives a CM from some procedure $Pa(k)$. When a procedure $Pi(k+j)$, $j \geq 0$, recursively calls $MP(k+j+1)$, the CM must contain at least one assignment constraint and so at least one variable would be eliminated from MP's problem set. By Lemma B1, n_v is a finite number and so the solver can recurse at most n_v times before all variables will be eliminated.

Now assume that after the solver has recursed $k+n$ times, $n \geq 1$, all variables have been eliminated and a descendent $Pq(k+n)$ of $MP(k+n)$ makes a recursive call. $Pq(k+n)$ could do so only if it called one of its satisfaction functions and the function returned YES(S) or MAYBE(S). By Corolary A1.3, $Pq(k+n)$'s problem set would consist of a subset of the constraints in $MP(k+n)$'s problem set but since all variables were eliminated, Restriction SF4 prevents any satisfaction function from returning YES(S) or MAYBE(S). This raises a contradiction and so the assumption must be wrong. Therefore, the solver can recurse only a finite number of times before no procedure can make a recursive call.

Q.E.D.

Lemma C1

If $MP(k)$ is sent a CM M_i and it reaches state 4 before it replies, it will not have any MP-solvable or MP-rejectable constraints in its problem set when it reaches state 4.

Proof:

First assume that $MP(k)$ has an MP-solvable constraint c in its problem set when it reaches state 4. $MP(k)$ must move to state 7 after receiving M_i and in order to reach state 4, it must make transition 2 or transitions 24 and 4. If $c \in PS_{M_i}^*$, when $MP(k)$ reached state 4, it must have been present just before $MP(k)$ made transition 2 or 4 from state 7 or 6, respectively. However, if c is MP-solvable and $MP(k)$ was in state 7, $MP(k)$ would make transition 24 and not 2. Similarly, if $MP(k)$ was in state 6, it would make transition 3 and not 4. Both cases raise a contradiction and so $MP(k)$ could not have an MP-solvable constraint in its problem set when it reaches state 4.

Now assume that $MP(k)$ has an MP-rejectable constraint c in its problem set when it reaches state 4. As stated above, $MP(k)$ could reach state 4 only by making transitions 2 or 24 and 4. If $c \in PS_{M_i}^*$, when $MP(k)$ reached state 4, c must have been present before $MP(k)$ made transition 2 or 4 from state 7 or 6, respectively. If this were true, $MP(k)$ would make transition 14 or 26 from state 7 or 6 and return NO without ever reaching state 4. This raises a contradiction and so $MP(k)$ could not have an MP-rejectable constraint in its problem set when it reaches state 4.

Q.E.D.

Lemma C2

Given the following conditions:

1. $MP(k)$ replies to a CM using transition 15, 25, 5, 6, 27, 28, 10, 11, 12, or 13, i.e., after having reached state 4, and
2. no procedure $Pi(k)$ makes a recursive call to $MP(k+1)$ before $MP(k)$ replies.

then $MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set when it makes one of the listed transitions.

Proof:

Assume that $MP(k)$ has an MP-solvable or MP-rejectable constraint c in its problem set after it replies to a CM M_i using one of the above transitions. If $MP(k)$

replies using transition 15 or 25, every constraint in $PS_{M_i}^*$ must be totally reduced and satisfied, including c . If c is MP-solvable or MP-rejectable, a contradiction arises and so $MP(k)$ must reply using some other transition.

If $c \in PS_{M_i}^*$ when $MP(k)$ replies using transition 5, 6, 27, 28, 10, 11, 12 or 13, either (1) c must have been present when $MP(k)$ reached state 4, or (2) c must have been introduced by some other recursive instance of MP. Lemma C1 proves the first case could not arise and since it is specified that no procedure calls $MP(k+1)$, the second case cannot arise either. Thus, $MP(k)$ cannot reply using transition 5, 6, 27, 28, 10, 11, 12 or 13.

If $MP(k)$ cannot reply using one of the listed transitions, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ replies to a CM using transition 15, 25, 5, 6, 27, 28, 10, 11, 12, or 13, and no procedure made a recursive call to $MP(k+1)$, then $MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set when it replies.

Q.E.D.

Lemma C3

Given the following conditions:

1. $MP(k)$ replies to a CM M_i using transition 14, 26, or 21,
2. no procedure $P_i(k)$ makes a recursive call to $MP(k+1)$ before $MP(k)$ replies, and
3. $PS_{M_i}^-$ does not contain an MP-solvable or MP-rejectable constraint,

then $MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set after it replies to M_i .

Proof:

Assume that $MP(k)$ has an MP-solvable or MP-rejectable constraint c in its problem set after it replies to a CM M_i . If $MP(k)$ makes transition 14, 26, or 21, it discards $PS_{M_i}^*$ and makes $PS_{M_i}^-$ its problem set. Condition (3) states that $PS_{M_i}^-$ does not contain an MP-solvable or MP-rejectable constraint and so a contradiction arises.

Therefore, $MP(k)$ cannot have an MP-solvable or MP-rejectable constraint in its problem set if it replies using transition 14, 26, or 21, no procedure makes a recursive call, and $PS_{M_i}^-$ does not contain an MP-solvable or MP-rejectable constraint.

Q.E.D.

Lemma C4

If the problem set of $MP(k)$ does not contain an MP-solvable or MP-rejectable constraint when it receives a CM, then it will not have an MP-solvable or MP-rejectable constraint in its problem set when it replies.

Proof:

If no procedure makes a recursive call to $MP(k+1)$ between the time that $MP(k)$ is sent a CM M_i and the time that it replies, the result follows from Lemmas C2 and C3. It remains to be shown that an MP-solvable or MP-rejectable constraint cannot be introduced even if some procedure $P_i(k)$ does call $MP(k+1)$.

Assume that a procedure $P_i(k)$ calls $MP(k+1)$ and that $MP(k)$'s problem set does contain an MP-solvable or MP-rejectable constraint c when it replies. The solver can recurse only if $MP(k)$ calls one of its children after reaching state 4. However, by Lemma C1, $PS_{M_i}^*$ would not contain any MP-solvable or MP-rejectable constraints when $MP(k)$ reaches state 4, and so there must be some $MP(k+n)$, $n \geq 1$, which leaves c in the problem set.

By Lemmas C2 and C3, if $MP(k+1)$ replies to $P_i(k)$ without some procedure $P_j(k+1)$ calling $MP(k+2)$, the problem set of $M(k+1)$ would not contain c unless one was present when it received the CM. It has just been shown (using Lemma C2, C3) that c cannot be present and so some $P_j(k+1)$ must call $MP(k+2)$.

The above arguments now repeat. In general, if $MP(k+m)$ replies to a CM without a descendent making a recursive call, the problem set cannot contain c . This indicates that the solver must recurse and so $MP(k+m)$ must call one of its children after reaching state 4. However, Lemma C1 indicates that its problem set would not contain any MP-solvable or MP-rejectable constraints when it reaches state 4 and so c must be introduced by a recursive call to $MP(k+m+1)$, and so on.

By Lemma B2, the solver can recurse only a finite number of times before no procedure can make a recursive call. This raises a contradiction and so the original assumption must be wrong. Therefore, if the problem set of $MP(k)$ does not contain an MP-solvable or MP-rejectable constraint when it receives a CM, then such a constraint cannot be present when $MP(k)$ replies.

Q.E.D.

Lemma C5

$MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set when it replies to a CM.

Proof:

Assume that $MP(k)$ has an MP-solvable or MP-rejectable constraint c in its problem set after it replies to a CM M_i . By Lemma C4, if $PS_{M_i}^-$ did not contain an MP-solvable or MP-rejectable constraint, then the problem set of $MP(k)$ cannot contain c when it replies. Therefore, c must have been present when $MP(k)$ received M_i . Two cases must be considered: $k = 0$ and $k > 0$.

Case 1: $k = 0$

If $c \in PS_{M_i}^-$, it must have been present when $MP(0)$ replied to the last CM. However, by Lemma C4, this indicates that it must have been present when it received the previous CM, and so on. Since the problem set of $MP(0)$ is initially empty, a contradiction arises for the case of $k = 0$.

Case 2: $k > 0$

If $c \in PS_{M_i}^-$, it must have been present when some procedure $Pi(k-1)$ called $MP(k)$, and so it must have been present when $MP(k-1)$ called the child at root of the subtree containing $Pi(k-1)$. However, by Lemma C1, $MP(k-1)$ would not have any MP-solvable or MP-rejectable constraints in its problem set when it reaches state 4. Thus, a contradiction is raised for the case of $k > 0$.

Since both of the above cases lead to contradictions, the original assumption must be wrong. Therefore, $MP(k)$ will not have an MP-solvable or MP-rejectable

constraint in its problem set when it replies to a CM.

Q.E.D.

Lemma C6

$MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set when it receives a CM.

Proof:

Assume that $MP(k)$ has an MP-solvable or MP-rejectable constraint c in its problem set when it receives a CM M_i . Two cases must be considered: $k = 0$ and $k > 0$.

Case 1: $k = 0$

Since the problem set of $MP(0)$ is initially empty, c can be present when $MP(0)$ receives the CM only if it was present when $MP(0)$ replied to a previous CM. Lemma C5 indicates that this is not possible and so a contradiction is raised for the case of $k = 0$.

Case 2: $k > 0$

$MP(k)$ can receive a CM only if some other procedure $Pi(k-1)$ makes a recursive call, and $Pi(k-1)$ would make such a call only if $MP(k-1)$ called its child at root of the subtree containing $Pi(k-1)$. This requires $MP(k-1)$ to have first reached state 4 but by Lemma C1, $MP(k-1)$ would not have any MP-solvable or MP-rejectable constraints in its problem set when it reaches state 4. Thus, a contradiction is raised for the case of $k > 0$.

Since both of the above cases lead to contradictions, the original assumption must be wrong. Therefore, $MP(k)$ will not have an MP-solvable or MP-rejectable constraint in its problem set when it receives a CM.

Q.E.D.

Corollary C6

If $MP(k)$ receives some CM M_i and L_{M_i} does not contain an MP-solvable or MP-rejectable constraint, then $PS_{M_i}^+$ will not contain one either.

Proof:

Assume that $MP(k)$'s updated problem set $PS_{M_i}^+$ contains an MP-solvable or MP-rejectable constraint c_i . If $c_i \ni L_{M_i}$, c_i must be a member of $PS_{M_i}^-$ but by Lemma C6, $c \ni PS_{M_i}^-$. This raises a contradiction and so the assumption must be wrong. Therefore, if $MP(k)$ receives a CM M_i and L_{M_i} does not contain an MP-solvable or MP-rejectable constraint, then $PS_{M_i}^+$ will not contain such a constraint either.

Q.E.D.

Lemma D1

If (1) $MP(k)$ receives a CM M_a and (2) no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one constraint $c \in L_{M_a}$ is not already satisfied, then $MP(k)$ will call at least one of its children before replying.

Proof:

Assume that $MP(k)$ receives M_a but it does not call at least one of its children before replying. This indicates that $MP(k)$ must reply using transition 14, 15, 25, or 26. By condition (2), no constraint in L_{M_a} is MP-solvable or MP-rejectable and so Corolary C6 indicates that $PS_{M_a}^+$ would not contain an MP-solvable or MP-rejectable constraint. Therefore, $MP(k)$ cannot make transition 14 or 24. $MP(k)$ cannot make transition 15 either because c is not satisfied and so it must make transition 2 from state 7. If $MP(k)$ moves to state 4, it has no way of returning to state 6 in order to make transition 25 or 26, the only remaining possibilities. This raises a contradiction and so the original assumption must be wrong. Thus, if no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one is not already satisfied, then $MP(k)$ will call at least one of its children before replying.

Q.E.D.

Lemma D2

If $MP(k)$ replies to a CM using transition 10 after some child $Pi(k)$ returned MAYBE(S), then:

1. some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 32 ($P_j(k)$ could be $P_i(k)$ or one of its descendents),
2. $P_j(k)$ sent a CM M_j to $MP(k+1)$ just before making transition 32, and $MP(k+1)$ returned $MAYBE(S)$, and
3. $P_j(k)$ called a satisfaction function F just before sending M_j to $MP(k+1)$ and F returned $MAYBE(S)$.

Proof:

Each result is considered below.

Result 1: $P_j(k)$ made transition 32

Assume that $P_i(k)$ nor any of its descendents made transition 32. $P_i(k)$ could return $MAYBE(S)$ only by making transition 9 or 32. If it did not make the latter, then it must have made transition 9 after some child $P_b(k)$ returned $MAYBE(S)$. $P_b(k)$ could return such a response only by making transition 9 or 32, but it could not make the latter because it is a descendent of $P_i(k)$. Thus, it would have to make transition 32 after one of its children returned $MAYBE(S)$.

The above arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 9 after one of its children returns $MAYBE(S)$. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that cannot make transition 9 and so a contradiction is raised. Thus, the assumption must be wrong and so if $MP(k)$ makes transition 10, then $P_i(k)$ or one of its descendents made transition 32.

Result 2: $P_j(k)$ sent a CM and $MP(k+1)$ returned $MAYBE(S)$

The proof follows directly from the state transition diagram of Figure 4.7. It was shown in Result 1 that if $MP(k)$ made transition 10, then some procedure $P_j(k)$ made transition 32, and $P_j(k)$ would make the transition only if $MP(k+1)$ returned $MAYBE(S)$. As well, $P_j(k)$ would have to be in state 12 in order to make transition 32, and state 12 can be reached only by making transition 26 or 27, the actions of

which involve sending a CM to $MP(k+1)$. Thus, before $P_j(k)$ made transition 32, it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned $MAYBE(S)$.

Result 3: $P_j(k)$ called F and F returned $MAYBE(S)$

It was shown in Result 2 that $P_j(k)$ must have made transition 26 or 27 in order to reach state 12, and it could make these transitions only if it was in state 10. State 10 can be reached only by making transition 23, the action of which involves calling some satisfaction function F . As well, F would have to return $MAYBE(S)$ or $YES(S)$ in order for $P_j(k)$ to make transition 26 or 27. However, if F returned $YES(S)$, then $V_{P_j(k)}$ would be set to 1 and $P_j(k)$ would be prevented from making transition 32. Thus, F must have returned $MAYBE(S)$ and so the result follows.

Q.E.D.

Lemma D3

If $MP(k)$ replies to a CM using transition 13 after some child $P_i(k)$ returned $YES(S)$, then:

1. some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 33 ($P_j(k)$ could be $P_i(k)$ or one of its descendants),
2. $P_j(k)$ sent a CM M_j to $MP(k+1)$ just before making transition 33, and $MP(k+1)$ returned $YES(S)$,
3. $P_j(k)$ called a satisfaction function F just before sending M_j to $MP(k+1)$ and F returned $MAYBE(S)$ or $YES(S)$.

Proof:

Each result is considered below.

Result 1: $P_j(k)$ made transition 33

Assume that $P_i(k)$ nor any of its descendants made transition 33. $P_i(k)$ could return $YES(S)$ only by making transition 11 or 33. If it did not make the latter, then it must have made transition 11 after some child $P_b(k)$ returned $YES(S)$. $P_b(k)$ could return such a response only by making transition 11 or 33, but it could not

make the latter because it is a descendent of $P_i(k)$. Thus, it would have to make transition 33 after one of its children returned $YES(S)$.

The above arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 11 after one of its children returns $YES(S)$. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that cannot make transition 11 and so a contradiction is raised. Thus, the assumption must be wrong and so if $MP(k)$ makes transition 13, then $P_i(k)$ or one of its descendents made transition 33.

Result 2: $P_j(k)$ sent a CM and $MP(k+1)$ returned $YES(S)$

The proof follows directly from the state transition diagram of Figure 4.7. It was shown in Result 1 that if $MP(k)$ made transition 13, then some procedure $P_j(k)$ made transition 33, and $P_j(k)$ would make the transition only if $MP(k+1)$ returned $YES(S)$. As well, $P_j(k)$ would have to be in state 12 in order to make transition 33, and state 12 can be reached only by making transition 26 or 27, the actions of which involve sending a CM to $MP(k+1)$. Thus, before $P_j(k)$ made transition 33, it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned $YES(S)$.

Result 3: $P_j(k)$ called F and F returned $MAYBE(S)$ or $YES(S)$

It was shown in Result 2 that $P_j(k)$ must have made transition 26 or 27 in order to reach state 12, and it could make these transitions only if it was in state 10. State 10 can be reached only by making transition 23, the action of which involves calling some satisfaction function F . As well, F would have to return $MAYBE(S)$ or $YES(S)$ in order for $P_j(k)$ to make transition 26 or 27 and so the result follows. **Q.E.D.**

Lemma D4

If $MP(k)$ replies to a CM using transition 11 or 12 after some child $P_i(k)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$, then:

1. some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 31, 32, or 42 ($P_j(k)$ could be $P_i(k)$ or one of its descendents).

2. $P_j(k)$ sent a CM M_j to $MP(k+1)$ just before making transition 31, 32, or 42, and $MP(k+1)$ returned MAYBE(S) or YES+MAYBE(S), and
3. $P_j(k)$ called a satisfaction function F just before sending M_j to $MP(k+1)$ and F returned MAYBE(S) or YES(S).

Proof:

Each result is considered below.

Result 1: $P_j(k)$ made transition 33

Assume that $P_i(k)$ nor any of its descendents made transition 31, 32, or 42. By Lemma D2, if $P_i(k)$ returned MAYBE(S), then it or one of its descendents made transition 31, thus $P_i(k)$ must have returned YES+MAYBE(S). $P_i(k)$ could return YES+MAYBE(S) only by making transition 10, 41, 31, or 42. If it did not make either of the latter two, then it must have made transition 10 or 41 after some child $P_b(k)$ returned MAYBE(S) or YES+MAYBE(S), respectively. By Lemma D2, if $P_b(k)$ returned MAYBE(S), then it or one of its descendents made transition 31 and so $P_i(k)$ must have made transition 41 after $P_b(k)$ returned YES+MAYBE(S).

$P_b(k)$ could return such a response only by making transition 10, 41, 31, or 42. but now the above arguments repeat with $P_b(k)$ replacing $P_i(k)$. It could not have made either of the latter two because of the assumption and it could not have made transition 10 because of Lemma D2. Thus, it must have made transition 41 after one of its children return YES+MAYBE(S).

These arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 41 after one of its children returns YES+MAYBE(S). Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that cannot make transition 41 and so a contradiction is raised. Thus, the assumption must be wrong and so if $MP(k)$ makes transition 11 or 12, then $P_i(k)$ or one of its descendents made transition 31, 32, or 42.

Result 2: $P_j(k)$ sent a CM, $MP(k+1)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$

The proof follows directly from the state transition diagram of Figure 4.7. It was shown in Result 1 that if $MP(k)$ made transition 11 or 12, then some procedure $P_j(k)$ made transition 31, 32, or 42, and $P_j(k)$ would make one of these only if $MP(k+1)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$. As well, $P_j(k)$ would have to be in state 12 in order to make transition 31, 32, or 42, and state 12 can be reached only by making transition 26 or 27, the actions of which involve sending a CM to $MP(k+1)$. Thus, before $P_j(k)$ made transition 31, 32, or 42, it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$.

Result 3: $P_j(k)$ called F and F returned $MAYBE(S)$ or $YES(S)$

It was shown in Result 2 that $P_j(k)$ must have made transition 26 or 27 in order to reach state 12, and it could make these transitions only if it was in state 10. State 10 can be reached only by making transition 23, the action of which involves calling some satisfaction function F . As well, F would have to return $MAYBE(S)$ or $YES(S)$ in order for $P_j(k)$ to make transition 26 or 27 and so the result follows. **Q.E.D.**

Lemma D5

If $MP(k)$ replies to a CM using transition 10, 11, 12, or 13 after some child $P_i(k)$ returned $YES(S)$, $MAYBE(S)$, or $YES+MAYBE(S)$, then:

1. some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 31, 32, 33, or 42 ($P_j(k)$ could be $P_i(k)$ or one of its descendants),
2. $P_j(k)$ sent a CM M_j to $MP(k+1)$ just before making transition 31, 32, 33, or 42, and $MP(k+1)$ returned $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$,
3. $P_j(k)$ called a satisfaction function F just before sending M_j to $MP(k+1)$ and F returned $MAYBE(S)$ or $YES(S)$.

Proof:

Each result is considered below.

Result 1: $P_j(k)$ made transition 31, 32, 33, or 42

By Lemma D2, if $MP(k)$ made transition 10, then $P_j(k)$ made transition 32; by Lemma D3, if $MP(k)$ made transition 13, then $P_j(k)$ made transition 33; and by Lemma D4, if $MP(k)$ made transition 11, or 12, then $P_j(k)$ made transition 31, 32, or 42. Thus, if $MP(k)$ made transition 10, 11, 12, or 13, then $P_j(k)$ made transition 31, 32, 33, or 42.

Result 2: $P_j(k)$ sent a CM

By Lemma D2, if $MP(k)$ made transition 10, then $P_j(k)$ sent a CM to $MP(k+1)$ prior to making transition 32, and $MP(k+1)$ returned MAYBE(S); by Lemma D3, if $MP(k)$ made transition 13, then $P_j(k)$ sent a CM to $MP(k+1)$ just before making transition 33, and $MP(k+1)$ returned YES(S); and by Lemma D4, if $MP(k)$ made transition 11 or 12, then $P_j(k)$ sent a CM to $MP(k+1)$ just before making transition 31, 32, or 42, and $MP(k+1)$ returned MAYBE(S) or YES+MAYBE(S). Thus, if $MP(k)$ made transition 10, 11, 12, or 13, then $P_j(k)$ sent a CM to $MP(k+1)$ just before making transition 31, 32, 33, or 42, and $MP(k+1)$ returned MAYBE(S), YES+MAYBE(S), or YES(S).

Result 3: $P_j(k)$ called F and F returned MAYBE(S) or YES(S)

By Lemma D2, if $MP(k)$ made transition 10, then $P_j(k)$ called a function F just before sending a CM to $MP(k+1)$ and F returned MAYBE(S); by Lemma D3, if $MP(k)$ made transition 13, then $P_j(k)$ called a function F just before sending a CM to $MP(k+1)$ and F returned MAYBE(S) or YES(S); and by Lemma D4, if $MP(k)$ made transition 11 or 12, then $P_j(k)$ called a function F just before sending a CM to $MP(k+1)$ and F returned MAYBE(S) or YES(S). Thus, if $MP(k)$ made transition 10, 11, 12, or 13, then $P_j(k)$ called a function F just before sending a CM to $MP(k+1)$ and F returned MAYBE(S) or YES(S).

Q.E.D.

Corolary D5

If $MP(k)$ replies to a CM using transition 10, 11, 12, or 13 after some child $P_i(k)$ returns YES(S), MAYBE(S), or YES+MAYBE(S), then S was forwarded back through at least one level of recursion.

Proof:

By Lemma D5, if $MP(k)$ replies to a CM using transition 10, 11, 12, or 13 after some child $Pi(k)$ returns MAYBE(S), YES+MAYBE(S), or YES(S), then $MP(k+1)$ was sent a CM by some procedure $Pi(k)$ and $MP(k+1)$ returned MAYBE(S), YES+MAYBE(S), or YES(S). Thus, S was forwarded back through at least one level of recursion.

Q.E.D.

Lemma E1

If a procedure $Pi(k)$ is sent a CM and it returns YES, then at least one satisfaction function of $Pi(k)$ or one of its descendants returned YES.

Proof:

Assume that procedure $Pi(k)$ returns YES but no satisfaction function of $Pi(k)$ or its descendants returned YES. $Pi(k)$ can return YES only by making transition 39 or 19, both of which require flag $V_{Pi(k)}$ to be set to 1. This can occur only as a result of $Pi(k)$ making transition 11, 41, 33, 42, 24, 27, or 7; each case is considered below.

Case 3.1: $Pi(k)$ makes transition 24

$Pi(k)$ can make transition 24 only if some function F is called and F returns YES. This contradicts the original assumption and so $Pi(k)$ cannot make transition 24.

Case 3.2: $Pi(k)$ makes transition 33, 42, 11, or 41

If $Pi(k)$ were to make any of these transitions, it would return a response other than YES. Thus, $Pi(k)$ cannot make transition 33, 42, 11, or 41.

Case 3.3: $Pi(k)$ makes transition 7

If $Pi(k)$ made transition 7 after some child $Pj(k)$ returned YES, $Pj(k)$ must have made transition 24, 33, 42, 11, 41, or 7. The above arguments may now be repeated with $Pj(k)$ replacing $Pi(k)$. Thus, $Pj(k)$ cannot make any of the first five transitions, and so it must make transition 7 after one of its children returns YES.

The above arguments repeat indefinitely and every time that they are used, one more procedure at another level deeper in the hierarchy must make transition 7 after one of its children returns YES. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that cannot make transition 7 and so a contradiction is raised. Thus, $Pi(k)$ cannot make transition 7.

If $Pi(k)$ cannot make any of the above transitions, a contradiction arises and so the original assumption must be wrong. Therefore, $Pi(k)$ cannot return YES unless at least one satisfaction of $Pi(k)$ or one of its descendents returns YES.

Q.E.D.

Corollary E1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$, and
2. a procedure $Pi(k)$ returns YES in response to a CM M_b before $MP(k)$ replies to M_a ,

then the constraints in $PS_{M_a}^4$ are feasible.

Proof:

By Lemma E1, if $Pi(k)$ returned YES, then a satisfaction function F of $Pi(k)$ or one of its descendents returned YES. By Restriction SF5, F would return YES only if it was passed every constraint in $PS_{M_a}^4$ and the constraints were feasible. Thus, the result holds.

Q.E.D.

Lemma E2

If a procedure $Pi(k)$ returns NO in response to a CM or an AVR, then a satisfaction function of $Pi(k)$ or one of its descendents returned NO.

Proof:

Assume that a procedure $Pi(k)$ returns NO without a function of $Pi(k)$ or one of its descendents returning NO. $Pi(k)$ can return NO only by making transition 12 or 29; each case is considered below.

Case 1: $P_i(k)$ makes transition 29

$P_i(k)$ could make transition 29 only if it previously made transition 23 and called a function, and the function returned NO. However, it was assumed that no function would return NO and so $P_i(k)$ cannot make transition 29.

Case 2: $P_i(k)$ makes transition 12

$P_i(k)$ could make transition 12 only if some child $P_j(k)$ returned NO in response to a CM or an AVR. $P_j(k)$ must have returned NO using transition 12 or 29 but now the above arguments now repeat with $P_j(k)$ replacing $P_i(k)$. $P_i(k)$ cannot make transition 29 without raising a contradiction and so it must make transition 12 after one of its children returns NO.

The above arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 12 after one of its children returns NO. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that cannot make transition 12 and so a contradiction is raised. Thus, $P_i(k)$ cannot return NO using transition 12.

If $P_i(k)$ cannot make transition 12 or 29, a contradiction is raised and so the original assumption must be wrong. Therefore, $P_i(k)$ cannot return NO in response to a CM or an AVR without a function of $P_i(k)$ or one of its descendents returning NO.

Q.E.D.

Lemma E3

If a procedure $P_i(k)$ returns YES(S) or YES+MAYBE(S) in response to a CM, then:

1. at least one satisfaction function of $P_i(k)$ or one of its descendents returned YES,
2. at least one satisfaction function of $P_i(k)$ or one of its descendents returned YES(S), or

3. $MP(k+1)$ returned $YES(S)$ or $YES+MAYBE(S)$.

Proof:

Assume that $P_i(k)$ could return $YES(S)$ or $YES+MAYBE(S)$ without one of the above conditions being true. $P_i(k)$ can return $YES(S)$ or $YES+MAYBE(S)$ only by making transition 10, 41, 11, 31, 33, or 42; each case is considered below.

Case 1: $P_i(k)$ makes transition 31, 33, or 42

$P_i(k)$ can make one of these only if it is in state 12, but 12 can only be reached using transition 26 or 27. Transition 27 requires a function to have returned $YES(S)$ but this would contradict Condition (2). Therefore, $P_i(k)$ must make transition 26 if it is to make either transition 31, 33, or 42; each case is considered below.

Case 1.1: $P_i(k)$ makes transition 31

This requires $V_{P_i(k)}$ to be set to 1, in which case $P_i(k)$ must have made transition 27, 24, 11, 41, 33 or 42, or 7. It has been shown that transition 27 cannot be made and transition 24 cannot be made without contradicting the Condition (1). As well, $P_i(k)$ cannot make transition 11, 41, 33 or 42 because it has been stated that $P_i(k)$ replies using transition 31. Therefore, $P_i(k)$ must make transition 7 after some child $P_j(k)$ returns YES.

By Lemma E1, if $P_j(k)$ returns YES, then at least one satisfaction function of the procedure or its descendents returned YES. This violates Condition (1) and so $P_i(k)$ cannot make transition 7. It follows that $P_i(k)$ cannot make transition 31.

Case 1.2: $P_i(k)$ makes transition 33 or 42

$P_i(k)$ can make these transitions only if $MP(k+1)$ returns $YES+MAYBE(S)$ or $YES(S)$ but this would violate Condition (3). Therefore, $P_i(k)$ cannot make transition 33 or 42.

Case 2: $P_i(k)$ makes transition 10

This requires that $V_{P_i(k)}$ be set to 1 but it was shown in Case 1.1 that $V_{P_i(k)}$ cannot be set to 1 without violating the stated conditions. Therefore, $P_i(k)$ cannot make transition 10.

Case 3: $P_i(k)$ makes transition 11 or 41

$P_i(k)$ can make one of these transitions only if some child $P_j(k)$ returns YES(S) or YES+MAYBE(S). However, the same arguments that applied to $P_i(k)$ now apply to $P_j(k)$. $P_j(k)$ cannot make transition 31, 33, 42, or 10 and so it must return YES(S) or YES+MAYBE(S) after one of its children returns the same.

These arguments repeat indefinitely and every time that they are used, one more procedure at another level deeper in the hierarchy must make transition 11 or 41 after one of its children returns YES(S) or YES+MAYBE(S). Since the number of procedures in the solver is finite, eventually a childless procedure must be encountered that cannot make transition 11 or 41 and so a contradiction is raised. Thus, $P_i(k)$ cannot make transition 11 or 41.

If $P_i(k)$ cannot make transition 31, 33, 42, 10, 11, or 41, a contradiction arises and so the original assumption must be wrong. Therefore, $P_i(k)$ will return YES(S) or YES+MAYBE(S) only if the stated conditions hold.

Q.E.D.

Lemma F1

If a procedure $P_a(k)$ returns NO in response to a CM M_i , then a satisfaction function F of $P_a(k)$ or one of its descendants determined that L_{M_i} contained non-feasible constraints.

Proof:

Assume that $P_a(k)$ returns NO but no function of $P_a(k)$ or one of its descendants determined that the constraints in L_{M_i} were nonfeasible. By Lemma E2, if $P_a(k)$ returns NO, a satisfaction function F of $P_a(k)$ or one of its descendants returned NO. Let $P_j(k)$ denote the procedure to which F belongs. $P_j(k)$ may or may not have called other functions before calling F ; each case is considered below.

Case 1: F was the first function called

By Lemma A2, F was passed a subset of the constraints in L_{M_i} , and by Restriction SF3 F can return NO only if it was passed a nonfeasible set of constraints or

all possible solutions to the set had already been generated. Since no other solutions were generated, the set must be nonfeasible but this contradicts the original assumption. Thus, F could not have been the first function called.

Case 2: F was not the first function called

Restriction SF3 indicates that F can return NO only if it was passed a nonfeasible set of constraints or all possible solutions to the set recognized by $P_j(k)$ had already been generated. It must be shown that all of these solutions were nonfeasible.

Assume that one of the solutions which was generated by a previously called function was feasible. Let L_{M_b} denote such a solution. $P_j(k)$ must have returned to state 9 after calling each function in order to call F . Thus, $MP(k+1)$ must have returned NO after being passed L_{M_b} . $MP(k+1)$ could return NO only by making transition 14, 26, or 21; each case is considered below.

Case 2.1: $MP(k+1)$ made transition 14 or 26

$MP(k+1)$ can make one of these only if it detects a nonfeasible constraint in $PS_{M_b}^*$. Since $MP(k+1)$ can generate unique solutions only, this contradicts the statement that L_{M_b} contained a feasible solution. Thus, $MP(k+1)$ could not return NO using transition 14 or 26.

Case 2.2: $MP(k+1)$ made transition 21

If $MP(k+1)$ made transition 21, some child $Pz(k+1)$ returned NO. The above arguments now repeat with $Pz(k+1)$ replacing $Pa(k)$. Consider the general case. If $MP(k+i)$, $i > 0$ returned NO in response to a CM M_a by making transition 21, some child $Pz(k+i)$ returned NO in response to a CM. By Lemma E2, if $Pz(k+i)$ returned NO, then a satisfaction function F_y of $Pz(k+i)$ or one of its descendents returned NO. Let $Py(k+i)$ denote the procedure to which F_y belongs.

If F_y was the first function to have been called by $Py(k+i)$, then $PS_{M_a}^*$ contained nonfeasible constraints but this contradicts the assumption the L_{M_a} contained a feasible solution. If (1) F_y was not the first function called by $Py(k+i)$, (2) Restriction SF3 is to hold, and (3) $PS_{M_a}^*$ is feasible, then F_y can return NO only if all possible solutions to the set recognized by $Py(k+i)$ had already been generated.

Thus, a feasible solution must have been generated by a previously called function of $Py(k+i)$; let M_b denote such a solution.

$Py(k+i)$ must have returned to state 9 after calling each function in order to call F_y , thus, $MP(k+i+1)$ must have returned NO after being passed L_{M_b} . $MP(k+i+1)$ could return NO only by making transition 14, 26, or 21, but it cannot make either of the first two without raising a contradiction. Thus, it must make transition 21 after some child $Px(k+i+1)$ returns NO.

These arguments repeat indefinitely but by Lemma B2, there exists a value of n such that no procedure $Px(k+n)$ will be able to make a recursive call. If $MP(k+n)$ returns NO in response to a CM M_c containing a feasible solution, it must do so by making transition 21 after some descendent $Pv(k+n)$ calls a function F_v and F_v returns NO. If $Pv(k+n)$ did not make any recursive calls, then either (1) F_v returned NO before all possible solutions were generated, or (2) the constraints in $Pv(k+n)$'s problem set were nonfeasible. The first case cannot be true without violating Restriction SF3, and the second indicates that $Pv(k+n)$'s problem set was nonfeasible, in which case $PS_{M_c}^4$ contained nonfeasible constraints and a contradiction is raised. If neither (1) nor (2) is possible, it follows that $MP(k+1)$ could not have made transition 21.

If $MP(k+1)$ could not make transition 14, 26, or 21, it could not have returned NO and so a contradiction arises. Thus, no other function could have been called before F .

The result of Cases 1 and 2 are contradictory and so the original assumption must be wrong. Thus, if a procedure $Pa(k)$ returns NO in response to a CM M_i , then L_{M_i} contained nonfeasible constraints.

Q.E.D.

Lemma F2

If $MP(k)$ returns NO in response to a CM M_i , then L_{M_i} contains nonfeasible constraints.

Thus, a feasible solution must have been generated by a previously called function of $Py(k+i)$; let M_b denote such a solution.

$Py(k+i)$ must have returned to state 9 after calling each function in order to call F_v , thus, $MP(k+i+1)$ must have returned NO after being passed L_{M_b} . $MP(k+i+1)$ could return NO only by making transition 14, 26, or 21, but it cannot make either of the first two without raising a contradiction. Thus, it must make transition 21 after some child $Px(k+i+1)$ returns NO.

These arguments repeat indefinitely but by Lemma B2, there exists a value of n such that no procedure $Px(k+n)$ will be able to make a recursive call. If $MP(k+n)$ returns NO in response to a CM M_c containing a feasible solution, it must do so by making transition 21 after some descendent $Pv(k+n)$ calls a function F_v and F_v returns NO. If $Pv(k+n)$ did not make any recursive calls, then either (1) F_v returned NO before all possible solutions were generated, or (2) the constraints in $Pv(k+n)$'s problem set were nonfeasible. The first case cannot be true without violating Restriction SF3, and the second indicates that $Pv(k+n)$'s problem set was nonfeasible, in which case $PS_{M_c}^4$ contained nonfeasible constraints and a contradiction is raised. If neither (1) nor (2) is possible, it follows that $MP(k+1)$ could not have made transition 21.

If $MP(k+1)$ could not make transition 14, 26, or 21, it could not have returned NO and so a contradiction arises. Thus, no other function could have been called before F .

The result of Cases 1 and 2 are contradictory and so the original assumption must be wrong. Thus, if a procedure $Pa(k)$ returns NO in response to a CM M_i , then L_{M_i} contained nonfeasible constraints.

Q.E.D.

Lemma F2

If $MP(k)$ returns NO in response to a CM M_i , then L_{M_i} contains nonfeasible constraints.

Proof:

Assume that $MP(k)$ returns NO but the constraints in L_{M_i} are feasible. $MP(k)$ can return NO only by making transition 14, 26, or 21; each case is considered below.

Case 1: $MP(k)$ makes transition 14

$MP(k)$ can make transition 14 only if $PS_{M_i}^+$ contains some MP-rejectable constraint c . L_{M_i} cannot contain an MP-rejectable constraint if its contents are feasible but by Corolary C6, if $c \ni L_{M_i}$, then $c \ni PS_{M_i}^+$. This raises a contradiction and so $MP(k)$ cannot make transition 14.

Case 2: $MP(k)$ makes transition 26

$MP(k)$ can make transition 26 only if $PS_{M_i}^*$ contains some MP-rejectable constraint c . L_{M_i} cannot contain such a constraint and by Corolary C6, if $c \ni L_{M_i}$, then $c \ni PS_{M_i}^+$. Thus, c must be the result of $MP(k)$ reducing a constraint using values determined using transitions 24 and possibly 3. However, since these values are the only ones possible, the constraints in L_{M_i} cannot be feasible. This raises a contradiction and so $MP(k)$ cannot make transition 26.

Case 3: $MP(k)$ makes transition 21

$MP(k)$ can make transition 21 only if some child $Pj(k)$ returns NO. However, by Lemma F1, if $Pj(k)$ returns NO, then the constraints in $PS_{M_i}^4$ are nonfeasible. Since $PS_{M_i}^4$ is the result of applying the only possible solutions to $PS_{M_i}^+$, L_{M_i} must contain nonfeasible constraints and so a contradiction arises. Thus, $MP(k)$ cannot make transition 21.

If $MP(k)$ cannot make transition 14, 26, or 21, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ returns NO in response to a CM M_i , then the constraints in L_{M_i} are nonfeasible.

Q.E.D.

Lemma F3

If a procedure $Pa(k)$ returns NO in response to an AVR regarding some CM M_i , then no solutions to L_{M_i} are possible other than the ones that have been returned already.

Proof:

Assume that $Pa(k)$ is sent an AVR and it returns NO, but there is some solution to L_{M_i} that has not yet been returned. $Pa(k)$ can return NO after receiving an AVR only by making transition 12 or 29. Each case is considered below.

Case 1: $Pi(k)$ makes transition 29

$Pi(k)$ could make transition 29 only if it called some function F and F returned NO. $Pi(k)$ could have received the AVR while in state 5 or 11, and so F may or may not have been the first function called by $Pi(k)$; each case is considered below.

Case 1.1: F was the first function called

By Restriction SF3, F can return NO only if it was passed a nonfeasible set of constraints or all possible solutions to the set had already been generated. Since no other solutions could have been generated if F was the first function called, then the set must be nonfeasible. However, this contradicts the original assumption and some other function must have been called before F .

Case 1.2: F was not the first function called

Restriction SF3 indicates that F can return NO only if it was passed a nonfeasible set of constraints or all possible solutions to the set which was recognized by $Pi(k)$ had already been generated. It must be shown that none of these solutions was feasible.

Assume that some solution L_{M_b} generated by a previously called function of $Pi(k)$ was feasible. If $Pi(k)$ called F , it must have returned to state 9 after calling each function and so $MP(k+1)$ must have returned NO after being passed L_{M_b} . However, by Lemma F2, if $MP(k+1)$ returned NO, the constraints in L_{M_b} were nonfeasible. This raises a contradiction and so no other function could have been called before F .

The results of Cases 1.1 and 1.2 are contradictory and so $Pi(k)$ could not have returned NO using transition 29.

Case 2: $P_i(k)$ makes transition 12

$P_i(k)$ can make transition 12 only if some child $P_j(k)$ returns NO. $P_j(k)$ may have been sent an AVR (if it returned the last solution), or a CM (if $P_j(k)$ is to the right of the procedure which returned the last solution); each case is considered below.

Case 2.1: $P_j(k)$ was sent a CM

If $P_j(k)$ returned NO in response to a CM, Lemma F1 indicates that it was sent a nonfeasible list of constraints and so there are no solutions. This raises a contradiction and so $P_j(k)$ could not return NO in response to a CM.

Case 2.2: $P_j(k)$ was sent an AVR

If $P_j(k)$ returned NO in response to an AVR, the above arguments repeat with $P_j(k)$ replacing $P_i(k)$. If it made transition 29 after some function returned NO, then no other solutions are possible; if it makes transition 12 after some child $P_m(k)$ returns NO, then $P_m(k)$ returned NO after having been sent a CM or an AVR. If $P_m(k)$ had been sent a CM, no solutions are possible and so it must have returned NO after having been sent an AVR.

The above arguments repeat indefinitely and every time that they are used, some procedure must make transition 12 after a child returns NO in response to an AVR. Since the number of procedures in the solver is finite, eventually a childless procedure will be encountered that cannot make transition 12. This raises a contradiction and so if another solution exists, $P_j(k)$ could not have returned NO in response to an AVR.

If $P_j(k)$ could not have returned NO in response to a CM or an AVR, a contradiction arises and so $P_i(k)$ could have not returned NO using transition 12.

If $P_i(k)$ could not have made transition 12 or 29, it could not have returned NO. This raises a contradiction and so the original assumption must be wrong. Thus, if a procedure $P_a(k)$ returns NO in response to an AVR regarding some CM M_i , then no solutions to L_{M_i} are possible other than the ones that have been returned

already.

Q.E.D.

Lemma F4

If a procedure $Pi^+(k)$ returns MAYBE in response to a CM, then any descendent which was sent a CM returned MAYBE.

Proof:

Assume that $Pi^+(k)$ returns MAYBE but it has a descendent $Pj(k)$ which was sent a CM but it did not return MAYBE. $Pi^+(k)$ could be some procedure $Pi(k)$ or $MP(k)$; each case is considered below.

Case 1: $Pi^+(k)$ is some procedure $Pi(k)$

$Pj(k)$ could be a child of $Pi(k)$ or some other descendent; each case is considered below.

Case 1.1: $Pj(k)$ is a child of $Pi(k)$

In order to return MAYBE, $Pi(k)$ must make transition 2, 18, or 20. If it were to make transition 2, it would not call any of its children and so for $Pj(k)$ to exist as stated, $Pi(k)$ must make transition 18 or 20 after having exhausted its list of children. It could make transition 18 or 20 only if $V_{Pi(k)}$ was set to 0 and so all of its children must return MAYBE, including $Pj(k)$. This raises a contradiction and so $Pj(k)$ cannot be a child of $Pi(k)$.

Case 1.2: $Pj(k)$ is some other descendent of $Pi(k)$

Let $Pq(k)$ be the child of $Pi(k)$ at the root of the subtree containing $Pj(k)$. As stated in Case 1.1, $Pi(k)$ can return MAYBE using transition 2, 18, or 20, but for $Pj(k)$ to exist, $Pi(k)$ must make transition 18 or 20 after all of its children are called and return MAYBE, including $Pq(k)$.

If $Pq(k)$ returns MAYBE, it must make transition 2, 18, or 20 but now the above arguments repeat. $Pq(k)$ cannot make transition 2 because it would prevent $Pj(k)$ from being sent a CM, therefore it must make transition 18 or 20 after all of its children return MAYBE. By induction, every descendent of $Pi(k)$ that is

an ancestor of $P_j(k)$ must make transition 18 or 20 so that $P_j(k)$ is sent a CM. However, a contradiction arises if the parent of $P_j(k)$ must make transition 18 or 20 and $P_j(k)$ returns a response other than MAYBE as assumed. Thus, $P_j(k)$ cannot be some other descendent of $P_i(k)$.

If $P_j(k)$ cannot be a child of $P_i(k)$ or some other descendent, a contradiction arises and so $P_i^+(k)$ cannot be some procedure $P_i(k)$.

Case 2: $P_i^+(k)$ is $MP(k)$

If $P_i^+(k)$ is $MP(k)$, $MP(k)$ can return MAYBE only by making transition 5 after all of its children return MAYBE. Let $P_q(k)$ denote the child of $MP(k)$ at the root of the subtree containing $P_j(k)$. By the results of Case 1, if $P_q(k)$ returns MAYBE, $P_j(k)$ returned MAYBE if it was sent a CM. This raises a contradiction and so $P_i^+(k)$ cannot be $MP(k)$.

If $P_i^+(k)$ cannot be $P_i(k)$ or $MP(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i^+(k)$ returns MAYBE, any descendent which was sent a CM returned MAYBE.

Q.E.D.

Lemma F5

If a procedure $P_i(k)$ returns MAYBE in response to an AVR R_a regarding a CM M_a , then all of its children returned MAYBE in response to a CM or an AVR before $P_i(k)$ returned MAYBE.

Proof:

Assume that $P_i(k)$ returns MAYBE in response to an AVR but it has a some child $P_j(k)$ which was sent a CM and zero or more AVR's but it did not return MAYBE. In order to return MAYBE, $P_i(k)$ must make transition 2, 18, or 20. $P_i(k)$ can make transition 2 only after having reached state 1, and state 1 can be reached only by making transition 1, 16, 22, 36, or 37 after having received a CM. Thus, $P_i(k)$ cannot return MAYBE in response to an AVR using transition 2. Thus, for $P_j(k)$ to exist as stated, $P_i(k)$ must make transition 18 or 20 after having

exhausted its list of children. It could make transition 18 or 20 only if $V_{Pi(k)}$ was set to 0 and so all of its children must have been a CM or an AVR and returned MAYBE, including $Pj(k)$. This raises a contradiction and so the assumption must be wrong. Thus, if a procedure $Pi(k)$ returns MAYBE in response to an AVR R_a regarding a CM M_a , then all of its children returned MAYBE in response to a CM or an AVR before $Pi(k)$ returned MAYBE.

Q.E.D.

Lemma F6

If a procedure $Pi(k)$ returns YES in response to an AVR R_a regarding a CM M_a , then all of its children returned MAYBE or YES in response to a CM or an AVR before $Pi(k)$ returned YES.

Proof:

Assume that $Pi(k)$ returns YES in response to an AVR but it has a some child $Pj(k)$ which was sent a CM and zero or more AVR's but it did not return MAYBE or YES. In order to return YES, $Pi(k)$ must make transition 2, 18, 39, 19, or 20. $Pi(k)$ can make transition 2 only after having reached state 1, and state 1 can be reached only by making transition 1, 16, 22, 36, or 37 after having received a CM. Thus, $Pi(k)$ cannot return MAYBE in response to an AVR using transition 2. Thus, for $Pj(k)$ to exist as stated, $Pi(k)$ must make transition 18, 39, 19, or 20 after having exhausted its list of children and all returned MAYBE or YES, including $Pj(k)$. This raises a contradiction and so the assumption must be wrong. Thus, if a procedure $Pi(k)$ returns YES in response to an AVR R_a regarding a CM M_a , then all of its children returned MAYBE or YES in response to a CM or an AVR before $Pi(k)$ returned MAYBE.

Q.E.D.

Lemma F7

If a procedure $Pi(k)$ returns MAYBE in response to a CM M_i , then $Pi(k)$ nor any of its descendents had a satisfaction function which could:

1. determine that the constraints in L_M , were feasible or nonfeasible,

2. generate a definite partial or full solution to a subset of the constraints in L_{M_i} ,
or
3. generate a tentative partial or full solution to a subset of them which was not subsequently rejected by $MP(k+1)$.

Proof:

Each result is considered below.

Result 1: *Determining that the constraints are feasible or nonfeasible*

Assume that some procedure $P_j(k)$ in the same subtree as $P_i(k)$ had a function F which can determine if the constraints in L_{M_i} were feasible or nonfeasible; $P_j(k)$ could be $P_i(k)$ or one of its descendants. $P_j(k)$ was or was not sent a CM; each case is considered below.

Case 1.1: $P_j(k)$ was sent a CM

By Lemma F4, if $P_i(k)$ returned MAYBE, then $P_j(k)$ returned MAYBE using transition 2, 18, or 20. It could make transition 2 only if it failed to accept any constraints that were passed to it, but this would violate Restriction RS1. $P_j(k)$ can make transition 18 only if its sufficiency predicate fails, but this would violate Restriction RS3. Thus, $P_j(k)$ could not have made transition 2 or 18.

$P_j(k)$ could make transition 20 only if it had called all of its functions and $V_{P_j(k)}$ is set to 0. This requires all functions to have returned MAYBE or MAYBE(S), otherwise $V_{P_j(k)}$ would be set to 1 (if any function returned YES or YES(S)), or $P_j(k)$ would return NO (if any function returned NO). However, given the above conditions, Corolary A4.1 indicates that F would return YES. This raises a contradiction and so $P_i(k)$ could not have made transition 20. It follows that $P_j(k)$ could not have been sent a CM.

Case 1.2: $P_j(k)$ was not sent a CM

If $P_j(k)$ was not sent a CM, it cannot be $P_i(k)$ and so it must be one of its descendants. Let $P_z(k)$ denote the parent of $P_j(k)$; $P_z(k)$ could be $P_i(k)$ or one of its descendants. $P_z(k)$ was or was not sent a CM; each case is considered below.

Case 1.2.1: $Pz(k)$ was sent a CM

By Lemma F4, if $Pz(k)$ was sent a CM M_z , it would have returned MAYBE using transition 2, 18, or 20. It could avoid sending a CM to $Pj(k)$ only by making transition 2, and it could make transition 2 only by failing to accept any data in L_z . However, this would violate Restriction RS1 and so $Pz(k)$ could not have been sent a CM.

Case 1.2.2: $Pz(k)$ was not sent a CM

Let $Pq(k)$ denote the parent of $Pz(k)$. The above arguments may now be repeated with $Pq(k)$ replacing $Pz(k)$. $Pq(k)$ was or was not sent a CM. If it was sent a CM, it must have returned MAYBE using transition 2 so as not to send a CM to $Pz(k)$. However, this violates Restriction RS1 and so it could not have been sent a CM. These arguments repeat indefinitely and so no ancestor of $Pj(k)$ could have been sent a CM, including $Pi(k)$. However, it was stated that $Pi(k)$ was sent a CM and so a contradiction arises. Therefore, $Pj(k)$ must have been sent a CM.

A contradiction arises if $Pj(k)$ was or was not sent a CM and so the original assumption must be wrong. Therefore, if $Pi(k)$ returns MAYBE, $Pi(k)$ nor any of its descendants had a satisfaction function which could determine that the constraints in L_{M_i} were feasible or nonfeasible.

Result 2: *Generating a definite solution*

Assume that some procedure $Pj(k)$ in the same subtree as $Pi(k)$ had a function F which could generate a definite partial or full solution to a subset of the constraints in L_{M_i} ; that is, F would have returned YES(S) if called. $Pj(k)$ could be $Pi(k)$ or one of its descendants. $Pj(k)$ was or was not sent a CM; each case is considered below.

Case 2.1: $Pj(k)$ was sent a CM

By Lemma F4, if $Pi(k)$ returned MAYBE, any descendants which was sent a CM returned MAYBE. If $Pj(k)$ returned MAYBE, it made transition 2, 18, or 20. The arguments that were used in Case 1.1 may now be repeated to show that it cannot make transition 2 or 18, and it can make transition 20 only if it has exhausted all

of its functions and $V_{Pj(k)}$ is set to 0. This prevents F from returning YES(S) but given the above conditions, Corolary A4.2 indicates that F will return YES(S). This raises a contradiction and so $Pj(k)$ could not have been sent a CM.

Case 2.2: $Pj(k)$ was not sent a CM

Similarly, the arguments that were used in Case 1.2 may be used to show that if $Pj(k)$ was not sent a CM and Restriction RS1 was not violated, then none of its ancestors could have been sent a CM. Since $Pi(k)$ is at the root of the subtree containing $Pj(k)$ and it was sent a CM, a contradiction arises and so $Pj(k)$ must have been sent a CM.

The above results are contradictory and so the original assumption must be wrong. Therefore, if $Pi(k)$ returns MAYBE, then $Pi(k)$ nor any of its descendents had a satisfaction function F which could generate a definite partial or full solution to a subset of the constraints in L_{M_i} .

Result 3: *Generating a tentative solution*

Assume that some procedure $Pj(k)$ in the same subtree as $Pi(k)$ had a function F which could generate a tentative partial or full solution to a subset of the constraints in L_{M_i} which would not be rejected by MP($k+1$); that is, F would return MAYBE(S) if called. $Pj(k)$ could be $Pi(k)$ or one of its descendents. $Pj(k)$ was or was not sent a CM; each case is considered below.

Case 3.1: $Pj(k)$ was sent a CM

By Lemma F4, if $Pi(k)$ returned MAYBE, any descendent which was sent a CM returned MAYBE. If $Pj(k)$ returned MAYBE, it made transition 2, 18, or 20. The arguments that were used in Case 1.1 may now be repeated to show that it could not make transition 2 or 18, and it could make transition 20 only if it had returned to state 9 after calling all of its functions, including F . This requires MP($k+1$) to reject any tentative solution generated by F and so a contradiction arises. Thus, $Pj(k)$ could not have been sent a CM.

Case 3.2: $P_j(k)$ was not sent a CM

Similarly, the arguments that were used in Case 1.2 may be used to show that if $P_j(k)$ was not sent a CM and Restriction RS1 was not violated, then none of its ancestors could have been sent a CM. Since $P_i(k)$ is at the root of the subtree containing $P_j(k)$ and it was sent a CM, a contradiction arises and so $P_j(k)$ must have been sent a CM.

The above results are contradictory and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns MAYBE, then $P_i(k)$ nor any of its descendants had a satisfaction function F which could generate a tentative partial or full solution to a subset of the constraints in L_{M_i} which was not subsequently rejected by $MP(k+1)$.

Q.E.D.

Lemma F8

If $MP(k)$ returns MAYBE in response to a CM M_i , then:

1. $MP(k)$ could not determine if the constraints in L_{M_i} were feasible or nonfeasible,
2. $MP(k)$ could not generate any solutions to any subset of constraints in $PS_{M_i}^+$,
3. no procedure $Pa(k)$ had a satisfaction function which could determine if the constraints in L_{M_i} were feasible or nonfeasible,
4. no procedure $Pa(k)$ had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Proof:

Each result is considered below. Note that $MP(k)$ could return MAYBE only by making transition 5 and if it had not determined any values using transitions 24

and 3. Thus, in order to reach state 4 after receiving CM M_i , it must have made transition 2 and not transition 4.

Result 1: *MP(k) could not determine the constraint feasibility*

Assume that $MP(k)$ could determine if the constraints in L_{M_i} were feasible or nonfeasible. In order to determine that they were feasible or nonfeasible, $MP(k)$ would have to:

- detect that all constraints in $PS_{M_i}^+$ were satisfied, in which case it would make transition 15 from state 7,
- it would have to totally reduce and satisfy every constraint in $PS_{M_i}^+$ using transitions 24 and 3, in which case it would make transition 25,
- detect a contradictory constraint in $PS_{M_i}^+$, in which case it would make transition 14, or
- detect a contradiction in $PS_{M_i}^*$, in which case it would make transition 26.

These results indicate that if $MP(k)$ could determine that the constraints in L_{M_i} were feasible or nonfeasible, it would have to make transition 14, 15, or 24 from state 7. It was stated in the beginning that in order to make transition 5, $MP(k)$ would first have to make transition 2 from state 7. This raises a contradiction and so if $MP(k)$ returns MAYBE, then it could not determine if the constraints in L_{M_i} were feasible or nonfeasible.

Result 2: *MP(k) could not generate any solutions*

Assume that $MP(k)$ could generate one or more solutions to a subset of constraints in $PS_{M_i}^+$. It could do so only by making transitions 24 and 3, but it was stated in the beginning that $MP(k)$ could not make these transitions. This raises a contradiction and so the assumption must be wrong. Thus, if $MP(k)$ returns MAYBE, then the result follows.

Result 3: *Pi(k) could not determine the constraint feasibility*

Assume that some procedure $Pi(k)$ had a satisfaction function which could determine if the constraints in L_{M_i} were feasible or nonfeasible. Let $Pj(k)$ denote the

child of $MP(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ could be $Pj(k)$ or one of its descendants.

$MP(k)$ could make transition 5 only if it has exhausted all of its children and $V_{MP(k)}$ is set to 0. This indicates that each of its children returned MAYBE when it was sent a CM. If $Pj(k)$ returned MAYBE, Lemma F7 indicates that $Pj(k)$ nor any of its descendants had a satisfaction function which could determine if the constraints that were sent to $Pj(k)$, i.e., $PS_{M_i}^4$, were feasible or nonfeasible. Thus, $Pj(k)$ nor any of its descendants had a function which could determine that the constraints in L_{M_i} were feasible or nonfeasible. This raises a contradiction and so the result follows.

Result 4: *No procedure $Pa(k)$ could generate any substitutions*

Assume that some procedure $Pi(k)$ had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$. Let $Pj(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ could be $Pj(k)$ or one of its descendants.

$MP(k)$ can make transition 5 only if it has exhausted all of its children and $V_{MP(k)}$ is set to 0. This indicates that each of its children returned MAYBE when it was sent a CM. If $Pj(k)$ returned MAYBE, Lemma F7 indicates that $Pj(k)$ nor any of its descendants had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints that were sent to $Pj(k)$, i.e., $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$. This raises a contradiction and so the result follows.

Q.E.D.

Lemma F9

If $Pi(k)$ is sent a CM and it returns YES, any descendent which was sent a CM returned MAYBE or YES.

Proof:

Assume that a procedure $Pi(k)$ returns YES but some descendent $Pj(k)$ returned a response other than MAYBE or YES after it was sent a CM. $Pi(k)$ can return

YES only by making transition 39 or 19, and it cannot make either unless all of its children return MAYBE or YES. (Otherwise, $P_i(k)$ would make transition 9, 10, 11, 12, or 41 and return a response other than YES).

Let $P_z(k)$ denote the child of $P_i(k)$ at the root of the subtree containing $P_j(k)$. By Lemma F4, if $P_z(k)$ returns MAYBE, $P_z(k)$ and any descendent which was sent a CM returned MAYBE. Therefore, $P_z(k)$ must return YES and $P_j(k)$ must be one of its descendents.

The above arguments now repeat with $P_z(k)$ replacing $P_i(k)$. $P_z(k)$ can return YES only by making transition 39 or 19, and it cannot make either unless all of its children return MAYBE or YES. Let $P_r(k)$ be the child of $P_z(k)$ at the root of the subtree containing $P_j(k)$. By Lemma F4, if $P_r(k)$ returns MAYBE, $P_r(k)$ and any descendent which was sent a CM returned MAYBE. Therefore, $P_r(k)$ must return YES and $P_j(k)$ must be one of its descendents.

These arguments repeat indefinitely and every time that they are used, $P_j(k)$ must be at another level deeper in the hierarchy. However, since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_j(k)$. This raises a contradiction and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns YES, any descendent which was sent a CM returned MAYBE or YES.

Q.E.D.

Lemma F10

If a procedure $P_i(k)$ returns YES in response to a CM M_a , then:

1. the constraints in M_a are feasible.
2. $P_i(k)$ nor any of its descendents had a satisfaction function which could generate a definite partial or full solution to a subset of the constraints in L_{M_a} , and
3. $P_i(k)$ nor any of its descendents had a satisfaction function which could generate a tentative partial or full solution to a subset of the constraints in L_{M_a} which was not subsequently rejected by $MP(k+1)$.

Proof:

Each result is considered below.

Result 1: *The constraints in L_{M_a} are feasible*

Assume that the constraints in L_{M_a} are nonfeasible. By Lemma E1, if $Pi(k)$ returned YES, then a satisfaction function F of $Pi(k)$ or one of its descendants returned YES. By Restriction SF5, F would return YES only if (1) it was passed every constraint in $PS_{M_b}^4$ (where M_b denotes the last CM which was sent to $MP(k)$), and (2) these constraints were feasible. This indicates that the problem set of $Pi(k)$ contains every constraint in $PS_{M_b}^4$, and by Corolary A1.2, the problem set of each of its ancestors must also contain these same constraints. Since L_{M_a} consists of the problem set of $Pi(k)$'s parent, that is every constraint in $PS_{M_b}^4$ and F has determined these to be feasible, a contradiction is raised and so the assumption must be wrong. Thus, the result holds.

Result 2: *Generating a definite solution*

Assume that some procedure $Pj(k)$ in the same subtree as $Pi(k)$ had a function F which could generate a definite partial or full solution to a subset of the constraints in L_{M_i} ; $Pj(k)$ could be $Pi(k)$ or one of its descendants. $Pj(k)$ was or was not sent a CM; each case is considered below.

Case 2.1: *$Pj(k)$ was sent a CM*

By Lemma F9, if $Pi(k)$ returns YES, any descendent which was sent a CM returned MAYBE or YES. The arguments that were used in Case 1.1 may be repeated to show that if $Pj(k)$ was sent a CM, it returned MAYBE or YES using transition 2, 18, 39, 19, or 20. It could not have made transition 2, 18, or 39, and it could make transition 19 or 20 only if it returned to state 9 after calling all of its functions. This indicates that if F returned YES(S), $Pj(k)$ must have recursively called $MP(k+1)$ and received a NO reply. However, by Lemma F2, if $MP(k+1)$ returns NO, then S contained nonfeasible constraints. This contradicts the assumption that F could generate a definite partial or full solution to a subset of the constraints in L_{M_i} , and so $Pj(k)$ could not have been sent a CM.

Case 2.2: $P_j(k)$ was not sent a CM

The arguments that were used in Case 1.2 may be used again to show that if $P_j(k)$ was not sent a CM and Restriction RS1 was not violated, then none of its ancestors could have been sent a CM. Since $P_i(k)$ is at the root of the subtree containing $P_j(k)$ and it was sent a CM, a contradiction arises and so $P_j(k)$ must have been sent a CM.

The above results are contradictory and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns YES, then $P_i(k)$ nor any of its descendants had a satisfaction function F which can generate a definite partial or full solution to a subset of the constraints in L_M .

Result 3: *Generating a tentative solution*

Assume that some procedure $P_j(k)$ in the same subtree as $P_i(k)$ had a function F which could generate a tentative partial or full solution to a subset of the constraints in L_M , which would not be rejected by $MP(k+1)$; $P_j(k)$ could be $P_i(k)$ or one of its descendants. $P_j(k)$ was or was not sent a CM; each case is considered below.

Case 3.1: $P_j(k)$ was sent a CM

By Lemma F9, if $P_i(k)$ returns YES, $P_j(k)$ returned MAYBE or YES if it was sent a CM. The arguments that were used in Case 1.1 may be used again to show that if $P_j(k)$ was sent a CM, it returned MAYBE or YES using transition 19 or 20 after calling all of its satisfaction functions and returning to state 9. This indicates that if F returned MAYBE(S), $P_j(k)$ must have recursively called $MP(k+1)$ and received a NO reply so that it could return to state 9. This contradicts the assumption that F could generate a tentative partial or full solution to a subset of the constraints in L_M , which would not be rejected by $MP(k+1)$ and so $P_j(k)$ could not have been sent a CM.

Case 3.2: $P_j(k)$ was not sent a CM

Similarly, the arguments that were used in Case 1.2 may be used to show that if $P_j(k)$ was not sent a CM and Restriction RS1 was not violated, then none of its ancestors could have been sent a CM. Since $P_i(k)$ is at the root of the subtree

containing $P_j(k)$ and it was sent a CM, a contradiction arises and so $P_j(k)$ must have been sent a CM.

A contradiction arises if $P_j(k)$ was or was not sent a CM and so the original assumption must be wrong. Thus, if $P_i(k)$ returns YES, $P_i(k)$ nor any of its descendants had a satisfaction function which could generate a tentative partial or full solution to a subset of the constraints in L_{M_i} which was not subsequently rejected by $MP(k+1)$.

Q.E.D.

Lemma F11

If $MP(k)$ returns YES in response to a CM M_i , then:

1. the constraints in L_{M_i} are feasible,
2. $MP(k)$ could not generate any definite or tentative solutions to a subset of the constraints in $PS_{M_i}^+$,
3. no procedure $Pa(k)$ could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Proof:

Each result is considered below. Note that $MP(k)$ can return YES only by making transition 15 or 6, and it can make transition 6 only if it has not determined any values using transitions 24 and 3. Thus, if it reaches state 4 after receiving CM M_i , it must make transition 2 and not transition 4.

Result 1: *Determining that the constraints are feasible*

Assume that $MP(k)$ returns YES in response to a CM M_i but there exists a constraint $c \in L_{M_i}$ that is nonfeasible. $MP(k)$ can return YES only by making transition 15 or 6. $MP(k)$ will make transition 15 only if every constraint in $PS_{M_i}^+$ is satisfied, including c . This raises a contradiction and so $MP(k)$ cannot make transition 15.

$MP(k)$ can make transition 6 only if it has exhausted its list of children and $V_{MP(k)}$ is set to 1. This indicates that each of its children returned MAYBE or YES when it was sent a CM, and at least one returned YES. Let $Pa(k)$ be any child of $MP(k)$ which returned YES. By Lemma F10, if $Pa(k)$ returns YES, then the constraints in $PS_{M_i}^4$ are feasible and so a contradiction is raised. Therefore, $MP(k)$ cannot make transition 6. Since it could not make transition 15 either, the original assumption must be wrong and so the constraints in L_{M_i} must be feasible.

Result 2: *$MP(k)$ could not generate any solutions*

Assume that $MP(k)$ could generate one or more solutions to a subset of constraints in $PS_{M_i}^+$. It could do so only by making transitions 24 and 3, but it was stated in the beginning that $MP(k)$ could not make these transitions and still return YES. This raises a contradiction and so the result follows.

Result 3: *No procedure $Pa(k)$ could generate any substitutions*

Assume that some procedure $P_i(k)$ had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$. Let $P_j(k)$ denote the child of $MP(k)$ at the root of the subtree containing $P_i(k)$; $P_i(k)$ could be $P_j(k)$ or one of its descendants.

$MP(k)$ can make transition 6 only if it has exhausted all of its children and $V_{MP(k)}$ is set to 1. This indicates that each of its children returned MAYBE or YES when it was sent a CM, and at least one returned YES. If $P_j(k)$ returned MAYBE or YES, Lemmas F7 and F10 indicate that $P_j(k)$ nor any of its descendants had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$. This raises a contradiction and so the assumption must be wrong. Thus, if $MP(k)$ returns YES, the result follows. **Q.E.D.**

Lemma G1

If a procedure $P_i^+(k)$ is sent a CM M_i and some child $P_z(k)$ subsequently returns NO, all of the children of $P_i^+(k)$ to the left of $P_z(k)$ returned MAYBE.

Proof:

First let $Pi^+(k)$ be some procedure $Pi(k)$ and assume that $Pi(k)$ has a child $Pj(k)$ to the left of $Pz(k)$ which is sent a CM but it does not return MAYBE. $Pi(k)$ could call $Pz(k)$ only if it had already called all of its other children to the left. Each one must have returned MAYBE or YES, otherwise $Pi(k)$ would have made transition 9, 10, 41, 11 or 12 and returned MAYBE(S), YES(S), YES+MAYBE(S), or NO.

Now let $Pi^+(k)$ be $MP(k)$ and assume that $MP(k)$ has a child $Pj(k)$ to the left of $Pz(k)$ that does not return MAYBE. $MP(k)$ could call $Pz(k)$ only if it had already called all of its other children to the left and so each one must have returned MAYBE or YES, otherwise $MP(k)$ would have made transition 10, 11, 12, 13, or 21 and returned MAYBE(S), YES(S), YES+MAYBE(S), or NO.

In either case, if $Pi^+(k)$ has a child $Pj(k)$ to the left of $Pz(k)$ that returns a response other than MAYBE, it must have returned YES. Lemma E1 states that if $Pj(k)$ returned YES, one or more satisfaction functions of $Pj(k)$ or its descendants returned YES. Lemma E2 states that if $Pz(k)$ returns NO, then a satisfaction function F of $Pz(k)$ or one of its descendants returned NO. According to Restriction SF3, F can return NO only if a subset of the constraints in L_{M_i} are nonfeasible or all possible solutions to the set have already been returned. The first possibility raises a contradiction since $Pj(k)$ verified the constraints, and the second violates Restriction SF3 because if there is a solution (as indicated by $Pj(k)$), it was not returned by $Pz(k)$ before it returned NO. Therefore, $Pj(k)$ cannot return YES and the original assumption is contradicted. This indicates that if $Pi^+(k)$ has a child $Pz(k)$ that returns NO, all children to the left of $Pz(k)$ returned MAYBE.

Q.E.D.

Corolary G1

If a procedure $Pi^+(k)$ is sent a CM M_i and it has a child $Pz(k)$ that subsequently returns NO, any descendent of $Pi^+(k)$ in a subtree to the left of $Pz(k)$ returned MAYBE if it was sent a CM.

Proof:

Assume that $Pi^+(k)$ has a descendent $Pj(k)$ in a subtree to the left of $Pz(k)$ that is sent a CM and it returns a response other than MAYBE. Let $Pr(k)$ be the child of $Pi^+(k)$ at the root of the subtree containing $Pj(k)$. By Lemma G1, $Pr(k)$ must have returned MAYBE and by Lemma F4, any descendent of $Pr(k)$ which was sent a CM returned MAYBE. This includes $Pj(k)$ and so a contradiction is raised. Thus, the original assumption must be wrong and so if a procedure $Pi^+(k)$ has a child $Pz(k)$ that returns NO, any descendent of $Pi^+(k)$ in a subtree to the left of $Pz(k)$ returned MAYBE if it was sent a CM.

Q.E.D.**Lemma G2**

If a procedure $Pi^+(k)$ sends a CM to some child $Pz(k)$ and the child returns NO, no descendent of $Pi^+(k)$ in a subtree to the right of $Pz(k)$ will be sent a CM before $Pi^+(k)$ replies.

Proof:

$Pi^+(k)$ could be some procedure $Pi(k)$ or $MP(k)$; each case is considered below.

Case 1: $Pi^+(k)$ is some procedure $Pi(k)$

Assume that $Pi(k)$ has a child $Pj(k)$ to the right of $Pz(k)$ that is sent a CM before $Pi(k)$ replies, and let $Pm(k)$ denote the child of $Pi(k)$ at the root of the subtree containing $Pj(k)$. $Pj(k)$ could be $Pm(k)$ or one of its descendents. $Pm(k)$ will be called only if $Pi(k)$ can make transition 8 when $Pm(k)$ is the next child to be called. However, as soon as $Pz(k)$ returns NO, $Pi(k)$ must make transition 12 and return NO. Thus, no child to the right of $Pz(k)$ will be called before $Pi(k)$ replies, including $Pm(k)$. If $Pm(k)$ is not called, then none of its descendents can be called and so a contradiction arises. Thus, $Pi^+(k)$ cannot be some procedure $Pi(k)$.

Case 2: $Pi^+(k)$ is $MP(k)$

Assume that $MP(k)$ has a child $Pj(k)$ to the right of $Pz(k)$ that is sent a CM before $MP(k)$ replies, and let $Pm(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pj(k)$. $Pj(k)$ could be $Pm(k)$ or one of its descendents. $Pm(k)$

will be called only if $MP(k)$ can make transition 7 when $Pm(k)$ is the next child to be called. However, as soon as $Pz(k)$ returns NO, $MP(k)$ must make transition 21 and return NO. Thus, no child to the right of $Pz(k)$ will be called before $MP(k)$ replies, including $Pm(k)$. If $Pm(k)$ is not called, then none of its descendants can be called and so a contradiction arises. Thus, $Pi^+(k)$ cannot be $MP(k)$.

If $Pi^+(k)$ cannot be some procedure $Pi(k)$ or $MP(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if a procedure $Pi^+(k)$ has a child $Pz(k)$ that returns NO, no descendent in a subtree to the right of $Pz(k)$ will be sent a CM before $Pi^+(k)$ replies.

Q.E.D.

Lemma G3

If a procedure $Pi(k)$ is sent a CM and it returns NO, any descendent which was sent a CM returned MAYBE or NO.

Proof:

Assume that $Pi(k)$ has a descendent $Pj(k)$ which is sent a CM but it returns a response other than MAYBE or NO. $Pi(k)$ can return NO only by making transition 29 or 12; each case is considered below.

Case 1: $Pi(k)$ makes transition 29

Before $Pi(k)$ can make transition 29, it must have first exhausted its list of children and made transitions 5 and 17. Each child must have returned MAYBE or YES, otherwise $Pi(k)$ would have replied using transition 9, 10, 11, 41, or 12.

Let $Pr(k)$ denote the child of $Pi(k)$ at the root of the subtree containing $Pj(k)$; $Pj(k)$ could be $Pr(k)$ or one of its descendants. Lemma F4 indicates that if $Pr(k)$ returned MAYBE, then any descendent which was sent a CM returned MAYBE, including $Pj(k)$. This contradicts the assumption that $Pj(k)$ returned some other response and so $Pr(k)$ must have returned YES.

Lemma E1 states that if $Pr(k)$ returns YES, a satisfaction function of $Pr(k)$ or one of its descendants returned YES, indicating that the problem set of $Pi(k)$ was feasible. On the other hand, $Pi(k)$ can make transition 29 only if one of its

satisfaction functions returns NO. By Restriction SF3, a NO response indicates that the problem set is nonfeasible or all possible solutions have already been returned. Since the YES response indicates that the problem set is feasible and $P_i(k)$ has not returned YES, a contradiction arises and so $P_i(k)$ cannot make transition 29.

Case 2: $P_i(k)$ makes transition 12

Assume that $P_i(k)$ makes transition 12 after one of its children $P_z(k)$ returns NO. $P_j(k)$ must be in a subtree to the left or right of $P_z(k)$, or in the same subtree as $P_z(k)$; each case is considered below.

Case 2.1: $P_j(k)$ is in a subtree to the right of $P_z(k)$

By Lemma G2, no procedure in a subtree to the right of $P_z(k)$ will be sent a CM before $P_i(k)$ replies and so $P_j(k)$ cannot be in a subtree to the right of $P_z(k)$.

Case 2.2: $P_j(k)$ is in a subtree to the left of $P_z(k)$

By Corollary G1, any descendent in a subtree to the left of $P_z(k)$ returned MAYBE if it was sent a CM and so $P_j(k)$ cannot be in a subtree to the left of $P_z(k)$.

Case 2.3: $P_j(k)$ is in the same subtree as $P_z(k)$

If $P_z(k)$ returns NO, $P_j(k)$ cannot be $P_z(k)$ and so it must be one of its descendants. $P_z(k)$ can return NO only by making transition 12 or 29. The arguments used in Cases 1 and 2 may now be repeated with $P_z(k)$ replacing $P_i(k)$. $P_z(k)$ cannot make transition 29 and so it must make transition 12 after one of its children returns NO. $P_j(k)$ cannot be in a subtree to the left or right of that child and so it must be in the same subtree.

These arguments repeat indefinitely and every time that they are used, $P_j(k)$ must be at another level deeper in the hierarchy. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_j(k)$ and so a contradiction arises. Thus, $P_j(k)$ cannot be in the same subtree as $P_z(k)$.

If $P_j(k)$ cannot be in a subtree to the left or right of $P_z(k)$ or in the same subtree as $P_z(k)$, a contradiction arises and so $P_i(k)$ cannot make transition 12.

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

If $P_i(k)$ cannot make transition 29 or 12, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns NO, any descendent which was sent a CM returned MAYBE or NO.

Q.E.D.

Lemma G4

If a procedure $P_i^+(k)$ is sent a CM and it returns NO, then any descendent which was sent a CM returned MAYBE or NO.

Proof:

Assume that $P_i^+(k)$ has a descendent $P_j(k)$ which is sent a CM but it returns a response other than MAYBE or NO. $P_i^+(k)$ could be some procedure $P_i(k)$ or $MP(k)$; Each case is considered below.

Case 1: $P_i^+(k)$ is some procedure $P_i(k)$

The proof of Lemma G3 suffices and so $P_i^+(k)$ cannot be some procedure $P_i(k)$.

Case 2: $P_i^+(k)$ is $MP(k)$

$MP(k)$ can return NO only by making transition 14, 26, or 21. If it makes either of the first two, it could not have made transition 7 and so no procedures are called. Thus, $MP(k)$ must make transition 21 after some child $P_z(k)$ returns NO. $P_j(k)$ could be in a subtree to the left or right of $P_z(k)$ or in the same subtree as $P_z(k)$; each case is considered below.

Case 2.1: $P_j(k)$ is in a subtree to the right of $P_z(k)$

By Lemma G2, procedures in subtrees to the right of $P_z(k)$ will not be sent CM's and so $P_j(k)$ cannot be in a subtree to the right of $P_z(k)$.

Case 2.2: $P_j(k)$ is in a subtree to the left of $P_z(k)$

By Corollary G1, any descendent in a subtree to the left of $P_z(k)$ returned MAYBE if it was sent a CM and so $P_j(k)$ cannot be in a subtree to the left of $P_z(k)$.

Case 2.3: $P_j(k)$ is in the same subtree as $P_z(k)$

Since $P_z(k)$ returns NO, $P_j(k)$ cannot be $P_z(k)$ and so it must be one of its descendants. However, by Lemma G3, if $P_z(k)$ returns NO, any descendent which was sent a CM returned MAYBE or NO. Thus, $P_j(k)$ cannot be in the same subtree as $P_z(k)$.

If $P_j(k)$ cannot be to the left or right of $P_z(k)$ or in the same subtree as $P_z(k)$, a contradiction arises and so $MP(k)$ cannot make transition 21. It follows that $P_i^+(k)$ cannot be $MP(k)$.

If $P_i^+(k)$ cannot be some procedure $P_i(k)$ or $MP(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i^+(k)$ is sent a CM and it returns NO, then any descendent which was sent a CM returned MAYBE or NO.

Q.E.D.

Appendix D

Proof of Properties

D.1 Proofs of General Properties

Property A1.1

Given the following conditions:

1. a procedure $P_i(k)$ is sent a CM,
2. every function call made by $P_i(k)$ or one of its descendents terminates,
3. every recognition rule and sufficiency predicate that is tested by $P_i(k)$ or one of its descendents terminates, and
4. if any recursive calls are made, $MP(k+1)$ responds in a finite length of time,

then $P_i(k)$ will respond in a finite length of time.

Proof:

Assume that the satisfaction process does not terminate; i.e., $P_i(k)$ does not respond after being sent a CM. This could be caused in one of two ways:

- $P_i(k)$ or one of its descendents could become caught in a infinite loop of transitions; the loop may involve calls to other procedures but the calls are not the direct cause of the loop.

- A series of inter-procedure calls could result in situations analagous to that of deadlock in concurrent systems; i.e., a set of procedures could end up waiting indefinitely for each other to respond.

More specifically, the following cases have been identified as potential causes of nontermination:

1. $P_i(k)$ or one of its descendents may get caught in a loop of passing constraints to its children and getting MAYBE or YES responses,
2. $P_i(k)$ or one of its descendents may get caught in a loop of executing a function, getting a YES or MAYBE response, and then trying another function.
3. $P_i(k)$ or one of its descendents may get caught in a loop involving the execution of a function, calling $MP(k+1)$ to test the values, having the values rejected, and then executing another function.
4. There could be an infinite set of procedures $\{P_a(k), P_b(k), P_c(k), \dots\}$ such that (1) $P_a(k)$ is the parent of $P_b(k)$, $P_b(k)$ is the parent of $P_c(k)$, and so on, and (2) each procedure is waiting for a reply from its child.
5. There could be a set of procedures $\{P_a(k), P_b(k), P_c(k), \dots, P_n(k)\}$ such that $P_a(k)$ is waiting for a reply from $P_b(k)$, $P_b(k)$ is waiting for a reply from $P_c(k)$, and so on, up to and including $P_n(k)$ waiting on $P_a(k)$.
6. $P_i(k)$ or one of its descendents might send a CM to $MP(k+1)$ and never receive a reply.

Each case is considered below. Unless otherwise noted, $P_j(k)$ denotes some procedure in the same subtree as $P_i(k)$; i.e., $P_j(k)$ could be $P_i(k)$ or one of its descendents.

Case 1:

Assume that a procedure $P_j(k)$ could be caught in a loop involving transition 8 and some combination of 6 and 7. This could happen only if the condition on transition 8 remained true, and this would require $P_j(k)$ to have an infinite number

of children. This contradicts Restriction PO1 and so $P_i(k)$ nor any of its descendants can be caught in such a loop.

Case 2:

Assume that some procedure $P_i(k)$ could be caught in a loop involving transition 23 and some combination of 24 and 25, i.e., looping forever because every function returned YES or MAYBE. This would require the condition on transition 23 to always be true and $P_j(k)$ to have an infinite number of satisfaction functions. This contradicts Restriction SF1 and so $P_i(k)$ nor any of its descendants can be caught in such a loop.

Case 3:

Assume that some procedure $P_j(k)$ could be caught in a loop involving transition 23 and some combination of 26, 27, 28, and 43, i.e., $P_j(k)$ generates a solution, sends it to $MP(k)$ for confirmation, receives a NO response, and then loops to test another function. This would require the condition on transition 23 to always be true and $P_j(k)$ to have an infinite number of satisfaction functions. This contradicts Restriction SF1 and so $P_i(k)$ nor any of its descendants can be caught in such a loop.

Case 4:

Assume that there is a infinite set of procedures such that each one is waiting for one of its children to respond. This requires the tree of procedures to be infinitely deep but this contradicts Restriction PO1. Consequently, there cannot be such a set of procedures.

Case 5:

Assume that a set of procedure calls has occurred as described in Item 5 above. Since procedures can call only their children, such a series of calls would require $P_a(k)$ to be a child of $P_n(k)$ and $P_n(k)$ to be a descendent of $P_a(k)$. This violates Restriction PO2 which limits procedure configurations to n-way trees, thus such a series of calls cannot occur.

Case 6:

Assume that some procedure $P_j(k)$ waits forever for $MP(k+1)$ to terminate. This contradicts condition (4) and so $P_i(k)$ nor any of its descendents can wait indefinitely for $MP(k+1)$ to respond.

Since none of the cases above are possible, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $P_i(k)$ will respond in a finite length of time.

Q.E.D

Property A1.2

Given the following conditions:

1. $MP(k)$ is sent a CM,
2. every function call that is subsequently made terminates, and
3. every recognition rule and sufficiency predicate that is subsequently tested terminates,

then $MP(k)$ will respond in a finite length of time.

Proof:

Assume that the satisfaction process does not terminate; i.e., $MP(k)$ does not respond after being sent a CM. This could be caused by $MP(k)$ becoming caught in an infinite loop of transitions or by one of its children failing to respond. Each case is considered below.

Case 1: $MP(k)$ could be caught in an infinite loop of transitions

There are two possible loops of transitions that could prevent $MP(k)$ from replying. The first involves transition 3 only; the second involves transitions 7, 8, and 9. Each case is considered below.

Case 1.1: $MP(k)$ loops forever using transition 3

Assume that $MP(k)$ could be caught in a loop involving transition 3. Every time that $MP(k)$ makes transition 3, it solves an MP-solvable constraint and generates

a new substitution. This eliminates one variable but since the number of variables is finite (by Lemma B1), eventually the MP-solvable constraints must be exhausted. Therefore, $MP(k)$ cannot loop forever using transition 3.

Case 1.2: *$MP(k)$ loops forever using transitions 7, 8, and 9*

Assume that $MP(k)$ could be caught in a loop involving transition 7 and some combination of 8 and 9. This requires the condition on transition 7 to always be true, but that would require $MP(k)$ to have an infinite number of children. This violates Restriction PO1 and so $MP(k)$ cannot be caught in a loop involving transitions 7, 8, and 9.

Case 2: *$MP(k)$ waits forever for one of its children to respond*

Assume that $MP(k)$ does not respond after calling some child $Pa(k)$ because $Pa(k)$ did not reply. By the results of Property A1.1, $Pa(k)$ could fail to reply only if:

1. a recognition rule, sufficiency predicate, or satisfaction function belonging to it or one of its descendants failed to terminate, or
2. some procedure $Pi(k)$ in the same subtree as $Pa(k)$ made a recursive call and $MP(k+1)$ failed to respond; $Pi(k)$ could be $Pa(k)$ or one of its descendants.

The first case cannot occur because of Conditions (2) and (3), and so it remains to be shown that $MP(k+1)$ would respond if it was called.

Assume that $MP(k+1)$ did not respond after being called by some procedure $Pi(k)$. The arguments used in Cases 1 and 2 now repeat with $MP(k+1)$ replacing $MP(k)$. Thus, $MP(k+i)$, $i \geq 0$, would respond in a finite length of time as long as it did not wait forever for some child $Pa(k+i)$ to reply, and $Pa(k+i)$ could fail to reply only if it or one of its descendants made a recursive call and $MP(k+i+1)$ failed to reply, and so on.

Assume that such a series of calls could occur. By Lemma B2, there exists a finite value of n , $n \geq 1$, such that no procedure $Px(k+n)$ could make a recursive call and so $MP(k+n)$ could not fail to reply to the procedure that called it. This

raises a contradiction and so such series of calls could not occur. It follows that $MP(k)$ cannot wait forever for one of its children to respond.

Since $MP(k)$ cannot be caught in an infinite loop of transitions and all of its children must reply if they are called, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will respond in a finite length of time.

Q.E.D

Property A2

Given the following conditions:

1. $MP(k)$ is sent a CM M_a and zero or more AVR's,
2. some procedure $Pi(k)$ can recognize at least one constraint in $PS_{M_a}^4$,
3. no procedure $Ph^+(k)$ can determine that $PS_{M_a}^*$ or $PS_{M_a}^*$ contains nonfeasible constraints, and
4. based on the order in which procedures can call their satisfaction functions, no procedure $Pq^+(k)$ which is before $Pi(k)$ can determine that all possible solutions to the constraints in L_{M_a} have been returned already,

then:

1. $Pi(k)$ will be sent a CM before $MP(k)$ returns NO or ABANDON, and
2. its updated problem set will contain every constraint $c \in PS_{M_a}^4$ that $Pi(k)$ can recognize.

Proof:

Each result is considered below.

Result 1: $Pi(k)$ is sent a CM

Assume that some procedure $Pi(k)$ is not sent a CM before $MP(k)$ returns NO or ABANDON, subject to the above conditions. $MP(k)$ could return NO or

ABANDON only by making transition 14, 26, 16, 21, or 20; each case is considered below.

Case 1: $MP(k)$ makes transition 14

$MP(k)$ could return NO using transition 14 only if it detected a nonfeasible constraint in $PS_{M_a}^+$. However, it was assumed that no procedure $Ph^+(k)$ could determine that a subset of constraints in $PS_{M_a}^+$ was nonfeasible and so $MP(k)$ cannot make transition 14.

Case 2: $MP(k)$ makes transition 26

$MP(k)$ could return NO using transition 26 only if it detected a nonfeasible constraint in $PS_{M_a}^+$. However, it was assumed that no procedure $Ph^+(k)$ could determine that a subset of constraints in $PS_{M_a}^+$ was nonfeasible and so $MP(k)$ cannot make transition 26.

Case 3: $MP(k)$ makes transition 16

$MP(k)$ could return NO using transition 16 only if it had previously made transition 15 or 25. It could make transition 15 only if every constraint in $PS_{M_a}^+$ was satisfied, in which case there would be no values to return. $MP(k)$ could make transition 25 only if it had totally reduced and satisfied its problem set using transitions 24 and possibly 3, in which case no other solutions would be possible (since no choicepoints were involved). However, it was assumed that no procedure $Ph^+(k)$ could determine that all possible solutions to L_{M_a} had been generated already, and so $MP(k)$ cannot make transition 16.

Case 4: $MP(k)$ makes transition 20

$MP(k)$ could return ABANDON using transition 20 only by receiving an AVR after it had reached state 2. To reach state 2, $MP(k)$ must have called all of its children and all must have returned MAYBE or YES in response to a CM or an AVR. Let $Pm(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$. It was just shown that $Pm(k)$ was sent a CM and so $Pi(k)$ must be one of its descendants. The following two cases must be considered.

Case 4.1: *$P_m(k)$ returned MAYBE or YES in response to a CM*

If $P_m(k)$ returned MAYBE or YES in response to a CM, then Lemmas F4 and F9 indicate that any descendent which was sent a CM returned MAYBE or YES. If $P_i(k)$ was not sent a CM, its parent $P_n(k)$ did not receive a CM or it returned MAYBE using transition 2. (Any other transition resulting in a MAYBE or YES reply results in $P_i(k)$ being called.) However, if $P_n(k)$ made transition 2, Restriction RS1 would be violated and so it could not have been sent a CM.

Similarly, if $P_n(k)$ was not sent a CM, its parent $P_l(k)$ made transition 2 or it did not receive a CM. The above arguments now repeat with $P_l(k)$ replacing $P_n(k)$. $P_l(k)$ could not have made transition 2 without violating Restriction RS1 and so it could not have been sent a CM. These arguments repeat for every ancestor of $P_i(k)$ up to and including $P_m(k)$. However, $P_m(k)$ was sent a CM and so a contradiction is raised. Thus, $P_i(k)$ cannot be a descendent of $P_m(k)$ if $P_m(k)$ returned MAYBE or YES in response to a CM.

Case 4.2: *$P_m(k)$ returned MAYBE or YES in response to an AVR*

If $P_m(k)$ returned MAYBE or YES in response to an AVR, then Lemmas F5 and F6 indicate that all of its children returned MAYBE or YES in response to a CM or an AVR before $P_m(k)$ replied. Thus, $P_i(k)$ cannot be a child of $P_m(k)$ and so it must some other descendent. Let $P_n(k)$ denote the child of $P_m(k)$ at the root of the subtree containing $P_i(k)$. It was just shown that $P_n(k)$ returned MAYBE or YES in response to a CM or an AVR; each case is considered below.

Case 4.2.1: *$P_n(k)$ returned MAYBE or YES in response to a CM*

If $P_n(k)$ returned MAYBE or YES in response to a CM, then the arguments used in Case 4.1 may be repeated with $P_n(k)$ replacing $P_m(k)$. Thus, $P_i(k)$ could not be in the same subtree as $P_n(k)$ if $P_n(k)$ returned MAYBE or YES in response to a CM.

Case 4.2.2: *$P_n(k)$ returned MAYBE or YES in response to an AVR*

If $P_n(k)$ returned MAYBE or YES in response to an AVR, then the arguments of Case 4.2 repeat with $P_n(k)$ replacing $P_m(k)$. Lemmas F5 and F6 indicate that

all of its children returned MAYBE or YES in response to a CM or an AVR before $P_n(k)$ replied. Thus, $P_i(k)$ cannot be a child of $P_n(k)$ and so it must be some other descendent.

These arguments repeat indefinitely and every time that they are used, $P_i(k)$ must be at another level deeper in the hierarchy. Since the number of procedures in the solver is finite, eventually a childless procedure will be encountered that cannot be $P_i(k)$ and so a contradiction arises. Thus, $P_n(k)$ could not return MAYBE or YES in response to an AVR.

If $P_n(k)$ could not return MAYBE or YES in response to a CM or an AVR, then a contradiction arises and so $P_i(k)$ cannot be a descendent of $P_n(k)$. It follows that $P_m(k)$ could not return MAYBE or YES in response to an AVR.

If $P_m(k)$ could not return MAYBE or YES in response to a CM or an AVR, a contradiction is raised and so $MP(k)$ could not make transition 20.

Case 5: $MP(k)$ makes transition 21

$MP(k)$ could return NO using transition 21 only if some child $Pa(k)$ returned NO after having been sent a CM or an AVR; each case is considered below.

Case 5.1: $Pa(k)$ returns NO in response to a CM

If $Pa(k)$ returned NO in response to a CM, Lemma F1 indicates that $Pa(k)$ or one of its descendents determined that $PS_{M_a}^4$ contained nonfeasible constraints. However, it was assumed that no procedure $Ph^+(k)$ could do so and so $Pa(k)$ cannot return NO in response to a CM.

Case 5.2: $Pa(k)$ returns NO in response to an AVR

By Lemma F3, if $Pa(k)$ returns NO in response to an AVR, then no solutions to $PS_{M_a}^4$ are possible other than those that have been returned already. By Lemma E2, if $Pa(k)$ returned NO in response to an AVR, then a function F of $Pa(k)$ or one of its descendents returned NO. Let $Pb(k)$ denote the procedure in the same subtree as $Pa(k)$ to which F belongs; $Pb(k)$ could be $Pa(k)$ or one of its descendents.

Given condition (4) and the original assumption, $P_i(k)$ must be before $P_b(k)$. It could be in a subtree to the left of $P_a(k)$, or in the same subtree as $P_a(k)$; each case is considered below.

Case 5.2.1: $P_i(k)$ is in a subtree to the left of $P_a(k)$

Let $P_n(k)$ denote the child of $MP(k)$ at the root of the subtree containing $P_i(k)$; $P_i(k)$ could be $P_n(k)$ or one of its descendents. If $MP(k)$ sent an AVR to $P_a(k)$, it must have previously sent a CM to $P_a(k)$. However, $MP(k)$ could not call $P_a(k)$ until $P_n(k)$ returned MAYBE or YES in response to a CM or an AVR. In any case, $P_n(k)$ was sent a CM and so it cannot be $P_i(k)$, thus $P_i(k)$ must be one of its descendents.

If $P_n(k)$ returned MAYBE or YES in response to a CM or an AVR, the arguments of Cases 4.1 and 4.2 apply again and so $P_i(k)$ cannot be a descendent of $P_n(k)$. It follows that $P_i(k)$ cannot be in a subtree to the left of $P_a(k)$.

Case 5.2.2: $P_i(k)$ is in the same subtree as $P_a(k)$

If $P_i(k)$ is before $P_b(k)$ and in the same subtree as $P_a(k)$, then it must be either:

1. a descendent of $P_b(k)$, or
2. in a subtree such that the procedure $P_m(k)$ at its root is:
 - 2.1 a sibling of $P_b(k)$ and $P_m(k)$ is leftmost, or
 - 2.2 a sibling of some ancestor $P_d(k)$ of $P_b(k)$, and $P_m(k)$ is leftmost.

Each case is considered below.

Case 5.2.2.1: $P_i(k)$ is a descendent of $P_b(k)$

$P_b(k)$ could call one of its functions only if it had previously made transitions 17 and 5, and it could make 5 only if it had exhausted its list of children. Let $P_n(k)$ denote the child of $P_b(k)$ at the root of the subtree containing $P_i(k)$. Since $P_n(k)$ must have been sent a CM, $P_n(k)$ cannot be $P_i(k)$ and so $P_i(k)$ must be one of its descendents.

$Pn(k)$ must have returned MAYBE or YES in response to a CM or an AVR. However, the arguments of Cases 4.1 and 4.2 now repeat and a contradiction will arise in either case. This indicates that $Pn(k)$ could not have returned MAYBE or YES in response to a CM or an AVR and so another contradiction arises. It follows that $Pi(k)$ cannot be a descendent of $Pb(k)$.

Case 5.2.2.2: $Pi(k)$ is in a subtree to the left of $Pb(k)$ or $Pd(k)$

Let $Pm(k)$ denote the procedure at the root of the subtree containing $Pi(k)$ as described above. Without loss of generality, let $Pf(k)$ denote the sibling of $Pm(k)$ at the root of the subtree containing $Pb(k)$; i.e., $Pf(k)$ is $Pb(k)$ or $Pd(k)$. As well, let $Pe(k)$ denote the common parent of $Pm(k)$ and $Pf(k)$; $Pe(k)$ could be $Pa(k)$ or some other ancestor of $Pb(k)$.

If $Pb(k)$ had returned NO, then it was sent either a CM or an AVR. As well, each of its ancestors must have been sent a CM or an AVR. Thus, $Pe(k)$ must have sent a CM or an AVR to $Pf(k)$ after $MP(k)$ had sent an AVR to $Pa(k)$ (the message type depending on whether or not $Pf(k)$ had returned the last solution). $Pf(k)$ would not be sent an AVR unless it had previously been sent a CM, and $Pe(k)$ could not send $Pf(k)$ a CM until $Pm(k)$ had returned MAYBE or YES in response to a CM or an AVR. Similarly, $Pm(k)$ would not be sent an AVR unless it had previously been sent a CM and so $Pm(k)$ was sent a CM either before or after $MP(k)$ sent an AVR to $Pa(k)$. It follows that (1) $Pi(k)$ cannot be $Pm(k)$, (2) $Pm(k)$ returned MAYBE or YES in response to a CM or an AVR, and (3) $Pi(k)$ must be a descendent of $Pm(k)$.

Given these three conclusions, the arguments of Cases 4.1 and 4.2 may now be used to show that $Pi(k)$ cannot be in a subtree to the left of $Pf(k)$. It follows that $Pi(k)$ cannot be in the same subtree as $Pa(k)$,

If $Pi(k)$ cannot be in the same subtree as $Pa(k)$ or in a subtree to its left, then a contradiction arises and so $Pa(k)$ cannot return NO in response to an AVR. Since it could not return NO in response to a CM either (as determined in Case 5.1), it follows that $MP(k)$ could not make transition 21.

If $MP(k)$ cannot make transition 14, 25, 16, 21, or 21, then a contradiction

is raised and so the original assumption must be wrong. Thus, given the above conditions, $Pi(k)$ will be sent a CM before $MP(k)$ returns NO or ABANDON.

Result 2: $Pi(k)$'s problem set contains every recognizable $c \in PS_{M_a}^4$.

It was shown in Result 1 that $Pi(k)$ would be sent a CM, and so by Lemma A3, its updated problem set will contain every constraint $c \in PS_{M_a}^4$ that $Pi(k)$ can recognize.

Q.E.D.

Property A3.1

Given the following conditions:

1. a procedure $Pi(k)$ is sent a CM containing constraint c ,
2. $Pi(k)$ has a function F_i that can generate the assignment constraint $X = x_i$ for some variable X in c ,
3. $Pi(k)$ has a descendent $Pj(k)$ with a function F_j that can generate the assignment constraint $X = x_j$ given c , and
4. $Pi(k)$ will return both solutions if sent a sufficient number of AVR's, that is, neither will be rejected if sent to $MP(k+1)$,

then $Pi(k)$ will return the solution $X = x_j$ before returning $X = x_i$.

Proof:

Assume that given the above conditions, $Pi(k)$ returns the substitution $X = x_i$ first. Let $Pm(k)$ denote the child of $Pi(k)$ at the root of the subtree containing $Pj(k)$; $Pj(k)$ could be $Pm(k)$ or one of its descendants. In order for $Pi(k)$ to return $X = x_i$, it must call F_i and make transition 31, 32, 33, or 42. However, in order to make transition 23, $Pi(k)$ must make transitions 17 and 5, the latter requiring every child of $Pi(k)$ to have been sent a CM and returned MAYBE or YES, including $Pm(k)$.

By Lemmas F7 and F9, if $Pm(k)$ returns MAYBE or YES, then $Pm(k)$ nor any of its descendants could generate a definite solution to c , or a tentative solution

which was not subsequently rejected by $MP(k+1)$. Conditions (3) and (4) indicate that $Pm(k)$ or one of its descendents could generate a definite solution, or a tentative one which would not be rejected by $MP(k+1)$. This raises a contradiction and so the original assumption must be wrong. Thus, given the above conditions, $Pi(k)$ will return the solution $X = x_j$ before returning $X = x_i$.

Q.E.D.

Property A3.2

Given the following conditions:

1. a procedure $Pa(k)$ is sent a CM containing constraint c ,
2. $Pa(k)$ has a descendent $Pi(k)$ with a function F_i that, if given c , can generate the assignment constraint $X = x_i$ for some variable X in c ,
3. $Pa(k)$ has another descendent $Pj(k)$ with a function F_j that, if given c , can generate the assignment constraint $X = x_j$,
4. $Pi(k)$ is in a subtree to the left of the subtree containing $Pj(k)$, and
5. $Pa(k)$ will return both solutions if sent a sufficient number of AVR's, that is, neither will be rejected if sent to $MP(k+1)$,

then $Pa(k)$ will return the solution $X = x_i$ before returning $X = x_j$.

Proof:

Let $Pm(k)$ and $Pn(k)$ denote procedures such that:

- $Pm(k)$ and $Pn(k)$ are children of a common parent $Pl(k)$.
- $Pm(k)$ is the child of $Pl(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ is $Pm(k)$ or one of its descendents.
- $Pn(k)$ is the child of $Pl(k)$ at the root of the subtree containing $Pj(k)$; $Pj(k)$ is $Pn(k)$ or one of its descendents.

Note that $Pl(k)$ could be $Pa(k)$ or one of its descendants.

Assume that $Pa(k)$ returns the solution $X = x_j$ before returning $X = x_i$. This indicates that $Pl(k)$ must return $X = x_j$ first. Since (1) $Pm(k)$ is to the left of $Pn(k)$, and (2) $Pl(k)$ will not call $Pn(k)$ until all of its children to the left of $Pn(k)$ return YES or MAYBE, $Pm(k)$ must return MAYBE or YES after being sent a CM.

By Lemmas F7 and F9, if $Pm(k)$ returns MAYBE or YES, then $Pm(k)$ nor any of its descendants could generate a definite solution to c , or a tentative solution which was not subsequently rejected by $MP(k+1)$. Conditions (2) and (5) indicate that $Pm(k)$ or one of its descendants could generate a definite solution, or a tentative one which would not be rejected by $MP(k+1)$. This raises a contradiction and so the original assumption must be wrong. Thus, given the above conditions, $Pa(k)$ must return the solution $X = x_i$ before returning $X = x_j$.

Q.E.D.

D.2 Responses Occur only if Conditions Met

The following proofs indicate that the solver will return various results only if certain conditions are met.

Property B1

If $MP(k)$ returns MAYBE in response to a CM M_i , then:

1. $MP(k)$ could not determine if the constraints in L_{M_i} were feasible or nonfeasible,
2. $MP(k)$ could not generate any solutions to any subset of constraints in $PS_{M_i}^+$,
3. no procedure $Pa(k)$ had a satisfaction function which could determine if the constraints in L_{M_i} were feasible or nonfeasible, and
4. no procedure $Pa(k)$ had a satisfaction function which could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2)

a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Proof:

This property is identical to Lemma F8 and so the proof of the lemma suffices.

Q.E.D.

Property B2

If $MP(k)$ returns YES in response to a CM M_i , then:

1. the constraints in L_{M_i} are feasible,
2. $MP(k)$ could not generate any definite or tentative solutions to a subset of the constraints in $PS_{M_i}^+$, and
3. no procedure $Pa(k)$ could generate (1) a definite partial or full solution to a subset of the constraints in $PS_{M_i}^+$, or (2) a tentative partial or full solution to a subset which was not subsequently rejected by $MP(k+1)$.

Proof:

This property is identical to Lemma F11 and so the proof of the lemma suffices.

Q.E.D.

Property B3

If $MP(k)$ returns MAYBE(S) in response to a CM M_a , then:

1. $MP(k)$ could not determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible,
2. no procedure $Pa(k)$ had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible,
3. if the solution was forwarded back through n levels of recursion, $n \geq 1$, no $MP(k+i)$, $i=1, \dots, n$, could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, and

4. if the solution was forwarded back through n levels of recursion, $n \geq 1$, no procedure $Pj(k + i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Proof:

Each result is considered below.

Result 1:

Assume that $MP(k)$ returns $MAYBE(S)$ even though it determine that $PS_{M_a}^+$ was feasible or nonfeasible. $MP(k)$ could determine such by detecting that all constraints in $PS_{M_a}^+$ were satisfied when it reached state 7, in which case it would make transition 15, or by totally reducing and satisfying the constraints in $PS_{M_a}^+$ using transitions 24 and 3, in which case it would make transition 25. $MP(k)$ could determine that $PS_{M_a}^+$ was nonfeasible by detecting a contradictory constraint in either $PS_{M_a}^+$, in which case it would make transition 14, or in $PS_{M_a}^*$, in which case it would make transition 26. Thus, if $MP(k)$ could determine that $PS_{M_a}^+$ was feasible or nonfeasible, it would have to make transition 14, 15, or 24 from state 7, or 25 or 26 from state 6.

If $MP(k)$ returned $MAYBE(S)$ as stated, it must have made transition 27 or 10. In either case, it must have first reached state 4 after having made transition 2 from state 7, or 4 from state 6. This indicates that it could not have made transition 14, 15, 24, 25, or 26 and so a contradiction is raised. Thus, $MP(k)$ could not have been able to determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible.

Result 2:

Assume that some procedure $Pa(k)$ had a satisfaction function F (on an explored search path) which could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible. Let $Pm(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pa(k)$; $Pa(k)$ could be $Pm(k)$ or one of its descendents. $MP(k)$ could return $MAYBE(S)$ by making transition 27 or 10; each case is considered below.

Case 1: $MP(k)$ makes transition 27

$MP(k)$ could make transition 27 only if it exhausted all of its children and $V_{MP(k)}$

was set to 0. This requires every child to have returned MAYBE when it was sent a CM, but by Lemma F7, if $Pm(k)$ returned MAYBE, then $Pm(k)$ nor any of its descendants had a function which could determine if the constraints in $PS_{M_i}^4$ were feasible or nonfeasible. This raises a contradiction and so $MP(k)$ could not make transition 27.

Case 2: $MP(k)$ makes transition 10

$MP(k)$ could make transition 10 only if some child $Pb(k)$ returned MAYBE(S) and $V_{MP(k)}$ was set to 0. This indicates that all of $MP(k)$'s children to the left of $Pb(k)$ returned MAYBE and so by Lemma F7, $Pa(k)$ could not be in a subtree to the left of $Pb(k)$. Consequently, it must be in the same subtree as $Pb(k)$. $Pb(k)$ could return MAYBE(S) by making transition 9 or 32; each case is considered below.

Case 2.1: $Pb(k)$ makes transition 32

$Pb(k)$ could make transition 32 only if it had exhausted its list of children and all returned MAYBE (so that $V_{Pb(k)}$ was set to 0). Thus, by Lemma F7, $Pa(k)$ could not be a descendent of $Pb(k)$ and so $Pa(k)$ must be $Pb(k)$. It follows that F must be a satisfaction function of $Pb(k)$.

If $Pb(k)$ made transition 32 and F is on an explored search path, then $Pb(k)$ must have previously called F . By Lemma A4 and Corolary A4.1, F would have returned YES, YES(S), or NO and so either $V_{Pb(k)}$ was set to 1, or $Pb(k)$ returned NO using transition 29. Both cases lead to contradictions and so $Pb(k)$ could not make transition 32.

Case 2.2: $Pb(k)$ makes transition 9

If $Pb(k)$ made transition 32 after some child $Pc(k)$ returned MAYBE(S), $Pc(k)$ must have made transition 32 or 9. The above arguments now repeat with $Pc(k)$ replacing $Pb(k)$ and so $Pc(k)$ could not have made transition 32. If it made transition 9, then some child $Pd(k)$ returned MAYBE(S), and so on.

These arguments repeat indefinitely and every time that they are used, some procedure must make transition 9 after one of its children returns MAYBE(S). Since the number of procedures in the solver is finite, eventually a childless procedure

would be encountered that could not make transition 9. This raises a contradiction and so $Pb(k)$ could not make transition 9.

If $Pb(k)$ could not make transition 32 or 9, it could not return $MAYBE(S)$; it follows that $MP(k)$ could not make transition 10.

If $MP(k)$ could not make transition 27 or 10, it could not return $MAYBE(S)$. This raises a contradiction and so the original assumption must be wrong. Thus, given the above conditions, no procedure $Pa(k)$ had a satisfaction function F (on an explored search path) which could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible.

Result 3:

Assume that the solution was forwarded back through n levels of recursion, $n \geq 1$, and that some $MP(k+i)$, $i=1, \dots, n$, was able to determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. By Lemma D2, if the solution was forwarded back through recursive calls, then at each level $i=1, \dots, n$, some procedure $Pj(k+i-1)$ sent a CM to $MP(k+i)$ and $MP(k+i)$ returned $MAYBE(S)$.

The arguments of Result 1 now repeat with $MP(k+i)$ replacing $MP(k)$ for any $i=1, \dots, n$. Thus, if $MP(k)$ returned $MAYBE(S)$ after a solution was forwarded back through n levels of recursion, then no $MP(k+i)$, $i=1, \dots, n$, could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Result 4:

Assume that the solution was forwarded back through n levels of recursion, $n \geq 1$, and that some procedure $Pa(k+n)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. Let $Pm(k+i)$ denote the child of $MP(k+i)$ at the root of the subtree containing $Pa(k+i)$; $Pa(k+i)$ could be $Pm(k+i)$ or one of its descendents.

The arguments of Result 2 now repeat with $MP(k+i)$, $Pa(k+i)$, and $Pm(k+i)$ replacing $MP(k)$, $Pa(k)$, and $Pm(k)$, respectively. Consequently, if $MP(k)$ returned

MAYBE(S) after a solution was forwarded back through n levels of recursion, then no procedure $P_j(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Q.E.D.

Property B4

If $MP(k)$ returns YES+MAYBE(S) in response to a CM M_i , then:

1. the constraints in L_{M_a} are feasible,
2. if $MP(k)$ returned YES+MAYBE(S) after the substitutions in S were forwarded back through $n \geq 1$ levels of recursion, then no $MP(k+i)$, $i=1, \dots, n$, could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, and
3. if $MP(k)$ returned YES+MAYBE(S) after the substitutions in S were forwarded back through n levels of recursion, $n \geq 1$, then no procedure $P_j(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied.

Proof:

Each result is considered below.

Result 1: *The constraints in L_{M_a} are feasible*

Assume that $MP(k)$ returns YES+MAYBE(S) in response to CM M_a but the constraints in L_{M_a} are nonfeasible. $MP(k)$ could return YES+MAYBE(S) only by making transition 11 or 12; each case is considered below.

Case 1: *$MP(k)$ makes transition 11*

$MP(k)$ could make transition 11 only if $V_{MP(k)}$ was set to 1 as a result of some child $Pa(k)$ returning YES. By Lemma F10, if $Pa(k)$ returned YES, then the constraints in $PS_{M_a}^4$ were feasible and so the constraints in L_{M_a} were feasible as well. This raises a contradiction and so $MP(k)$ could not make transition 11.

Case 2: $MP(k)$ makes transition 12

$MP(k)$ could make transition 12 if some child $Pa(k)$ returned YES+MAYBE(S), and $Pa(k)$ could do so only by making transition 10, 41, 31, or 42; each case is considered below.

Case 2.1: $Pa(k)$ makes transition 10

$Pa(k)$ could make transition 10 only if $V_{Pa(k)}$ was set to 1 by some child $Pb(k)$ of $Pa(k)$ returning YES. By Corolary E1 and Restriction SF5, if $Pb(k)$ returned YES, then the constraints in $PS_{M_a}^4$ were feasible, and by Lemma A5, the constraints in L_{M_a} were feasible. However, this raises a contradiction and so $Pa(k)$ could not make transition 10.

Case 2.2: $Pa(k)$ makes transition 31

$Pa(k)$ could make transition 31 only if $V_{Pa(k)}$ was set to 1 by some child of $Pa(k)$ or by one of its satisfaction functions. The arguments used in Case 2.1 apply once more and so no child of $Pa(k)$ could return YES, and no satisfaction function of $Pa(k)$ could return YES without violating Restriction SF5. Thus, $Pa(k)$ could not make transition 31.

Case 2.3: $Pa(k)$ makes transition 41 or 42

Transition 41 requires some child $Pb(k)$ to return YES+MAYBE(S) by making transition 10, 31, 42, or 41. The arguments used in Cases 1.2.1–1.2.3 now repeat with $Pb(k)$ replacing $Pa(k)$. $Pb(k)$ could not make transition 10 or 31 and so it must make transition 41 or 42 after one of its children returns YES+MAYBE(S), or it makes a recursive call and $MP(k+1)$ returns the same.

Assume that every procedure returns YES+MAYBE(S) after one of its children returns the same response. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that could not make transition 41. Consequently, some procedure $Pc(k)$ in the same subtree as $Pa(k)$ must make transition 42; $Pc(k)$ could be $Pa(k)$ or one of its descendents.

$Pc(k)$ could make transition 42 only if it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned YES+MAYBE(S). $MP(k+1)$ could return such a response only by making transition 11 or 12 but now the arguments used in Cases 1 and 2 apply once

more. Thus, no $MP(k+i)$, $i > 0$, could make transition 11 and so each must make transition 12 after one of its descendents makes a recursive call and $MP(k+i+1)$ returns YES+MAYBE(S). These arguments repeat indefinitely but by Lemma B2, the solver can recurse only a finite number of times before no procedure can make a recursive call. This raises a contradiction and so $Pa(k)$ could not make transition 41 or 42.

$Pa(k)$ could not return YES+MAYBE(S) if it could not make transition 10, 31, 41, or 42, and so $MP(k)$ could not make transition 12.

If $MP(k)$ could not make transition 11 or 12, a contradiction is raised and so the original assumption must be wrong. Therefore, if $MP(k)$ returns YES+MAYBE(S), then the constraints in L_{M_a} are feasible.

Result 2 and 3:

$MP(k)$ could return YES+MAYBE(S) by making transition 11 or 12; the results will be proven for each one separately. Note that by Corolary D5, if $MP(k)$ made either transition 11 or 12, then substitutions were forwarded back through $n \geq 1$ levels of recursion.

Part 1: $MP(k)$ makes transition 11

Assume that (1) some $MP(k+i)$, $i=1 \dots n$, could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, or (2) some procedure $Pj(k+i)$, $i=1 \dots n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. $MP(k)$ could make transition 11 only if some child $Pi(k)$ returned MAYBE(S) by making transition 32 or 9; each case is considered below.

Case 1: $Pi(k)$ makes transition 32

If $Pi(k)$ made transition 32 after sending a list of substitutions to $MP(k+1)$, then $MP(k+1)$ returned MAYBE(S). By Property B3, if $MP(k+1)$ returned MAYBE(S), then (1) no $MP(k+i)$, $i=1 \dots n$, could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions applied, and (2) no procedure $Pj(k+i)$,

$i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. This raises a contradiction and so $Pi(k)$ could not make transition 32.

Case 2: $Pi(k)$ makes transition 9

$Pi(k)$ could make transition 9 only if some child $Pj(k)$ returned MAYBE(S) and $Pj(k)$ could do so only by making transition 32 or 9. The above arguments now repeat with $Pj(k)$ replacing $Pi(k)$ and so it could not have made transition 32 without raising a contradiction. Thus, it must have made transition 9 after some child returned MAYBE(S).

These arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 9 after one of its children returns MAYBE(S). Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that could not make transition 9. This raises a contradiction and so $Pi(k)$ could not make transition 9.

If $Pi(k)$ could not make transition 9 or 32, then it could not return MAYBE(S) and $MP(k)$ could not make transition 11. This raises a contradiction and so if $MP(k)$ returns YES+MAYBE(S) using transition 11, then Results (2) and (3) hold.

Part 2: $MP(k)$ makes transition 12

Assume that (1) some $MP(k+i)$, $i=1, \dots, n$, could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied, or (2) some procedure $Pj(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. $MP(k)$ could make transition 12 only if some child $Pa(k)$ returned YES+MAYBE(S), and $Pa(k)$ could do so only by making transition 10, 41, 31, or 42; each case is considered below.

Case 2.1: $Pa(k)$ makes transition 31

$Pa(k)$ could make transition 31 only if it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned MAYBE(S). However, by Property B3, if $MP(k+1)$ returned MAYBE(S).

then (1) no $MP(k+i)$, $i=1, \dots, n$, could determine if the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions applied, and (2) no procedure $Pj(k+i)$, $i=1, \dots, n$, had a satisfaction function (on an explored search path) which could determine that the constraints in $PS_{M_a}^+$ were feasible or nonfeasible with the substitutions in S applied. This raises a contradiction and so $Pi(k)$ could not make transition 31.

Case 2.2: $Pa(k)$ makes transition 10

$Pa(k)$ could make transition 10 only if some child $Pb(k)$ of $Pa(k)$ returned $MAYBE(S)$, and $Pb(k)$ could do so only by making transition 31 or 9. If it made transition 31, the arguments of Case 2.1 now repeat and a contradiction arises. If it made transition 9 after some child $Pc(k)$ returned $MAYBE(S)$, then $Pc(k)$ made transition 31 or 9, and so on.

These arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 9 after one of its children returns $MAYBE(S)$. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that could not make transition 9. This raises a contradiction and so $Pa(k)$ could not make transition 10.

Case 2.3: $Pa(k)$ makes transition 41 or 42

$Pa(k)$ could make transition 41 if some child $Pb(k)$ returned $YES+MAYBE(S)$ using transition 10, 31, 42, or 41. The arguments used in Cases 2.1–2.3 now repeat with $Pb(k)$ replacing $Pa(k)$. $Pb(k)$ could not make transition 10 or 31 and so it must make transition 41 or 42 after one of its children returns $YES+MAYBE(S)$ or it makes a recursive call and $MP(k+1)$ returns the same.

Assume that every procedure returns $YES+MAYBE(S)$ after one of its children returns the same. Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that could not make transition 41. Consequently, some procedure $Pc(k)$ in the same subtree as $Pa(k)$ must make transition 42; $Pc(k)$ could be $Pa(k)$ or one of its descendants.

$Pc(k)$ could make transition 42 only if it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned $YES+MAYBE(S)$. $MP(k+1)$ could return such a response only by making

transition 11 or 12 but now the arguments used in Parts 1 and 2 apply again. No $MP(k+i)$, $i > 0$, could make transition 11 and so each must make transition 12 after one of its descendants makes a recursive call (using transition 42) and $MP(k+i+1)$ returns YES+MAYBE(S).

These arguments repeat indefinitely but by Lemma B2, the solver can recurse only a finite number of times before no procedure could make a recursive call. This raises a contradiction and so $Pa(k)$ could not make transition 41 or 42.

$Pa(k)$ could not return YES+MAYBE(S) if it could not make transition 10, 31, 41, or 42, and so a contradiction arises. Thus, if $MP(k)$ returns YES+MAYBE(S) using transition 12, then Results (2) and (3) hold.

Q.E.D.

Property B5

If $MP(k)$ returns YES(S) in response to a CM M_a , then (1) the constraints in L_{M_a} are feasible, and (2) S contains a definite partial or full solution to the constraints in L_{M_a} .

Proof:

Each result is considered below.

Result 1: The constraints in L_{M_a} are feasible

Assume that $MP(k)$ returns YES(S) in response to CM M_a but L_{M_a} contains some nonfeasible constraint c . $MP(k)$ could return YES(S) only by making transition 25, 28, or 13; each case is considered below.

Case 1: $MP(k)$ makes transition 25

$MP(k)$ could make transition 25 only if it totally reduced and satisfied every constraint in its problem set, including c . However, if c is nonfeasible, a contradiction is raised and so $MP(k)$ could not make transition 25.

Case 2: $MP(k)$ makes transition 28

$MP(k)$ could make transition 28 only if $V_{MP(k)}$ was set to 1 as a result of some child $Pj(k)$ returning YES. By Lemma F10, if $Pj(k)$ returned YES, then the con-

straints in $PS_{M_a}^4$ were feasible, and by Lemma A5, the constraints in L_{M_a} are feasible. However, if L_{M_a} contains some nonfeasible constraint c , a contradiction is raised and so $MP(k)$ could not make transition 28.

Case 3: $MP(k)$ makes transition 13

$MP(k)$ could make transition 13 only if some child $Pj(k)$ returned $YES(S)$, and $Pj(k)$ could return $YES(S)$ only by making transition 11 or 33; each case is considered below.

Case 3.1: $Pj(k)$ made transition 11

$Pj(k)$ could make transition 11 only if it received $YES(S)$ from some child $Pm(k)$. $Pm(k)$ could return $YES(S)$ only if it made transition 11 or 13 but if it made 11, one of its children made transition 11 or 13, and so on. Assume that $Pj(k)$ and all of its descendents made transition 11 after some child returned $YES(S)$. Since the number of procedures is finite, eventually a childless procedure would be encountered that could not make transition 11. This raises a contradiction and so some procedure in the same subtree as $Pj(k)$ must make transition 33. Assume without loss of generality that it is $Pj(k)$.

Case 3.2: $Pj(k)$ made transition 33

$Pj(k)$ could make transition 33 only if it sent a CM to $MP(k+1)$ and $MP(k+1)$ returned $YES(S)$. $MP(k+1)$ could return such a response only by making transition 25, 28, or 13, but now the arguments used in Cases 1–3 apply again. No $MP(k+i)$, $i > 0$, could make transition 25 or 28 and so each $MP(k+i)$ must make transition 13 after one of its descendents makes a recursive call and $MP(k+i+1)$ returns $YES(S)$. These arguments repeat indefinitely but by Lemma B2, the solver can recurse only a finite number of times before no procedure can make a recursive call. This raises a contradiction and so $Pj(k)$ could not make transition 33.

If $Pj(k)$ could not make transition 11 or 33, then it could not return $YES(S)$; it follows that $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, then it could not return $YES(S)$. This raises a contradiction and so the original assumption must be wrong.

Therefore, if $MP(k)$ returns $YES(S)$ in response to a CM M_a , then the constraints in L_{M_a} are feasible.

Result 2: *S contains a definite partial or full solution*

Assume that S contains a list of substitutions which makes some constraint $c \in L_{M_a}$ nonfeasible. $MP(k)$ could return $YES(S)$ only by making transition 25, 28, or 13; each case is considered below.

Case 1: *$MP(k)$ makes transition 25*

$MP(k)$ could make transition 25 only if it totally reduced and satisfied every constraint in $PS_{M_a}^+$ using the substitutions in S . However, if the substitutions make constraint c nonfeasible as assumed, a contradiction arises and so $MP(k)$ could not make transition 25.

Case 2: *$MP(k)$ makes transition 28*

$MP(k)$ could make transition 28 only if $V_{MP(k)}$ was set to 1 as a result of some child $Pj(k)$ returning YES. By Lemma F10, if $Pj(k)$ returned YES, then the constraints in $PS_{M_a}^4$ were feasible. Since the substitutions in S are those which $MP(k)$ used to transform $PS_{M_a}^+$ into $PS_{M_a}^4$, then the substitutions did not cause any constraint $c \in L_{M_a}$ to become nonfeasible. This raises a contradiction and so $MP(k)$ could not make transition 28.

Case 3: *$MP(k)$ makes transition 13*

$MP(k)$ could make transition 13 only if some child $Pj(k)$ returned $YES(S)$, and $Pj(k)$ could do so only by making transition 11 or 33. The arguments which were used in Cases 3.1 and 3.2 of Result 1 may now be repeated and so if $Pj(k)$ returned $YES(S)$, then $MP(k+1)$ was called and returned the same. $MP(k+1)$ could return $YES(S)$ only by making transition 25, 28, or 13, but now the arguments of Cases 1–3 of Result 2 apply again. Thus, no $MP(k+i)$, $i > 0$, could make transition 25 or 28 and so each $MP(k+i)$ must make transition 13 after one of its descendants makes a recursive call and $MP(k+i+1)$ returns $YES(S)$. These arguments repeat indefinitely but by Lemma B2, the solver can recurse only a finite number of times

before no procedure can make a recursive call. This raises a contradiction and so $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, then it could not return $YES(S)$. This raises a contradiction and so the original assumption must be wrong. Thus, if $MP(k)$ returns $YES(S)$, then S contains a definite full or partial solution.

Q.E.D.

Property B6.1

If $MP(k)$ returns NO in response to a CM M_i , then $PS_{M_i}^+$ contained nonfeasible constraints.

Proof:

This property is identical to Lemma F2 and so the proof of the lemma suffices.

Q.E.D.

Property B6.2

If $MP(k)$ returns NO in response to an AVR regarding some CM M_i , then all possible solutions to the constraints in L_{M_i} have been returned already.

Proof:

Assume that $MP(k)$ returns NO in response to an AVR regarding some CM M_i , but all possible solutions to the constraints in L_{M_i} have not yet been returned. $MP(k)$ could return NO after receiving an AVR only by making transition 16 or 21; each case is considered below.

Case 1: $MP(k)$ makes transition 16

$MP(k)$ could make transition 16 only if it has first reached state 1. This requires $MP(k)$ to have first made transition 15 or 25 after (1) detecting that every constraint in $PS_{M_i}^+$ is satisfied or (2) totally reducing and satisfying every constraint in $PS_{M_i}^+$ using the only possible substitutions. Since no other solutions are possible, a contradiction arises and so $MP(k)$ could not make transition 16.

Case 2: *MP(k) makes transition 21*

By Property A4.1, an AVR causes the forward search to be continued from the last choicepoint. If $MP(k)$ subsequently makes transition 21 after some child $Pa(k)$ returns NO, $Pa(k)$ was sent a CM or an AVR, depending on whether or not it returned the last solution that was forwarded by $MP(k)$; each case is considered below.

Case 2.1: *Pa(k) returns NO in response to a CM*

By Lemma F1, if $Pa(k)$ returned NO in response to a CM, then $PS_{M_i}^4$ contained nonfeasible constraints. Since $PS_{M_i}^4$ is the result of applying the only possible solutions to $PS_{M_i}^+$, L_{M_i} must contain nonfeasible constraints and so no solution is possible. (Any solution previously returned by $MP(k)$ must have been tentative.) It follows that $Pa(k)$ could not return NO in response to a CM.

Case 2.2: *Pa(k) returns NO in response to an AVR*

By Lemma F3, if $Pa(k)$ returned NO in response to an AVR, then no solutions to $PS_{M_i}^4$ are possible other than the ones that have been returned already. Since $PS_{M_i}^4$ is the result of applying the only possible solutions to $PS_{M_i}^+$, there can be no other solutions to L_{M_i} . This raises a contradiction and so $Pa(k)$ could not return NO in response to an AVR.

If $Pa(k)$ could not return NO in response to a CM or an AVR, then $MP(k)$ could not make transition 21.

If $MP(k)$ could not make transition 16 or 21, it could not return NO after receiving an AVR. Thus, the original assumption must be wrong and so if $MP(k)$ returns NO in response to an AVR regarding some CM M_i , then all possible solutions to L_{M_i} have been returned already.

Q.E.D.

D.3 Responses Occur if Conditions Met

The following proofs indicate that the solver will or will not return various results if certain conditions are met.

Property C1.1

For any $k \geq 1$, if $MP(k)$ is sent some CM M_a , then $MP(k)$ will not return MAYBE or YES.

Proof:

Assume that for some $k \geq 1$, $MP(k)$ could return MAYBE or YES in response to a CM; it could do so only by making transition 15, 5, or 6. $MP(k)$ would receive M_a only if some procedure $Pi(k-1)$ made transition 26 or 27 after calling some satisfaction function F and F returned a non-empty list of substitutions. Thus, L_{M_a} would contain at least one assignment constraint.

$MP(k)$ would move to state 7 after receiving M_a but because there must have been an unsatisfied constraint in the problem set of $MP(k-1)$ for F to generate a substitution, $MP(k)$ could not make transition 15. Assignment constraints cannot be MP-rejectable and so $MP(k)$ would have to make transition 24 and generate at least one substitution for $D(k)$. However, if $D(k)$ is non-empty, $MP(k)$ could not make transition 5 or 6, the only transitions remaining which allow $MP(k)$ to return MAYBE or YES. This raises a contradiction and so the original assumption must be wrong. Thus, for any $k \geq 1$, if $MP(k)$ is sent a CM, then $MP(k)$ will not return MAYBE or YES.

Q.E.D.

Property C1.2

If $MP(k)$ receives a CM M_a and the constraints in L_{M_a} are feasible, then $MP(k)$ will not return NO.

Proof:

Assume that the constraints in L_{M_a} are feasible but $MP(k)$ could return NO. By Lemma F2, if $MP(k)$ returned NO, then L_{M_a} contained nonfeasible constraints. This

raises a contradiction and so the assumption must be wrong. Thus, the property holds.

Q.E.D.

Property C1.3

If $MP(k)$ receives a CM M_a containing a nonfeasible constraint c , then YES, YES(S), and YES+MAYBE(S) will not be returned.

Proof:

Assume that (1) $MP(k)$ receives a CM M_a , (2) some constraint $c \in L_{M_a}$ is nonfeasible, and (3) $MP(k)$ could return YES, YES(S) or YES+MAYBE(S). By Properties B2, B4, and B5, if $MP(k)$ returns any one of these replies, then the constraints in L_{M_a} are feasible. This raises a contradiction and so the assumption must be wrong. Thus, if $MP(k)$ receives a CM containing a nonfeasible constraint, then YES, YES(S), and YES+MAYBE(S) will not be returned.

Q.E.D.

Property C2.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one is not satisfied,
3. no procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^+$,
4. no procedure has a satisfaction function which can determine that a subset of constraints in $PS_{M_a}^+$ is nonfeasible,
5. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, and

6. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$,

then $MP(k)$ will return MAYBE.

Proof:

Assume that given the above conditions, $MP(k)$ could return a response other than MAYBE. By Lemma D1, $MP(k)$ would call at least one of its children before returning a reply, and so it must return NO, MAYBE(S), YES, YES+MAYBE(S), or YES(S) using transition 21, 6, 27, 28, 10, 11, 12, or 13; each case is considered below. Note that by Corolary C6, $PS_{M_a}^+$ could not contain any MP-solvable or MP-rejectable constraints, and so $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical.

Case 1: $MP(k)$ makes transition 21

Given the above conditions, $MP(k)$ could return NO only by making transition 21 after some child $Pa(k)$ returned NO in response to a CM. By Lemma F1, if $Pi(k)$ returned NO, then a satisfaction function of $Pi(k)$ or one of its descendents determined that $PS_{M_a}^4$ contained nonfeasible constraints. However, it was assumed in condition (4) that no procedure could find constraints in $PS_{M_a}^+$ to be nonfeasible and since $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, a contradiction is raised. Thus, $Pa(k)$ could not return NO and $MP(k)$ could not make transition 21.

Case 2: $MP(k)$ makes transition 6

$MP(k)$ could return YES using transition 6 only if $V_{MP(k)}$ was set to 1 by some child $Pi(k)$ returning YES. By Lemma E1, at least one satisfaction function F of $Pi(k)$ or one of its descendents returned YES: i.e., F determined that the constraints in $PS_{M_a}^4$ were feasible. However, it was assumed in condition (3) that no procedure had a function which could determine that the constraints in $PS_{M_a}^+$ were feasible and since $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, a contradiction is raised. Thus, $MP(k)$ could not make transition 6.

Case 3: $MP(k)$ makes transition 27 or 28

$MP(k)$ could return MAYBE(S) or YES(S) by making transition 27 or 28 (or

10 or 13, but these are discussed in Case 4). As stated in the beginning, Corolary C6 indicates that $MP(k)$ could not determine any values using transitions 24 and 3 and so $D(k)$ would be empty. Thus, $MP(k)$ could not make transition 27 or 28.

Case 4: $MP(k)$ makes transition 10, 11, 12, or 13

By Lemma D5, if $MP(k)$ made transition 10, 11, 12, or 13 after some child $P_i(k)$ returned $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$, then some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 31, 42, 10, or 41 after:

- it called a function F ,
- F returned $MAYBE(S')$ or $YES(S')$,
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned $MAYBE(S)$, $YES+MAYBE(S)$, or $YES(S)$.

However, F could not return $YES(S')$ because it was assumed in condition (5) that no procedure had a function which could generate a definite solution to some subset of constraints in $PS_{M_a}^+$, and F could not return $MAYBE(S')$ because it was assumed in condition (6) that no procedure had a function which could generate a tentative solution which would not be rejected by $MP(k+1)$. This raises a contradiction and so $P_j(k)$ could not make transition 31, 42, 10, or 41, and $MP(k)$ could not make transition 10, 11, 12, or 13.

If $MP(k)$ could not make any of the above transitions, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will return $MAYBE$.

Q.E.D.

Property C2.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraint in L_{M_a} is MP-solvable or MP-rejectable and at least one is not satisfied, and

3. no constraint in $PS_{M_a}^+$ is recognizable by any procedure $P_i(k)$,
then $MP(k)$ will return MAYBE.

Proof:

By Property C2.1, given conditions 1 and 2 above plus the following:

- P1. no procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^+$,
- P2. no procedure has a satisfaction function which can determine that a subset of constraints in $PS_{M_a}^+$ is nonfeasible,
- P3. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, and
- P4. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$,

then $MP(k)$ will return MAYBE. Thus, it suffices to show that if no procedure can recognize any constraint in $PS_{M_a}^+$, then conditions P1 – P4 will hold.

Assume that no procedure can recognize any constraint in $PS_{M_a}^+$, but at least one of P1 – P4 does not hold, that is, some procedure $P_i(k)$ has a satisfaction function F which can:

- 1. verify the constraints in $PS_{M_a}^+$,
- 2. determine that a subset of constraints in $PS_{M_a}^+$ is nonfeasible,
- 3. generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, or
- 4. generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which would not be rejected by $MP(k+1)$.

Let $Pj(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ could be $Pj(k)$ or one of its descendants.

By Restriction RS1, if $Pi(k)$ has a function F as described above, then the recognition rules of $Pj(k)$ must recognize at least one constraint in $PS_{M_a}^4$, but by condition (3), $Pj(k)$ could not recognize any constraint in $PS_{M_a}^+$. Since the two sets of constraints are identical, a contradiction is raised and so the assumption must be wrong. Thus, $Pi(k)$ cannot exist and so conditions P1-P4 must hold. It follows by Property C2.1 that $MP(k)$ will return MAYBE.

Q.E.D.

Property C2.3

If $MP(k)$ is sent an AVR after having returned MAYBE, it will return ABANDON.

Proof:

The result follows directly from the state transition diagram for $MP(k)$. $MP(k)$ could return MAYBE only by making transition 5 to state 2. If $MP(k)$ received an AVR while in state 2, it would make transition 20 and return ABANDON; thus, the property holds.

Q.E.D.

Property C3.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$, and
2. $MP(k)$ can determine that every constraint in $PS_{M_a}^+$ is satisfied,

then $MP(k)$ will return YES.

Proof:

The result follows directly from the state diagram for $MP(k)$. Whenever $MP(k)$ is sent a CM, it makes transition 1, 17, 18, or 19 to state 1. If $MP(k)$ could determine that every constraint in $PS_{M_a}^+$ was satisfied, then it would make transition 15 and

in so doing return YES. Thus, the property holds.

Q.E.D.

Property C3.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. no constraints in L_{M_a} are MP-solvable and at least one is not satisfied,
3. at least one procedure $P_i(k)$ has a function F which can verify the constraints in L_{M_a} ,
4. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of $PS_{M_a}^+$, and
5. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$,

then $MP(k)$ will return YES.

Proof:

Assume that given the above conditions, $MP(k)$ could return a response other than YES. If $P_i(k)$ has a function F which can verify the constraints in L_{M_a} , then there can be no MP-rejectable constraints in L_{M_a} . It follows, given condition (2) and Lemma D1, that $MP(k)$ would call at least one of its children before replying, and so it must return NO, MAYBE, MAYBE(S), YES+MAYBE(S), or YES(S) using transition 21, 5, 27, 28, 10, 11, 12, or 13; each case is considered below. Note that by Corolary C6, $PS_{M_a}^+$ could not contain any MP-solvable or MP-rejectable constraints, and so $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical.

Case 1: $MP(k)$ makes transition 21

$MP(k)$ could return NO only by making transition 21. However, it was determined that the constraints in L_{M_a} are feasible and so by Property C1.2, $MP(k)$ could not return NO.

Case 2: $MP(k)$ makes transition 5

Lemma F8 indicates that if $MP(k)$ returned MAYBE, then no procedure had a satisfaction function which could determine that the constraints in L_{M_a} were feasible. However, this contradicts condition (3) and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ makes transition 27 or 28

$MP(k)$ could return MAYBE(S) or YES(S) by making transition 27 or 28 (or 10 or 13, but these cases are discussed in Case 4). As stated in the beginning, Corolary C6 indicates that $MP(k)$ could not determine any values using transitions 24 and 3 and so $D(k)$ would be empty. Thus, $MP(k)$ could not make transition 27 or 28.

Case 4: $MP(k)$ makes transition 10, 11, 12, or 13

By Lemma D5, if $MP(k)$ made transition 10, 11, 12, or 13 after some child $Pm(k)$ returned MAYBE(S), YES+MAYBE(S), or YES(S), then some procedure $Pn(k)$ in the same subtree as $Pm(k)$ made transition 31, 42, 10, or 41 after:

- it called a function F_n ,
- F_n returned MAYBE(S') or YES(S'),
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned MAYBE(S), YES+MAYBE(S), or YES(S).

F_n could not return YES(S') because it was assumed in condition (4) that no procedure had a function which could generate a definite solution to some subset of constraints in $PS_{M_a}^+$, and F_n could not return MAYBE(S') because it was assumed in condition (5) that no procedure had a function which could generate a tentative solution which would not be subsequently rejected by $MP(k+1)$. This raises a contradiction and so $MP(k)$ could not make transition 10, 11, 12, or 13.

If $MP(k)$ could not make any of the above transitions, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions,

MP(k) will return YES.

Q.E.D.

Property C3.3

If MP(k) is sent an AVR after having returned YES, it will return NO or ABANDON.

Proof:

The proof follows directly from the state transition diagram for MP(k). MP(k) could return YES only by making transition 15 to state 1 or transition 6 to state 2. If MP(k) received an AVR while in state 1, it would make transition 16 and return NO. If it was in state 2, it would make transition 20 and return ABANDON. Thus, if MP(k) is sent an AVR after having returned YES, it will return NO or ABANDON. **Q.E.D.**

Property C4.1

Given the following conditions:

1. a CM M_a is sent to MP(k),
2. L_{M_a} contains one or more MP-solvable constraints,
3. MP(k) cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints in L_{M_a} ,
4. MP(k) cannot detect any nonfeasible constraints in any instance of $PS_{M_a}^*$,
5. no procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^4$,
6. no procedure has a satisfaction function which can determine that a subset of constraints in $PS_{M_a}^4$ is nonfeasible,
7. if any procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$, then MP($k+1$) will return MAYBE(S) when sent the substitutions, and

8. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of $PS_{M_a}^4$,

then $MP(k)$ will return $MAYBE(S)$.

Proof:

Assume that given the above conditions, $MP(k)$ could return some other solution, i.e., NO, MAYBE, YES, YES+MAYBE(S), or YES(S). Each case is considered below.

Case 1: $MP(k)$ returns NO

$MP(k)$ could return NO only by making transition 14, 26, or 21. It could make one of the first two only if it detected a contradiction in some instance of $PS_{M_a}^*$, that is, after zero or more MP-solvable constraints had been satisfied. However, this would violate condition (4) and so $MP(k)$ could not make transition 14 or 26.

$MP(k)$ could make transition 21 only if some child $Pi(k)$ returned NO in response to a CM, but by Lemma F1, if $Pi(k)$ returned NO, then a function of $Pi(k)$ or one of its descendents determined that $PS_{M_a}^4$ contained nonfeasible constraints. This contradicts condition (6) and so $MP(k)$ could not make transition 21. It follows that $MP(k)$ could not return NO.

Case 2: $MP(k)$ returns MAYBE

$MP(k)$ could return MAYBE only by making transition 5 after it had exhausted its list of children and all returned MAYBE; as well, $MP(k)$ could not have generated any solutions using transition 24 and 3. However, condition (2) states that L_{M_a} contains at least one MP-solvable constraint and so $MP(k)$ would make transition 24 after reaching state 7. This raises a contradiction and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

$MP(k)$ could return YES only by making transition 15 or 6. It could make transition 15 only if every constraint in $PS_{M_a}^+$ was satisfied but since L_{M_a} contains at least one MP-solvable constraint, it could not do so. $MP(k)$ could make transition

6 only if it had exhausted its list of children, at least one returned YES, and it had not generated any values using transition 24 and 3. However, condition (2) states that L_{M_a} contains at least one MP-solvable constraint and so $MP(k)$ would make transition 24 after reaching state 7. This raises a contradiction and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns YES+MAYBE(S)

$MP(k)$ could return YES+MAYBE(S) only by making transition 11 or 12; each case is considered below.

Case 4.1: $MP(k)$ makes transition 11

$MP(k)$ could make transition 11 only if $V_{MP(k)}$ was set to 1 as a result of some child $Pi(k)$ returning YES. By Lemma E1, $Pi(k)$ could return YES only if a satisfaction functions of $Pi(k)$ or one of its descendents returned YES, that is, determined that the constraints in $PS_{M_a}^4$ were feasible. However, this violates condition (5) and so $MP(k)$ could not make transition 11.

Case 4.2: $MP(k)$ makes transition 12

This transition requires some child $Pi(k)$ to return YES+MAYBE(S), but by Lemma E3, it could do so only if:

1. at least one satisfaction function of $Pi(k)$ or one of its descendents returned YES,
2. at least one satisfaction function of $Pi(k)$ or one of its descendents returned YES(S), or
3. $MP(k+1)$ returned YES(S) or YES+MAYBE(S) after being sent a CM.

The first case could not arise without violating condition (5); the second case could not arise because condition (8) indicates that no function could have been called and returned YES(S); the third case could not arise because condition (7) indicates that if $MP(k+1)$ was sent a CM after a function returned MAYBE(S), then $MP(k+1)$ would return MAYBE(S). Thus, if none of the above conditions are possible, a contradiction arises and so $MP(k)$ could not make transition 12.

If $MP(k)$ could not make transition 11 or 12, it follows that it could not return $YES+MAYBE(S)$.

Case 5: $MP(k)$ returns $YES(S)$

$MP(k)$ could return $YES(S)$ only by making transition 25, 28, or 13; each case is considered below.

Case 5.1: $MP(k)$ makes transition 25

$MP(k)$ could make transition 25 only if it had totally reduced and satisfied every constraint in $PS_{M_a}^+$ using the values determined from the MP-solvable constraints. However, this would violate condition (3) and so $MP(k)$ could not make transition 25.

Case 5.2: $MP(k)$ makes transition 28

$MP(k)$ could make transition 28 only if it had exhausted its list of children and $V_{MP(k)}$ was set to 1 after some child $Pi(k)$ returned YES. By Lemma E1, if $Pi(k)$ returned YES, then at least one satisfaction function of $Pi(k)$ or one of its descendants returned YES, that is, determined that the constraints in $PS_{M_a}^4$ were feasible. However, this would violate condition (5) and so $MP(k)$ could not make transition 28.

Case 5.3: $MP(k)$ makes transition 13

By Lemma D3, if $MP(k)$ made transition 13 after some child $Pi(k)$ returned $YES(S)$, then $Pi(k)$ or one of its descendants made transition 33 after calling a function F and sending a CM to $MP(k+1)$, and $MP(k+1)$ returned $YES(S)$. As well, F must have returned $YES(S)$ but by condition (8), F could not do so. This raises a contradiction and so $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, it follows that it could not return $YES(S)$.

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions,

MP(k) will return MAYBE(S).

Q.E.D.

Property C4.2

Given the following conditions:

1. a CM M_a is sent to MP(k),
2. no constraint in L_{M_a} is MP-rejectable and at least one is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then MP(k) cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints, and MP(k) cannot detect any nonfeasible constraints in any instance of $PS_{M_a}^*$,
4. at least one procedure $Pi(k)$ has a function F which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that MP($k+1$) returns MAYBE(S) when sent the substitutions,
5. if any procedure has a satisfaction function which can verify the constraints in $PS_{M_a}^4$, it does not call the function before $Pi(k)$ calls F ,
6. if any procedure has a function which can determine that a subset of the constraints in $PS_{M_a}^4$ is nonfeasible, it does not call the function before $Pi(k)$ calls F ,
7. if any procedure has a function which can generate a definite partial or full solution to a subset of $PS_{M_a}^4$, it does not call the function before $Pi(k)$ calls F , and
8. if any procedure has a function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that MP($k+1$) would return YES+MAYBE(S) or YES(S) if sent the solution, it does not call the function before $Pi(k)$ calls F ,

then $MP(k)$ will return $MAYBE(S)$.

Proof:

Let $Pj(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ could be $Pj(k)$ or one of its descendents. Assume that given the above conditions, $MP(k)$ could return some other response, that is, NO, YES, MAYBE, YES(S), or YES+MAYBE(S); each case is considered below.

Case 1: $MP(k)$ returns NO

$MP(k)$ could return NO only by making transition 14, 26, or 21; each case is considered below.

Case 1.1: $MP(k)$ makes transition 14 or 26

$MP(k)$ could make transition 14 or 26 only if it detected a nonfeasible constraint in some instance of $PS_{M_e}^*$. However, if it did, condition (1) or (2) would be violated and so $MP(k)$ could not make transition 14 or 26.

Case 1.2: $MP(k)$ makes transition 21

$MP(k)$ could make transition 21 only if some child $Pm(k)$ returned NO in response to a CM, but by Lemma F1, if $Pm(k)$ returned NO, then some function F' of $Pm(k)$ or one of its descendents determined that $PS_{M_a}^4$ contained nonfeasible constraints. If F' was called before $Pi(k)$ called F , condition (6) would be violated and so $Pi(k)$ must have called F already. If it did, $Pi(k)$ would have made transition 32 and $MAYBE(S)$ would have been forwarded back up through the hierarchy to $MP(k)$. It follows that $MP(k)$ would have made transition 10 and so a contradiction is raised. Thus, $MP(k)$ could not make transition 21.

If $MP(k)$ could not make transition 14, 26, or 21, it follows that $MP(k)$ could not return NO.

Case 2: $MP(k)$ returns MAYBE

$MP(k)$ could return MAYBE only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical. By Lemma F8, if $MP(k)$ returned MAYBE, then no procedure had a function which could generate

a tentative solution to a subset of $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, condition (4) is violated and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

$MP(k)$ could return YES only by making transition 15 or 6. It could make transition 15 only if every constraint in $PS_{M_a}^+$ was satisfied but this would violate condition (2). It could make transition 6 only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ would be identical. By Lemma F11, if $MP(k)$ returned YES, then no procedure had a function which could generate a tentative solution to a subset of $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, this would violate condition (4) and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns YES+MAYBE(S)

$MP(k)$ could return YES+MAYBE(S) only by making transition 11 or 12; each case is considered below.

Case 4.1: $MP(k)$ makes transition 11

$MP(k)$ could make transition 11 only if $V_{MP(k)}$ was set to 1 by some child $Pm(k)$ returning YES before $MP(k)$ sent a CM to $Pj(k)$. By Lemma E1, $Pm(k)$ could return YES only if a satisfaction function F' of $Pm(k)$ or one of its descendants returned YES. However, F' would have to be called before $Pi(k)$ called F and this would violate condition (5). Thus, $MP(k)$ could not make transition 11.

Case 4.2: $MP(k)$ makes transition 12

This transition requires some child $Pm(k)$ to return YES+MAYBE(S) by making transition 10, 41, 31 or 42; each case is considered below.

Case 4.2.1: $Pm(k)$ makes transition 31

$Pm(k)$ could make transition 31 only if $V_{Pm(k)}$ was set to 1 before $MP(k+1)$ returned MAYBE(S). $V_{Pm(k)}$ could be set to 1 only by (1) some descendent of $Pm(k)$ returning YES, or (2) $Pi(k)$ calling some other function before F . However,

given Lemma E1 and condition (5), this is not possible and so $Pm(k)$ could not make transition 31.

Case 4.2.2: $Pm(k)$ makes transition 10

$Pm(k)$ could make transition 10 only if $V_{Pm(k)}$ was set to 1 before some child returned MAYBE(S), and $V_{Pm(k)}$ could be set to 1 only by some other descendent of $Pm(k)$ returning YES. However, given Lemma E1 and condition (5), this is not possible and so $Pm(k)$ could not make transition 10.

Case 4.2.3: $Pm(k)$ makes transition 42

$Pm(k)$ could make transition 42 only if (1) a CM was sent to $MP(k+1)$ after some function F' was called, and (2) $MP(k+1)$ returned YES+MAYBE(S). Given condition (7), F' would have to return MAYBE(S) but if it did, condition (8) would be violated. Thus, $Pm(k)$ could not make transition 42.

Case 4.2.4: $Pm(k)$ makes transition 41

This transition requires some child $Pd(k)$ to return YES+MAYBE(S), but now the above arguments repeat with $Pd(k)$ replacing $Pm(k)$. $Pd(k)$ could not make transition 10, 31, or 42 and so it would have to make transition 41 after one of its children returned YES+MAYBE(S). These arguments repeat indefinitely and every time that they are used, some procedure at another level deeper in the hierarchy must make transition 41 after one of its children returns YES+MAYBE(S). Since the number of procedures in the solver is finite, eventually a childless procedure would be encountered that could not make transition 41 and so a contradiction is raised. Thus, $Pm(k)$ could not make transition 41.

If $Pm(k)$ could not make transition 10, 41, 31, or 42, it follows that $MP(k)$ could not make transition 12, and if $MP(k)$ could not make transition 11 or 12, it could not return YES+MAYBE(S).

Case 5: $MP(k)$ returns YES(S)

$MP(k)$ could return YES(S) only by making transition 25, 28, or 13; each case is considered below.

Case 5.1: $MP(k)$ makes transition 25

$MP(k)$ could make transition 25 only if it had totally reduced and satisfied every constraint in $PS_{M_a}^+$ using the values determined from the MP-solvable constraints. However, this would violate condition (3) and so $MP(k)$ could not make transition 25.

Case 5.2: $MP(k)$ makes transition 28

Let $Pa(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$; $Pi(k)$ could be $Pa(k)$ or one of its descendants. $MP(k)$ could make transition 28 only if it had exhausted its list of children and all had returned MAYBE or YES. By Lemmas F7 and F10, if $Pa(k)$ returned MAYBE or YES, then $Pa(k)$ nor any of its descendants had a satisfaction function which could generate a tentative solution which was not subsequently rejected by $MP(k+1)$. This contradicts condition (4) and so $MP(k)$ could not make transition 28.

Case 5.3: $MP(k)$ makes transition 13

By Lemma D3, if $MP(k)$ made transition 13 after some child $Pm(k)$ returned YES(S), then $Pm(k)$ or one of its descendants made transition 33 after:

- it called a function F' ,
- F' returned MAYBE(S') or YES(S'),
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned YES(S).

If F' was called before $Pi(k)$ called F , condition (7) would be violated and so $Pi(k)$ must have called F already. If it did, $Pi(k)$ would have made transition 32 and MAYBE(S) would have been forwarded back up through the hierarchy to $MP(k)$. It follows that $MP(k)$ would have made transition 10 and so a contradiction is raised. Thus, $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, it follows that it could not return YES(S).

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will return $MAYBE(S)$.

Q.E.D.

Property C5.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains at least one constraint which is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints,
4. at least one procedure $Pa(k)$ has a satisfaction function F_a which can verify the constraints in $PS_{M_a}^4$,
5. at least one procedure $Pb(k)$ has a function F_b which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return $MAYBE(S)$ if sent the substitutions,
6. $Pa(k)$ calls F_a before $Pb(k)$ calls F_b , and
7. if any procedure has a satisfaction function which can generate a definite or tentative solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return $YES(S)$ or $YES+MAYBE(S)$ if sent the solution, then the procedure does not call the function before $Pb(k)$ calls F_b ,

then $MP(k)$ will return $YES+MAYBE(S)$.

Proof:

Let $Pc(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pb(k)$; $Pb(k)$ could be $Pc(k)$ or one of its descendents. Let $Pd(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pa(k)$; $Pa(k)$ could be $Pd(k)$ or one of

its descendents. Now assume that given the above conditions, $MP(k)$ could return a response other than YES+MAYBE(S), that is, NO, MAYBE, YES, MAYBE(S), or YES(S); each case is considered below:

Case 1: $MP(k)$ returns NO

If function F_a of $Pa(k)$ can determine that the constraints in $PS_{M_a}^4$ are feasible, then the constraints in L_{M_a} must be feasible. However, by Property C1.2, if L_{M_a} contains feasible constraints, then $MP(k)$ could not return NO.

Case 2: $MP(k)$ returns MAYBE

$MP(k)$ could return MAYBE only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical. By Lemma F8, if $MP(k)$ returned MAYBE, then no procedure had a function which could generate a tentative solution to any constraint in $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, this violates condition (5) and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

$MP(k)$ could return YES only by making transition 15 or 6. It could make transition 15 only if every constraint in $PS_{M_a}^+$ was satisfied but this would violate condition (2). It could make transition 6 only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ would be identical. By Lemma F11, if $MP(k)$ returned YES, then no procedure had a function which could generate a tentative solution to any constraint in $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, this violates condition (5) and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns MAYBE(S)

$MP(k)$ could return MAYBE(S) only by making transition 10 or 27; each case is considered below.

Case 4.1: $MP(k)$ makes transition 27

$MP(k)$ could make transition 27 only if it had exhausted its list of children and all returned MAYBE (so that $V_{MP(k)}$ was set to 0). By Lemma F7, if $Pd(k)$ returned

MAYBE, then $Pd(k)$ nor any of its descendants could verify the constraints in $PS_{M_a}^4$. This contradicts condition (4) and so $MP(k)$ could not make transition 27.

Case 4.2: $MP(k)$ makes transition 10

By Lemma D2, if $MP(k)$ made transition 10 after some child $Pi(k)$ returned $MAYBE(S)$, then some procedure $Pm(k)$ in the same subtree as $Pi(k)$ made transition 32 after:

- it called a function F' ,
- F' returned $MAYBE(S')$,
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned $MAYBE(S)$.

Hence, F' must be F_b . If F_b was called before $Pa(k)$ called F_a , condition (6) would be violated and so $Pa(k)$ must have called F_a already. However, by Property B3, if $MP(k)$ returned $MAYBE(S)$, then no procedure had a satisfaction function on an explored search path which could determine that the constraints in $PS_{M_a}^4$ were feasible. This raises a contradiction and so $MP(k)$ could not make transition 10.

If $MP(k)$ could not make transition 27 or 10, it follows that $MP(k)$ could not return $MAYBE(S)$.

Case 5: $MP(k)$ returns $YES(S)$

$MP(k)$ could return $YES(S)$ only by making transition 25, 28, or 13; each case is considered below.

Case 5.1: $MP(k)$ makes transition 25

$MP(k)$ could make transition 25 only if it had totally reduced and satisfied every constraint in $PS_{M_a}^+$ using the values determined from the MP-solvable constraints. However, this would violate condition (3) and so $MP(k)$ could not make transition 25.

Case 5.2: $MP(k)$ makes transition 28

$MP(k)$ could make transition 28 only if it had exhausted its list of children and all had returned MAYBE or YES. By Lemmas F7 and F10, if $Pc(k)$ returned MAYBE or YES, then $Pc(k)$ nor any of its descendants had a satisfaction function which could generate a tentative solution which was not rejected by $MP(k+1)$. This violates condition (5) and so $MP(k)$ could not make transition 28.

Case 5.3: $MP(k)$ makes transition 13

By Lemma D3, if $MP(k)$ made transition 13 after some child $Pi(k)$ returns $YES(S)$, then some procedure $Pm(k)$ in the same subtree as $Pi(k)$ made transition 33 after:

- it called a function F' ,
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned $YES(S)$.

If F' was called before $Pb(k)$ called F_b , condition (7) would be violated and so $Pb(k)$ must have called F_b already.

If $Pb(k)$ had called F_b , condition (5) indicates that $Pb(k)$ would have made transition 32 and returned $MAYBE(S)$ to whichever procedure had sent it a CM. If its parent was a procedure other than $MP(k)$, then it and any other ancestor would have made transition 9, resulting in $MAYBE(S)$ being forwarded back up through the hierarchy to $MP(k)$. Thus, $MAYBE(S)$ would have been returned to $MP(k)$ before $Pi(k)$ could return $YES(S)$.

If $MAYBE(S)$ was returned to $MP(k)$, then $MP(k)$ would have made transition 10 or 11 and returned $MAYBE(S)$ or $YES+MAYBE(S)$. In either case, it would not return $YES(S)$ and so a contradiction arises. Thus, $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, it follows that $MP(k)$ could not return $YES(S)$.

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will return YES+MAYBE(S).

Q.E.D.

Property C5.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains at least one constraint which is not satisfied,
3. if L_{M_a} contains one or more MP-solvable constraints, then $MP(k)$ cannot totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints,
4. at least one procedure $Pb(k)$ has a function F_b which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ returns YES+MAYBE(S) when sent the substitutions, and
5. if any procedure has a satisfaction function which can generate a definite or tentative solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would return MAYBE(S) or YES(S) if sent the substitutions, then the procedure does not call the function before $Pb(k)$ calls F_b ,

then $MP(k)$ will return YES+MAYBE(S).

Proof:

Let $Pc(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pb(k)$; $Pb(k)$ could be $Pc(k)$ or one of its descendents. Assume that given the above conditions, $MP(k)$ could return a response other than YES+MAYBE(S), that is, NO, MAYBE, YES, MAYBE(S), or YES(S); each case is considered below:

Case 1: $MP(k)$ returns NO

By condition (4) and Property B4, if $MP(k+1)$ returned YES+MAYBE(S) after being sent a CM, then the (assignment) constraints which were sent to it were

feasible. It follows that the constraints in L_{M_a} are feasible, but by Property C1.2, $MP(k)$ could not return NO if it is sent a feasible list of constraints. This raises a contradiction and so $MP(k)$ could not return NO.

Case 2: $MP(k)$ returns MAYBE

$MP(k)$ could return MAYBE only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ are identical. By Lemma F8, if $MP(k)$ returned MAYBE, then no procedure had a function which could generate a tentative solution to any constraint in $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, this violates condition (4) and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

$MP(k)$ could return YES only by making transition 15 or 6. It could make transition 15 only if every constraint in $PS_{M_a}^+$ was satisfied but this would violate condition (2). It could make transition 6 only if it did not determine any values using transition 24 and 3, in which case $PS_{M_a}^4$ and $PS_{M_a}^+$ would be identical. By Lemma F11, if $MP(k)$ returned YES, then no procedure had a function which could generate a tentative solution to any constraint in $PS_{M_a}^+$ which was not subsequently rejected by $MP(k+1)$. Given that $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, this violates condition (4) and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns MAYBE(S)

$MP(k)$ could return MAYBE(S) only by making transition 10 or 27; each case is considered below.

Case 4.1: $MP(k)$ makes transition 27

$MP(k)$ could make transition 27 only if it had exhausted its list of children and all returned MAYBE (so that $V_{MP(k)}$ was set to 0). By Lemma F7, if $Pc(k)$ returned MAYBE, then $Pc(k)$ nor any of its descendants had a function which could generate a definite solution to some subset of constraints in $PS_{M_a}^4$, or a tentative solution which was not subsequently rejected by $MP(k+1)$. This contradicts condition (4) and so $MP(k)$ could not make transition 27.

Case 4.2: $MP(k)$ makes transition 10

By Lemma D2, if $MP(k)$ made transition 10 after some child $Pi(k)$ returned $MAYBE(S)$, then (1) $Pi(k)$ or one of its descendants made transition 32 after calling a function F' and sending a CM to $MP(k+1)$, and (2) $MP(k+1)$ returned $MAYBE(S)$. If F' was called before $Pb(k)$ called F_b , condition (5) would be violated and so $Pb(k)$ must have called F_b already.

If $Pb(k)$ had called F_b , condition (4) indicates that $Pb(k)$ would have made transition 42 and returned $YES+MAYBE(S)$ to whichever procedure had sent it a CM. If its parent was a procedure other than $MP(k)$, then it and any other ancestor would have made transition 41, resulting in $YES+MAYBE(S)$ being forwarded back up through the hierarchy to $MP(k)$. Thus, $YES+MAYBE(S)$ would be returned to $MP(k)$ before $Pi(k)$ could return $YES(S)$.

If $YES+MAYBE(S)$ was returned to $MP(k)$, then $MP(k)$ would have made transition 12 and returned the same, that is, it would not return $MAYBE(S)$. This raises a contradiction and so $MP(k)$ could not make transition 10.

If $MP(k)$ could not make transition 27 or 10, it follows that $MP(k)$ could not return $MAYBE(S)$.

Case 5: $MP(k)$ returns $YES(S)$

$MP(k)$ could return $YES(S)$ only by making transition 25, 28, or 13; each case is considered below.

Case 5.1: $MP(k)$ makes transition 25

$MP(k)$ could make transition 25 only if it had totally reduced and satisfied every constraint in $PS_{M_a}^+$ using the values determined from the MP-solvable constraints. However, condition (3) states that it could not do so and so $MP(k)$ could not make transition 25.

Case 5.2: $MP(k)$ makes transition 28

$MP(k)$ could make transition 28 only if it had exhausted its list of children and all had returned $MAYBE$ or YES . By Lemma F7, if $Pc(k)$ returned $MAYBE$, then $Pc(k)$ nor any of its descendants had a satisfaction function which could generate a

tentative solution which was not rejected by $MP(k+1)$, and by Lemma F10, if $Pd(k)$ returned YES, the same result holds. In either case, condition (5) is violated and so $MP(k)$ could not make transition 28.

Case 5.3: $MP(k)$ makes transition 13

By Lemma D3, if $MP(k)$ made transition 13 after some child $Pi(k)$ returned YES(S), then some procedure $Pm(k)$ in the same subtree as $Pi(k)$ made transition 33 after:

- it called a function F' ,
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned YES(S).

If F' was called before $Pb(k)$ called F_b , condition (5) would be violated and so $Pb(k)$ must have called F_b already.

If $Pb(k)$ had called F_b , the same line of reasoning that was used in Case 4.2 may be used to show that YES+MAYBE(S) would have been returned to $MP(k)$ before $Pi(k)$ could return YES(S). If YES+MAYBE(S) was returned to $MP(k)$, then $MP(k)$ would have returned the same using transition 12. This raises a contradiction and so $MP(k)$ could not make transition 13.

If $MP(k)$ could not make transition 25, 28, or 13, it follows that $MP(k)$ could not return YES(S).

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will return YES+MAYBE(S).

Q.E.D.

Property C6.1

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,

2. L_{M_a} contains one or more MP-solvable constraints, and
3. $MP(k)$ can totally reduce and satisfy its problem set based on the solutions of the MP-solvable constraints in L_{M_a} ,

then $MP(k)$ will return $YES(S)$.

Proof:

Assume that given the above conditions, $MP(k)$ could return some other response. After receiving M_a , $MP(k)$ would move to state 7 and make transition 14, 15, 2, or 24. If $MP(k)$ could totally reduce and satisfy its constraints by solving one or more MP-solvable constraints, then the constraints must be feasible and so L_{M_a} could not contain any MP-rejectable constraints. It follows, by Lemma C6, that $PS_{M_a}^+$ could not contain any MP-rejectable constraints and so $MP(k)$ could not make transition 14.

If L_{M_a} contains at least one MP-solvable constraints as stated, then $PS_{M_a}^+$ would contain an MP-solvable constraint. Thus, $MP(k)$ could not make transition 2 or 15 and so it must make transition 24. Given condition (3), it follows that $MP(k)$ would either make transition 25 immediately or it would solve another constraint using transition 3 and return to state 6. If $MP(k)$ does not return $YES(S)$ as assumed, then it must make transition 3 but the same arguments apply again and so eventually it must make transition 25. As soon as it does, it would return $YES(S)$ but this raises a contradiction. Thus, the original assumption must be wrong and so given the above conditions, $MP(k)$ will return $YES(S)$.

Q.E.D.

Property C6.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains one or more MP-solvable constraints,
3. at least one procedure $Pi(k)$ has a satisfaction function which can verify the constraints in $PS_{M_a}^4$,

4. no procedure has a satisfaction function which can generate a definite partial or full solution to a subset of the constraints in $PS_{M_a}^+$, and
5. no procedure has a satisfaction function which can generate a tentative partial or full solution to a subset of $PS_{M_a}^+$ which is not subsequently rejected by $MP(k+1)$,

then $MP(k)$ will return $YES(S)$.

Proof:

Let $Pb(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pa(k)$, and assume that given the above conditions, $MP(k)$ could return some other solution, that is NO, YES, MAYBE, MAYBE(S), or YES+MAYBE(S); each case is considered below.

Case 1: $MP(k)$ returns NO

If a function of $Pi(k)$ can determine that the constraints in $PS_{M_a}^4$ are feasible, then the constraints in L_{M_a} must be feasible. However, by Property C1.2, if L_{M_a} contains feasible constraints, then $MP(k)$ would not return NO.

Case 2: $MP(k)$ returns MAYBE

Lemma F8 indicates that if $MP(k)$ returned MAYBE, then no procedure had a satisfaction function which could determine that the constraints in $PS_{M_a}^4$ were feasible. However, this contradicts condition (3) and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

$MP(k)$ could return MAYBE only by making transition 15 or 6. It could not make transition 15 because condition (2) indicates that L_{M_a} contains at least one constraint which is MP-solvable. $MP(k)$ could make transition 6 only if it did not generate any values using transition 24 and 3. However, if L_{M_a} contains one or more MP-solvable constraints, $MP(k)$ would make transition 24 after reaching state 7 and in so doing, it would generate a substitution. This raises a contradiction and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns $MAYBE(S)$

$MP(k)$ could return $MAYBE(S)$ only by making transition 10 or 27. It could make transition 27 only if it had exhausted its list of children and all returned $MAYBE$ (so that $V_{MP(k)}$ was set to 0). By Lemma F7, if $Pb(k)$ returned $MAYBE$, then $Pb(k)$ nor any of its descendants could verify the constraints in $PS_{M_a}^4$. However, $Pi(k)$ is a descendent of $Pb(k)$ and so a contradiction is raised. Thus, $MP(k)$ could not make transition 27.

$MP(k)$ could make transition 10 if some child $Pc(k)$ returned $MAYBE(S)$. If it were to do so, Lemma D2 indicates that $Pc(k)$ or one of its descendants sent a CM to $MP(k+1)$ after calling a function and $MP(k+1)$ returned $MAYBE(S)$. However, this contradicts condition (4) and so $MP(k)$ could not make transition 10 or return $MAYBE(S)$.

Case 5: $YES+MAYBE(S)$ is returned

$MP(k)$ could return $YES+MAYBE(S)$ only by making transition 11 or 12. By Lemma D4, if $MP(k)$ made either one after some child $Pi(k)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$, then (1) $Pi(k)$ or one of its descendants made transition 31, 32 or 42 after calling some function F and sending a CM to $MP(k+1)$, and (2) $MP(k+1)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$. As well, F must have returned $YES(S)$ or $MAYBE(S)$. By condition (4), F could not have generated a definite solution and by condition (5), if it generated a tentative solution, $MP(k+1)$ would have returned NO. This raises a contradiction and so $MP(k)$ could not return $YES+MAYBE(S)$.

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Thus, given the above conditions, $MP(k)$ will return $YES(S)$.

Q.E.D.

Property C6.3

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,

2. there is at least one procedure $P_i(k)$ with a function F which can generate a definite or tentative, partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ will return YES(S) when sent the substitutions, and
3. if any procedure sends a CM to $MP(k+1)$ before $P_i(k)$ calls F , it will receive a NO reply,

then $MP(k)$ will return YES(S).

Proof:

Let $P_j(k)$ denote the child of $MP(k)$ at the root of the subtree containing $P_i(k)$; $P_i(k)$ could be $P_j(k)$ or one of its descendents. Assume that given the above conditions, $MP(k)$ could return some other response, that is, NO, YES, MAYBE, MAYBE(S), or YES+MAYBE(S); each case is considered below.

Case 1: $MP(k)$ returns NO

By Property B5, if $MP(k+1)$ could return YES(S) after being sent a list of (assignment) constraints, then the constraints were feasible. It follows that the constraints in L_{M_a} must be feasible but by Property C1.2, $MP(k)$ could not return NO if it was sent a feasible list of constraints. This raises a contradiction and so $MP(k)$ could not return NO.

Case 2: $MP(k)$ returns MAYBE

By Lemma F8, if $MP(k)$ returned MAYBE, then no procedure could generate (1) a definite solution to a subset of the constraints in $PS_{M_a}^+$ or (2) a tentative partial or full solution which was not subsequently rejected by $MP(k+1)$. This contradicts condition (2) and so $MP(k)$ could not return MAYBE.

Case 3: $MP(k)$ returns YES

By Lemma F11, if $MP(k)$ returned YES, then no procedure could generate (1) a definite solution to a subset of the constraints in $PS_{M_a}^+$ or (2) a tentative partial or full solution which was not subsequently rejected by $MP(k+1)$. This contradicts condition (2) and so $MP(k)$ could not return YES.

Case 4: $MP(k)$ returns $MAYBE(S)$

$MP(k)$ could return $MAYBE(S)$ only by making transition 10 or 27; each case is considered below.

Case 4.1: $MP(k)$ makes transition 27

$MP(k)$ could make transition 27 only if it had exhausted its list of children and all had returned $MAYBE$ (so that $V_{MP(k)}$ was set to 0). By Lemma F7, if $Pj(k)$ returned $MAYBE$, then no procedure in the same subtree as $Pj(k)$ could (1) generate a definite solution to a subset of the constraints in $PS_{M_a}^+$ or (2) a tentative partial or full solution which was not subsequently rejected by $MP(k+1)$. This contradicts condition (2) and so $MP(k)$ could not make transition 27.

Case 4.2: $MP(k)$ makes transition 10

$MP(k)$ could make transition 10 only if some child $Pc(k)$ returned $MAYBE(S)$. By Lemma D2, if $Pc(k)$ returned $MAYBE(S)$, then some procedure in the same subtree as $Pc(k)$ sent a CM to $MP(k+1)$ after calling a function F' , and $MP(k+1)$ returned $MAYBE(S)$. If F' was called before $Pi(k)$ called F , condition (3) would be violated and so $Pi(k)$ must have called F already.

If $Pi(k)$ had called F , condition (2) indicates that $Pi(k)$ would have made transition 33 and returned $YES(S)$ to whichever procedure had sent it a CM. If its parent was a procedure other than $MP(k)$, then it and any other ancestor would have made transition 11, resulting in $YES(S)$ being forwarded back up through the hierarchy to $MP(k)$. Thus, $YES(S)$ would have to be returned to $MP(k)$ before $Pc(k)$ could return $MAYBE(S)$.

If $YES(S)$ was returned to $MP(k)$, then $MP(k)$ would have made transition 13 and returned the same, that is, it would not return $MAYBE(S)$. This raises a contradiction and so $MP(k)$ could not make transition 10.

If $MP(k)$ could not make transition 27 or 10, it follows that it could not return $MAYBE(S)$.

Case 5: $YES+MAYBE(S)$ is returned

$MP(k)$ could return $YES+MAYBE(S)$ by making transition 11 or 12; each case

is considered below.

Case 5.1: $MP(k)$ makes transition 11

$MP(k)$ could make transition 11 only if some child returned $MAYBE(S)$, but it was shown in Case 4.2 that no child could return such a response. Thus, $MP(k)$ could not make transition 11.

Case 5.2: $MP(k)$ makes transition 12

This transition requires some child $Pc(k)$ to return $YES+MAYBE(S)$. If $Pc(k)$ returned $YES+MAYBE(S)$, then some procedure in the same subtree as $Pc(k)$ sent a CM to $MP(k+1)$ after calling a function F' , and $MP(k+1)$ returned $MAYBE(S)$ or $YES+MAYBE(S)$. If F' was called before $Pi(k)$ called F , condition (3) would be violated and so $Pi(k)$ must have called F already. The arguments that were used in Case 4.2 may now be repeated to show that if F had been called already, $YES(S)$ would have been forwarded back to $MP(k)$ before $Pc(k)$ could return $YES+MAYBE(S)$, and $MP(k)$ would make transition 13. This raises a contradiction and so $MP(k)$ could not make transition 12.

If $MP(k)$ could not make transition 11 or 12, it follows that it could not return $YES+MAYBE(S)$.

If $MP(k)$ could not return any of the above responses, a contradiction arises and so the original assumption must be wrong. Thus, given the above conditions, $MP(k)$ will return $YES(S)$.

Q.E.D.

Property C7.1

If (1) a CM M_a is sent to $MP(k)$ and (2) there exists a constraint $c \in L_{M_a}$ such that c is MP-rejectable, then $MP(k)$ will return NO.

Proof:

The result follows directly from the state transition diagram for $MP(k)$. When $MP(k)$ is sent a CM, it makes transition 1, 17, 18, or 19 to state 1. If $MP(k)$ detects an MP-rejectable constraint $c \in PS_{M_a}^+$, then it will make transition 14

and in so doing return NO.

Q.E.D.

Property C7.2

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. L_{M_a} contains one or more MP-solvable constraints, and
3. the solution to some MP-solvable constraint $c \in L_{M_a}$ results in some other constraint $d \in PS_{M_a}^*$ becoming MP-rejectable,

then $MP(k)$ will return NO.

Proof:

Assume that given the above conditions, $MP(k)$ could return a response other than NO. If $MP(k)$ was sent a CM, it would make transition 1, 17, 18, or 19 to state 1. If L_{M_a} contained one or more MP-solvable constraints, $MP(k)$ would make transition 24 and solve one of them. Note that it could not make transition 15 because all constraints in $PS_{M_a}^+$ could not be satisfied, and it could not make transition 2 if any MP-solvable constraints were present. $MP(k)$ could make transition 14 if L_{M_a} contained an MP-rejectable constraint but if it did, it would still return NO.

After making transition 24, $MP(k)$ would apply the solution to c to every other constraint and one of the following cases would arise:

1. an MP-rejectable constraint would appear in which case $MP(k)$ would make transition 26 and return NO,
2. no MP-rejectable constraints would appear and one of the following cases would arise:
 - (a) All constraints would be totally reduced and satisfied (in which case no MP-rejectable constraints could have arisen), in which case $MP(k)$ would return YES(S) using transition 25.

- (b) Another MP-solvable constraint would be detected, in which case $MP(k)$ would solve it and apply the solution to every other constraint. Outcomes (1) and (2) would apply once more.
- (c) The problem set would not contain an MP-solvable or MP-rejectable constraint, in which case $MP(k)$ would make transition 4. Note that after doing so, it could not further reduce the constraints in $PS_{M_a}^4$.

Thus, if an MP-rejectable constraint ever arises in $PS_{M_a}^*$, $MP(k)$ would return NO. This contradicts the original assumption and so it must be wrong. Thus, given the above conditions, $MP(k)$ will return NO.

Q.E.D.

Property C7.3

Given the following conditions:

1. a CM M_a is sent to $MP(k)$,
2. at least one procedure $Pi(k)$ has a function F which can determine that $PS_{M_a}^4$ contains one or more nonfeasible constraints, and
3. if any procedure has a satisfaction function which could generate a tentative partial or full solution to a subset of $PS_{M_a}^4$ such that $MP(k+1)$ would not reject the solution, then the procedure does not call the function before $Pi(k)$ calls F ,

then $MP(k)$ will return NO.

Proof:

Assume that given the above conditions, $MP(k)$ could return some other response, that is, MAYBE, YES, MAYBE(S), YES+MAYBE(S), or YES(S); each case is considered below.

Case 1: $MP(k)$ returns MAYBE

If $MP(k)$ returned MAYBE, then it could not have determined any substitutions using transitions 24 and 3, and so $PS_{M_a}^+$ and $PS_{M_a}^4$ must be identical. By Lemma F8,

if $MP(k)$ returned MAYBE, then $MP(k)$ nor any of its descendents could determine that $PS_{M_a}^+$ contained nonfeasible constraints. Since $PS_{M_a}^+$ and $PS_{M_a}^4$ are identical, condition (2) would be violated and so $MP(k)$ could not return MAYBE.

Case 2: *$MP(k)$ returns MAYBE(S) using transition 27 or 10*

$MP(k)$ could return MAYBE(S) by making transition 27 or 10; each case is considered below.

Case 2.1: *$MP(k)$ makes transition 27*

Let $Pm(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$. $MP(k)$ could make transition 27 only if it had exhausted its list of children and all returned MAYBE (so that $V_{MP(k)}$ was set to 0). However, by Lemma F7, if $Pm(k)$ returned MAYBE, then $Pm(k)$ nor any of its descendents could find any constraints in $PS_{M_a}^4$ to be nonfeasible. This violates condition (2) and so $MP(k)$ could not return MAYBE(S) using transition 27.

Case 2.2: *$MP(k)$ makes transition 10*

By Lemma D2, if $MP(k)$ made transition 10 after some child $Pj(k)$ returned MAYBE(S), then some procedure $Pm(k)$ in the same subtree as $Pj(k)$ made transition 32 after:

- it called a function F_m ,
- F_m returned MAYBE(S'),
- a CM was sent to $MP(k+1)$, and
- $MP(k+1)$ returned MAYBE(S).

However, F_m could not return MAYBE(S) because it was assumed in condition (3) that no procedure had a function which could generate a tentative solution which would not be rejected by $MP(k+1)$ before $Pi(k)$ called F . This raises a contradiction and so $MP(k)$ could not return MAYBE(S) using transition 10.

If $MP(k)$ could not make transition 27 or 10, it follows that $MP(k)$ could not return MAYBE(S).

Case 3: $MP(k)$ returns YES, YES+MAYBE(S), or YES(S)

If $Pi(k)$ had a function which could determine that $PS_{M_a}^4$ contained nonfeasible constraints, then the constraints in L_{M_a} could not be feasible (because the constraints in $PS_{M_a}^+$ were reduced using the only possible solutions). However, by Property C1.3, if $MP(k)$ is sent a CM containing one or more nonfeasible constraints, then it not return YES, YES+MAYBE(S), or YES(S).

If $MP(k)$ could not return any of the listed responses, a contradiction arises and so the original assumption must be wrong. Therefore, given the above conditions, $MP(k)$ will return NO.

Q.E.D.

Appendix E

Proof of Backtrack Properties

Property 1

If $MP(k)$ returns NO in response to a CM M_a without calling any of its children, then the status of every procedure $Pi^+(k)$ will be the same after $MP(k)$ replies as before $MP(k)$ was sent the CM.

Proof:

Assume that the status of some procedure $Pi^+(k)$ is not restored to what it was before $MP(k)$ received M_a . $Pi^+(k)$ could be $MP(k)$ or some other procedure; each case is considered below.

Case 1: $Pi^+(k)$ is $MP(k)$

For the status of $MP(k)$ not to be restored, it must fail to discard some changes which were made to its data stack, VV, or SL. $MP(k)$ could return NO only by making transition 14, 26, or 21. If it made either of the first two, it would pop its stack and discard any changes made to VV and SL, contrary to the assumption. $MP(k)$ could not make transition 21 without calling at least one of its children, therefore $Pi^+(k)$ could not be $MP(k)$.

Case 2: $Pi^+(k)$ is some procedure $Pi(k)$

It was shown in Case 1 that $MP(k)$ could return NO only by making transition 14 or 26. In either case, $MP(k)$ could not call any of its children and so the status of every descendent remains unchanged. Thus, $Pi^+(k)$ could not be $Pi(k)$.

If $Pi^+(k)$ could not be $MP(k)$ or any other procedure $Pi(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ returns NO in response to a CM without calling any children, then the status of every procedure $Pi^+(k)$ is the same after $MP(k)$ replies as before $MP(k)$ was sent the CM. **Q.E.D.**

Corolary 1

If $MP(k)$ returns NO in response to a CM M_a without calling any of its children, then the status of every procedure $Pi^+(k+n)$, $n \geq 0$, will be the same after $MP(k)$ replies as before $MP(k)$ was sent M_a .

Proof:

Consider the cases of $n=0$ and $n \geq 1$ separately.

Case 1: $n=0$

The proof of Property 1 suffices.

Case 2: $n \geq 1$

Status changes at deeper levels of recursion could occur only if some procedure called $MP(k+1)$. However, if $MP(k)$ returns NO without calling any of its children, no procedure could call $MP(k+1)$; it follows that the status of every procedure $Pi^+(k+n)$, $n \geq 1$, will be the same after $MP(k)$ replies as before $MP(k)$ was sent CM M_a .

Q.E.D.

Property 2

If $MP(k)$ returns YES in response to a CM M_a without calling any of its children, then the status of every procedure $Pi(k)$ will be the same after $MP(k)$ replies as before $MP(k)$ was sent the CM.

Proof:

Assume that $MP(k)$ returns YES in response to a CM M_a without calling any of its children, but the status of at least one procedure $Pi(k)$ is not the same after

$MP(k)$ replies. $MP(k)$ could return YES only by making transition 15 or 6; each case is considered below.

Case 1: $MP(k)$ makes transition 15

After receiving M_a , $MP(k)$ would leave some current state PS and move to state 7. If $MP(k)$ then detected a totally reduced and satisfied problem set, it would make transition 15 and returns YES without calling any of its children. Since the status of every procedure $Pi(k)$ could not have changed, the assumption is violated and so $MP(k)$ could not make transition 15.

Case 2: $MP(k)$ makes transition 6

$MP(k)$ could make transition 6 only if it had sent a CM to each of its children. Since this violates the assumption that it does not call any child. $MP(k)$ could not make transition 6.

If $MP(k)$ could not make transition 15 or 6, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ returns YES in response to a CM without calling any of its children, then the status of every procedure $Pi(k)$ will be the same after $MP(k)$ replies as before $MP(k)$ was sent the CM.

Q.E.D.

Property 3

If $MP(k)$ is sent an AVR after it returns YES in response to a CM without calling any of its children, then it will return NO and the status of every procedure $Pi^+(k)$ will be restored to what it was before $MP(k)$ received the CM.

Proof:

Assume that the status of some procedure $Pi^+(k)$ will not be restored to what it was before $MP(k)$ received some CM M_a . $Pi^+(k)$ could be $MP(k)$ or some other procedure; each case is considered below.

Case 1: $Pi^+(k)$ is $MP(k)$

Property 2 proved that if $MP(k)$ returned YES without calling any of its children, it must have made transition 15 to state 1. A subsequent AVR forces it to make

transition 16 and return NO, pop its data stack, and discard any changes to VV and SL. Since these actions restore its status to what it was before it received M_a , $Pi^+(k)$ cannot be $MP(k)$.

Case 2: $Pi^+(k)$ is some procedure $Pi(k)$

By Property 2, the status of every other procedure $Pi(k)$ will be the same after $MP(k)$ returns YES as it was before $MP(k)$ was sent M_a . Since transition 16 does not cause $MP(k)$ to send any messages to its children, the status of its descendants could not have changed. Therefore, $Pi^+(k)$ cannot be some procedure $Pi(k)$.

Since $Pi^+(k)$ cannot be $MP(k)$ or any other procedure $Pi(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ returns YES in response to a CM without calling any of its children and it then sent an AVR, $MP(k)$ will return NO and the status of every procedure $Pi^+(k)$ will be restored to what it was before $MP(k)$ received the CM.

Q.E.D.

Property 4

Given the following conditions:

1. a procedure $Pi(k)$ returns MAYBE in response to a CM,
2. no recursive calls are made by $Pi(k)$ or any of its descendants, and
3. $Pi(k)$ is then sent a BACKTRACK message,

then the status of $Pi(k)$ and all of its descendants will return to what it was before $Pi(k)$ received the CM.

Proof:

Assume that a procedure $Pi(k)$ returns MAYBE in response to a CM M_a and it is then sent a BACKTRACK message, but the status of some procedure $Pz(k)$ in the same subtree as $Pi(k)$ is not restored to what it was before $Pi(k)$ received M_a . $Pz(k)$ could be $Pi(k)$ or one of its descendants. $Pi(k)$ could return MAYBE only by making transition 2, 18, or 20; each case is considered below.

Case 1: $P_i(k)$ makes transition 2

If $P_i(k)$ makes transition 2 (after it failed to find any new data in M_a) and it then receives a BACKTRACK message, it must make transition 38 and pop its data stack. In so doing, $P_i(k)$ discards any changes resulting from the CM and so its status is restored. As well, since $P_i(k)$ could not have called any of its children between the time that it received M_a and the time that it made transition 38, the status of all of its descendents could not have changed. Therefore, $P_z(k)$ cannot be $P_i(k)$ or one of its descendents if $P_i(k)$ makes transition 2.

Case 2: $P_i(k)$ makes transition 18 or 20

If $P_i(k)$ made transition 18 or 20, it would move to state 8. Upon receipt of a BACKTRACK message, it would make transition 21 and in so doing pop its data stack and return to its previous state. Therefore, the status of $P_i(k)$ would be restored to what it was before $P_i(k)$ received M_a . This indicates that $P_z(k)$ cannot be $P_i(k)$ and so it must be one of its descendents.

Let $P_j(k)$ denote the child of $P_i(k)$ at the root of the subtree containing $P_z(k)$; $P_z(k)$ could be $P_j(k)$ or one of its descendents. Since $P_i(k)$ returned MAYBE, Lemma F4 indicates that $P_j(k)$ returned MAYBE if it was sent a CM. If the status of $P_z(k)$ was changed, a CM must have been sent to $P_z(k)$ and so one must have been sent to $P_j(k)$.

The above arguments now repeat with $P_j(k)$ replacing $P_i(k)$. $P_j(k)$ must make transition 2, 18, or 20; $P_z(k)$ cannot be $P_j(k)$ or one of its descendents if $P_j(k)$ makes transition 2; $P_z(k)$ cannot be $P_j(k)$ if $P_j(k)$ makes transition 18 or 20 and so $P_z(k)$ must be a descendent of $P_j(k)$ (after $P_j(k)$ makes transition 18 or 20). If $P_q(k)$ is the child of $P_j(k)$ at the root of the subtree containing $P_z(k)$, $P_q(k)$ must have been sent a CM and returned MAYBE, and so on.

The above arguments repeat indefinitely and so by induction $P_z(k)$ must always be at another level deeper in the tree. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_z(k)$. Therefore, $P_z(k)$ cannot exist if $P_i(k)$ makes transition 18 or 20.

If $P_i(k)$ cannot make any of these transitions, a contradiction is raised and so

the original assumption must be wrong. Therefore, if $P_i(k)$ is sent a BACKTRACK message after returning MAYBE in response to a CM, the status of $P_i(k)$ and all of its descendants will be restored to what it was before $P_i(k)$ was sent the CM.

Q.E.D.

Property 5

Given the following conditions:

1. a procedure $P_i(k)$ returns YES in response to a CM,
2. no recursive calls were made by $P_i(k)$ or any of its descendants, and
3. $P_i(k)$ is then sent a BACKTRACK message,

then the status of $P_i(k)$ and all of its descendants will return to what it was before $P_i(k)$ received the CM.

Proof:

Assume that a procedure $P_i(k)$ returns YES in response to a CM M_a and it is then sent a BACKTRACK message, but the status of some procedure $P_z(k)$ in the same subtree as $P_i(k)$ is not restored to what it was before $P_i(k)$ received M_a . $P_z(k)$ could be $P_i(k)$ or one of its descendants.

$P_i(k)$ can return YES only by making transition 39 or 19 to state 8. In either case, a subsequent BACKTRACK message would cause it to make transition 21, pop its data stack, and return to the state it was in before it received the CM. Since this restores the status of $P_i(k)$, $P_z(k)$ must be a descendant of $P_i(k)$.

Let $P_j(k)$ denote the child of $P_i(k)$ at the root of the subtree containing $P_z(k)$; $P_z(k)$ could be $P_j(k)$ or one of its descendants. Since $P_i(k)$ previously returned YES, Lemma F9 indicates that $P_j(k)$ returned MAYBE or YES, or it was not sent a CM. Each case is considered below.

Case 1: $P_j(k)$ was not sent a CM

If $P_i(k)$ made transition 19 or 39, it must have previously called all of its children, including $P_j(k)$. Therefore, $P_j(k)$ must have returned MAYBE or YES in response to a CM.

Case 2: $P_j(k)$ returned MAYBE

If $P_j(k)$ returned MAYBE, by Property 4 and the fact that $P_i(k)$ will send $P_j(k)$ a BACKTRACK message as part of transition 21, the status of $P_j(k)$ and all of its descendants will be restored to what it was before $P_i(k)$ was sent M_a . Therefore, $P_z(k)$ is not in the same subtree as $P_j(k)$ if $P_j(k)$ returned MAYBE.

Case 3: $P_j(k)$ returned YES

The above arguments now repeat with $P_j(k)$ replacing $P_i(k)$. In order to return YES, $P_j(k)$ must have made transition 39 or 19; $P_j(k)$ will receive a BACKTRACK message when $P_i(k)$ makes transition 21 and so its status will be restored. This indicates that $P_z(k)$ cannot be $P_j(k)$ and so it must be one of its descendants, and so on.

These arguments repeat indefinitely and so by induction $P_z(k)$ must always be at another level deeper in the tree. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_z(k)$. Therefore, $P_z(k)$ cannot be in the same subtree as $P_j(k)$ if $P_j(k)$ returns YES.

If $P_j(k)$ cannot return one of the above responses, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns YES in response to a CM and it is then sent a BACKTRACK message, the status of $P_i(k)$ and all of its descendants will be restored to what it was before $P_i(k)$ was sent the CM.

Q.E.D.

Property 6

If a procedure $P_i(k)$ returns NO in response to a CM, then the status of $P_i(k)$ and all of its descendants will return to what it was before $P_i(k)$ received the CM.

Proof:

Assume that after $P_i(k)$ returns NO in response to a CM M_a , there exists a procedure $P_z(k)$ in the same subtree as $P_i(k)$ that does not return to its original state. $P_z(k)$ could be $P_i(k)$ or one of its descendants. $P_i(k)$ can return NO only by making transition 29 or 12; each case is considered below.

Case 1: $P_i(k)$ makes transition 29

If $P_i(k)$ were to make transition 29, it would pop its data stack and return to the state it was in before it received M_a , therefore the status of $P_i(k)$ is restored. This indicates that $P_z(k)$ must be a descendent of $P_i(k)$.

Let $P_p(k)$ be the child of $P_i(k)$ at the root of the subtree containing $P_z(k)$; $P_z(k)$ could be $P_p(k)$ or one of its descendents. Since the status of a procedure cannot change unless it has been sent a CM, Lemma G3 indicates that $P_p(k)$ must have returned MAYBE. (It could not have returned NO otherwise $P_i(k)$ would have made transition 12 and not 29 as assumed.) Since $P_i(k)$ sends BACKTRACK messages to its children as part of transition 29, Property 4 indicates that the status of $P_p(k)$ and all of its descendents will be restored to what it was before $P_p(k)$ was sent the CM, that is, the status before $P_i(k)$ was called. This indicates that $P_z(k)$ cannot be a descendent of $P_i(k)$ makes transition 29. It has already been shown that $P_z(k)$ cannot be $P_i(k)$ and so $P_z(k)$ cannot be in the same subtree as $P_i(k)$ if $P_i(k)$ makes transition 29.

Case 2: $P_i(k)$ makes transition 12

If $P_i(k)$ returns NO by making transition 12 after some child $P_j(k)$ returns NO, $P_z(k)$ could be $P_i(k)$, in a subtree to the left or right of $P_j(k)$, or in the same subtree as $P_j(k)$. Each case is considered below.

Case 2.1: $P_z(k)$ is $P_i(k)$

$P_i(k)$ pops its data stack and returns to its previous state after making transition 12, therefore its status is restored and so $P_z(k)$ cannot be $P_i(k)$.

Case 2.2: $P_z(k)$ is in a subtree to the left of $P_j(k)$

By Lemma G1, if $P_j(k)$ returns NO, all of the other children of $P_i(k)$ to the left of $P_j(k)$ must have returned MAYBE. Let $P_s(k)$ be the child of $P_i(k)$ at the root of the subtree containing $P_z(k)$. When $P_i(k)$ makes transition 12, it will send $P_s(k)$ a BACKTRACK message and so by Property 4, the status of $P_s(k)$ and all of its descendents will return to what it was before $P_i(k)$ was sent message M_a . Thus, $P_z(k)$ cannot be in a subtree to the left of $P_j(k)$.

Case 2.3: $P_z(k)$ is in a subtree to the right of $P_j(k)$

By Lemma G2, $P_i(k)$ would not send any messages to its children to the right of $P_j(k)$ and so the status of all descendants to the right of $P_j(k)$ would not have changed. Thus, $P_z(k)$ cannot be in a subtree to the right of $P_j(k)$.

Case 2.4: $P_z(k)$ is in the same subtree as $P_j(k)$

The above arguments now repeat with $P_j(k)$ replacing $P_i(k)$. $P_j(k)$ can return NO only by making transition 12 or 29; $P_z(k)$ cannot be $P_j(k)$ or one of its descendants if $P_j(k)$ makes transition 29; if $P_j(k)$ makes transition 12 after some child $P_q(k)$ returns NO, $P_z(k)$ cannot be $P_j(k)$ or in a subtree to the left or right of $P_q(k)$, therefore it must be in the same subtree as $P_q(k)$.

The above arguments repeat indefinitely and so by induction $P_z(k)$ must always be at another level deeper in the tree. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_z(k)$. Therefore, $P_z(k)$ cannot be in the same subtree as $P_j(k)$ and $P_i(k)$ cannot return NO using transition 12.

If $P_i(k)$ cannot make either of the above transitions, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns NO in response to a CM, the status of $P_i(k)$ and all of its descendants will be restored to what it was before $P_i(k)$ was sent the CM.

Q.E.D.

Property 7

If $MP(k)$ returns NO in response to a CM, the status of every procedure $P_i^+(k)$ will be the same after $MP(k)$ replies as before $MP(k)$ received the CM.

Proof:

Assume that $MP(k)$ receives a CM M_a and returns NO, but there is a procedure $P_i^+(k)$ whose status is not the same after $MP(k)$ returns NO as it was before $MP(k)$ received M_a . $MP(k)$ can return NO by making transitions 14, 26, or 21. If it makes either of the first two, it will not have called any of its children and by Corolary 1, the status of every procedure will be the same after $MP(k)$ replies as before $MP(k)$

was sent the CM. Therefore, $MP(k)$ must make transition 21 after some child $Pj(k)$ returns NO. $Pz^+(k)$ could be $MP(k)$ or some other procedure $Pz(k)$; each case is considered below.

Case 1: $Pz^+(k)$ is $MP(k)$

If $MP(k)$ makes transition 21, it pops its data stack, discards any changes to VV and SL, and returns to the state it was in before it received message M_a . Since its status is restored, $Pz^+(k)$ cannot be $MP(k)$.

Case 2: $Pz^+(k)$ is some procedure $Pz(k)$

$Pz(k)$ must be in a subtree to the left or right of $Pj(k)$, or in the same subtree as $Pj(k)$. Each case is considered below:

Case 2.1: $Pz(k)$ is in a subtree to the left of $Pj(k)$

By Lemma G1, if $Pj(k)$ returns NO, all of the other children of $MP(k)$ to the left of $Pj(k)$ must have returned MAYBE. Let $Pq(k)$ be the child of $MP(k)$ at the root of the subtree containing $Pz(k)$. When $MP(k)$ makes transition 21, it will send $Pq(k)$ a BACKTRACK message and so by Property 4, the status of $Pq(k)$ and all of its descendents will be restored to what it was before $Pq(k)$ and $MP(k)$ were sent the CM's. Therefore, $Pz(k)$ cannot be in a subtree to the left of $Pj(k)$.

Case 2.2: $Pz(k)$ is in a subtree to the right of $Pj(k)$

By Lemma G2, $MP(k)$ would not send any messages to its children to the right of $Pj(k)$ and so the status of all procedures in subtrees to the right of $Pj(k)$ would not have changed. Thus, $Pz(k)$ cannot be in a subtree to the right of $Pj(k)$.

Case 2.3: $Pz(k)$ is in the same subtree as $Pj(k)$

By Property 6, if a procedure $Pj(k)$ returns NO in response to a CM, then the status of $Pj(k)$ and all of its descendents will be restored to what it was before $Pj(k)$ received the CM. Therefore, $Pz(k)$ cannot be in the same subtree as $Pj(k)$.

If $Pz^+(k)$ cannot be $MP(k)$ or some other procedure $Pz(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ returns NO in response to a CM, the status of every procedure $Pi^+(k)$ will be the same after $MP(k)$

replies as before $MP(k)$ was sent the CM.

Q.E.D.

Property 8

If a procedure $Pi(k)$ returns YES or MAYBE in response to a CM M_a , then:

1. $Pi(k)$ did not make any recursive calls, or
2. if it did call $MP(k+1)$, any status changes that resulted at deeper levels of recursion were discarded.

Proof:

$Pi(k)$ can return YES or MAYBE by making transition 2, 18, 39, 19, or 20; each case is considered below.

Case 1: $Pi(k)$ makes transition 2

If $Pi(k)$ makes transition 2, it cannot make a recursive call because there is no way for it to return to state 1 after calling $MP(k+1)$.

Case 2: $Pi(k)$ makes transition 18 or 39

If $Pi(k)$ makes transition 18 or 39, it cannot make a recursive call because there is no way for it to return to state 7 after calling $MP(k+1)$.

Case 3: $Pi(k)$ makes transition 19 or 20

$Pi(k)$ could make transition 19 or 20 without making a recursive call if all of its satisfaction functions returned MAYBE or YES. It remains to be shown that if $Pi(k)$ does call $MP(k+1)$ and then makes transition 19 or 20, any resulting status changes at deeper levels of recursion will be discarded.

Assume that $Pi(k)$ calls $MP(k+1)$ and there exists a procedure $Pj^+(k+1)$ whose status is not restored to what it was before $Pi(k)$ was sent M_a . If $Pi(k)$ returns YES or MAYBE after calling $MP(k+1)$, it must have returned to state 9 using transition 28 in order to make transition 19 or 20. This indicates that $MP(k+1)$ must have returned NO but by Property 7, the status of all procedures $Px^+(k+1)$ will be

restored to what it was before $MP(k+1)$ was sent the CM; i.e., what it was before $Pi(k)$ received M_a . This raises a contradiction and so $Pj^+(k+1)$ cannot exist.

The only other way for there to be a procedure whose status is not restored is for it to be at a deeper level of recursion. If there is a procedure $Pj^+(k+n)$, $n \geq 1$, that does not discard any status changes after $Pi(k)$ returns MAYBE or YES, some procedure $Pm(k+n-1)$ made a recursive call to $MP(k+n)$ and by Property 7, $MP(k+n)$ could not have returned NO. Similarly, $Pm(k+n-1)$ could make the call only if some procedure $Pl(k+n-2)$ made a recursive call to $MP(k+n-1)$ and so on, up to and including some $Pq(k+n-(n-1))$, that is, $Pq(k+1)$. Finally, $Pq(k+1)$ can only exist if some procedure called $MP(k+1)$; this condition is satisfied by $Pi(k)$.

It was shown above that $MP(k+1)$ had to return NO so that $Pi(k)$ could make transition 19 or 20, and by Lemma G4, $Pq(k+1)$ must have returned MAYBE or NO. If $Pq(k+1)$ returned either of these after calling $MP(k+2)$, it must have made transition 20 or 29 after it called a satisfaction function, sent a CM to $MP(k+2)$, and $MP(k+2)$ returned NO. By Property 7, if $MP(k+2)$ returns NO, then the status of all procedures $Px^+(k+2)$ will be restored to what it was before $MP(k+2)$ was called, and so the arguments repeat.

In general, by Lemma G4, each procedure $Pz(k+m-1)$ that makes a recursive call returns MAYBE or NO using transition 20 or 29 because $MP(k+m-1)$ returns NO, and $MP(k+m)$ must return NO in order for $Pz(k+m-1)$ to make either of these transitions. However, it was shown that for $Pa^+(k+n)$ to not discard any status changes, $MP(k+n)$ cannot return NO. This raises a contradiction and so $Pj^+(k+n)$ cannot exist if $Pi(k)$ makes transition 19 or 20.

If $Pi(k)$ cannot make any of the above transitions, a contradiction arises and so the original assumption must be wrong. Therefore, if a procedure $Pi(k)$ returns YES or MAYBE in response to a CM, then (1) $Pi(k)$ did not make any recursive calls or (2) if it did, any status changes that resulted at deeper levels of recursion were discarded.

Q.E.D.

Property 9

If a procedure $P_i(k)$ returns NO in response to a CM, then $P_i(k)$ did not make any recursive calls or if such calls were made, any status changes that resulted at deeper levels of recursion were discarded.

Proof:

$P_i(k)$ can return NO by making transition 12 or 29; each case is considered below.

Case 1: $P_i(k)$ returns NO using transition 12

If $P_i(k)$ makes transition 12, it could not have made a recursive call because there is no way for it to return to state 4 after calling $MP(k+1)$.

Case 2: $P_i(k)$ returns NO using transition 29

If $P_i(k)$ makes transition 29, it did so after calling one of its satisfaction functions. It could return such a response without making a recursive call and so it remains to be shown that if such a call was made, any status changes that resulted at deeper levels of recursion were discarded.

Assume that there exists a procedure $Pa^+(k+1)$ whose status is not restored to what it was before $P_i(k)$ was sent the CM. Before $P_i(k)$ can make transition 29 after calling $MP(k+1)$, it must first return to state 9 using transition 28. This indicates that $MP(k+1)$ returns NO but by Property 7, the status of all procedures $Px^+(k+1)$ will be restored to what they were before $MP(k+1)$ was sent the CM. This includes $Pa^+(k+1)$ and so $Pa^+(k+1)$ cannot exist if $P_i(k)$ returns NO. The only other way for there to be a procedure that does not discard any status changes is for it to be at a deeper level of recursion.

If there is a procedure $Pa^+(k+n)$, $n \geq 1$, that does not discard any status changes after $P_i(k)$ returns NO, some procedure $Pm(k+n-1)$ made a recursive call to $MP(k+n)$ and by Property 7, $MP(k+n)$ could not have returned NO. Similarly, $Pm(k+n-1)$ could only make the call if some procedure $Pl(k+n-2)$ made a recursive call to $MP(k+n-1)$ and so on, up to and including some $Pj(k+n-(n-1))$, that is, $Pj(k+1)$. Finally, $Pj(k+1)$ can only exist if some procedure called $MP(k+1)$; this condition is satisfied by $P_i(k)$.

It was shown above that $MP(k+1)$ had to return NO so that $Pi(k)$ could make transition 29, and by Lemma G4, $Pj(k+1)$ must have returned MAYBE or NO. If $Pj(k+1)$ returned such a response after calling $MP(k+2)$, it must have made transition 20 or 29 after it called a satisfaction function and $MP(k+2)$ returned NO. By Property 7, if $MP(k+2)$ returns NO, the status of all procedures $Px^+(k+2)$ will be restored to what it was before $MP(k+2)$ was sent the CM, and so the arguments repeat.

In general, by Lemma G4, each procedure $Pz(k+m-1)$ that makes a recursive call returns MAYBE or NO using transition 20 or 29 because $MP(k+m-1)$ returns NO, and $MP(k+m)$ must return NO in order for $Pz(k+m-1)$ to make either of these transitions. However, it was shown that for $Pa^+(k+n)$ to exist, $MP(k+n)$ cannot return NO. This raises a contradiction and so $Pa^+(k+n)$ cannot exist if $Pi(k)$ makes transition 29.

If $Pi(k)$ cannot make either of these transitions, a contradiction arises and so the original assumption must be wrong. Therefore, if a procedure $Pi(k)$ returns NO in response to a CM, then (1) $Pi(k)$ did not make any recursive calls or (2) if it did, any status changes that resulted at deeper levels of recursion were discarded.

Q.E.D.

Property 10

If a procedure $Pi(k)$ returns YES or MAYBE in response to a CM, then $Pi(k)$ and its descendents did not make any recursive calls, or if such calls were made by $Pi(k)$ or a descendent, any status changes that resulted at deeper levels of recursion were discarded.

Proof:

Assume that a recursive call is made by some procedure $Pj(k)$ in the same subtree as $Pi(k)$ and that there exists a procedure $Pa^+(k+n)$, $n > 0$, that does not discard all changes resulting from the recursive call. $Pj(k)$ could be $Pi(k)$ or one of its descendents; each case is considered below.

Case 1: $P_j(k)$ is $P_i(k)$

By Property 8, if $P_i(k)$ returns YES or MAYBE in response to a CM, $P_i(k)$ did not make any recursive calls or if such calls were made, any status changes that resulted at deeper levels of recursion were discarded. Therefore, if any status changes were not discarded, they resulted from a recursive call made by a descendent of $P_i(k)$.

Case 2: $P_j(k)$ is a descendent of $P_i(k)$

Assume that a recursive call is made by $P_j(k)$. By Lemmas F4 and F9, if $P_i(k)$ returns MAYBE or YES, each of its descendents was not sent a CM or it was sent a CM and it returned MAYBE or YES. Since $P_j(k)$ must have been called in order for it to make a recursive call, it must have returned MAYBE or YES. By Property 8, if $P_j(k)$ returns MAYBE or YES after making a recursive call, then any status changes that resulted at deeper levels of recursion were discarded and so $Pa^+(k+n)$ cannot exist if $P_j(k)$ is a descendent of $P_i(k)$.

If $P_j(k)$ cannot be $P_i(k)$ or one of its descendents, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns YES or MAYBE in response to a CM, $P_i(k)$ and its descendents did not make any recursive calls, or if such calls were made, any status changes that resulted at deeper levels of recursion were discarded.

Q.E.D.

Property 11

If a procedure $P_i(k)$ returns NO in response to a CM, $P_i(k)$ and its descendents did not make any recursive calls, or if such calls were made by $P_i(k)$ or a descendent, any status changes that resulted at deeper levels of recursion were discarded.

Proof:

Assume that a recursive call is made by some procedure $P_j(k)$ in the same subtree as $P_i(k)$ and that there exists a procedure $Pa^+(k+n)$, $n > 0$, that does not discard changes resulting from the recursive call. Each case is considered below.

Case 1: $P_j(k)$ is $P_i(k)$

By Property 9, if $P_i(k)$ returns NO, $P_i(k)$ did not make any recursive calls or if such calls were made, any status changes associated with deeper levels of recursion were discarded. Thus, if $Pa^+(k+n)$ exists, it must result from some descendent of $P_i(k)$ making a recursive call.

Case 2: $P_j(k)$ is a descendent of $P_i(k)$

By Lemma G3, if $P_i(k)$ returns NO, each of its descendents returned MAYBE or NO if they were called at all. Since $P_j(k)$ must have been sent a CM in order for it to call $MP(k+1)$, it must have returned MAYBE or NO. By Properties 8 and 9, if $P_j(k)$ returns MAYBE or NO after making a recursive call, any status changes that resulted at deeper at levels of recursion were discarded. Therefore, $Pa^+(k+n)$ cannot exist if $P_j(k)$ is a descendent of $P_i(k)$.

If $P_j(k)$ cannot be $P_i(k)$ or one of its descendents, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns NO, $P_i(k)$ and its descendents did not make any recursive calls, or if such calls were made, any status changes that resulted at deeper levels of recursion were discarded.

Q.E.D.

Property 12

Given the following conditions:

1. a procedure $P_i(k)$ returns MAYBE or YES in response to a CM, and
2. $P_i(k)$ is then sent a BACKTRACK message,

then the status of $P_i(k)$ and all of its descendents will return to what it was before $P_i(k)$ received the CM and any status changes that resulted at deeper levels of recursion will be discarded.

Proof:

First assume that there exists a procedure $Pa^+(k+n)$, $n \geq 1$, whose status after $P_i(k)$ processes the BACKTRACK message is not the same as it was before

$P_i(k)$ received some CM M_b . By Property 10, if a procedure $P_i(k)$ returns YES or MAYBE in response to a CM, then no recursive calls were made, or if they were, any status changes that resulted at deeper levels of recursion were discarded before $P_i(k)$ replied. Therefore, for $Pa^+(k+n)$ to exist, status changes must be caused by some procedure $P_z(k)$ in the same subtree as $P_i(k)$ making a recursive call.

$P_z(k)$ could be $P_i(k)$ or one of its descendants. $P_i(k)$ can only receive BACKTRACK messages while it is waiting in state 2 or 8, and if it does, it makes transition 16 or 21, respectively. In either case, $P_i(k)$ does not make any recursive calls and so it cannot cause any status changes at deeper levels of recursion. Therefore, $P_z(k)$ cannot be $P_i(k)$ and so it must be one of its descendants.

Let $P_j(k)$ be the child of $P_i(k)$ at the root of the subtree containing $P_z(k)$. The above arguments now repeat with $P_j(k)$ replacing $P_i(k)$; i.e., $P_j(k)$ can receive BACKTRACK messages only while waiting in state 2 or 8 and if it does, it must make transition 16 or 21. In either case, it does not make a recursive call and so $P_z(k)$ cannot be $P_j(k)$, therefore it must be one of its descendants.

These arguments repeat indefinitely and each time they are used, $P_z(k)$ must be at another level deeper in the tree. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $P_z(k)$. This raises a contradiction and so the BACKTRACK message cannot cause any status changes at deeper levels of recursion.

Now assume that the status of $P_i(k)$ or one of its descendants is not restored after $P_i(k)$ replies to the BACKTRACK message. This contradicts Properties 4 and 5 and so the status of $P_i(k)$ and all of its descendants is restored.

Since neither of the above cases are possible, a contradiction arises and so the original assumption must be wrong. Therefore, if $P_i(k)$ returns MAYBE or YES in response to a CM and it is then sent a BACKTRACK message, the status of $P_i(k)$ and all of its descendants will be restored to what it was before $P_i(k)$ received the CM, and any status changes that resulted at deeper levels of recursion will be discarded.

Q.E.D.

Property 13

If $MP(k)$ receives a CM and returns NO after having called at least one of its children, then the status of every procedure $Pi^+(k+n)$, $n \geq 0$, will be the same after $MP(k)$ replies as before $MP(k)$ was sent message the CM.

Proof:

Assume that $MP(k)$ returns NO in response to a CM M_b after calling at least one of its children, but the status of some procedure $Pi^+(k+n)$, $n \geq 0$, is not the same after $MP(k)$ replies as before $MP(k)$ was sent the CM. If $MP(k)$ returns NO after receiving M_b and calling a child, it must make transition 21 after some child $Pj(k)$ returns NO. When $MP(k)$ makes transition 21, it discards any changes to VV and SL resulting from the CM and it returns to the state it was in before M_b was received. Therefore, the status of $MP(k)$ is restored and so $Pi^+(k+n)$ cannot be $MP(k)$.

If $Pi^+(k+n)$ is a procedure other than $MP(k)$, then four cases are possible: (1) it is in a subtree to the left of $Pj(k)$, (2) it is in a subtree to the right of $Pj(k)$, (3) it is in the same subtree as $Pj(k)$, or (4) it is a procedure at a deeper level of recursion. The first three cases are considered separately below; the fourth is included as part of them.

Case 1: $Pi(k)$ is in a subtree to the left of $Pj(k)$

If $Pi(k)$ is in a subtree to the left of $Pj(k)$, let $Pq(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$. By Lemma G1, $Pq(k)$ must have returned MAYBE and $MP(k)$ will send $Pq(k)$ a BACKTRACK message when it makes transition 21. By Property 12, the status of $Pq(k)$ and all of its descendants will return to what it was before $Pq(k)$ was sent the CM; therefore, $Pi(k)$ cannot be in a subtree to the left of $Pj(k)$. As well, any status changes resulting from recursive calls by procedures in subtrees to the left of $Pj(k)$ will be discarded.

Case 2: $Pi(k)$ is in a subtree to the right of $Pj(k)$

If $Pi(k)$ is in a subtree to the right of $Pj(k)$, let $Pq(k)$ denote the child of $MP(k)$ at the root of the subtree containing $Pi(k)$. By Lemma G2, $Pq(k)$ could not

have have been called before $MP(k)$ replied and so the status of $Pq(k)$ and all its descendents could not have changed, nor could they have made any recursive calls. Therefore, $Pi(k)$ cannot be in a subtree to the right of $Pj(k)$ and no status changes at deeper levels of recursion could have occurred.

Case 3: $Pi(k)$ is in the same subtree as $Pj(k)$

By Property 6, the status of $Pj(k)$ and all of its descendents will return to what it was before $Pj(k)$ received the CM, therefore $Pi(k)$ cannot be in the same subtree as $Pj(k)$. As well, by Property 11, $Pj(k)$ and its descendents did not make any recursive calls, or if such calls were made, any status changes that resulted at deeper levels of recursion were discarded.

If none of the above cases are possible, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ receives a CM and returns NO after having called at least one of its children, the status of every procedure $Pi^+(k + n)$, $n \geq 0$, will be the same after $MP(k)$ replies as before $MP(k)$ was sent message the CM.

Q.E.D.

Property 14

Given the following conditions:

1. a procedure $Pi^+(k)$ forwards a child's reply that includes a list of substitutions, and
2. it is then sent an AVR,

then its status will be restored to what it was before the reply was returned and it will issue an AVR to the child.

Proof:

First let $Pi^+(k)$ be $MP(k)$. When $MP(k)$ forwards a reply that includes a list of substitutions, it makes transition 10, 11, 12, or 13 from state 5 to 3. If it then receives an AVR, it must make transition 22 back to state 5 without any changes to

its problem set and so the status of $MP(k)$ has been restored to what it was before the solution was returned. As well, in making transition 22, it issues an AVR to the child that returned the solution.

Now let $P_i^+(k)$ be some other procedure $P_j(k)$. When $P_j(k)$ forwards a solution returned by a child, it makes transition 9, 10, 11, or 41 from state 4 to state 5. If it receives a subsequent AVR, it will make transition 14 back to state 4 without any changes to its problem set and so the status of $P_j(k)$ is restored to what it was before the solution was returned. As well, in making transition 14, it issues an AVR to the child that returned the solution.

Since $P_i^+(k)$ must be $MP(k)$ or some other procedure $P_j(k)$, if $P_i^+(k)$ forwards a child's reply that includes a list of substitutions and it is then sent an AVR, its status will be restored to what it was before the reply was returned and it will issue an AVR to the child.

Q.E.D.

Property 15

Given the following conditions:

1. a procedure $P_i(k)$ returns YES(S), MAYBE(S), or YES+MAYBE(S) after calling one of its own satisfaction functions, and
2. it is then sent an AVR,

then:

1. the status of $P_i(k)$ will be restored to what it was before the solution was sent to $MP(k+1)$ with the exception that $P_i(k)$ will be restored to state 6 rather than 12, and
2. $P_i(k)$ will send an AVR to $MP(k+1)$.

Proof:

The proof follows directly from the state transition diagram for $P_i(k)$. When a procedure passes a solution to $MP(k+1)$ for verification, it waits in state 12 for a

reply. If a response other than NO is returned, $P_i(k)$ makes transition 31, 32, 33, or 42 to state 11 and in so doing returns a tentative or definite solution to its parent. If $P_i(k)$ then receives an AVR, it must make transition 34 to state 6 and in so doing send an AVR to $MP(k+1)$. Since its problem set is not affected, (1) the status of $P_i(k)$ has been restored to what it was before $MP(k+1)$ had replied except that it now waits in state 6 for $MP(k+1)$ to respond, and (2) it sends an AVR to $MP(k+1)$. Q.E.D.

Property 16

Given the following conditions:

1. $MP(k)$ is sent a CM,
2. it makes transition 10, 11, 12, or 13 after some child $P_i(k)$ returns YES(S), MAYBE(S), or YES+MAYBE(S), and
3. $MP(k)$ is then sent an AVR,

then the status of every procedure $P_{x^+}(k)$ will be restored to what it was before $MP(k+1)$ replied most recently, with the exception that the procedure to call $MP(k+1)$ will be waiting in state 6 rather than 12.

Proof:

If $MP(k)$ forwards a reply from $P_i(k)$, some procedure $P_j(k)$ in the same subtree as $P_i(k)$ made transition 31, 32, 33, or 42 after making a recursive call and $MP(k+1)$ returned a reply other than NO. $P_j(k)$ could be $P_i(k)$ or one of its descendents.

Assume that after $MP(k)$ receives the AVR, the status of some procedure $P_{a^+}(k)$ is not restored to what it was before $MP(k+1)$ replied to $P_j(k)$. $P_{a^+}(k)$ could be $MP(k)$, in a subtree to the left or right of $P_i(k)$, or in the same subtree as $P_i(k)$; each case is considered below.

Case 1: $P_{a^+}(k)$ is $MP(k)$

By Property 14, the status of $MP(k)$ must be restored when it is sent an AVR and so $P_{a^+}(k)$ cannot be $MP(k)$.

Case 2: $Pa^+(k)$ is in a subtree to the left or right of $Pi(k)$

When $MP(k)$ forwards a solution returned by $Pi(k)$, it makes transition 10, 11, 12, or 13 and moves to state 3. A subsequent AVR forces it to make transition 22 back to state 5 and issue an AVR to $Pi(k)$. Since $MP(k)$ does not send messages to any other children, the status of the procedures in subtrees to the left and right of $Pi(k)$ will not be affected. Thus, $Pa^+(k)$ cannot be in a subtree to the left or right of $Pi(k)$.

Case 3: $Pa^+(k)$ is in the same subtree as $Pi(k)$

Two cases are possible; $Pa^+(k)$ could be $Pi(k)$ or it could be one of its descendants; each case is considered below.

Case 3.1: $Pa^+(k)$ is $Pi(k)$

If $Pi(k)$ made transition 31, 32, 33, or 42, then $Pi(k)$ is $Pj(k)$. When $MP(k)$ makes transition 22, it will send an AVR to $Pi(k)$ and by Property 15, if $Pi(k)$ made one of the listed transitions, its status will be restored with the exception of its new state being state 6. Therefore, $Pa^+(k)$ cannot be $Pi(k)$ if $Pi(k)$ made one of the listed transitions.

If $Pi(k)$ forwards a solution generated by some child $Pl(k)$, Property 14 indicates that the status of $Pi(k)$ will be restored to what it was before it called $Pl(k)$. Therefore, $Pa^+(k)$ cannot be $Pi(k)$ and so it must be one of its descendants.

Case 3.2: $Pa^+(k)$ is a descendent of $Pi(k)$

The following two cases must be considered.

Case 3.2.1: $Pi(k)$ is $Pj(k)$

If $Pi(k)$ made transition 31, 32, 33, or 42, a subsequent AVR would force it to make transition 34 to state 6 and issue an AVR to $MP(k+1)$. Since $Pi(k)$ would not send a message to any other procedure, the status of all of its children would not change and so $Pa^+(k)$ cannot be a descendent of $Pi(k)$ if $Pi(k)$ made one of the listed transitions.

Case 3.2.2: $Pi(k)$ is not $Pj(k)$

Now assume that $Pi(k)$ forwarded a solution returned by some child $Pl(k)$. By

Cases 3.1 and 3.2.1, $Pa^+(k)$ cannot be $Pi(k)$ and so it must be one of its descendents. The above arguments may now be repeated with $Pl(k)$ replacing $Pi(k)$; i.e., $Pa^+(k)$ could be in a subtree to the left or right of $Pl(k)$ or in the same subtree as $Pl(k)$. Using analagous arguments to those used in Case 2, $Pa^+(k)$ cannot be in a subtree to the left or right of $Pl(k)$ and so it must be in the same subtree as $Pl(k)$; using the arguments of Case 3.1, $Pa^+(k)$ cannot be $Pl(k)$ and so it must one of its descendents.

Every time these arguments are used, $Pa^+(k)$ must be at another level deeper in the hierarchy. Since the number of procedures is finite, eventually a childless procedure will be encountered that cannot be $Pa^+(k)$. This raises a contradiction and so $Pa^+(k)$ cannot be a descendent of $Pi(k)$ if $Pi(k)$ is not $Pj(k)$. Combined with the results of Case 3.2.1, it follows that $Pa^+(k)$ cannot be a descendent of $Pi(k)$. Similarly, since $Pa^+(k)$ cannot be $Pi(k)$ according to Case 3.1, it follows that $Pa^+(k)$ cannot be in the same subtree as $Pi(k)$.

If $Pa^+(k)$ cannot be $MP(k)$, in a subtree to the left or right of $Pi(k)$, or in the same subtree as $Pi(k)$, a contradiction arises and so the original assumption must be wrong. Therefore, if $MP(k)$ is sent a CM and it forwards a reply involving a list of substitutions as returned by some child $Pi(k)$ and $MP(k)$ is sent an AVR, then the status of every procedure $Px^+(k)$ will be restored to what it was before $MP(k+1)$ replied to $Pj(k)$, with the exception that the most recent procedure to call $MP(k+1)$ will be waiting in state 6 rather than 12.

Q.E.D.

Property 17

If the solver recurses n times, $n > 0$, before returning a solution, then:

1. $MP(n)$ returned MAYBE(S) or YES(S) by making transition 25, 27, or 28, and
2. every descendent of $MP(n)$ returned MAYBE or YES if it was called at all.

Proof:**Part 1: Proving $MP(n)$ makes transition 25, 27, or 28**

First assume that $MP(n)$ replies using a transition other than those listed above; i.e., it uses transition 5, 6, 15, 14, 26, 21, 10, 11, 12, or 13. If $MP(n)$ were to make one of the first three, it could not have determined any values since it was passed M_b , the most recent CM. However, whenever the solver recurses, some procedure $P_j(k)$ must send a list of assignment constraints to $MP(k+1)$ and so $MP(n)$ must determine some values. Thus, $MP(n)$ cannot make transition 5, 6, or 15.

If $MP(n)$ were to make transition 14, 26, or 21, it would return NO but this would mean that the solver has not recursed n times as stated. Therefore, $MP(n)$ cannot make transition 14, 26, or 21.

$MP(n)$ can make transition 10, 11, 12, or 13 only if some child $P_i(n)$ returns MAYBE(S), YES(S), or YES+MAYBE(S). $P_i(n)$ could do so only by forwarding the response of one of its children using transition 9, 10, 11, 41, 11, or 12, or by making transition 29, 31, 32, 33, or 42 after executing one of its satisfaction functions. Each case is considered below.

Case 1: $P_i(n)$ makes transition 31, 32, 33, or 42

$P_i(n)$ can make the above transitions only if it has reached state 12, and $P_i(n)$ can reach state 12 only if it sends a CM to $MP(n+1)$ and $MP(n+1)$ returns a response other than NO. However, this constitutes $n+1$ recursions which contradicts the statement that the solver recursed n times. Consequently, $P_i(n)$ cannot make transition 31, 32, 33, or 42.

Case 2: $P_i(n)$ makes transition 29

If $P_i(n)$ made transition 29, it would return NO to $MP(n)$ and $MP(n)$ would make transition 21. However, it has already been shown that $MP(n)$ cannot make transition 21 and so $P_i(n)$ cannot make transition 29.

Case 3: $P_i(n)$ makes transition 9, 10, 11, 41, 11, or 12

If $P_i(n)$ makes one of the above transitions, some child $P_j(n)$ returned a solution other than YES or MAYBE and so the above arguments apply again with

$P_j(n)$ replacing $P_i(n)$. Using the arguments of Cases 1 and 2, $P_j(n)$ cannot make transition 29, 31, 32, 33, or 42 and so it must make transition 9, 10, 11, 41, 11, or 12 after one of its children returns a solution other than YES or MAYBE.

The above arguments repeat indefinitely and each time they are used, there must be a procedure at another level deeper in the solver that forwards a solution returned by one of its children. Since the number of procedures is finite, eventually a childless procedure will be encountered and so a contradiction arises. Therefore, $P_i(n)$ cannot make transition 9, 10, 11, 41, 11, or 12.

If $P_i(n)$ cannot make any of the above transitions, then $P_i(n)$ must reply using transition 2, 18, 39, 19 or 20 and return MAYBE or YES. Therefore, $MP(n)$ cannot reply using transition 5, 6, 15, 14, 26, 21, 10, 11, 12, or 13, and so it must reply using transition 25, 27, or 28 and return MAYBE(S) or YES(S).

Part 2: Proving every descendent returned MAYBE or YES if it was called

Now assume that some descendent of $MP(n)$ does not return MAYBE or YES after being called. It was shown in Part 1 that $MP(n)$ must reply using transition 25, 27, or 28. If it uses the first, it does not call any of its children, therefore it must make transition 27 or 28.

By the results of Part 1, all of its children must return MAYBE or YES if called and by Lemmas F4 and F9, every descendent of these procedures must return MAYBE or YES if they are called at all. Therefore, if the solver recurses n times, $n > 0$, before returning a solution, every descendent of $MP(n)$ returned MAYBE or YES if it was called at all.

Q.E.D.

Property 18

If the solver recurses n times, $n > 0$, before returning a solution and $MP(n)$ is sent an AVR, then:

1. every procedure $P_i^+(n)$ will discard any status changes associated with the n^{th} level of recursion, and
2. NO or ABANDON will be returned to the procedure that called $MP(n)$.

Proof:

By Property 17, if the solver recurses n times, $MP(n)$ made transition 25, 27, or 28 before replying. If $MP(n)$ made transition 25 after receiving some CM M_b , it moved to state 1 after its problem set was totally reduced and satisfied. Since $MP(n)$ did not call any of its children before making transition 25, it is the only procedure to have been affected by M_b and so the status of all other procedures at the n^{th} level of recursion remains unchanged. A subsequent AVR causes $MP(n)$ to make transition 16, discard any changes to VV and SL, return to the state it was in before M_b was received, and return NO to its caller. Since $MP(n)$ did not call any of its children before making transition 16, no procedure at the n^{th} level of recursion is affected. Therefore, if $MP(n)$ makes transition 25 and it is then sent an AVR, all procedures $Pi^+(n)$ discard any status changes associated with the n^{th} level of recursion and NO is returned to the procedure that called $MP(n)$.

If $MP(n)$ made transition 27 or 28 after receiving some CM M_b , it moved to state 2 and by Property 17, all of its children returned MAYBE or YES after being sent CM's. A subsequent AVR causes $MP(n)$ to make transition 20 and issue BACKTRACK messages to all of its children. By Properties 4 and 5, this restores the status of all of its descendents to what it was before $MP(n)$ was sent M_b , therefore they discard any changes associated with the n^{th} level of recursion. As well, transition 20 causes $MP(n)$ to undo any changes to VV and SL, return to its previous state, and return ABANDON to its caller. Thus, all procedures $Pi^+(n)$ discard any status changes associated with the n^{th} level of recursion and NO or ABANDON is returned to the procedure that called $MP(n)$.

Q.E.D.

Property A4.1

Given the following conditions:

1. $MP(0)$ receives a CM.
2. it returns MAYBE(S), YES+MAYBE(S), or YES(S) using transition 10, 11, 12, or 13 after S has been forwarded back through $n > 0$ levels of recursion, and

3. $MP(0)$ subsequently receives an AVR.

then:

1. the status of every procedure $Pi^+(k)$, $k=0, \dots, n-1$, will be restored to what it was before $MP(n)$ replied, with the exception that any procedure $Pa(j)$, $j=0, \dots, n-1$ that called $MP(j+1)$ will return to state 6 rather than 12,
2. any status changes associated with the n^{th} level of recursion will be discarded.
3. NO or ABANDON will be returned to the procedure that had previously called $MP(n)$, and
4. the procedure that called $MP(n)$ will return to state 9 in order to test its next satisfaction function.

Consequently, the search will be continued from the last choicepoint.

Proof:

Each result is considered below.

Result 1: Restoring procedure status

It is stated that $MP(0)$ is sent an AVR after it forwards a solution returned by one of its children, therefore, by Property 16, the status of every procedure $Pi^+(0)$ will be restored to what it was before $MP(0)$ received the AVR (except for the procedure that called $MP(1)$), and by Property 15, an AVR will be sent to $MP(1)$. By induction and Properties 15 and 16, an AVR will be sent to $MP(k)$, $k=0, \dots, n-1$, since each $MP(k)$ forwarded a solution returned by one of its children. Consequently, the status of every procedure $Pi^+(k)$, $k=0, \dots, n-1$, will be restored to what it was before $MP(n)$ returned the solution, with the exception that any procedure $Pa(j)$, $j=0, \dots, n-1$ that called $MP(j+1)$ will be restored to state 6 rather than 12.

Result 2: Discarding status changes at the n^{th} level of recursion

By Result 1 and Property 15, an AVR will be sent to $MP(n)$ and so by Property 18, any status changes associated with the n^{th} level of recursion will be discarded.

Result 3: *Returning NO to $Pa(n-1)$*

Result 2 showed that an AVR will be sent to $MP(n)$ and so by Property 18, NO or ABANDON will be returned to some procedure $Pa(n-1)$, that is, the procedure that called $MP(n)$.

Result 4: *$Pa(n-1)$ continues the search with its next function*

By Results 1 and 2, $Pa(n-1)$ is waiting in state 6 for a reply from $MP(n)$ and by Result 3, $MP(n)$ returns NO or ABANDON. Therefore, $Pa(n-1)$ makes transition 43 or 58 back to state 9 and the search continues with the next satisfaction function.

The above results indicate that an AVR causes the status of procedures at all but the deepest level of recursion to be restored and the search to continue with the next choicepoint. The only exception is that the procedures which made recursive calls are restored to state 6 rather than 12.

Q.E.D.

Property A4.2

Given the following conditions:

1. a procedure $Pi(k)$ sends $MP(k+1)$ a CM.
2. $MP(k+1)$ replies so that $Pi(k)$ returns MAYBE(S), YES+MAYBE(S), or YES(S) using transition 31, 32, 33, or 42, and
3. $Pi(k)$ subsequently forwards an AVR to $MP(k+1)$,

then:

1. $Pi(k)$ will treat ABANDON results returned from $MP(k+1)$ as if they were NO results returned while it was waiting in state 12, and
2. $Pi(k)$ will treat all other responses from $MP(k+1)$ as if it received them while waiting in state 12.

Proof:

Result 1: *Treating ABANDON results as NO responses*

Assume that $Pi(k)$ does not treat an ABANDON response as if it were a NO response while waiting in state 12. If $Pi(k)$ receives an ABANDON reply, it makes transition 43 from state 6 to state 9 and it calls its next satisfaction function. If it had received a NO response in state 12, it would make transition 28 to state 9 and call its next satisfaction function. The results are the same and so $Pi(k)$ treats ABANDON results as if they were NO results received while in state 12.

Result 2: *Treating the other results*

If $Pi(k)$ receives a NO response in state 6, it makes transition 58 to state 9 and calls its next function; similarly, if it receives a NO while in state 12, it returns to state 9 and calls its next function. If $Pi(k)$ receive any other response, the following pairs of transitions cause $Pi(k)$ to move to state 11 and return identical results: 51 and 31; 52 and 32; 53 and 33; and 54 and 42. Therefore, it will treat all other responses from $MP(k+1)$ as if it received them while waiting in state 12.

Q.E.D.

Property A4.3

If $MP(\theta)$ returns NO in response to a CM M_b , then:

1. the status of every procedure $Pi^+(\theta)$ in the solver will be returned to what it was before $MP(\theta)$ received the CM, and
2. any status changes resulting from recursive calls which were made since M_b was received will be discarded.

Proof:

$MP(\theta)$ can return NO by making transition 14, 26, or 21. If it makes either of the first two, the result follows from Corollary 1 (with $k=0$). If $MP(\theta)$ makes transition 21, the result follows from Property 13 (with $k=0$).

Q.E.D.

Property A4.4

If $MP(\theta)$ returns NO or ABANDON in response to an AVR, then:

1. the status of every procedure $Pi^+(\theta)$ in the solver will be restored to what it was before $MP(\theta)$ received the most recent CM M_b , and
2. any status changes at deeper levels of recursion which were made since M_b was received will be discarded.

Proof:

$MP(\theta)$ can return NO or ABANDON in response to an AVR by making transition 16, 20, or 21: each case is considered below.

Case 1: $MP(\theta)$ makes transition 16

If $MP(\theta)$ makes transition 16, it previously made transition 15 or 25 after receiving a CM. If $MP(\theta)$ made either of these, it could not have called any of its children and so by Property 3, the status of every procedure $Pi^+(\theta)$ will be restored to what it was before $MP(\theta)$ received the CM.

Case 2: $MP(\theta)$ makes transition 20

Before $MP(\theta)$ can make transition 20, it must make transition 5, 6, 27, or 28 after all of its children returned MAYBE or YES. Transition 20 forces $MP(\theta)$ to issue BACKTRACK messages to all of its children and so by Property 12, (1) the status of all procedures $Pi(\theta)$ will be restored to what it was before $MP(\theta)$ received the CM, and (2) any status changes resulting from recursive calls will be discarded. As well, when $MP(\theta)$ makes transition 20, it discards any changes to VV and SL resulting from the previous CM, it returns to the state it was in before the CM was received, and it returns ABANDON to PE. Therefore, the status of all procedures is restored.

Case 3: $MP(\theta)$ makes transition 21

It was shown in Property A4.1 that if (1) $MP(\theta)$ is sent a CM M_b , (2) it forwards a solution returned by one of its children, and (3) it is then sent an AVR, then (1)

any procedures which made recursive calls are restored to state 6 rather than 12. and (2) the solver continues searching as if the innermost call to $MP(k)$ had failed. By Property A4.2, procedures restored to state 6 behave as if they were in state 12. (handling ABANDON responses as if they were NO replies), and so when the solver replies to an AVR message, it may be considered to have responded to CM M_b .

If $MP(0)$ returns NO in response to an AVR, it may be considered to be replying to the most recent CM for which NO or ABANDON has not already been returned. Consequently, the status of the procedures is affected in the same way. By Property A4.2, when $MP(0)$ returns NO in response to a CM, the status of every procedure will be restored to what it was before $MP(0)$ received the CM and any changes resulting from recursive calls will be discarded. Therefore, if $MP(0)$ returns NO in response to an AVR, the status of every procedure is restored and any changes resulting from recursive calls are discarded.

In conclusion, when $MP(0)$ returns NO in response to an AVR, (1) the status of every procedure in the solver will be restored to what it was before $MP(0)$ received the associated CM. and (2) any status changes that occurred at deeper levels of recursion will be discarded.

Q.E.D.

Appendix F

Combining Metalevel Data

In principle, the algebraic classification of an expression does not affect the way it could be grouped, nor does its group influence how it could be classified. However, the solver's search mechanism will have difficulties when sparse metalevel data is provided. Consider the case when a constraint is classified one way and at the same time it belongs to a group containing expressions of another class. For instance:

group: set1.

a(Y, Z) :- c1: $Y - Z = 0$ #[isa(c1, linear), belongs_to(c1, set1)],
 c2: $Y \times Z = 1$ #[isa(c2, nonlin), belongs_to(c2, set1)].

Although both *c1* and *c2* belong to *set1*, the procedure corresponding to the “linear” classification would receive *c1*, while that for “nonlin” would receive *c2*. Thus the data is split between two different procedures, each with partial knowledge of the group. Neither can attempt to find a solution even though there is enough data available.

Such problems could be avoided if every expression was described using a series of classifications from very limited to the most general solvable by the tree. A suitable procedure could be found for a group *Q* of constraints by looking for the procedure which is deepest in the hierarchy that has as descendants, all procedures with any group *Q* data. Unfortunately, this approach would be a burden on the user and it

5 of/de 5

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

would introduce another level of searching into the solver. Whenever a constraint of some group was sent to a procedure P_i , a means would be necessary to ensure that P_i had an ancestor with access to any other data associated with the group. More control mechanisms for passing and retracting data would also be needed.

END

05-06-92

FIN