REENGINEERING UNIFICATION AND T-ENTAILMENT FOR MANTRA IN C++

TANIA KHARMA

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MAY 1996

ISBN 0-612-18409-9

Abstract

Reengineering Unification and t-Entailment for Mantra in C++

Tania Kharma

The objective of this thesis lies in two directions. In one hand, understanding the concepts of unification and decidable inference mechanism, used in building the logic formalism of Mantra and implemented in its previous version in Common Lisp in 1991. On the other hand, learning about the object oriented paradigm and applying it in the design and implementation through the use of design patterns, the OMT notations, and C++ as the implementation language.
Mantra is a shell for hybrid knowledge representation and hybrid inferences. It supports three different formalisms: logic, frames, and semantic networks. In representing any kind of knowledge through a knowledge base system, one is faced with choosing among a broad repertoire of formalisms. Mantra provides a combination of knowledge representation formalisms, so the user can decide which representation is convenient for each piece of knowledge, and is not limited to only one formalism.
Our concentration is on reengineering the logic formalism module. Mantra uses a four-valued logic which has the syntax of standard first order order along with a decidable inference algorithm, called *t*-entailment.
Unification is at the heart of the inference mechanisms in Mantra. The efficient algorithm of Martelli and Montanari is used.
In view of its many advantages and its popularity, the object oriented paradigm is used in the design and implementation of this work. The primary design considerations are correctness, understandability, and extensibility.

Acknowledgments

I would like to thank my supervisor Dr. Gregory Butler for his continuous guidance and support. I am also grateful for his valuable suggestions and technical assistance especially in the design and implementation phases.

I also would like to express my gratitude to Dr. Tjandra Indra Adiono for his help and for all the theoretical background he was patient in teaching me.

Special thanks to my fiancé Wissam who gave me the moral support I needed.

I want to give my thanks to my friends and the many other people working in the department, for helping me in various ways during my work.

Finally, my efforts would not have been equal to this task had they not been supported and encouraged by my parents to whom I dedicate this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of this thesis lies in two directions. In one hand, understanding the concepts of unification and decidable inference mechanism, used in building the logic formalism of Mantra and implemented in its previous version in Common Lisp in 1991. On the other hand, learning about the object oriented paradigm and applying it in the design and implementation through the use of design patterns, the OMT notations and C++ as the implementation language.

To represent any kind of knowledge through a knowledge base system, one is faced with choosing among a broad repertoire of formalisms to achieve this goal. In making this choice, four properties should be taken into consideration [34]:

- Representational adequacy —the ability to represent the required knowledge.
- Inferential adequacy —the ability to manipulate the knowledge and infer new knowledge from the old.
- Inferential efficiency —the ability to direct the inference mechanism into the most efficient direction by appropriate guides.
- Acquisitional efficiency —the ability to acquire new knowledge using automatic methods.

To date no single system optimises all four properties. The main reason behind this is the "trade-off between expressive power and computational tractability" [24]. Mantra [8, 9], a shell for knowledge representation, offers three different formalisms:

logic, semantic nets, and frames. The user can represent his knowledge using any of the formalisms. Then infer knowledge from one or a combination of formalism, through this hybrid inference the functionality of one formalism could be used to enhance the inferential adequacy and the efficiency of the other. Our concern in this thesis is the logic formalism.

Logic has played an important role in knowledge representation because of its formal semantics, expressive power, and well understood properties as regards their completeness, soundness, and decidability. However, very little was done in defining decidable logics for knowledge representation. Most modifications are either extensions to first-order logic or ad-hoc changes to inference mechanism [29]. in Mantra, a four-valued logic, based on the work of Patel-Schneider [31] is used. The four-valued logic weakens the first order logic just enough to achieve a logic with a decidable inference process. The semantic-net and the frame formalisms are also based on the four-valued semantics. The main advantage of having unified semantics among the three formalisms, is the facility for definition of the interaction between different formalisms. Part of this thesis's work involved the design and implementation of this inference mechanism.

This inference requires its input to be formulae in Conjunctive Normal Form (CNF). Thus, the first step involved the design and implementation of the transformation of a first order logic formula into CNF.

Unification is at the heart of the inference mechanisms in Mantra. From the execution time point of view, unification is expensive and tricky. This is the reason why it has been introduced and studied by a number of researchers. In this thesis, we are concerned both in the design and implementation of an efficient unification module for Mantra based on the algorithm described by Martelli and Montanari [25].

Since 1980s, the object oriented approach has been getting more popular, because of its many advantages, such as reusability of software components leading to productivity and rapid system development, maintainability where an object can be replaced with a new implementation without affecting the other objects, and modularity. In addition, this paradigm is closer to the real world since it deals with objects, and it offers key features such as polymorphism, encapsulation, abstraction, and inheritance, which differentiate it from the traditional paradigm .

Along with the object oriented approach, the roots of object oriented design patterns

go back to the late 1970s and early 1980s. The design pattern concept can be viewed as an abstraction of the imitation activity people tend to do in copying parts of programs or designs written by others. More precisely, as Christopher Alexander says: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [20, page 2]. Design patterns make it easier to reuse successful designs, to choose design alternatives, and to improve the documentation and maintenance. Put simply, design patterns help a designer get a design "right" fast [33, page 2]. In this thesis, design patterns are used to solve the design problems encountered.
In the design description, the Object Modeling Technique (OMT) notations are used [10]. The OMT methodology, is developed by J. Rumbaugh and his colleagues, it supports the same notation during the three different phases of the software life cycle. And the notations and terminologies it uses are not dependent on any programming language. The selection of OMT for this work, is based on its popularity and enforced by the fact that "All of the object-oriented methodologies, have much in common, and should be contrasted more with non-object-oriented methodologies than with each other" [33].
The programming language C++ is used in this work since it supports a combination of object-oriented and traditional procedure-oriented programming.

Chapter 2

Background

2.1 Logic

Logic studies proofs, theorem inference from other theorems or axioms, and how assertions are combined and connected. It is separated into two aspects: One aspect is concerned with syntax — rules stating how strings of symbols can be built up; and the other with semantics — the meaning or interpretation of the syntactical objects. There are many logics: first order or predicate, second order, propositional, modal, temporal, many-valued, and many others. They differ in the concepts and the basic features they consider. Throughout the years, computer science has utilised many of them in areas such as programming languages, logic programming, proving the correctness of programs, automated theorem proving, artificial intelligence, and knowledge representation. Our interest is theorem proving and knowledge representation using logic. In this section, the classical propositional and predicate logic notations [17, 37] are studied and a wide variety of their theorem proving formalisms are discussed namely: tableaux and resolution mechanisms, Hilbert systems, natural deduction, the sequent calculus, and the Davis-Putman procedure. Then, a decidable four-valued logic is described.

2.1.1 Propositional Logic

Propositional logic starts with the simple connectives and $\wedge$, or $\vee$, and not $\neg$. We formalise atomic sentences which can be either true or false and then we combine them to form more complex sentences. The main interest is to show how the truth value of an atomic sentence extends to the truth of more complex sentences. The most important features studied in propositional logic are given below and the notations used are based on [17] and [37].

2.1.1.1 Syntax

- *Propositional letters* express propositions: P,Q,...
- *Formulae* are propositional letters connected using propositional connectives.
- *Propositional connectives*: $\vee, \wedge, \subset, \supset, \equiv, \not\supset, \not\subset, \not\equiv, \uparrow, \downarrow$.
- *Constants*: $\bot$ false , $\top$ true.

  $\top$, **t**, and true are used interchangeably. And $\bot$, **f**, and false are used interchangeably.
- An *atomic formula* is either a propositional letter or a constant.
- *Formulae* are defined as:
  1. All atomic formulae are formulae.
  2. For every formula F, $\neg F$ is a formula.
  3. For all formulae F and G, F op G is a formula, for any propositional connective op.
- Many transformations can be applied to a formula while preserving its semantics. Some of the transformations are: commutativity, associativity, and distributivity.

2.1.1.2 Semantics

Propositional logic is two-valued : true (**t**) and false (**f**). Table 1 summarises the interpretations of the binary connectives. The unary operator $\neg$ is simple : $\neg$ true = false and $\neg$ false = true.

2.1.1.3 Properties

1. Boolean Valuation:

		$\wedge$	$\vee$	$\supset$	$\subset$	$\uparrow$	$\downarrow$	$\not\supset$	$\not\subset$	$\equiv$	$\not\equiv$
t	t	t	t	t	t	f	f	f	f	t	f
t	f	f	t	f	t	t	f	t	f	f	t
f	t	f	t	t	f	t	f	f	t	f	t
f	f	f	f	t	t	t	t	f	f	t	f

Table 1: Interpretation of the Binary Connectives in Classical Logic

- *Boolean valuation* v is a mapping from the set of propositional formulae to the set {true, false} with the following conditions: v($\top$)=t; v($\bot$)=**f**; v($\neg X$)=$\neg$v(X); v(X op Y)=v(X) op v(Y) where op is any propositional connective.

  A boolean valuation is also known as an assignment.
- A *tautology* is a formula X for which v(X)=**t** for every boolean valuation v.
- A set S of formulae is *satisfiable* if there is a boolean valuation v that maps every formula in the set to **t**.
- Two formulae F and G are *equivalent* if for every boolean valuation v(F)=v(G).

2. Uniform Notation: In his book [17], M. Fitting used the notations: α formulae, for those that act conjunctively, and β formulae, for those that act disjunctively. v(α)=v(α_1) $\wedge$ v(α_2) and v(β)=v(β_1) $\vee$ v(β_2)

   In Table 2, all connectives are grouped under two categories : conjunctives and disjunctives. These notations are used in section 2.1.1.4.

3. Normal Forms: Formulae exist in many forms. However, some of these forms play an important role in theorem proving and they are known as normal forms.

   - Let $X_1, X_2, X_3, ..., X_n$ be a list of propositional formulae. $[X_1, X_2, X_3, ..., X_n]$ is the *generalised disjunction* of $X_1, X_2, X_3, ..., X_n$, where each X_i is a propositional formula and the only connective between formulae is $\vee$. And $< X_1, X_2, X_3, ..., X_n >$ is the *generalised conjunction*, where each X_i is a propositional formula and the only connective between formulae is $\wedge$.
   - A *literal* is either a propositional letter, its negation, or a constant.
   - A *clause* is a generalised disjunction where each X is a literal.

Conjunctive			Disjunctive		
α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \supset Y)$	X	$\neg Y$	$X \supset Y$	$\neg X$	Y
$\neg(X \subset Y)$	$\neg X$	Y	$X \subset Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\supset Y$	X	$\neg Y$	$\neg(X \not\supset Y)$	$\neg X$	Y
$X \not\subset Y$	$\neg X$	Y	$\neg(X \not\subset Y)$	X	$\neg Y$

Table 2: Conjunctive and Disjunctive Formulae

- A clause C_1 *subsumes* a clause C_2 if every literal in C_1 is in C_2.
- A *dual clause* is a generalised conjunction where each X is a literal.
- A formula is in *Conjunctive Normal Form* (CNF) if it is $< C_1, C_2, ..., C_n >$ where each C is a clause.
- A formula is in *Disjunctive Normal Form* (DNF) if it is $[D_1, D_2, D_3, ..., D_n]$ where each D is a dual clause.
- A *Horn formula* is in CNF and every disjunction contains at most one positive literal.

4. Hintikka's Lemma: *Hintikka's Lemma* helps us extend the proof of some theorem provers to first order logic. It is an abstract and generalised approach which connects syntax and semantics. A set H of formulae is called a propositional Hintikka set if it satisfies the following:

   - If A is a propositional letter then both $A \in \mathsf{H}$ and $\neg A \in \mathsf{H}$
   - $\bot \notin \mathsf{H}$;$\neg\top \notin \mathsf{H}$
   - If $\neg\neg Z \in \mathsf{H}$ then $Z \in \mathsf{H}$
   - If $\alpha \in \mathsf{H}$ then $\alpha_1 \in \mathsf{H}$ and $\alpha_2 \in \mathsf{H}$ (see Table 2)

- If $\beta \in \mathsf{H}$ then $\beta_1 \in \mathsf{H}$ or $\beta_2 \in \mathsf{H}$ (see Table 2)

The important result here is that every propositional Hintikka set is satisfiable.

5. Model Existence Theorem: The *Model Existence Theorem* relates syntax and semantics. The arguments in this theorem are given abstractly so we can use it to deal with the completeness of several proof procedures in propositional logic. A set of formulae is said to be *consistent* if no contradiction occurs when we apply a proof procedure to it.
The various features of consistency are used to construct a boolean valuation for this set. This construction will identify the essential features of consistency, which are grouped in an abstract consistency property.
The Model Existence Theorem is the assertion that the abstract consistency property is enough for the construction of a boolean valuation.
The idea of abstract consistency property is very general, so instead of talking about a consistency property C of a set of formulae, we talk about the collection of all sets having property C. If a set S is in the collection C we say that S is *C-consistent* (has property C). For C to be a *propositional consistency property* each set $S \in \mathsf{C}$ should meet the following:

   - If A is a propositional letter then both $A \in S$ and $\neg A \in S$
   - $\bot \notin S$; $\neg\top \notin S$
   - If $\neg\neg Z \in S$ then $S \cup \{Z\} \in \mathsf{C}$
   - If $\alpha \in S$ then $S \cup \{\alpha_1, \alpha_2\} \in \mathsf{C}$
   - If $\beta \in S$ then $S \cup \{\beta_1\} \in \mathsf{C}$ or $S \cup \{\beta_2\} \in \mathsf{C}$

   Two very important theorems: 1. If C is a propositional consistency property and $S \in \mathsf{C}$ then S is satisfiable (that is there is a boolean valuation that maps every element of S to true). 2. Let S be a set of propositional formulae. If every nonempty finite subset of S is satisfiable, then S is satisfiable.

6. Compactness Theorem: The *Compactness Theorem* states that a set of formulae is satisfiable if and only if every finite subset of it is satisfiable.

2.1.1.4 Propositional Logic Theorem Provers:

2.1.1.4.1 Semantic Tableau Method

Semantic tableaux are an extremely elegant and efficient proof procedure for propositional logic. It can also be extended to first order logic. Some proof procedures, like analytic tableaux, are variations of semantic tableaux, but the main idea is the same for all tableaux methods. Semantic tableaux are related to formulae in disjunctive normal form (DNF), and they are a refutation system since they involve working on $\neg X$ and concluding the opposite for X. For instance: if $\neg X$ has a tableau proof then we conclude that X does not have a tableau proof. The idea of a tableau proof is to expand X by giving it the form of a tree whose branches are conjunction of formulae and the tree itself is the disjunction of its branches.

$$\begin{array}{c} A_1 \\ A_2 \\ \cdot \\ \cdot \\ A_n \end{array}$$

is the one branch tableau for the finite set of propositional formulae $\{A_1, A_2, ..., A_n\}$ The expansion of a formula takes place according to rules known as the tableau expansion rules:

$$\frac{\neg\neg Z}{Z} \quad \frac{\neg\top}{\bot} \quad \frac{\neg\bot}{\top} \quad \frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \quad \frac{\beta}{\beta_1 \mid \beta_2}$$

Suppose that θ is a branch on a finite tree with nodes labelled by propositional formulae, and X a formula on θ. The first rule states that if X is $\neg\neg Z$, then a new node Z is added to the end of θ. Similarly, in the second rule if X is $\neg\top$, then the node $\bot$ is added. And in the third rule X is $\neg\bot$ and the node $\top$ is added. If X is α, the fourth rule adds a node to the end of θ labelled α_1 and another node after that labelled α_2. If X is β, the fifth rule adds left and right children to the final node of θ, one is labelled β_1 and the other is β_2.

The expansion resulting for a formula F is known as the tableau for this formula.

A branch is closed if X and $\neg X$ occur on it, or $\perp$ occurs on it. A tableau is closed if all its branches are closed. When a tableau is closed for $\neg X$ then X has a propositional tableau proof.

We summarise the semantic tableaux evaluation as follows:

- They are suitable for automation since there is a very well defined algorithm to implement them.

- Tableau proofs are shorter than using truth table verifications for a formula.

- Tableau proofs are sound: any formula that could be proved using tableau is a tautology.
  A set S of propositional formulae is satisfiable if there is a boolean valuation that maps every formula in S to true.
  A branch is satisfiable if the set of its formulae is satisfiable, since a branch is the conjunction of its formulae so all of them should be true. For a tableau to be satisfiable it is enough to have at least one satisfiable branch, since a tableau is a disjunction of its branches.
  If we apply any expansion rule to a satisfiable tableau, the result is a satisfiable tableau. From this we get the proposition that if there is a closed tableau for a set S, then S is not satisfiable. And we conclude that if any formula X has a tableau proof then X is a tautology.

- Tableau proofs are complete: any tautology could be proved using tableau.
  A finite set S of propositional formulae is tableau consistent if none of its formulae has a closed tableau. The collection C of all tableau consistent sets is a propositional consistency property. Then, according to the Model Existence theorem, S is satisfiable, so each of its formulae is a tautology.

- Implementation of tableau proofs requires imposing rules regarding: when to stop, skipping formulae, reusing formulae, and choosing the next formula. This is not very difficult to do except deciding whether a formula should be reused or not. Reusing formulae over and over in tableaux proofs is a controversial issue: non-classical logicians claim we should, while classical ones claim we should not. The idea behind this is that reusing formulae makes completeness easy to prove by a general method but implementation becomes difficult [17, page 39].

2.1.1.4.2 Resolution Method

The resolution principle was originally proposed by J.A. Robinson in the early 1960s [36]. The motivation for developing this principle was to improve the efficiency of earlier proof methods. He admitted that the single inference rule of his calculus does not necessarily lead to easily understandable proofs, but he said that all that is required is that it is sound and recursive. Moreover, this inference rule constitutes a complete calculus for first order logic. Resolution is a refutation proof procedure that operates on formulae in conjunctive normal form (CNF).

In a resolution proof, we represent a conjunction of disjunctions by listing its members in a sequence, one disjunction on each line.

$$\begin{array}{c} [A_1] \\ [A_2] \\ \cdot \\ \cdot \\ [A_n] \end{array}$$

is a resolution expansion for the finite set of propositional formulae $\{ A_1, A_2, \ldots, A_n \}$. For adding new lines, we either apply one of the resolution expansion rules or the resolution rule. We keep doing this until we get the empty clause([]). A resolution containing the empty clause is *closed.* Resolution is a refutation system, so a closed resolution expansion for $\neg X$ is a resolution proof of X. X is a theorem of the resolution system if it has a resolution proof.

The Resolution Expansion Rules are the following:

$$\frac{\neg\neg Z}{Z} \quad \frac{\neg\top}{\bot} \quad \frac{\neg\bot}{\top} \quad \frac{\beta}{\begin{array}{c}\beta_1 \\ \beta_2\end{array}} \quad \frac{\alpha}{\alpha_1 \mid \alpha_2}$$

The first rule states that if we have a disjunction D containing $\neg\neg Z$ then a disjunction follows that is like D except that it contains an occurrence of Z instead of $\neg\neg Z$. Similarly the second and third rules replace occurrences of $\neg\top$ and $\neg\bot$ by $\bot$ and $\top$ respectively. In the fourth rule, if β is a disjunction in D then another D follows containing occurrences of both β_1 and β_2 where D contained β, for example, $[\neg Q, \neg R]$ follows from $[\neg(Q \wedge R)]$. In the fifth rule, having a formula α two disjunctions follow, one like D but with α replaced by α_1 and the other with α replaced by α_2, for example, $[\neg P]$ and $[\neg(Q \wedge R)]$ follow from $[\neg P \wedge \neg(Q \wedge R)]$.

The resolution rule is of a quite different nature than the resolution expansion rules:

$$\frac{\begin{array}{c}[X_{11}, X_{12}, ..., X_{1n}, X] \\ [X_{21}, X_{22}, ..., X_{2m}, \neg X]\end{array}}{[X_{11}, X_{12}, ..., X_{1n}, X_{21}, X_{22}, ..., X_{2m}]}$$

It states the following:

Having D_1 and D_2 are two disjunctions, with X a formula occurring in D_1 and $\neg X$ in D_2. Let D be the result of first, deleting all X from D_1. Second, deleting all $\neg X$ from D_2. Third, combining the resulting disjunctions. D is said to be the result of resolving D_1 and D_2 on X. D is called the *resolvent* of D_1 and D_2 and X is the formula being resolved on. We say that D follows from the disjunctions D_1 and D_2 by the application of the resolution rule.

In evaluating resolution, the following points were raised:

- It is quite popular in automatic theorem proving because it is simple to implement.
- It has a single, simple inference rule known as the resolution rule.
- The resolution method requires that the proposition should be in conjunctive normal form.
- The resolution method is a sound proof procedure.
  A resolution expansion is satisfiable if there exists some boolean valuation that maps every line of it to true. Since resolution expansion is a generalised conjunction of its disjunctions then all disjunctions should map to true for the resolution to map to true. If we apply a resolution expansion rule or the resolution rule to a satisfiable resolution expansion then the result is another satisfiable resolution expansion. Thus, satisfiability is preserved. If we find a closed resolution expansion for a set S, then S is not satisfiable because at least one of its elements is not true. So we conclude that if any formula X has a resolution proof then X is a tautology.

- The resolution method is a complete proof procedure.
  A finite set S of propositional formulae is resolution consistent if there is no closed resolution expansion for S. A resolution derivation from a set S of disjunctions is a sequence of disjunctions each of which is a member of S, or comes from an earlier term in the sequence by applying one of the resolution expansion rules or by applying the resolution rule. The collection of all resolution consistent sets is a propositional consistency property. Then, according to the Model Existence theorem, S is satisfiable, so each of its formulae is a tautology.

- The controversy on re-using formulae in tableau applies for resolution method as well. Thus, non-strict versions are easy to prove but difficult to implement and vice-versa.

2.1.1.4.3 Hilbert or Axiom Systems

Over the last hundred years many different calculi for logic have been developed. This development started with Frege in 1879, and later in 1928 led to the Hilbert type calculi in which valid formulae are derived from a sequence of basic logical formulae using few forms of inferences [16]. Unlike tableaux and resolution mechanisms which start with the formula and try to work with it in order to prove it, Hilbert systems start with known tautologies, derive consequences, derive consequences from consequences until the formula we want to prove is reached.
A Hilbert system consists of the following important features:

- *Axioms* which are the starting point and assumed to be true, and *rules of inference* used to show that a formula follows from another one.

- A *proof* in Hilbert system is nothing but a sequence $X_1, X_2, ..., X_n$ of formulae such that each X_i is either an axiom or follows from earlier formulae $X_1 ... X_{i-1}$ by applying one of the rules of inference.

- X is a *theorem* of a Hilbert System if X is the last line (X_n) of a proof.

- The most common rule of inference provided is the modus ponens:

$$\frac{X \quad X \supset Y}{Y}$$

This means that Y is accepted to be true if X and $X \supset Y$ have been previously shown to be true. In other words, we infer Y.

An infinite number of axioms is available but they all follow a finite number of axiom schemes. Assuming that P and Q are propositional letters and that X and Y are formulae, we have the following axiom schemes:

1. $X \supset (Y \supset X)$
2. $(X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z))$
3. $\perp \supset X$
4. $X \supset \top$
5. $\neg\neg X \supset X$
6. $X \supset (\neg X \supset Y)$
7. $\alpha \supset \alpha_1$
8. $\alpha \supset \alpha_2$
9. $(\beta_1 \supset X) \supset ((\beta_2 \supset X) \supset (\beta \subset X))$

In evaluating Hilbert Systems, the following points were raised:

- Hilbert systems are sound: If X has a Hilbert proof then X is a tautology.
- Hilbert systems are complete: If X is a tautology then X has a Hilbert proof.
- Hilbert systems rely on heuristics for the choice of axioms. The length of the proof can vary greatly depending on the heuristics.
- A user can derive a huge number of consequences as work progresses, which makes this system again not good for automation.
- A proof in a Hilbert system is very easy to explain to people and may provide insights that other systems do not.

- Certain non-classical logics use only Hilbert systems. Hilbert systems are historically very important and are widely used.

- Hilbert systems are simple because they consist of a set of axioms together with a few rules of inference. But the problem is that in practice proofs tend to be long, tedious and difficult to perform.

2.1.1.4.4 Natural Deduction

During 1934-35, Gerhard Gentzen [21] developed a system of natural deduction intended to allow proofs to be performed in a way which corresponds to human reasoning. The major principle of natural deduction is that there should be separate rules that allow the introduction and removal of each logical symbol. Thus, we find many rules of inference and few axioms. The way to proceed is to start from the assumed axioms (taken as true always) and get to the goal (that is the theorem we wish to prove) using the rules of inference. In a natural deduction system, we have subordinate proofs which are conclusions derived from premises, and once we finish with these premises we discharge them in order to produce an assumptio [illegible] ce result. To make this idea clear we use the following notation: each subordinate proof is written in a box, with the first line inside the box being the assumption made in that subordinate proof, and the first line below the box being the result of discharging the assumption. Note that once the box is closed the assumption is discharged, which means that the assumption cannot be used in other boxes. The box is an analogy of the scope of the assumption. At any stage of the proof, the formulae that are in boxes which are not closed are the active formulae. The following is a list of rules which are used in natural deduction systems:

1. Implication rule: if we can derive Y from X then we can discharge the assumption X and conclude that we have proved $X \supset Y$.

2. Modus Ponens rule: From X and $X \supset Y$ we conclude Y.

3. Constant rules:

$$\frac{\bot}{X} \quad \frac{}{\top}$$

4. Negation rules:

$$\frac{\begin{array}{c}X\\ \neg X\end{array}}{\bot} \qquad \begin{array}{c}\boxed{\begin{array}{c}X\\ \cdot\\ \cdot\\ \bot\end{array}}\\ \neg X\end{array} \quad \begin{array}{c}\boxed{\begin{array}{c}\neg X\\ \cdot\\ \cdot\\ \bot\end{array}}\\ X\end{array}$$

5. Connective rules:

$$\alpha \text{ Elimination:} \quad \frac{\alpha}{\alpha_1} \quad \frac{\alpha}{\alpha_2}$$

$$\alpha \text{ Introduction:} \quad \frac{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}{\alpha}$$

$$\beta \text{ Elimination:} \quad \frac{\begin{array}{c}\neg\beta_1\\ \beta\end{array}}{\beta_2} \quad \frac{\begin{array}{c}\neg\beta_2\\ \beta\end{array}}{\beta_1}$$

$$\beta \text{ Introduction:} \quad \begin{array}{c}\boxed{\begin{array}{c}\neg\beta_1\\ \cdot\\ \cdot\\ \beta_2\end{array}}\\ \beta\end{array} \quad \begin{array}{c}\boxed{\begin{array}{c}\neg\beta_2\\ \cdot\\ \cdot\\ \beta_1\end{array}}\\ \beta\end{array}$$

The following is an example of a natural deduction proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

1. $\neg(P \wedge Q)$

2. $\neg\neg P$

3. $\neg Q$

4. $\neg P \vee \neg Q$

5. $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$

1. is an assumption.

2. is a second assumption.

3. follows from 1 and 2 using the first β elimination rule where we take β as $\neg(P \wedge Q)$.

4. follows from the first β introduction rule where we take β as $\neg P \vee \neg Q$.

5. follows from 1 and 4 using the implication rule.

A natural deduction derivation of X from a set S of formulae is a proof of X. Natural deduction derivation allows at any stage, any member of S to be used as a line (axiom), and all assumptions are discharged before the end of the proof so X is true without any assumptions.

The evaluation of Natural Deduction System is summarised as follows:

- Natural deduction system is sound: If X has a natural deduction proof then X is a tautology.
- Natural deduction system is complete: If X is a tautology then X has a natural deduction proof.
- Natural deduction is very close to human reasoning, thus easy to understand.
- Natural deduction is not efficient in developing proofs. However, natural deduction provides a clear form for displaying proofs once they have been found.

- Proofs tend to be lengthy and complex.
- Choosing the right and most appropriate instances of the axiom schema for a particular proof is a difficult task. Usually, it requires enumerating all possible deductions until the required proof is generated. This could be overcome by performing the proof backwards.
- Deciding which rule should be chosen to be applied at each step is another difficult task. Of course, the straightforward way is to try all applicable rules at each step of the proof but this makes the proof very lengthy and time consuming.

2.1.1.4.5 Gentzen Sequents or Sequent Calculus

Historically, both natural deduction and sequent calculus were introduced by Gentzen. However, sequent calculus can be regarded as intermediate between tableaux and natural deduction. It was in realising the limitations of the natural deduction calculus for developing proofs that Gentzen was motivated to design the Sequent Calculus. This involved showing that proofs are performed in a direct way, starting with axiom sequents and successively introducing connectives until we get the theorem we want to prove [17]. Later, tableaux systems used a backward application of the rules of sequent calculus.

Some notations and definitions used in sequent calculus:

- The connectives used are $\vee$, $\wedge$, and $\supset$.
- A *sequent* is a pair $< \Gamma , \Delta >$ of finite sets of formulae. This could also be written as $\Gamma \rightarrow \Delta$. A sequent is like an assertion: if all the formulae on the left of the arrow are true, then at least one of the formulae on the right is true. This is described as follows: A boolean valuation v($\Gamma \rightarrow \Delta$) = true if v(X) = false for some $X \in \Gamma$ or v(Y) = true for some $Y \in \Delta$.

The axioms and rules used are as follows:

1. Axioms:

$$X \rightarrow X$$
$$\bot \rightarrow$$
$$\rightarrow \top$$

2. Structural Rule, Thinning: If $\Gamma_1 \subseteq \Gamma_2$ and $\Delta_1 \subseteq \Delta_2$ then

$$\frac{\Gamma_1 \rightarrow \Delta_1}{\Gamma_2 \rightarrow \Delta_2}$$

3. Negation Rules:

$$\frac{\Gamma \rightarrow \Delta, X}{\Gamma, \neg X \rightarrow \Delta} \qquad \frac{\Gamma, X \rightarrow \Delta}{\Gamma \rightarrow \Delta, \neg X}$$

4. Conjunction Rules:

$$\frac{\Gamma, X, Y \rightarrow \Delta}{\Gamma, X \wedge Y \rightarrow \Delta} \qquad \frac{\Gamma \rightarrow \Delta, X \quad \Gamma \rightarrow \Delta, Y}{\Gamma \rightarrow \Delta, X \wedge Y}$$

5. Disjunction Rules:

$$\frac{\Gamma, X \rightarrow \Delta \quad \Gamma, Y \rightarrow \Delta}{\Gamma, X \vee Y \rightarrow \Delta} \qquad \frac{\Gamma \rightarrow \Delta, X, Y}{\Gamma \rightarrow \Delta, X \vee Y}$$

6. Implication Rules:

$$\frac{\Gamma \rightarrow \Delta, X \quad \Gamma, Y \rightarrow \Delta}{\Gamma, X \supset Y \rightarrow \Delta} \qquad \frac{\Gamma, X \rightarrow \Delta, Y}{\Gamma \rightarrow \Delta, X \supset Y}$$

A *proof* in sequent calculus is a tree labelled with sequents meeting the following conditions: leaf nodes are axioms; if a node has children then those children follow from the sequent labelling the node by applying one of the sequent calculus rules. The label o.. the root node is the sequent we want to prove. Finally, a formula X is a theorem of the sequent calculus if the sequent $\rightarrow X$ has a proof.
The following is an example of the sequent calculus proof of $\neg(P \wedge Q) \supset (\neg P \vee \neg Q)$:

$P \rightarrow P$	axiom	$Q \rightarrow Q$	axiom
$P, Q \rightarrow P$	thinning	$P, Q \rightarrow Q$	thinning

$P, Q \rightarrow P \wedge Q$ conjunction

$Q \rightarrow P \wedge Q, \ \neg P$ negation

$\rightarrow P \wedge Q, \ \neg P, \neg Q$ negation

$\neg(P \wedge Q) \rightarrow \neg P, \ \neg Q$ negation

$\neg(P \wedge Q) \rightarrow \neg P \vee \neg Q$ disjunction

$\rightarrow \neg(P \wedge Q) \supset (\neg P \vee \neg Q)$ implication

The following points summarise the Sequent Calculus evaluation:

- Sequent calculus is sound: if X is a theorem of sequent calculus then X is a tautology.
- Sequent calculus is complete: if X is a tautology then X is a theorem of the sequent calculus.
- Sequent calculus can be easily implemented. In fact most systems that have been implemented on computers for automatic proving have sequent calculus bases.
- All proof systems designed mainly for the purpose of efficient proof development have a great relationship with sequent calculi.
- Tableaux systems, which are very good for automation, correspond to the backward application of rules of sequent calculus.

2.1.1.4.6 Davis Putnam Procedure

The Davis Putnam procedure was introduced in 1960 [15]. It is a refutation method. In order to apply the Davis Putnam procedure on a formula, first it must be converted into its CNF. A *block* is a disjunction of formulae in CNF. The second step involves transforming the blocks into new blocks by applying rules. These transformations will preserve the satisfiability of the block. We keep transforming until we reach a block that is obviously satisfiable or unsatisfiable. The rules that could be applied are the following:

1. Preliminary 1: Remove any repetition within a clause in a block, and arrange the literals.
2. Preliminary 2: Delete any clause that contains both a literal and its complement (P and $\overline{P} = \neg P$ are complementary literals). Delete any clause that contains $\top$. Delete all occurrences of $\bot$.
3. One Literal Rule: If B is a block, S is a CNF formula in B, and S contains the one-literal clause $[L]$. Then, remove from S all clauses containing L, and delete all occurrences of $\overline{L}$ from the remaining clauses.

4. Affirmative Negative Rule: If B is a block, S is a CNF formula in B, and some clauses in S contain the literal L and none of them contains $\overline{L}$. Then, we can remove from S all clauses containing L.

5. Subsumption Rule: If B is a block, S is a CNF formula in B, C_1 and C_2 are clauses in S, and C_1 subsumes C_2. Then, we can remove C_2 from S.

6. Splitting Rule: If B is a block, and S a CNF formula in this block. In S some clauses contain the literal L others contain $\overline{L}$. S_L is the CNF formula that results when all clauses in S containing L are removed, and all occurrences of $\overline{L}$ are deleted. In the same way, we could have $S_{\overline{L}}$

A Davis Putnam *derivation* for a block B is the finite sequence of blocks B_1,B_2,B_3...B_n. Where each block follows from the previous one by applying one of the rules. A derivation succeeds if each CNF formula in the last block contains the empty clause. A derivation fails if the last block contains an empty CNF formula (no literals in all its disjunctions). Any application of the Davis Putnam procedure on a formula X must terminate. If it is a success then X is a tautology otherwise it is not a tautology.

In evaluating Davis Putnam Procedure, the following point are raised:

- The Davis Putnam procedure is complete.
- The Davis Putnam procedure is sound.
- Davis Putnam procedure is very suitable for automation since its algorithm is not difficult to implement. And it is considered among the fastest.
- Some transformations are required before the procedure starts.
- A clear and simple set of rules are applied to transform the block. These transformations will always terminate with an answer that either the formula is a tautology or it is not.

2.1.1.5 Summary of Propositional logic theorem proving

The following Table 3 is a summary of the theorem provers just described. The information is based on readings from [17, 19, 16, 5, 12]

	Semantic Tableau	Resolution System	Hilbert System	Natural Deduction	Gentzen Sequents	Davis Putnam
Sound Complete	yes	yes	yes	yes	yes	yes
Suitable for Automation	very good	very good	very bad	very bad	very good	very good
Transformation Required	yes	yes	no	no	no	yes
Understanding Complexity	fair	fair	very easy	very easy	fair	fair
Rely on Heuristics	no	no	yes	yes	no	no
Fast	average	average	no	no	average	very

Table 3: Summary of Propositional Logic Theorem Provers

2.1.2 Predicate Logic

Predicate logic is an extension of propositional logic thus, it allows us to describe concepts we were not able to express with the limited tools of propositional logic. For instance, how to express that a property is true for certain objects or for all objects. The most important features studied in predicate logic are given below and the notations used are also based on [17, 37].

2.1.2.1 Syntax

- The *quantifiers* are: existential quantifier (there exists) $\exists$ and universal quantifier (for all) $\forall$.
- A *first order language* L(R,F,C) is determined by:
  - A finite set R of *relation* symbols where each relation has a positive integer which determines its arity.
  - A finite set F of *function* symbols where each function has a positive integer which also determines its arity.
  - A finite set C of *constant* symbols.
- Variables are identifiers other than those in R,F, or C.

- A term of L(R,F,C) is:
  - Any constant symbol member of C.
  - Any variable.
  - Any n-place function symbol $f(t_1, ..., t_n)$ where $f \in$ F of arity n, and each t_i is a term of L(R,F,C).
- A term is closed if it contains no variables.
- An *atomic formula* is a constant notation ($\top$ or $\bot$) or a relation $r(t_1, ..., t_n)$ where $r \in$ R of arity n, and each t_i is a term of L(R,F,C).
- *Formulae* are defined as:
  - All atomic formulae are formulae.
  - For every formula F, $\neg F$ is a formula.
  - For all formulae F and G, F op G is a formula, for any propositional connective op.
  - For every formula F and variable x, $(\exists x)F$ is a formula.
  - For every formula F and variable x, $(\forall x)F$ is a formula.
- Variables within a formula occur either *bound* or *free*.
  A *free variable occurrence* is defined as:
  - Any variable in an atomic formula.
  - Any variable in $\neg A$ that is free in A.
  - Any variable in A op B that is free in both A and B and for any propositional connective op.
  - Any variable in $(\forall x)A$ or $(\exists x)A$ that is free in A and is not x.

  A variable occurrence is bound if it is not free. For example in the formula $\forall x$ the variable x is bound, while in the formula $\exists x(Py \vee Qy)$ the variable y is free. A formula is *closed* if it has no free variable.
- The concept of equivalence as well as the transformation rules described in propositional logic can be extended to predicate logic.

- A *substitution* is defined as a mapping from a set of variables to a set of terms S: $V \rightarrow T$. Applying a substitution to a formula F replaces all free occurrences of a variable in F by the corresponding term. $F[x/t]$ denotes the application of the substitution of x by the term t.

2.1.2.2 Semantics

- In first order logic to give meaning to a formula we need to specify a model which consists of two items: a *domain* D for the quantifiers, and an *interpretation* I for the constants, functions and relational symbols with respect to the domain.

- A *model* of L(R,F,C) will be a pair M $= <$ D,I $>$ where D is a non-empty set and I is a mapping such that:

    - For every constant $c \in$ C, $c^{\mathsf{I}} \in$ D.
    - For every n-ary function symbol $f \in$ F, f^{I}:$D^n \rightarrow$ D.
    - For every n-ary relation symbol $r \in$ R, $r^{\mathsf{I}} \subseteq D^n$.

- An *assignment* in a model M $= <$ D,I $>$ is a mapping A from the set of variables to the set D.
  An assignment A in M $= <$ D,I $>$ of L(R,F,C) associate to each term t of L(R,F,C) a value t^{IA} in D as follows:

    - For a constant c, $c^{\mathsf{I,A}} = c^{\mathsf{I}}$
    - For a variable v, $v^{\mathsf{I,A}} = v^{\mathsf{A}}$
    - For a function symbol f, $[f(t_1, ..., t_n)]^{\mathsf{I,A}} = f^{\mathsf{I}}(t_1^{\mathsf{I,A}}, ..., t_n^{\mathsf{I,A}})$

- The interpretation of terms and formulae can be defined recursively.

- We associate a truth value (true or false) to each formula F in M $= <$ D,I $>$ for a language L (R,F,C) and an assignment A: $F^{\mathsf{I,A}}$.

- A formula F is ***true in a model*** if $F^{\mathsf{I,A}}$ = true for all assignments A. It is *valid* if it is true in all models of L. A set of formulae is *satisfiable* if there is some assignment A for which $F^{\mathsf{I,A}}$ = true for all F member of the set. The set is satisfiable if it is satisfiable in some model.

Universal		Existential	
γ	$\gamma(t)$	δ	$\delta(t)$
$(\forall x)\Phi$	$\Phi[x/t]$	$(\exists x)\Phi$	$\Phi[x/t]$
$\neg(\exists x)\Phi$	$\neg\Phi[x/t]$	$\neg(\forall x)\Phi$	$\neg\Phi[x/t]$

Table 4: Universal and Existential Formulae and Instances

- A model $\mathsf{M} = <\mathsf{D},\mathsf{I}>$ for L is a *Herbrand model* if D is the set of closed terms and for each closed term t, $t^{I}=t$.
  If $\mathsf{M} = <\mathsf{D},\mathsf{I}>$ is a Herbrand model: 1. for any term t of L, $t^{\mathsf{I},\mathsf{A}} = (t\mathsf{A})^{\mathsf{I}}$ (in $t^{\mathsf{I},\mathsf{A}}$ A is an assignment which gives values to variables, and in $(t\mathsf{A})^{\mathsf{I}}$ A is a substitution which substitutes variables by terms) 2. for any formula F of L $F^{\mathsf{I},\mathsf{A}} = (F\mathsf{A})^{\mathsf{I}}$ 3. if $(\forall x)F$ is true in M then $F[x/d]$ is also true for every $d \in \mathsf{D}$ and if $(\exists x)F$ is true in M then $F[x/d]$ is also true for some $d \in \mathsf{D}$.

2.1.2.3 Properties

1. Uniform Notations: In Table 4, γ *formulae* are associated with universal quantifiers and δ *formulae* with existential quantifiers.

   If γ(t) and δ(t) have property Q for each term t, then γ and δ have property Q. In a Herbrand model $\mathsf{M}=<\mathsf{D},\mathsf{I}>$ for a language L, a formula γ of L is true in M if and only if γ(d) is true in M for every $d \in \mathsf{D}$, and a formula δ of L is true in M if and only if δ(d) is true in M for some $d \in \mathsf{D}$.

2. Normal Forms: A very important feature of predicate logic is the transformation of any formula into standard forms while preserving its semantics: In a *rectified formula*, no variable is bound and free and all quantifiers refer to different variables. A *prenexed formula* has all the quantifiers at the front. A formula in RPF is in a rectified and prenexed form. A skolemized formula is a formula in RPF where each existentially bound variable is replaced by its *skolem function* — a skolem function is a new function symbol introduced into the formula and has as arguments the variables in the universal scope of the existentially bound variable to be replaced. A formula in CNF is a skolemized formula with its *matrix* part —part of the formula without quantifiers, as a conjunction of disjunctions.

All the transformations can be done algorithmically. Many proof mechanisms, like resolution for example, require the input formula to be in CNF.

3. Hintikka's Lemma: A set of sentences H is called a *first order Hintikka set with respect to a language L* if:

   - It satisfies all the properties for the Hintikka's Lemma for propositional logic.
   - If $\gamma \in$ H then $\gamma(c) \in$ H for every closed term c of L.
   - If $\delta \in$ H then $\delta(c) \in$ H for some closed term c of L.

   The important property of Hintikka set is that it is satisfiable in a Hebrand Model.

4. Parameters: Whenever within a proof we want to introduce a new item having a given property we should use a constant symbol that has not been used yet, so we introduce a set called *parameters* and we denote it by L^{par} of the language L.

5. The *Model Existence Theorem*: Let L be a language, L^{par} an extension of L, and C a collection of sets of sentences of L^{par}. C is said to be a *first order consistency property* if in addition to the Model Existence properties for propositional logic:

   - If $\gamma \in S$ then $S \cup \{\gamma(t)\} \in$ C for every closed term t of L^{par}.
   - If $\delta \in S$ then $S \cup \{\delta(t)\} \in$ C for some parameter t of L^{par}.

   If C is a first order consistency property with respect to L, S a set of se. ̣ nces of L, and $S \in$ C then S is satisfiable in a Herbrand model. The main use of the model existence theorem just stated is to prove completeness. There are also some other applications:

   - Let S be a set of sentences of the language L, if every finite subset of S is satisfiable so is S. This is the *Compactness Theorem.*
   - A set S of sentences of a language L is *satisfiable* if and only if it is satisfiable in a Herbrand Model with respect to L^{par}. A sentence X of L is valid if and only if X is true in all models that are Herbrand with respect to L^{par}.

6. Logical Consequences: In addition to knowing whether a formula is valid or not we also want to know which formulae follow from which formulae. A sentence X is a *logical consequence* of a set of sentences S if X is true in every model in which all the members of S are true.

2.1.2.4 First Order Logic Theorem Provers:

2.1.2.4.1 Semantic Tableaux

In first order logic, tableau proofs are constructed exactly as in propositional logic but with two additional tableau expansion rules which are the following:

$$\begin{array}{cc} \dfrac{\gamma}{\gamma(t)} & \dfrac{\delta}{\delta(p)} \\ \text{for any closed} & \text{for a new} \\ \text{term t of } L^{par} & \text{parameter p} \end{array}$$

The first rule suggests that we replace $\forall x \Phi$ with $\Phi[x/t]$ where t is any closed term. The second rule is to replace $\exists x \Phi$ with $\Phi[x/t]$ where t is a new parameter that does not have an intended meaning in the original language L. The reason why the parameter should be new is that: Suppose we have proved the statement $(\exists x)Px$ so we say let a be such as x and we write Pa so P holds for at least one a. Then we want to show for another property Q that there exists x such that Qx, we cannot say let a be such x because we have already committed a for x such as Px and we do not know whether there exists a that can hold for both properties, so we chose another parameter.

A tableau branch is satisfiable if the sentences on it are satisfiable and the tableau is satisfiable if at least one of its branches is satisfiable. Whenever a tableau expansion rule is applied to a satisfiable tableau, the result is another satisfiable tableau.

A finite set S of sentences of L^{par} is *tableau consistent* if there is no closed tableau for S. The collection of all tableau consistent sets is a first order consistency property.

In evaluating Semantic Tableau the following points were raised:

- Semantic Tableaux are sound and complete.

- The rules are non-deterministic. If the right choice is not made at each stage there is a possibility to go on forever, continually doing something new without reaching a closed tableau. This is possible when applying the first rule infinitely. To avoid this problem we need to restrict reusing formulae.

2.1.2.4.2 Resolution Method

Just as we did with tableau proofs, we have two additional resolution expansion rules which are the following:

$$\begin{array}{cc} \dfrac{\gamma}{\gamma(t)} & \dfrac{\delta}{\delta(c)} \\ \text{for any closed} & \text{for a new} \\ \text{term t of } L^{par} & \text{parameter c} \end{array}$$

The same explanations and restrictions apply to both tableau and resolution methods.

A resolution expansion is satisfiable if there is a model in which every disjunction in it is true. This means that each line should be satisfiable, and for each line to be satisfiable at least one of its sentences should be satisfiable. Whenever a Resolution Expansion rule is applied to a satisfiable resolution expansion, the result is another satisfiable resolution expansion. A finite set S of sentences of L^{par} *is resolution consistent* if there is no closed resolution expansion for S (A closed resolution is a resolution containing the empty clause). The collection of all resolution consistent sets is a first order consistency property.

The following points summarise the Resolution evaluation:

- Resolution is sound and complete.
- The rules are non-deterministic. It is possible to continually do something new without reaching a closed resolution. When applying the first rule infinitely.
- A disadvantage of using resolution as a proof procedure is that by transforming everything to CNF, some obvious, useful, and valuable things become hidden such as the existentially quantified variables.
- Resolution requires the use of a unification algorithm to match two literals.
- Resolution can be used to answer questions which instantiate variables, not only yes/no questions.

2.1.2.4.3 Hilbert Systems

In addition to the axioms and rules defined for predicate logic, one axiom and one rule are added for the first order logic version of Hilbert Systems :

Axiom: $\gamma \supset \gamma(t)$ where t is any closed term of L^{par}

Universal Generalisation Rule:

$$\frac{\Phi \supset \gamma(p)}{\Phi \supset \gamma}$$

where p is a parameter not occurring in $\Phi \supset \gamma$

Hilbert Systems are sound and complete. As in propositional logic, Hilbert Systems rely heavily on heuristics in the choice of axioms and rules.

2.1.2.4.4 Natural Deduction

Just by adding the following two quantifier rules, the natural deduction explained for propositional logic becomes applicable to first order logic:

γ Elimination	δ Elimination
$\dfrac{\gamma}{\gamma(t)}$	$\dfrac{\delta}{\delta(p)}$
for any closed term t of L^{par}	for a new parameter p

Natural Deduction is sound and complete.

2.1.2.4.5 Gentzen Sequents

Gentzen Sequents is extended to a first order version by adding the following two quantifier rules:

γ Rules:

$$\frac{\Gamma, \gamma(t) \rightarrow \Delta}{\Gamma, \gamma \rightarrow \Delta} \qquad \frac{\Gamma \rightarrow \Delta, \gamma(p)}{\Gamma \rightarrow \Delta, \gamma}$$

	Semantic Tableau	Resolution System	Hilbert System	Natural Deduction	Gentzen Sequents
Sound Complete	yes	yes	yes	yes	yes
Suitable for Automation	yes	yes	no	no	yes
Transformation Required	yes	yes	no	no	no
Understanding Complexity	fair	fair	very easy	very easy	fair
Rely on Heuristics	yes	yes	yes	yes	no

Table 5: Summary of First Order Logic Theorem Provers

t is any closed term and p is a parameter that does not yet occur in the proof.
δ Rules:

$$\frac{\Gamma \rightarrow \Delta, \delta(t)}{\Gamma \rightarrow \Delta, \delta} \qquad \frac{\Gamma, \delta(p) \rightarrow \Delta}{\Gamma, \delta \rightarrow \Delta}$$

t is any closed term and p is a parameter that does not yet occur in the proof.
Gentzen Sequents are sound and complete.

2.1.2.5 Summary of Theorem Proving in Predicate Logic

The following Table 5 is a summary of the theorem provers just described. The information is based on readings from [17, 19, 16, 5, 12]

2.1.3 Computability and Decidability

Computability deals with functions that could or could not be computed by algorithms. An algorithm is a well defined set of instructions that work on some input data to produce useful output. Both the input and output data must have finite descriptions and the algorithm should run for a finite amount of time. A function is *computable* if there is an algorithm for computing it [3]. A problem is *decidable* if there is an algorithm which can always give the answer to the problem.

There is no algorithm that is always able to determine whether an arbitrary formula or sentence of first order logic is valid. That is, first order logic is not decidable. There are, however, subsets of first order logic for which all theories are decidable. If a formulae is a theorem we can show that it is valid in a finite time. However, if it is not a theorem, the procedure may loop for ever. Thus, first order logic is semi-decidable [16, pages 34–35]. The theorem provers described in section 2.1.1.4 and section 2.1.2.4 will not always terminate when attempting to prove that a particular formula is or is not valid in first order logic.

2.1.4 Four-Valued Logic

Many-valued logics were developed based on the argument that by considering facts as either true or false we are oversimplifying things. So we need our facts to be mapped not only over two values (true and false) but over many values [2]. Historically, the first many-valued system (propositional logic) was constructed by Lukasiewicz in 1920 [43] in which he introduced the notion of three-valued logic : true, false, and neutral(an intermediary value). Shortly after Lukasiewicz in 1921, Post published his many-valued system [43] where he allowed arguments and functions to take values out of a given number of n values (say 1,2,...,n) regardless of what meaning each value i could have. In the early 1940s, the first complete algebra of n-valued logic corresponding to the work of Post, was formulated [35]. After that, many-valued logic started to gain interest in many areas: logic design, switching theory, programming languages, pattern recognition, artificial intelligence and many others.
Our concern here is the decidable four-valued logic introduced by Belnap [4] and extended to a version suitable for knowledge representation by Patel-Schneider [30, 31].

2.1.4.1 Why Four-Valued Logic ?

There are several reasons behind the need to use four-valued logic:
Firstly, there is a trade-off between expressive power and computational tractability in knowledge representation formalisms [24]. When the formalism is very expressive like first order logic, then reasoning becomes time consuming and, in the case of first order logic, semi-decidable. Being semi-decidable, first order logic is not suitable for knowledge representation systems. What four-valued logic does is weaken the standard first order logic just enough to get a decidable inference process. Other

systems might solve the undecidability problem in different ways: add restrictions and heuristic steps to the inference mechanisms to make them decidable. But this will produce a system which cannot justify its answers except by giving the actions taken at each step, and it will not have an adequate semantics. Another approach is to design semantics that will fit a particular inference process. Again, this approach has problems in choosing its semantics and it usually results in a very complicated semantics which reflects the inference process rather than the meaning of a formula. Secondly, we should be able to consider cases where something is both true and false (contradiction) and deal with it without polluting the remaining knowledge [4]. We should also be able to express the fact that we know nothing about a particular knowledge (neither true nor false).

2.1.4.2 Syntax of Four-Valued Logic

The syntax of four-valued logic is essentially the same as that of classical logic:

- Primitive elements are variables, n-place function letters, and n-place relation letters.
- A *term* is a variable or a function application $f(t_1, ..., t_n)$ where f is a function of arity n, and each t_i is a term.
- An *atomic formula* is a relation application $r(t_1, ..., t_n)$ where r is a relation letter of arity n, and each t_i is a term.
- A *formula* is either:

  1. atomic formula
  2. $\neg\alpha$
  3. $\alpha \vee \beta$
  4. $\alpha \wedge \beta$
  5. $\forall x \alpha$
  6. $\exists x \alpha$

  where α and β are formulae and x is a variable.
- A *sentence* is a formula with no free variables.

	None	F	T	Both
$\neg$	None	T	F	Both

Table 6: Negation Evaluation in Four-Valued Logic

2.1.4.3 Semantics of Four-Valued Logic

The four-valued logic truth values are:

1. True: when there is existence of evidence supporting the truth of a fact.
2. False: when there is existence of evidence supporting the falsity of a fact.
3. Both: when there is existence of evidence supporting both the falsity and the truth of a fact. (contradiction).
4. None: when there are no evidence supporting either the falsity or the truth of a fact. (ignorance).

The four truth values form the set of subsets of {true, false}: {true}, {false}, {true, false} for both, and {} for none.

Table 6 through Table 8 summarise the values taken by compound sentences with the $\neg$ negation, $\wedge$ conjunction, and $\vee$ disjunction connectives [4]. The entries are based on the truth table of classical logic, applying monotonicity using the lattice in figure 1, or by induction from the following two equivalences:

- $a \wedge b = a$ iff $a \vee b = b$
- $a \wedge b = b$ iff $a \vee b = a$

Just as an assignment assigns one of the two truth values to a propositional letter, a *setup* in four-valued logic maps each propositional letter into one of the four subsets of {true, false}. Unlike standard logic, no sentence is true in all setups and no sentence is false in all setups either.

$\wedge$	None	F	T	Both
None	None	F	None	F
F	F	F	F	F
T	None	F	T	Both
Both	F	F	Both	Both

Table 7: Conjunction Evaluation in Four-Valued Logic

$\vee$	None	F	T	Both
None	None	None	T	T
F	None	F	T	Both
T	T	T	T	T
Both	T	Both	T	Both

Table 8: Disjunction in Four-Valued Logic

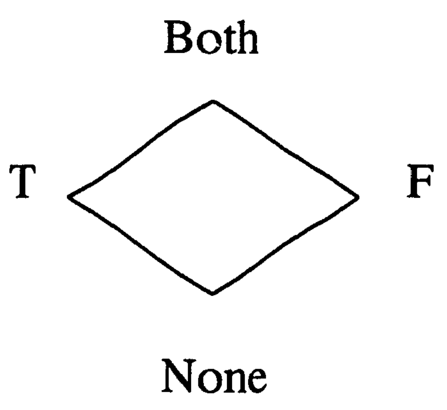


Figure 1: Approximation Lattice

Situations are the basis of the semantics. They correspond to models of the classical logic. A *situation* consists of:

- A domain, which is a non-empty set D.
- A mapping, k, which maps each function letter into a function over the domain thus giving meaning to them.
- A mapping t which maps each n-place relation letter ,r, into a function from D^n to subsets of {true,false}. So it gives meanings and values to predicate letters.

A situation s might support the truth of a predicate P on some objects $\vec{d}$ in the domain (t $\in t_s(P)(\vec{d})$) , the falsity on some objects $\vec{d}$ in the domain (f $\in t_s(P)(\vec{d})$) , it might be missing information ($\emptyset \in t_s(P)(\vec{d})$) , or even inconsistent (t and **f** $\in t_s(P)(\vec{d})$). A model in classical logic would correspond to a situation without any contradictions or missing information.

A *Variable Map* is defined exactly as in classical logic. It maps variables into some non-empty set. For instance : if v is a variable map into D, x is a variable, and d is an element of D, then v^x_d is a variable map into D. In a situation s:

$$\begin{aligned} v^*_s &= v(x) \quad \textit{if } x \textit{ is a variable}, \\ v^*_s(f(t_1, ..., tn)) &= (k_s(f))(v^*_s(t_1), ..., v^*_s(t_n)) \quad \textit{otherwise}. \end{aligned}$$

Formulae are given meaning using an *interpretation function* i(s, v, formula) (v is a variable map and s the situation) which maps each formula into subsets of $\{\mathbf{t}, \mathbf{f}\}$. For example: $\mathbf{t} \in i(s, v, \alpha \vee \beta)$ iff $\mathbf{t} \in i(s, v, \alpha)$ or $\mathbf{t} \in i(s, v, \beta)$.

For two formulae α and β, we say α *entails* β ($\alpha \rightarrow \beta$) : if for all situations and all variable maps β is true whenever α is, and α is false whenever β is.
With the definitions we already have, computing this entailment is still undecidable. The problem that causes undecidability is still present: quantification is equivalent to infinite conjunction or disjunction. For example: $Pa \vee Pb$ entails $\exists x Px$ because if $Pa \vee Pb$ is true either Pa is true or Pb is true, so Px is true for some x, however, x is not the same individual all the time. To solve this problem, the quantification equivalence is changed by evaluating formulae in a set of *compatible* sets of situations rather than in a situation.
A *compatible set of situations* is a set of situations with the same domain (D) and the same mapping of function letters to functions over the domain (k). But they differ in the truth conditions they give to predicate letters (t).
The interpretation function will then map compatible sets of situations, variable maps, and formulae into subsets of {t,f}. For example: $\mathbf{t} \in i(S, v, \forall x\alpha)$ iff for all $d \in D$ $\mathbf{t} \in i(S, v_d^s, \alpha)$ where S is a set of compatible situations, v a variable mapping, D the domain, and α a formula. Now for a set of situations to support the truth of $\exists x Px$ there has to be a single object common across all the situations, and all the situations support the truth of P for that object. We read $\exists x Px$ "there exists a known individual for which P is true " [6]. More generally, S supports the truth of $\exists x \alpha$ under the variable map v if there is some domain element ($\in D$), common across all the situations in S, which when taken as the mapping of x, is such that each situation in S supports the truth of α.
In four-valued logic, there are three different versions of *entailment*, with corresponding versions of equivalence. They are for all compatible sets of situations S, and variable maps v:

t-entailment: $\alpha \rightarrow_t \beta$ iff $\mathbf{t} \in i(S,v,\alpha)$ then $\mathbf{t} \in i(S,v,\beta)$. That is β is true whenever α is true.

f-entailment: $\alpha \rightarrow_f \beta$ iff $\mathbf{f} \in i(S,v,\beta)$ then $\mathbf{f} \in i(S,v,\alpha)$. That is α is false whenever β is false.

tf-entailment: $\alpha \rightarrow_{tf} \beta$ iff $\alpha \rightarrow_t \beta$ and $\alpha \rightarrow_f \beta$

We summarise some properties of these entailments:

- $\forall x \not{t}\ x \rightarrow_{tf} Pa$: A universal entails an instance
- $Pa \rightarrow_{tf} \exists x Px$: An instance entails an existential
- $Pa \vee Pb \not\rightarrow_{tf} \exists x Px$: A disjunction does not entail an existential
- $\forall x Px \not\rightarrow_{tf} Pa \wedge Pb$: A universal does not entail a conjunction
- $\forall x Px \rightarrow_t Pa \wedge Pb$: A universal t-entails a conjunction but
- $\forall x Px \not\rightarrow_f Pa \wedge Pb$: A universal does not f-entail a conjunction
- $Pa \vee Pb \not\rightarrow_t \exists x Px$: A disjunction does not t-entail an existential but
- $Pa \vee Pb \rightarrow_f \exists x Px$: A disjunction f-entails an existential

A formula is in *t-quantifier normal form* if it is prenexed, rectified, all the universally bound variables are skolemized, and its matrix is in CNF.

2.1.4.4 Four-Valued Logic Evaluation

We summarise the four-valued logic evaluation in the following points:

- It is decidable: there is an algorithm to compute t-entailment.
- Its syntax is the same as the classical first order logic, making it easy to understand.
- Its semantics is based on logical situations which correspond to models in the classical first order logic.
- It is weak:
  1. All its entailments are weaker than logical implication.
  2. Some valuable rules are not valid, for example modus ponens, disjunctive syllogism ($a \wedge (\neg a \vee b) \not\rightarrow b$).

- The paradoxes of implication: $A \wedge \neg A \rightarrow B$ and $A \rightarrow B \vee \neg B$ are not valid. The failure of the first means that just because we have been told that A is true and that A is false, we cannot conclude everything. Actually we might not know anything about B (none). And the failure of the second means that we cannot conclude that we know something about B just by knowing that A is true. Their absence is welcome because we guarantee that the presence of a contradiction will not contaminate the system [4].

- It is rooted in reality where we encounter contradictory facts (both) or we ignore the truth or falsity of some others (none) [4].

2.2 Knowledge Representation

In order to use knowledge in a machine, we must first choose a way of representing it, and a way of manipulating it in order to create solutions to problems related to this knowledge. Some very general methods exist for manipulating knowledge, but these methods are made so general that their strength is very limited. This is the reason why we use specific knowledge representation models which have more powerful inference mechanisms to manipulate them.

2.2.1 Logic

One way of representing facts is to use the language of logic. This representation has many advantages [22]: First, logic has a formal semantics, which gives a precise meaning for each expression. Second, logic has well defined properties for which it is possible to prove their soundness, completeness, and decidability. For powerful logics, proof theories are sound, complete, and semi-decidable. Third, logic has predefined techniques to manipulate and reason with that knowledge and to derive new knowledge from old, such as the resolution method for predicate logic. The final but not least important advantage is its expressive power. Logic can express, not only asserted knowledge, but also incomplete knowledge, or information about incompletely known situations [27]. For instance, in predicate logic disjunction could be used to express that either this pen is blue or this pen is red, which is an incomplete knowledge about the colour of the pen. As Levesque and Brachman [24] say about the expressive power of logic:"determines not so much what can be said, but what can be left unsaid".

2.2.2 Structured Representations of Knowledge

In logic, complex structures and relationships cannot be easily represented so we have other systems for representing complex structured knowledge. Those systems should have the following four properties [34]:

- Representational adequacy: be able to represent all the kinds of knowledge needed.
- Inferential adequacy: be able to manipulate the representational structures to derive new ones.
- Inferential efficiency: be able to incorporate information that will help in the inference mechanism.
- Acquisitional efficiency: be able to acquire information easily.

Two representations fulfill these objectives: Declarative representation, where knowledge is represented as a static collection of facts with a small set of procedures on how to manipulate them; and procedural representation, where knowledge is represented as procedures for using it. Each of the methods has its advantages and disadvantages and in practice most representations employ a combination of both. In declarative representations, choosing the level of representation is an important issue. For instance: if "Tania ate an ice cream" and then I am asked: "Did Tania use a spoon?" I cannot answer this question unless the fact that Tania ate an ice cream is broken down into taking a spoon, moving it towards the mouth and so on. The advantages of representing knowledge in terms of a small set of primitives is that the inference rules will be written in terms of the primitives rather than in terms of the many ways in which knowledge may appear. The disadvantages are that a lot of work is needed to convert knowledge into its primitives, and a lot of storage is also required.
A knowledge structure is a data structure in which knowledge can be stored about a particular problem domain. Two very common and useful ways of decomposing things in knowledge structures are: the ISA relationship between two objects in a hierarchy, and the ISPART relationship between an object and the parts that make

it up. Transitivity is a property of both relationships.
A *schema* gives a method on how to organise things in a particular way because of past reactions and experiences. Examples of schemas are: *frames* to describe attributes of an object, *scripts* to describe a sequence of events, and *stereotypes* to describe characteristics found in a particular group of people.
The representational schema can be categorised into two categories by considering the syntax-semantics dimension: *Syntactic systems* do not have any concern for the meaning, they are concerned with manipulating the representation. *Semantic systems* are concerned with the meaning behind each representation. Most of the available structures tend to use a combination of both.

2.2.2.1 Declarative Representations

Using declarative representations has many advantages: each fact is stored only once and could be used in many ways. It is also easy to add new facts without changing anything. Several declarative mechanisms for representing knowledge were developed. Some of them are general, others are used for specific kinds of knowledge. But they all share some common features: the way they are implemented in main memory using associative memory technique or other techniques, each has a set of inference rules which emphasises computational efficiency rather than completeness.
Some of the declarative representations are briefly described below:

- *Semantic Nets* are used to describe both events and objects. They are one of the first knowledge representation structures developed, they were proposed by Minsky in 1975 in [26]. They were originally designed to represent the meanings of English words. The two components of a semantic net are the *nodes* and the *labelled arcs*. Information, an object, an event or a concept, is represented by the nodes and relationships are represented by the labelled arcs between nodes. Semantic net is particularly good for representing binary relations. Non-binary relations are handled by using arcs with more than two endpoints. The advantages of semantic nets are that they easily represent inheritance, they are flexible so it is easy to modify them by adding, deleting or changing a node or link as necessary. However, this flexibility tends to become a disadvantage because semantic nets lack a formal structuring, so complex ones become messy and hard to understand. To overcome this problem, some techniques were developed such as the Warnier/Orr approach, and normalisation [11].

- *Frames* are used to represent complex objects from different points of view. They describe an object by placing all the information related to that object in *slots*. Each slot either describes an aspect of the object, is another frame, or has a default value. Slots could also be attached to procedures that describe them: this is known as procedural attachment.
  One way of selecting a frame that is applicable to a particular situation is: Select an initial candidate frame and instantiate it. Find values for each slot in the frame. If a value is not found then select another frame using clues for the failure or use fragments that are correct and try to match them with another frame.
  In real life, we tend to proceed in a similar fashion. When something happens we do not analyse it from scratch. We have in mind a collection of structures which are like the frames and whenever something happens, we try to find the frame that best fits the situation. That's why using frames is very common when modelling real world situations. Procedural attachment allow attributes to have more general values rather than simply data values.

- *Conceptual Dependency* describes the relationship between the components of an action. It is a theory on how to represent the meaning of natural language sentences in a way that has an easy inference mechanism, and that is independent of the language in which the sentences are stated. A conceptual dependency consists of:
  - A group of primitive actions, such as ATRANS for a transfer of things.
  - A group of conceptual categories included in a sentence, such as ACT for an action, and PP for an object.
  - A list of semantic relations among the concepts and dependencies so it is clear how things could be related in a sentence. For instance there is a rule which describes the relationship between two PPs one of which belongs to the set defined by the other.

The advantage of using Conceptual Dependency is that we have less inference rules because many inferences are included in the representation, and the structure contains holes to be filled with the correct values.

The disadvantages of using Conceptual Dependency are that: a low-level decomposition is needed, and there is a need to represent more than just events.

- *Scripts* were first proposed by Schank and Abelson in 1977 [1]. They are used to represent a common sequence of events. For example, what happens when one goes to a restaurant. A script is a description of a class of events in terms of contexts, participants, and sub-events. It consists of a set of slots. Each slot has a default value and information about what it could take as values. Scripts are useful because in the real world there is a pattern behind each event. However, for a system to be able to reason about a variety of things, many scripts are needed. This makes scripts not a desirable representation for general cases, but they may be suitable for very specific events.

- *Demons* are procedures which activate an associated process when some conditions become true. For example, we might have a procedure to compute the weight whenever needed. This procedure is invoked when the volume and density are available and the weight is needed. Demons are difficult to reason about because they are not invoked explicitly. However, they allow the representation of knowledge at a global level. Demons are used when coding active databases and knowledge systems.

- A *Decision Table* is a collection in any tabular form used to describe a rule by providing all the possible conditions and the corresponding actions to be taken. The advantages of using decision tables is that they are easily understood and they are good to capture and store a large amount of data.

- *Decision Trees* are like hierarchical semantic nets where nodes represent goals or decision points and the links are alternative decisions. They are examined from left to right. Decision trees are especially good to represent cause and effect and they are easy to validate.

2.2.2.2 Procedural Representation

So far we have discussed declarative representation where knowledge is represented as a set of facts. Procedural representation is another way of representing knowledge as a set of procedures for using knowledge. Procedural representation of a piece of information is a plan on how to use that information. In other words, knowledge is represented by rules about a particular problem. This approach was taken by Winograd [41, 42] in his SHRDLU system which takes English sentences as input and produces a set of procedures for doing what the statement requested. Each procedure has a set goals and there is a mechanism to satisfy this goal. In practice, very few systems use procedural knowledge as the only way to represent knowledge. Rather, a combination of declarative and procedural knowledge is used.

2.3 Mantra

2.3.1 Definition

Mantra is a general purpose shell for hybrid knowledge representation and hybrid inferences. It supports three different knowledge representation formalisms: logic, frames, and semantic nets. Knowledge is represented in one of the formalisms and then inferred from one or a combination of the formalisms.
Mantra was developed in Karlsruhe, Germany, in 1991 by G. Bittencourt [6, 8, 7] and implemented in Common Lisp. The motivations behind its development were:

- To provide a combination of knowledge representation formalisms since each formalism is usually suitable for a particular kind of knowledge. The user can then decide which representation is convenient for each piece of knowledge, and is not limited to only one formalism.

- To allow the user to control the interactions between the different knowledge representation formalisms.

- To have decidable inference procedures based on a clear semantics.

- To provide a shell for the development of expert systems.

The interface of Mantra is very simple. It provides two primitives: *Tell*, to store knowledge into one of the three formalisms, and *Ask*, to query knowledge from one or a combination of formalisms. For example:

- Tell(KBS-name,to-logic(formula))

- Ask(KBS-name,from-logic(logic-question))

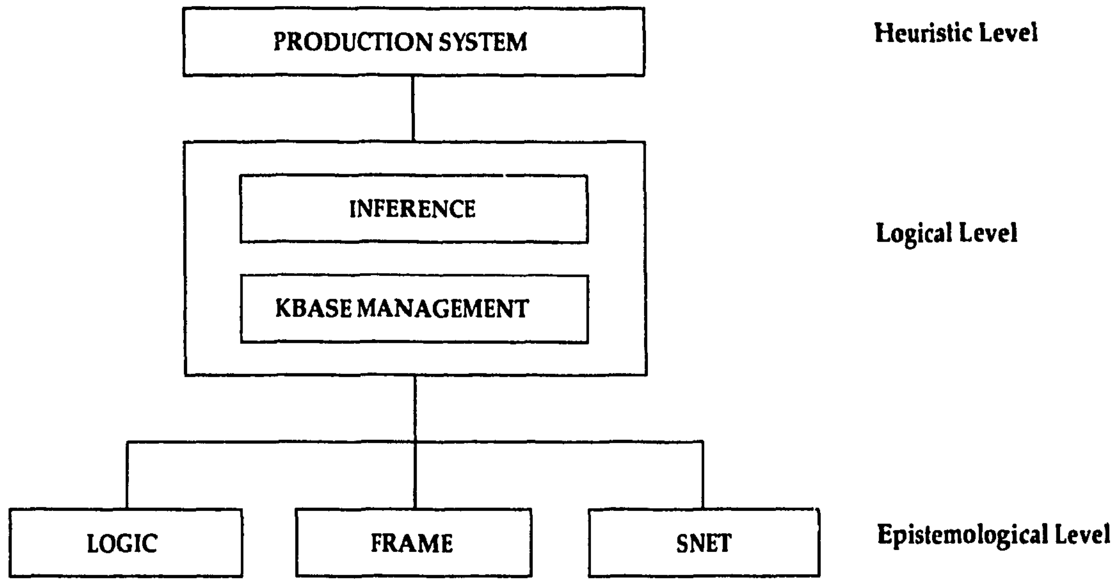


Figure 2: Mantra Architecture

- Ask(KBS-name,from-logic-frame(logic-frame-question))

2.3.2 Architecture

The architecture of Mantra consists of three levels: Epistemological level, Logical level, and Heuristic level, as shown in Figure 2.

- The **Epistemological Level** provides three formalisms to represent facts about the world. It consists of three modules each based on one formalism:
  - The **Assertional Module** is used to represent asserted facts about a particular domain. Its formalism is first order logic language, based on the decidable four-valued logic. Example:

    Tell(k,to-logic(robin(tweety)))
    Tell(k,to-logic(size(tweety,small)))
    Ask(k,from-logic(!Ex(robin(x) & size(x,small))))
    Answer: yes x=tweety
    Ask(k,from-logic(!Ex(robin(x) & size(x,big))))
    Answer: no

- The **Frame Module** is used to represent concepts, which are categories of objects, and relations, which are the properties of objects. The syntax of this module is very rich: it pro[illegible] rich set of primitives such as negation of concepts and relations, and special symbols and tests. The semantics are also based on the four-valued logic. Example:

  Tell(k,to-frame(mammal :c=animal & !V blood:[warm] & !V reproduction:[viviparous]))
  describes the concept of mammal as all animals having a warm blood and a viviparous reproduction system.
  Tell(k,to-frame(elephant :c=mammal & !V food:[plant] & !E organ:[trunk]))
  describes the concept of elephant as a mammal whose food is only plant and one of its organs is trunk.
  Tell(k,to-frame(bird :c=animal & !V blood:[warm] & !V reproduction:[oviparous]))
  describes the concept of bird as an animal whose blood is warm and whose reproduction system is oviparous.
  Tell(k,to-frame(robin :c=bird & !V size:[small] & !E organ:[wing]))
  describes the concept of robin as a bird whose size is small and one of its organs is wing.
  Ask(k,from-frame(mammal > robin))
  asks whether the concept of mammal subsumes the concept of robin (Concept C_1 subsumes concept C_2 if all instances of C_2 are instances of C_1)
  Answer: no

  The four Tell statements describe the concepts on Figure 3.

- The **Semantic Net Module** allows the definition of classes of objects and hierarchies. Hierarchies are the links between classes, with default and exception links provided. The syntax provides the necessary primitives and operators to describe them. Example:

  Tell(k,to-snet(circus-elephant :k=normal-elephant + flying elephant))

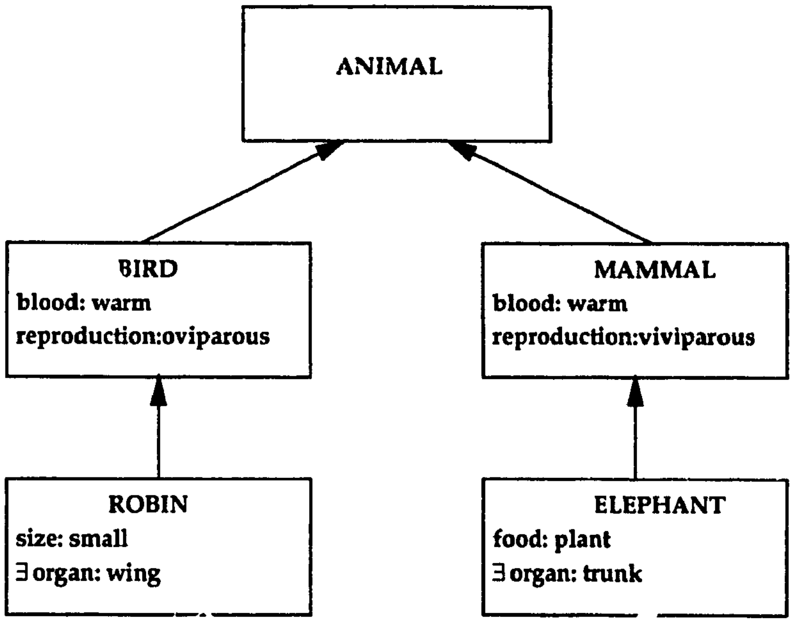


Figure 3: Example of Concepts

takes the classes of normal-elephant and flying elephant and creates a new class circus-elephant more specific.

Tell(k,to-snet(colour :h=elephant $\rightarrow$ gray ++ royal-elephant $\not\rightarrow$ gray))

constructs the colour hierarchy: elephant has a positive link with gray, and royal-elephant has a negative link with gray

Tell(k,to-snet(circus :h=african-elephant $\rightarrow$ elephant ++ royal-elephant $\rightarrow$ elephant ++ circus-elephant $\rightarrow$ royal-elephant))

constructs the circus hierarchy out of three hierarchies: african-elephant and circus-elephant have a positive link with elephant and circus-elephant has a positive link with royal-elephant.

Ask(k,from-snet(colour ++ circus(circus-elephant $\not\rightarrow$ gray)))

asks whether in the hierarchies colour and circus, circus-elephant is not a sub-class of ıy.

Answer: yes

This example describes the hierarchies shown on Figure 4

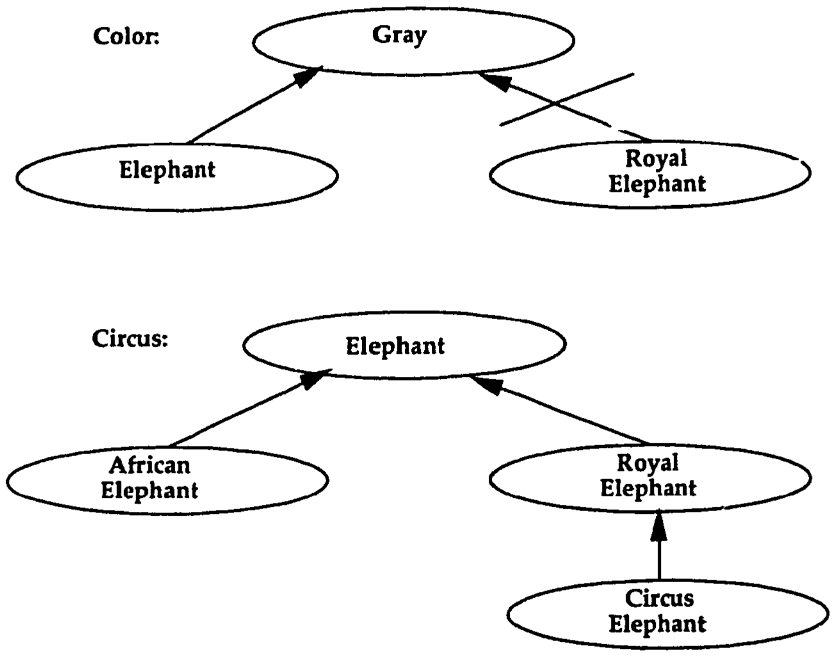


Figure 4: Example of Two Hierarchies

The three modules use the same semantics basis, which facilitates the definition of their interactions. Through these interactions, the functionalities of one module increase the inference power of another module. An examples on the interactions based on the facts stored in the examples above:

> Ask(k,from-logic-frame(!Ex(size(x,small) & animal(x))))
> Answer: yes x=tweety.

The division of the knowledge language into formalisms has two advantages: (1) each formalism solves its computability problems independently; and (2) it is very easy to integrate new formalisms to the language without changing the others.

- At the **Logical Level**, the knowledge bases and the functions that manipulate them are defined. Each knowledge base is a set of partitions, one for each formalism, with the two primitives; Tell and Ask. The inference defined on this level is both for a single or a combination of formalisms: logic, frame, semantic nets, logic and frames, logic and semantic nets, or frame and semantic nets.

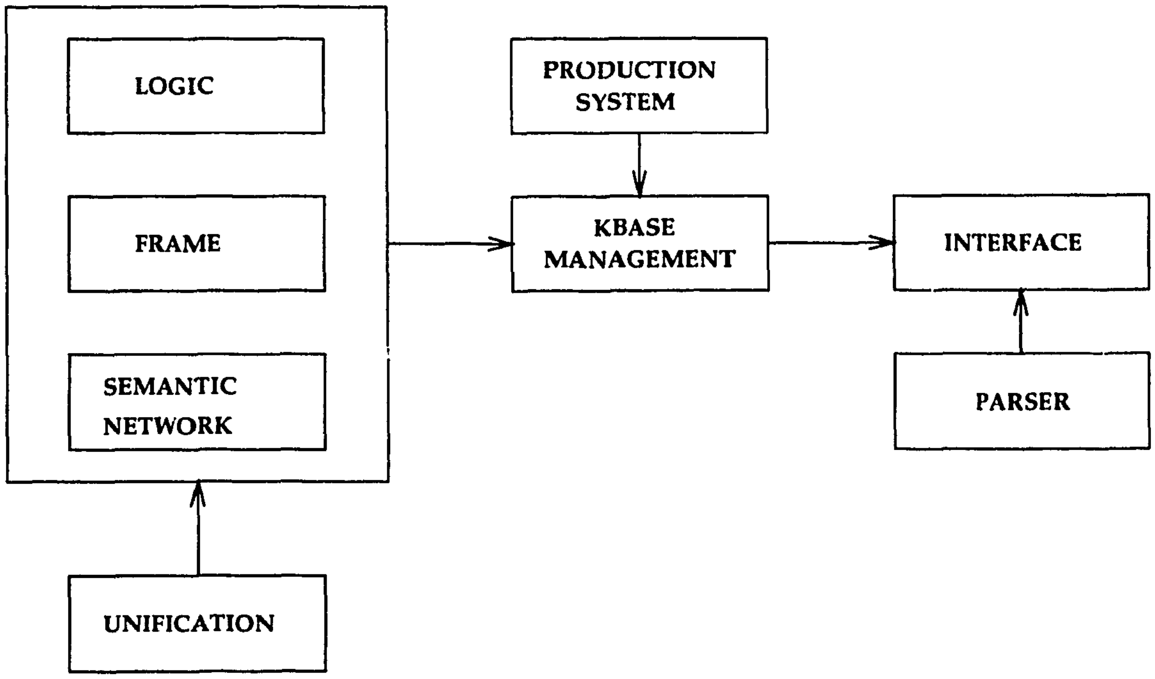


Figure 5: Mantra Modules

- While the epistemological level is for declarative knowledge, the **Heuristic Level** is for procedural knowledge. It defines the primitives that allow the definition of production systems for the automatic manipulation of knowledge bases. Each production rule consists of: a rule identifier, a list of variables, a condition, and an action to be taken if the condition is satisfied.

2.3.3 Modules

The modules of Mantra are shown on Figure 5. They are

Interface contains the functions that define and control the menus presented to the user.

Parser parses the input and validates it.

Knowledge Base Management holds the definitions of the knowledge bases and the functions that control the interactions between the three epistemological formalisms.

Production System implements the primitives of the heuristic level to define production rules and manipulate them.

Unification defines the functions and structures used in the unification algorithm.

Logic Module contains the functions used to manipulate logical formula, including the t-entailment algorithm.

Frame Module contains the functions used in the manipulation of frames: definition of new frames, and subsumption tests.

Semantic Network Module contains the functions that manipulate semantic nets and compute inheritance.

Chapter 3

Transforming a Formula into CNF

A first order logic formula can be transformed into various normal forms while preserving its semantics. Thus, the various transformations applied on the formula preserve its truth value and no information is lost or modified. Conjunctive Normal Form, known as CNF, is the most common form. It is a standard way of representing a formula to be processed by various algorithms especially algorithms based on resolution. In our case, the inference mechanism requires that the asserted facts be in CNF, and the questions to be in t-quantifier normal form, which is a variation of CNF. The knowledge base stores the facts as $F = D_1 \wedge D_2 \wedge D_3 ... D_n$, that is, a conjunction of disjunctions. Each disjunction is a collection of terms. Terms are the input to the unification module.

3.1 Steps in Transforming a Formula into CNF

The steps necessary to transform a formula into CNF are the following [37]:

1. Rectify the formula:
   - No variable should occur both bound and free. For variables occurring bound and free, the free occurrences are renamed.
   - All quantifiers should refer to different variables. Whenever a variable is bound twice, one of its occurrences is renamed and all its occurrences within that scope are also renamed.

2. Prenex the formula: All the quantifiers are moved to the beginning of the formula. In a prenexed formula, no quantifier occurs within a negation, a conjunction, or a disjunction. The part of the formula without quantifiers is called the *matrix*. For instance, in $\forall x \exists y(Px \wedge Qy)$, the matrix is $Px \wedge Qy$.

3. Skolemizing a formula: For each variable bound by an existential quantifier, all occurrences are substituted by skolem functions. A *skolem function* is a new function symbol that does not yet occur in the formula, and takes as its arguments all the variables in the universal scope of the variable being replaced. After skolemizing the formula, the existential quantifiers can be removed.

4. CNF: The matrix of the formula is transformed into conjunctions of disjunctions by applying the following rules:

Replace $\neg\neg F$ by F	(double negation)
Replace $\neg(F \wedge G)$ by $\neg F \vee \neg G$	(deMorgan's law)
Replace $\neg(F \vee G)$ by $\neg F \wedge \neg G$	(deMorgan's law)
Replace $T \vee (F \wedge G)$ by $(T \vee F) \wedge (T \vee G)$	(distributivity)
Replace $(F \wedge G) \vee T$ by $(T \vee F) \wedge (T \vee G)$	(distributivity)

   Every variable is understood as universally bounded. Hence universal quantifiers are eliminated.

The following is an example showing all the steps involved in the transformation of

$$F = (\neg\exists x(P(x) \vee \forall y Q(x, f(y))) \vee \forall y P(g(x,y), a))$$

into its CNF form:

1. $F = (\neg\exists x(P(x) \vee \forall y Q(x, f(y))) \vee \forall w P(g(x,w), a))$
   The variable y is bound twice, it is renamed to w in the second disjunct. None of the variables are bound and free at the same time. The formula is in rectified form.

2. $F = \forall x \exists y \forall w((\neg P(x) \wedge \neg Q(x, f(y))) \vee P(g(x,w), a))$
   All the quantifiers are moved to the beginning of the formula. The formula is in prenex form.

3. $F = \forall x \forall w((\neg P(x) \wedge \neg Q(x, f(h(x)))) \vee P(g(x, w), a))$
   A new function symbol h is introduced, $h(x)$ is substituted for y because y depends on x. The existential quantifiers are removed. The formula is in skolemized form.

4. $((\neg P(x) \vee P(g(x, w), a)) \quad \wedge \quad (\neg Q(x, f(h(x)))) \vee P(g(x, w), a))$
   The matrix of F is transformed into CNF by applying distributivity, and the universal quantifiers are eliminated.

3.2 CNF Implementation

3.2.1 Object Model

The Object Model in figure 11 describes the classes used in the implementation of the transformation of a formula into CNF form. There are nine classes namely: LIST, ITERATOR, FORMULA, CNF FORMULA, TRANSFORMER, EXISTENTIAL TRANSFORMER, UNIVERSAL TRANSFORMER, TERM, and DISJUNCTION. The associations between classes are simple: the TRANSFORMER uses the LIST in its computations (list of strings, list of terms, list of disjunctions, list of formulae) the LIST is traversed by LIST ITERATOR, the FORMULA has many transformers, the TRANSFORMER is a base class for EXISTENTIAL TRANSFORMER and UNIVERSAL TRANSFORMER, and the CNF FORMULA is a FORMULA which consists of DISJUNCTIONS each having at least one TERM.

The iterator pattern [20, pages 257-273] is used to access the elements of the LIST sequentially, regardless of the internal representation of the LIST. This pattern uses the LIST class and the LIST ITERATOR class. The LIST ITERATOR defines an interface to access and traverse elements of the LIST. While the LIST defines an interface for creating an iterator object. The main advantages of using this pattern are: the list representation can be changed without affecting the iterator since the list does not need to expose its internal structure to the iterator; different list traversals can be defined and used on the same list depending on the need of the transformer.

The template method [20, pages 325-331] is another pattern, used in the relation between the TRANSFORMER and its two sub-classes. It allows us to define the skeleton of the CNF transformation in the TRANSFORMER class and defer some sub-steps to the

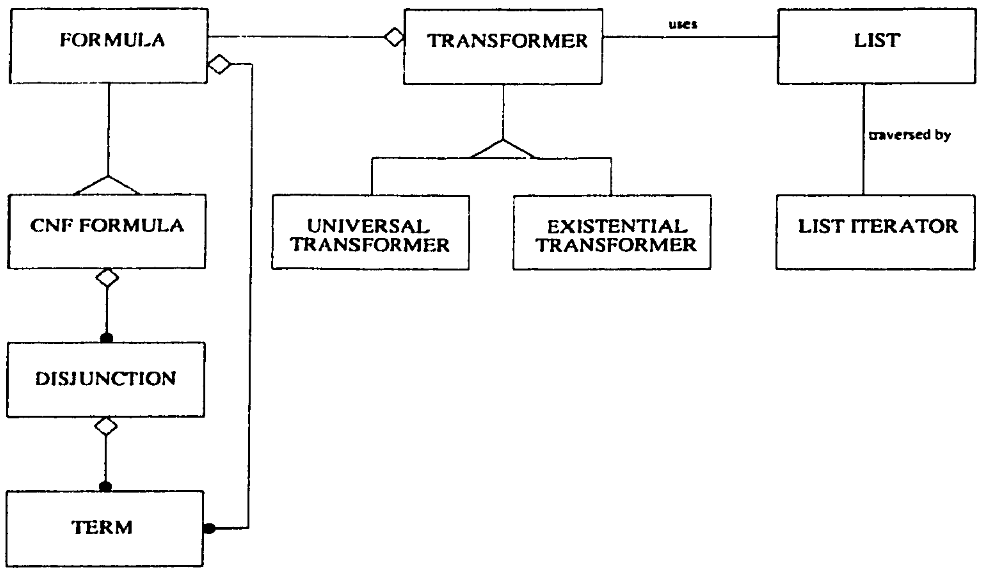


Figure 6: Object Model

two sub-classes: UNIVERSAL TRANSFORMER and EXISTENTIAL TRANSFORMER. The UNIVERSAL TRANSFORMER defines the steps for the facts transformation, while the EXISTENTIAL TRANSFORMER defines the steps for the query transformation. Each of the sub-classes has a different implementation for some of the steps. For example, in the scope computation, UNIVERSAL TRANSFORMER finds the universal scope of variables while the EXISTENTIAL TRANSFORMER finds their existential scope. Each sub-class has its own version of those sub-steps, and the invariant sub-steps, common to both, are defined in the TRANSFORMER class. The main advantages of using this pattern are: certain steps of the transformation are defined at the sub-class level without changing the transformation structure —it is left to sub-classes to implement the behaviour that can vary; the common steps which are invariant are localised under the base class to avoid code duplication.
Figure 11 shows the object model without any details. Figure 9 through Figure 15 show details of each class in the object model with its attributes and methods.

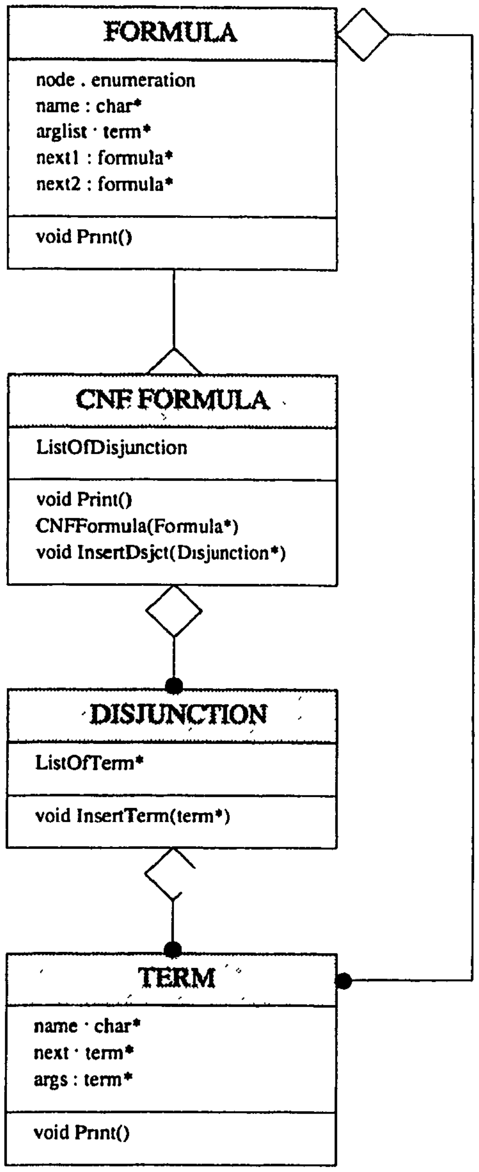


Figure 7: Object Model Details

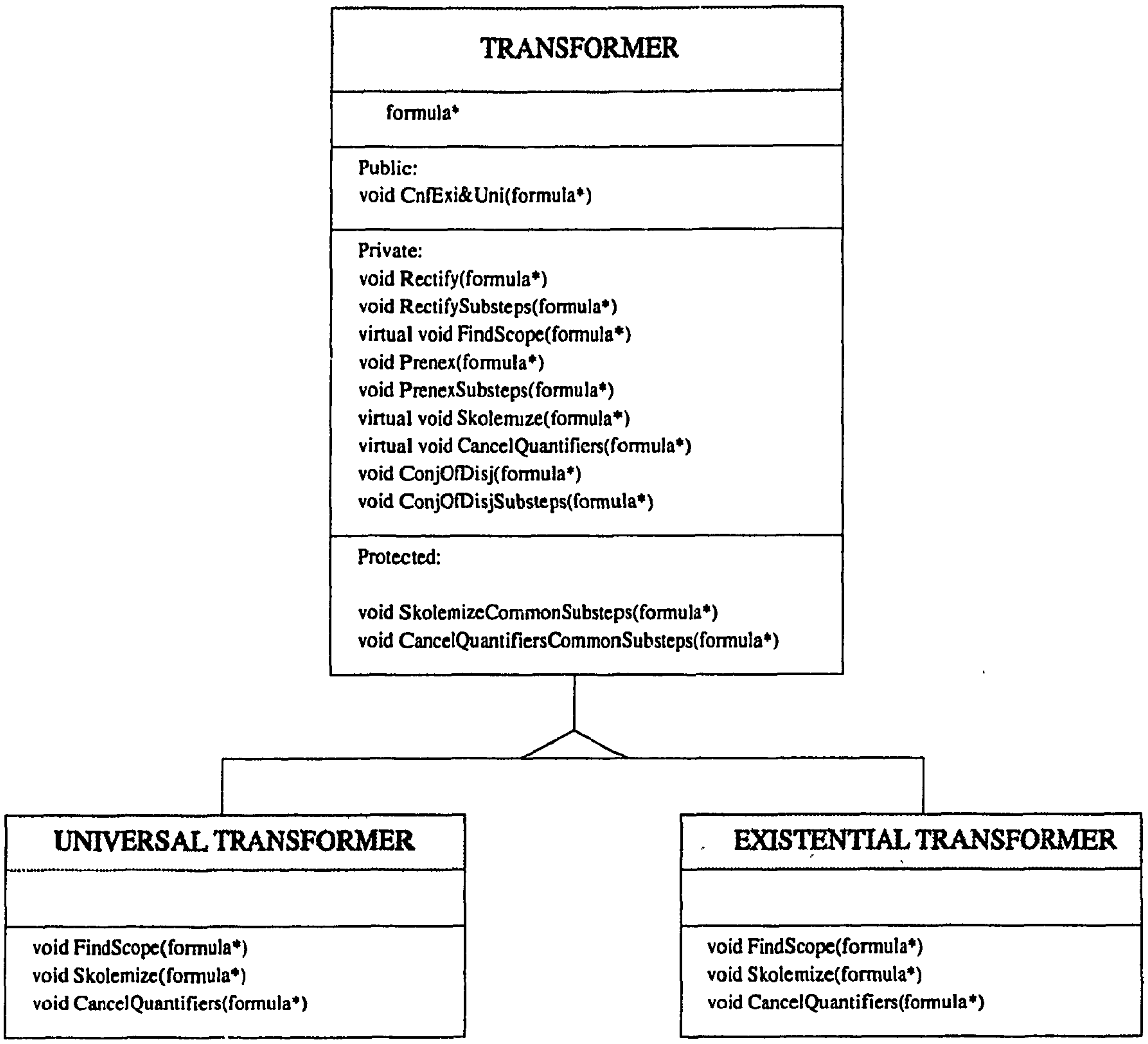

TRANSFORMER
formula*
Public:
void CnfExi&Uni(formula*)
Private:
void Rectify(formula*)
void RectifySubsteps(formula*)
virtual void FindScope(formula*)
void Prenex(formula*)
void PrenexSubsteps(formula*)
virtual void Skolemize(formula*)
virtual void CancelQuantifiers(formula*)
void ConjOfDisj(formula*)
void ConjOfDisjSubsteps(formula*)
Protected:
void SkolemizeCommonSubsteps(formula*)
void CancelQuantifiersCommonSubsteps(formula*)
UNIVERSAL TRANSFORMER
void FindScope(formula*)
void Skolemize(formula*)
void CancelQuantifiers(formula*)
EXISTENTIAL TRANSFORMER
void FindScope(formula*)
void Skolemize(formula*)
void CancelQuantifiers(formula*)

Figure 8: Object Model Details

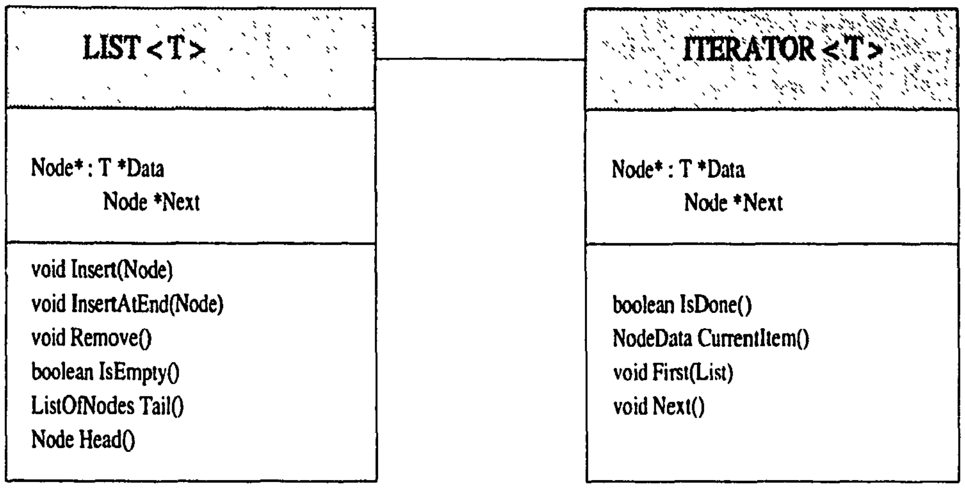


Figure 9: Object Model Details

3.2.2 Data Structures

The major data structures used are briefly described below.

1. List Node is a structure:

   - Data: any type depending on the instantiation of the list, since a list is a template.
   - Next: a list node

2. Formula is a structure:

   - Node: enumeration { EX, ALL, NOT, AND, OR, PRED }
     It determines the type of the formula: EX for $\exists$, ALL for $\forall$, NOT for $\neg$, AND for $\wedge$, OR for $\vee$, and PRED for a predicate (relation symbol).
   - Name: a string
     If the node is EX or ALL, name is the name of the variable bound by $\exists$ or $\forall$. If the node is NOT, name could be either null if a formula is negated or name could hold the name of the negated variable. For AND and OR, name is null. For PRED, name is the predicate symbol.
   - Argument list: a pointer to a term. It is only used if the node is a PRED, it's the argument list of the predicate.

- Next1: a pointer to a formula. For the unary operators (EX, ALL, NOT), next1 is the formula following. For the binary operators (AND, OR), next1 is the left-hand branch.
- Next2: a pointer to a formula. For the unary operators (EX, ALL, NOT), next2 is null. For the binary operators (AND, OR), next2 is the right-hand branch.

3. CNF formula is a list of Disjunctions.
4. Disjunction is a list of Terms.
5. Term is a structure:
   - Name: a string. It is the term name.
   - Next: a pointer to a term. Next points to the subsequent terms if there are any.
   - Argument list: a pointer to a term. It is null for a constant or variable term and holds the arguments of a composite term.

3.2.3 Data Dictionary

- **List Class**
  - **Description**
    It is a container class used to store elements of any type. It could be used either with the LIFO strategy or the FIFO strategy depending on the need.
  - **Methods**
    1. Insert: inserts an element at the head of the list.
    2. InsertAtEnd: inserts an element at the end of the list.
    3. Remove: removes the element at the head of the list.
    4. IsEmpty: checks whether the list is empty or not.
    5. Tail: returns the tail of the list.
    6. Head: returns the first element in the list.
- **Iterator Class**

  - **Description**

    It is used to traverse the list sequentially regardless of its internal representation.

  - **Methods**

    1. IsDone: checks whether the end of the list has been reached or not.
    2. CurrentItem: always returns the data in the node it is currently on during the traversal.
    3. First: initialises the iterator to point to the first element in the list.
    4. Next: moves to the next element in the list.

- **Formula Class**

  - **Description**

    Defines a logical formula, as used by Mantra to represent asserted facts.

3.2.3.1 Methods

1. Print: prints the formula.

- **CNF Formula Class**

  - **Description**

    Is an instance of the formula in a Conjunctive Normal Form. It is represented as a list of disjunctions since it is a conjunction of disjunctions.

  - **Methods**

    1. Print: overloads the print version of formula base class. It prints the CNF formula as a list of disjunctions. One disjunction on each line.
    2. CNFFormula: constructs a CNF formula as a list of disjunctions given a formula containing only conjunctions of disjunctions —a formula transformed into CNF form but still implemented as a tree rather than a list of disjunctions.
    3. InsertDisjunction: inserts a disjunction into the CNF formula.

- **Disjunction Class**

- **Description**

  A disjunction is a list of terms.

- **Methods**

  1. InsertItem: inserts a term into the disjunction

- **Term Class**

  - **Description**

    Terms are the leafs of the formula. They also represent the atomic formulae that constitute each disjunction.

  - **Methods**

    1. Print: prints a term.

- **Transformer Class**

  - **Description**

    The transformer defines each step for transforming a formula into its Conjunctive Normal Form. It is also the base class for two sub-classes which override some of the steps of the transformation.

  - **Methods**

    Some of the transformer methods are private being only used by the public method CnfExiUni. Others are protected because they are used by the two sub-classes.

    1. Rectify: produces a rectified formula.
    2. RectifySubsteps: sub-methods to perform each task in rectifying a formula.
    3. FindScope: is a virtual function overridden by each of the sub-classes depending on whether the universal or the existential scope needs to be found.
    4. Prenex: produces a prenexed formula.
    5. PrenexSubsteps: sub-methods to perform each task in prenexing a formula.
    6. Skolemize: is a virtual function implemented by each of the sub-classes depending on whether skolemization is on existential or universal quantifiers.

7. CancelQuantifiers: is a virtual function implemented by each of the subclasses depending on the type of quantifiers being removed from the formula.
8. ConjOfDisj: is the last stage in normalising a formula. It transforms its matrix into a conjunction of disjunctions.
9. ConjOfDisjSubsteps: sub-methods used in transforming the matrix into conjunctions of disjunctions. Each of the sub-steps apply a modification on the matrix to finally reach a CNF.
10. SkolemizeCommonSubsteps, and CancelQuantifiersCommonSubsteps : protected methods used by the methods in the sub-classes. They are implemented in the same way regardless of the sub-class to which the transformer belongs and they are used by each of the sub-classes' methods. CancelQuantifiersCommonSubsteps is the same since it removes all the quantifiers at the beginning of the formulae regardless of whether they are existential or universal. In SkolemizeCommonSubsteps, sub-steps common to both transformers in the skolemization process are defined such as the construction of the skolem function once the scope is found.
11. CnfExiUni: Is the high level method that calls all the others in a certain order. Depending on whether this method is called for an existential or a universal transformer, the appropriate virtual methods are executed.

- **Universal Transformer Class**

  - **Description**

    This is a Transformer sub-class. It inherits all its methods and overrides the virtual ones. This transformer deals with the steps that keep the universal quantifiers in the formula, producing a CNF formula.

  - **Methods**

    1. FindScope: finds the universal scope for each term. In skolemization, the variables in the scope are the arguments of the skolem function.
    2. Skolemize: replaces all the existentially bound variables by their skolem functions.

3. CancelQuantifiers: removes the existential quantifiers from the formula.

- **Existential Transformer Class**
  - **Description**
    This is a Transformer sub-class. It inherits all its methods and overrides the virtual ones. This transformer deals with the steps that keep the existential quantifiers in the formula, producing a formula in t-quantifier normal form.
  - **Methods**
    1. FindScope: finds the existential scope for each term. In skolemization, the variables in the scope are the arguments of the skolem function.
    2. Skolemize: replaces all the universally bound variables by their skolem functions.
    3. CancelQuantifiers: Removes the universal quantifiers from the formula.

Chapter 4

Unification Module

Given two terms in first order logic t_1 and t_2, a *unifier* is a substitution θ such that $t_1\theta=t_2\theta$. Unification is the process of finding the most general unifier (*mgu*).
The earliest reference for unification of general terms goes back to 1920 when E. Post mentioned in his diary and notes, partially published in [14], a hint of the concept of a unification algorithm that computes a most general representation as opposite to all possible instantiations. The first published unification algorithm was given in J. Herbrand's thesis in 1930 [23]. In his algorithm, Herbrand described three properties related to the validity of a formula. In 1960, D. Prawitz [32] worked on a logic which did not contain any function symbols, and for which he computed a most general representative of all possible instantiations. In 1963, M. Davis [13] published a proof procedure which was implemented on an IBM 7090 at Bell Telephone Laboratories in November 1962, and which used a unification algorithm. It was the first fully implemented unification algorithm. It was not until 1965 that J.A Robinson [36] published a formal unification algorithm for first order terms, and a proof that this algorithm computes a most general unifier. Robinson's algorithm is simple and applicable to any finite nonempty set A of terms. The process always terminates either with failure when the set is not unifiable or with success when it returns the most general unifier. This algorithm requires time exponential in the size of the terms.
Robinson introduced unification to implement resolution. Since then, it became universal in automated theorem proving. Due to its pattern matching nature, unification is applicable in a wide variety of areas in computer science [18, 38, 39] such as deductive databases, information retrieval, type checking, natural language processing, and logic programming. Unification is at the heart of these applications. Thus, its

performance is crucial to the overall efficiency. Some algorithms having almost linear time complexity were published for example [19]: M.S. Paterson and M.N. Wegman in 1978, Kapur, Krishnamoorthy, and Narendran in 1982, G. Huet in 1976, L.D. Baxter in 1973, Martelli and Montanari in 1982. The Martelli and Montanari unification algorithm and its implementation are described in detail below.

4.1 Martelli and Montanari Unification Algorithm

The algorithm of A. Martelli and U. Montanari [25] is based on a technique that was first described by Herbrand. It operates on a system of term equations, and it transforms this system until a solved form is reached. In their description of the unification solution, Martelli and Montanari start with a nondeterministic algorithm. Then they give another algorithm which works on a system of multi-equations but is also nondeterministic. Then with few changes, the algorithm is improved and made deterministic. Later, an enhancement is suggested to improve the efficiency of the algorithm. Finally, a comparison with other algorithms is made.

4.1.1 Description of the algorithm

4.1.1.1 A first nondeterministic algorithm

Before describing the actual algorithm, few definitions are necessary:

- Term reduction and variable elimination are transformations which preserve the sets of all unifiers. Let $f(t_1, t_2, ..., t_n) = f(w_1, w_2, ..., w_n)$ be an equation. *Term reduction* is a process that constructs an equivalent set of equations:

$$\begin{array}{rcl} t_1 & = & w_1 \\ t_2 & = & w_2 \\ & \cdot & \\ & \cdot & \\ t_n & = & w_n \end{array}$$

  Let $x = t$ be an equation, where x is a variable and t is a constant term. *Variable elimination* is the process of applying the substitution of t for x to a set of equations.

- A set of equations is in *solved form* if:
  - the equations are $x_n = t_n$
  - every variable in the left of an equation occurs only there.

  A solved set of equations has a unifier $V = \{[x_1/t_1], [x_2/t_2], ..., [x_n/t_n]\}$.

The first algorithm in figure 10 shows how a set of equations is transformed into an equivalent set of equations in solved form.

```
Any equation of the form t=x is rewritten as x=t
Any equation of the form x=x is eliminated
Apply term reduction to equations of the form
   $f(t_1...t_n) = f(w_1...w_n)$
Apply variable elimination to equations of the form x=t
   if x does not occur in t.
Failure occurs if there is an equation of the form
   $f(t_1...t_n) = g(w_1...w_n)$ where $f \neq g$
   or an equation x=t where x occurs in the term t
```

Figure 10: A First Nondeterministic Algorithm

This algorithm always terminates either with failure if there is no unifier or with success when the set has been transformed into a set in solved form. However, it is not deterministic because there is no fixed order in applying the steps, or selecting the equations.

4.1.1.2 A second nondeterministic algorithm

A *multi-equation* is a pair < S,M >, written as S=M where S is a non-empty set of variables and M is a multi-set of non-variable terms.
The *unifier of a multi-equation* is the substitution θ such that $S\theta = M\theta$.
A set of equations SE *corresponds* to a multi-equation S=M iff:

- All terms of SE belong to $S \cup M$
- For every t_1 and $t_2 \in S \cup M$ we have $t_1 =^*_{SE} t_2$, where $=^*_{SE}$ is the reflexive, symmetric, and transitive close of $=_{SE}$, where $t_1 =_{SE} t_2$ iff the equation $t_1 = t_2$ belongs to SE.

So the multi-equation $\{x_1...x_n\} = (t_1...t_m)$ is a way of grouping many equations $x_i = x_j$ and $x_i = t_k$ and $t_j = t_k$.

The *common part* of a multi-set of terms (M) is a term constructed by superimposing all terms, and taking the part which is common to all of them starting from the root. For instance:

$$f(x_1, g(a, f(x_5, b))),\ f(h(c), g(x_2, f(b, x_6))),\ and\ f(h(x_4), g(x_6, x_3))) \qquad (1)$$

have common part : $f(x_1, g(x_2, x_3))$ as shown on figure 11.

A variable term is more general than a function term, and more general than a constant term. Whenever we have two functions, they should have the same root symbol and the same number of arguments. Otherwise there is no common part. Whenever we have two different constant terms or a constant and a function then again there is no common part. The cases, where no common part is computed, are called a *clash*. For example, in computing the common part of $\{x_1\}$ = (g(x_2,x_3),g(x_4)) we get a clash.

The *frontier* of a multi-set of terms (M) is a set of multi-equations where each multi-equation is constructed by taking one leaf of the common part and all sub-terms corresponding to that leaf. For example, the frontier of the multi-set given in (1) is:

$$\{\{x_1\} = (h(c), h(x_4)),\ \{x_2, x_6\} = (a),\ \{x_3\} = (f(x_5, b), f(b, x_6))\}$$

Figure 11 shows the correspondence of each leaf (x_1, x_2, x_3) in the common part to a set of sub-terms.

The process of *multi-equation reduction* of $S = M$ applied to a set Z of multi-equations containing $S = M$ yields the set $Z' = (Z - \{S = M\} \cup \{S = common\ part\ of\ M\}$ $\cup \{frontier\ of\ M\}$.

Two multi-equations $S_1 = M_1$ and $S_2 = M_2$ where $S_1 \cap S_2 \neq \emptyset$ are *merged* to produce $S_1 \cup S_2 = M_1 \cup M_2$. A set Z of multi-equations is *compact* if $S_1 \cap S_2 = \emptyset$ for all multi-equations $S_1 = M_1$ and $S_2 = M_2$ in Z. A set Z may be *compactified* by merging multi-equations that violate $S_1 \cap S_2 = \emptyset$.

A multi-equation S=M has a *cycle* if some variable x in S occurs in some term t in M. For example, $\{x_1\} = f(x_1, x_2)$ has a cycle.

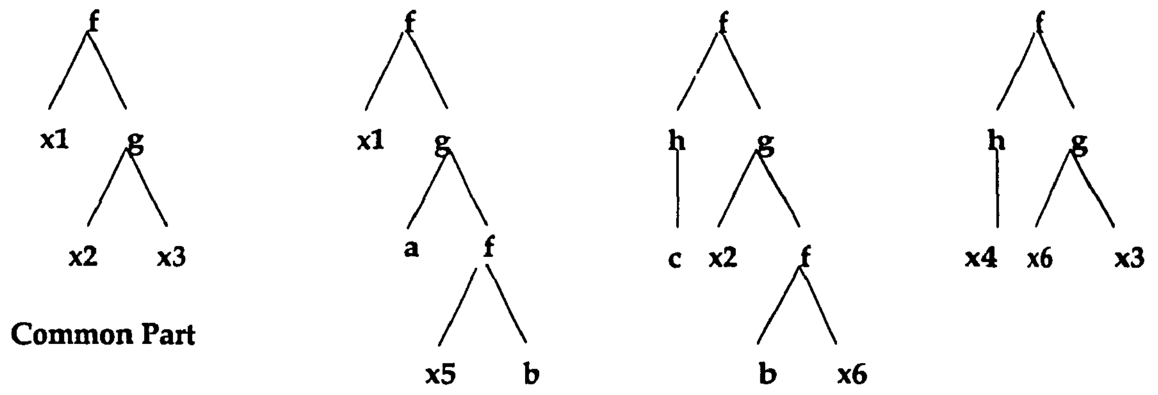


Figure 11: Common Part and The Frontier Computation

The second algorithm in figure 12 solves a set Z of multi-equations. The *system* consists of sets of multi-equations T and U. Where T is the *solved*, triangular part and U is the *unsolved* part. At first, T is empty and U is Z. This algorithm always terminates. If it terminates with failure then the system has no unifier. If it terminates with success then it has an empty unsolved part U and the result is T which is in solved form. The algorithm is nondeterministic because the order of selection of multi-equations is arbitrary.

4.1.1.3 The unification algorithm

The algorithm in figure 13 is a simplified and deterministic version of the previous one. It eliminates the need to apply substitutions and to check for cycles during each iteration. During each iteration only those multi-equations which definitely do not lead to cycles are selected.

4.1.1.4 Improvement on the unification algorithm

An improvement, in order to make an efficient multi-equation selection, is to associate a *counter* with each multi-equation. The counter contains the number of occurrences of its left-hand side variables in the right-hand side of all multi-equations in U. When the counter is zero, the multi-equation can be selected. The counters are initially computed by scanning U. Then, each time a reduction is applied, counters are decremented if an occurrence of one of its variables appears in the left hand side of a multi-equation in the frontier (because the right-hand side frontier variables were

```
Transform the two terms to unify into a set Z of multi-equations
Repeat
   Select a multi-equation S=M from U with non-empty M
   Compute the common part and frontier of M.
   If there is no common part then stop with failure.
      (This indicates that there is a clash)
   If any of the frontier multi-equations computed for M contains
      in its left-hand side some variables of S
      then stop with failure
      (This indicates that there is a cycle)
   Transform U using multi-equation reduction on S=M
   Apply compactification on U
   Apply the substitution, where $t_c$ is the common part of M
      $\{[x/t_c] \mid \forall x \in S\}$
      to the RHS of multi-equations in U.
   Transfer S=common part of M from U to the end of T.
Until U is empty or U contains only multi-equations
   with empty RHS.
Transfer all the multi-equations of U with empty RHS to the end of T
Stop with success
T contains a substitution that unifies the two terms
```

Figure 12: A Second Nondeterministic Algorithm

```
Transform the two terms to unify into a set Z of multi-equations
Repeat
   Select a multi-equation S=M of the unsolved part
       such as the variables in its S part do not occur anywhere else.
   If there is no such multi-equation then stop with failure
       since there is a cycle.
   If M is empty then
       transfer the multi-equatic ' to the end of T.
   Else
       begin
          Compute the common part and the frontier of M.
          If there is no common part then stop with failure
             since there is a clash.
          Apply multi-equation reduction and compactification to U.
          Transfer S=common part of M from U to the end of T.
       end
Until U is empty
Stop with success.
T contains a substitution that unifies the two terms.
```

Figure 13: Martelli and Montanari Unification Algorithm

in the M terms of the selected multi-equation). When multi-equations are merged, counters are added.

4.1.2 Comparison with other algorithms

The Martelli and Montanari algorithm offers many advantages:

1. The representation it uses is very abstract, it does not refer to any specific control, or any data structure for representing terms and systems of equations. This makes the idea very transparent, and it gives freedom of choice to any person who wants to implement the algorithm [18].

2. Their approach is flexible. This is clearly demonstrated by the fact that this algorithm became a standard in the field and its idea was used to describe other unification algorithms [18].

3. It allows terms of any depth. This leads to a more efficient performance because the computation of the common part and the frontier is the same for any term.

Martelli and Montanari's algorithm has an almost linear time complexity. In the remaining part of this section, we will compare it to two other algorithms: Huet's algorithm and Paterson and Wegman's algorithm both with a linear complexity. The three algorithms stop either with success, or with failure when a cycle or a clash is detected. We consider the performance of the three algorithms in three extreme cases:

1. When there is a high probability of stopping with success: Paterson and Wegman's is asymptotically the best, because it has a linear complexity while the two others have a comparable nonlinear complexity. However, in theorem provers, where unification is mostly used, we usually deal with small terms and in this case the asymptotically growing difference between linear and nonlinear algorithms will depend on the efficiency of the implementation. An experiment was carried out by Trum and Winterstein [40]: They implemented the three algorithms in Pascal using the same data structures. The result was that Martelli and Montanari's algorithm had the lowest running time for all test data. The reason behind this is that it uses simpler data structures than Paterson and Wegman's and it does not need a final check for cycles like Huet's algorithm.

Algorithm/Case	Cycle	Clash	Success	Small terms
Paterson and Wegman	Very Good	Poor	Very Good	Good
Huet	Poor	Very Good	Good	Good
Martelli and Montanari	Good	Good	Good	Very Good

Table 9: Unification Algorithms Comparison Summary

2. When there is a high probab'lity of detecting a cycle: Paterson and Wegman's algorithm is the best because it does not merge any two multi-equations before making sure that there are no cycles. So it saves merges. Huet's has a very poor performance because it checks for cycles after each merging. Martelli and Montanari's algorithm is good because the detection of cycles is efficiently handled by the counters.

3. When there is a high probability of detecting a clash: Huet's algorithm is the best because it stops on clashes before wasting overhead of cycle detection. Martelli and Montanari's algorithm is better than Paterson and Wegman's algorithm because clashes are detected when computing the common part and this takes place earlier than in Paterson and Wegman's algorithm .

Some algorithms perform better than Martelli and Montanari in some cases, but Martelli and Montanari's algorithm is better in general since it has a good performance in all cases. Table 9 is a summary of the performance of these three algorithms.

4.2 Unification Implementation

4.2.1 Object Model

The object model shows the classes that were used to implement Montanari and Martelli's unification algorithm. Figure 14 shows the object model in overview. Figure 15 and Figure 16 show details of each class in the object model with its attributes. The object model has ten different classes : SYSTEM, SOLVED PART, UNSOLVED PART, MULTI-EQUATION, S PART, M PART, TERM, V 'ABLE, CONSTANT, COMPOSITE TERM, LIST, and ITERATOR. The relation between the first seven is aggregation (is-part): A SYSTEM consists of a SOLVED PART and an UNSOLVED PART. Each of the two parts consists of zero or many MULTI-EQUATIONS and each MULTI-EQUATION has an S PART and an M PART. Both multi-equation parts have terms. The TERM has an isa relation with three sub-classes: VARIABLE, CONSTANT, and COMPOSITE TERM which has terms as arguments. The two input terms to unify are transformed into a system. The unification algorithm uses a LIST container to handle intermediate computations and to manipulate the other classes. The LIST is traversed by LIST ITERATOR.

Besides the ubiquitous iterator pattern, unification uses double dispatching. This is a combination of the composite pattern [20, pages 163–175], which allows us to treat individual objects (variable, constant) and compositions of objects (composite term) in the same manner, and the visitor pattern [20, pages 331-345], which let us add operations to classes without changing them. Double dispatching means that the operation that is executed depends on the request (operation) and the type of each of two arguments. Instead of binding operations statically into the term interface, the operation that gets executed each time depends on the operation itself and the types of terms we are dealing with (variable, constant, or composite term). The binding to the exact operation takes place at run-time. Figure 17 through Figure 19 show examples of dispatching.

4.2.2 Pseudocodes

A pseudocode is provided for some of the methods with complex steps:

- **Unify Algorithm**

  Unify two input terms t_1 and t_2.

  In the implementation, each combination of two terms has its own method (being called using dispatching). The pseudocode on Figure 20 shows the steps taken by all methods using the if statements.

- **Solve System**

  Refer to Figure 21

- **Common Part Computation**

  Refer to Figure 22 for the pseudocode of the common part computation of M=(t_1,t_2).

- **Frontier Computation**

  Refer to Figure 23 for the pseudocode of the frontier computation of M=(t_1 ...

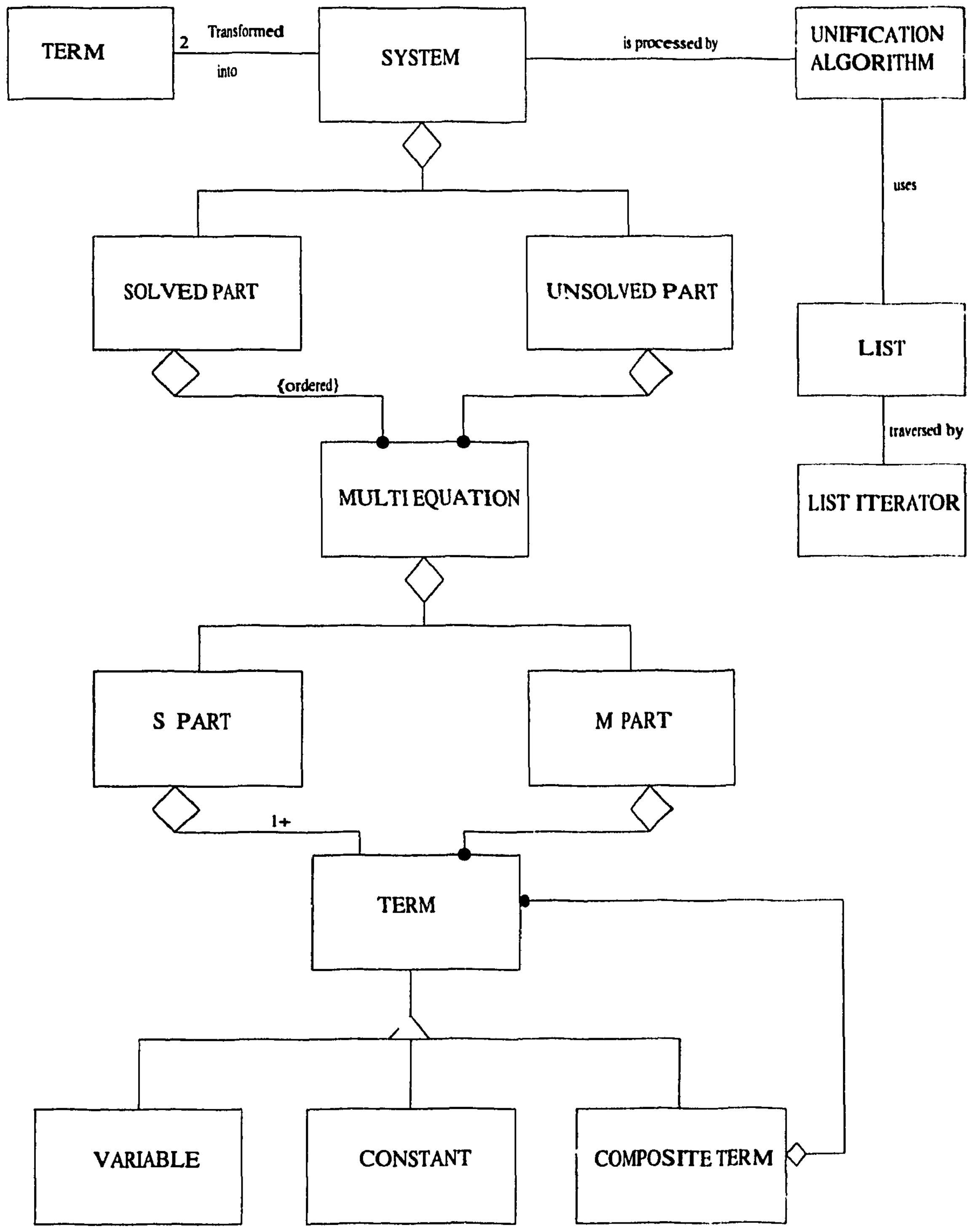


Figure 14: Object Model

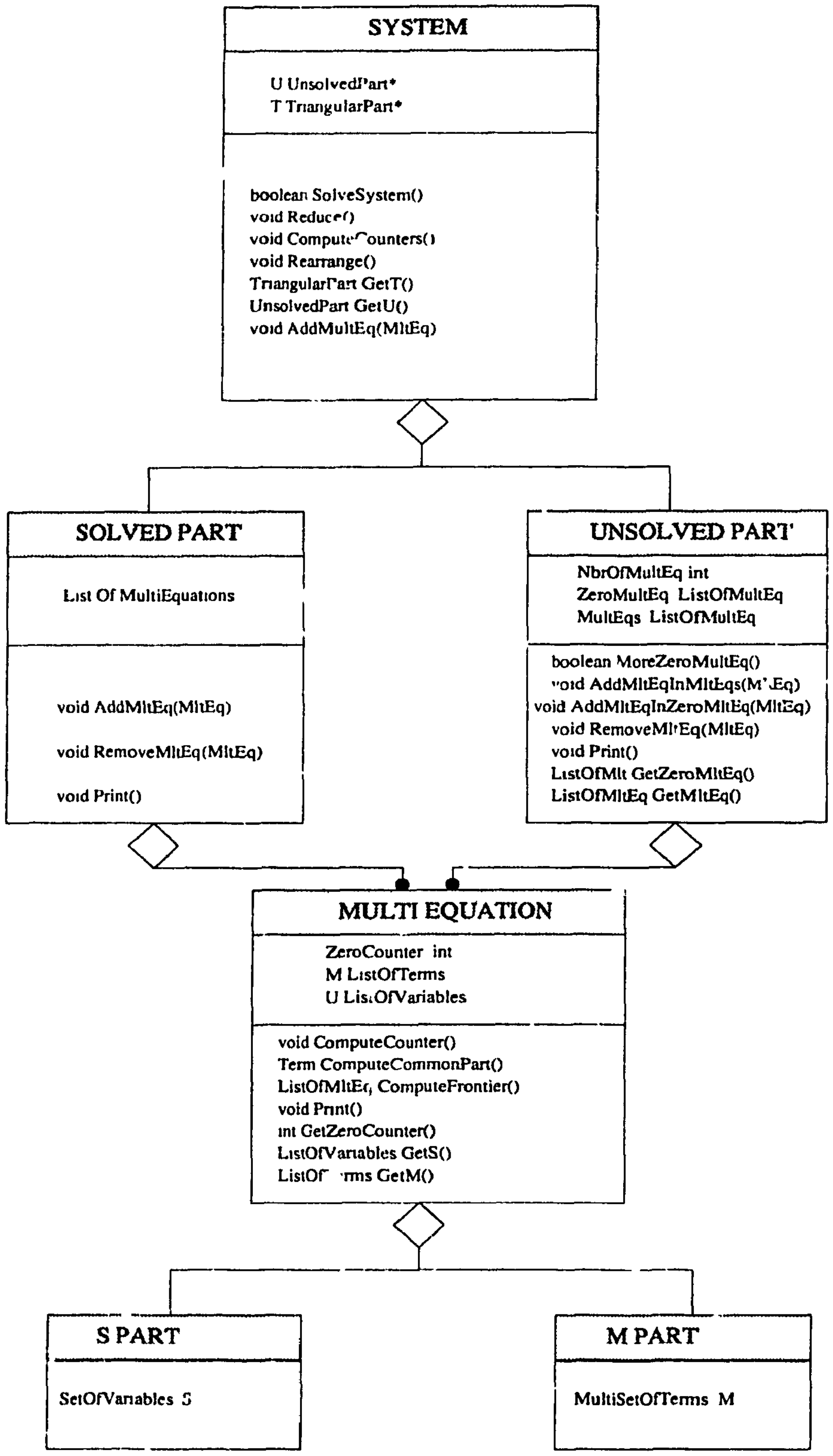


Figure 15: Object Model Details

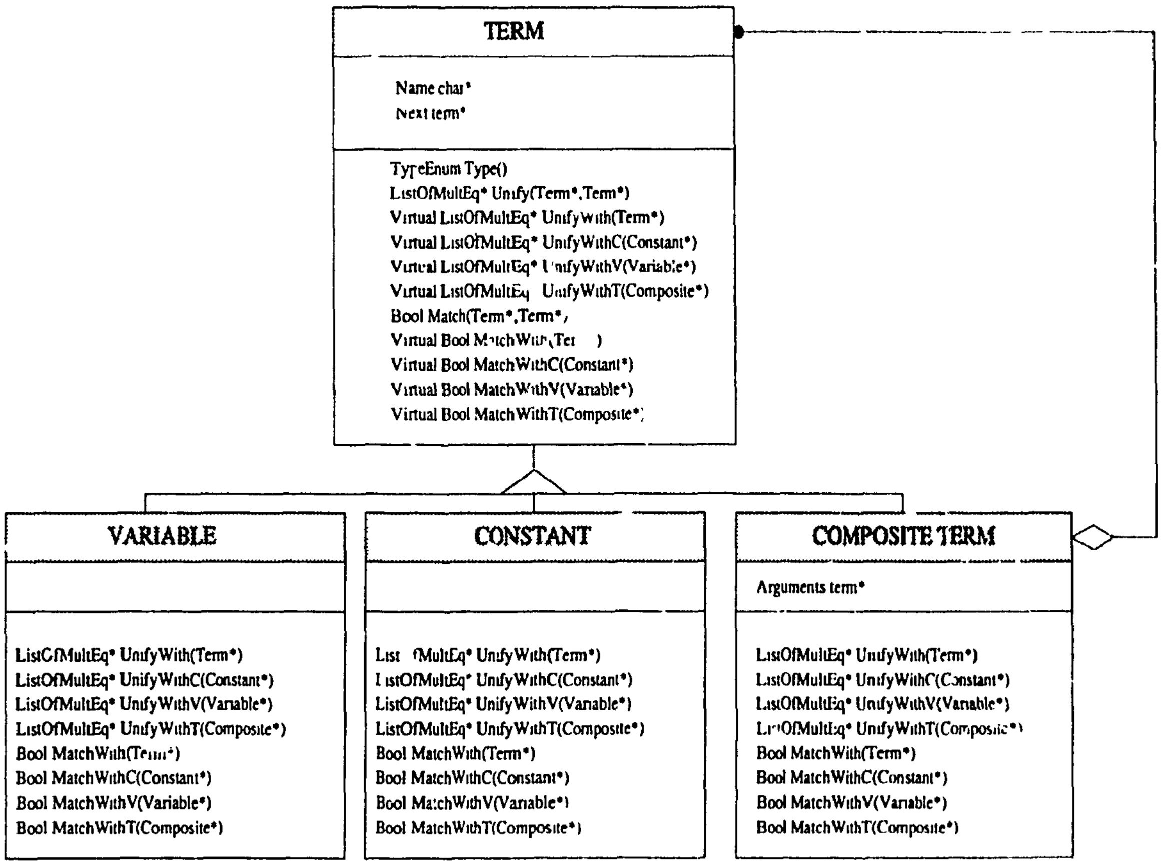

TERM
Name char*
Next term*
TypeEnum Type()
ListOfMultEq* Unify(Term*,Term*)
Virtual ListOfMultEq* UnifyWith(Term*)
Virtual ListOfMultEq* UnifyWithC(Constant*)
Virtual ListOfMultEq* UnifyWithV(Variable*)
Virtual ListOfMultEq UnifyWithT(Composite*)
Bool Match(Term*,Term*)
Virtual Bool MatchWith(Term*)
Virtual Bool MatchWithC(Constant*)
Virtual Bool MatchWithV(Variable*)
Virtual Bool MatchWithT(Composite*)
VARIABLE
ListOfMultEq* UnifyWith(Term*)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable*)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)
CONSTANT
ListOfMultEq* UnifyWith(Term*)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable*)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)
COMPOSITE TERM
Arguments term*
ListOfMultEq* UnifyWith(Term*)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable*)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)

Figure 16: Object Model Details

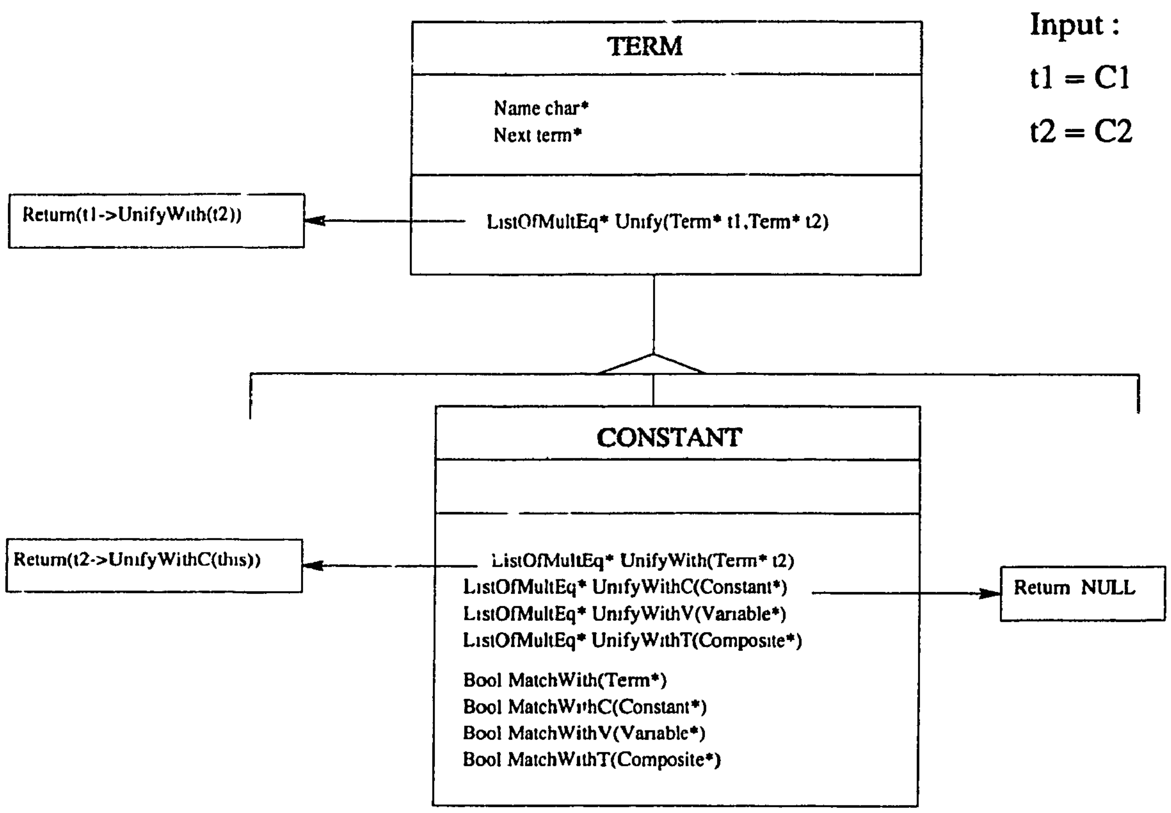

Input :
t1 = C1
t2 = C2
TERM
Name char*
Next term*
ListOfMultEq* Unify(Term* t1,Term* t2)
Return(t1->UnifyWith(t2))
CONSTANT
ListOfMultEq* UnifyWith(Term* t2)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable*)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)
Return(t2->UnifyWithC(this))
Return NULL

Figure 17: Dispatching Example

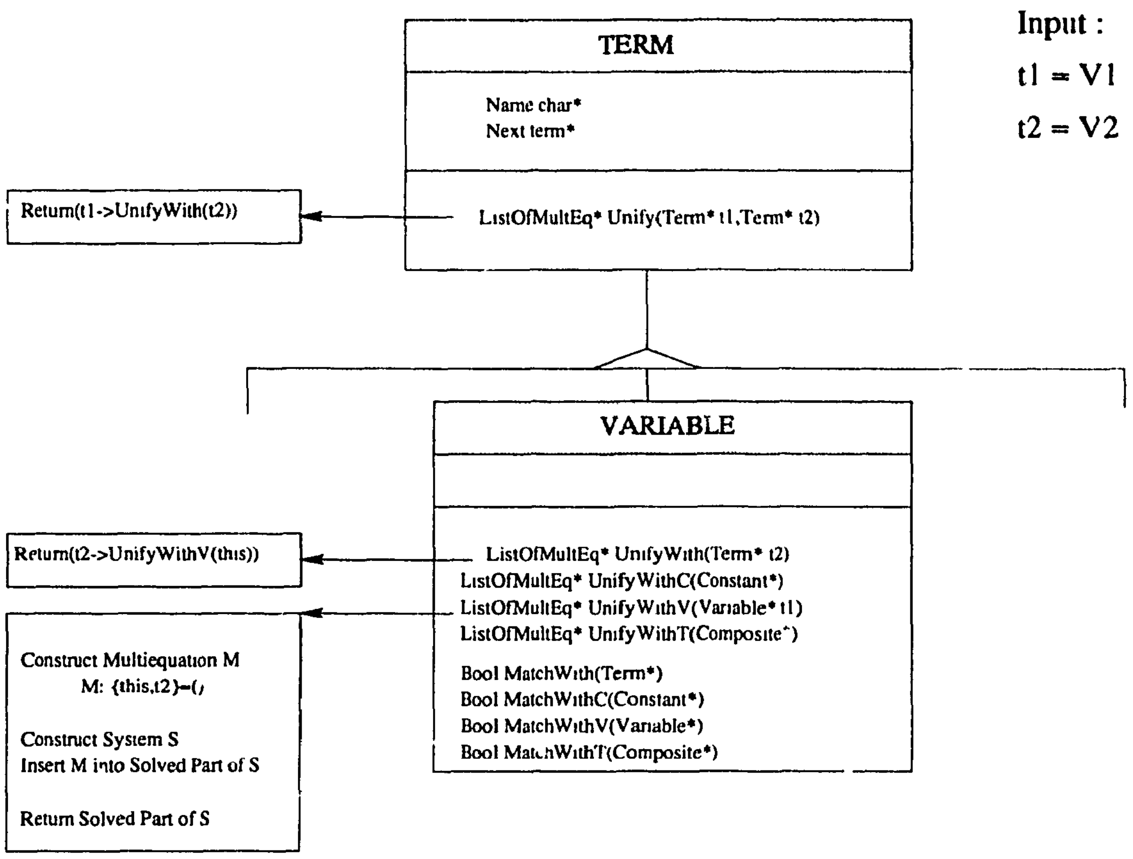
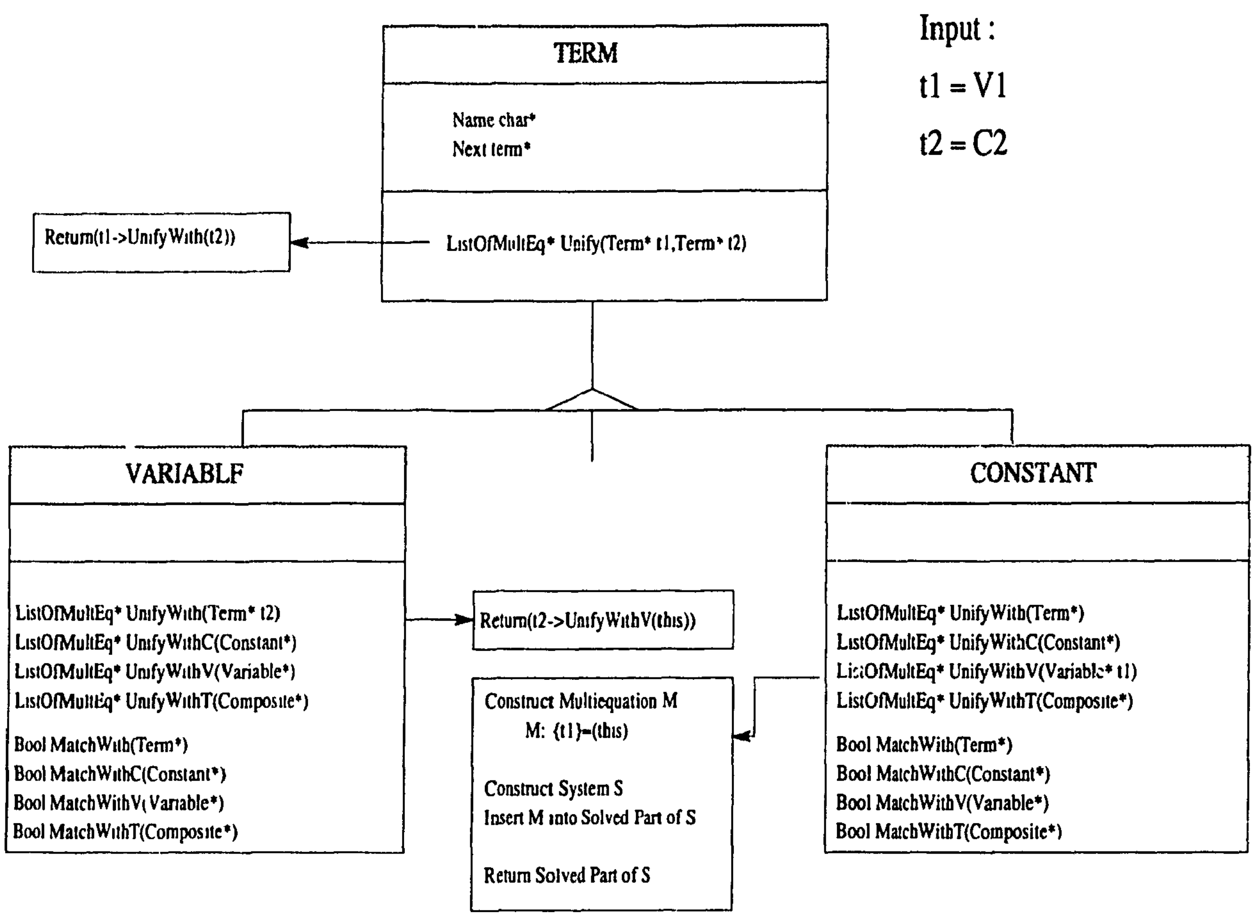

Input :
t1 = V1
t2 = V2
TERM
Name char*
Next term*
ListOfMultEq* Unify(Term* t1,Term* t2)
Return(t1->UnifyWith(t2))
VARIABLE
ListOfMultEq* UnifyWith(Term* t2)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable* t1)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)
Return(t2->UnifyWithV(this))
Construct Multiequation M
M: {this,t2}=()
Construct System S
Insert M into Solved Part of S
Return Solved Part of S

Figure 18: Dispatching Example



Input :
t1 = V1
t2 = C2
TERM
Name char*
Next term*
ListOfMultEq* Unify(Term* t1,Term* t2)
Return(t1->UnifyWith(t2))
VARIABLF
ListOfMultEq* UnifyWith(Term* t2)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable*)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)
Return(t2->UnifyWithV(this))
Construct Multiequation M
M: {t1}=(this)
Construct System S
Insert M into Solved Part of S
Return Solved Part of S
CONSTANT
ListOfMultEq* UnifyWith(Term*)
ListOfMultEq* UnifyWithC(Constant*)
ListOfMultEq* UnifyWithV(Variable* t1)
ListOfMultEq* UnifyWithT(Composite*)
Bool MatchWith(Term*)
Bool MatchWithC(Constant*)
Bool MatchWithV(Variable*)
Bool MatchWithT(Composite*)

Figure 19: Dispatching Example

```
If we have a constant term and a function term then
      return that there is no unification
If we have two constant term then
      If the constants are identical
            return an empty triangular part
      Otherwise
            return that there is no unification
If we have two variable terms then
      construct a multi-equation with the two variables in S and M empty
      insert the multi-equation into the triangular part of the system
      return the triangular part
If we have a variable term and a constant term then
      construct a multi-equation with the variable in S and constant in M
      insert the multi-equation into the triangular part of the system
      return the triangular part
If we have a variable term and a composite term then
      If the variable is not in any argument then
         construct a multi-equation with
            the variable in S and the function in M
         insert the multi-equation into the triangular part of the system
         return the triangular part
      Otherwise
            return that there is no unification because of a cycle
If we have two composite terms then
      If they have different root names or different number of arguments
            return that there is no unification because of a clash
      Otherwise
            begin
               for each variable term in the arguments of the functions
                  begin
                     construct a multi-equation S has the variable and M empty
                     add it to the unsolved part
                  end
               construct an additional multi-equation:
                     insert in S a new variable not occurring anywhere
                     insert in M part the two composite terms
               insert the multi-equation into the unsolved part
               Solve the system
               Return either no unification or the triangular part
            end
```

Figure 20: Unify Algorithm

```
Compute counters of all multi-equations in the unsolved part.
Rearrange: store multi-equations with counter zero into the zero list
      and the others in the multi-equation list.
Repeat
      Take a multi-equation from the zero list
      If its M part is empty
            Insert it in the triangular part
      Otherwise
            begin
               Compute its common part
               If it does not have a common part
                  Stop with failure, there is a clash
               Otherwise
                  begin
                     Compute its frontier
                     Apply multi-equation reduction on the system
                     Compute the counters again
                     Rearrange the system
                  end
            end
Until the zero list is empty
If the unsolved part is empty
      return success, solution is in T part
Otherwise
      failure, there is a cycle
```

Figure 21: Solve System Algorithm

```
Case
One of $t_1$, $t_2$ is a variable x :
      common part = x
Both $t_1$ and $t_2$ are constants c :
      common part = c
$t_1 = f(a_1...a_n)$ and $t_2 = f(b_1...b_n)$ :
      common part = $f(c_1...c_n)$
      where $c_i$= common part of $a_i$ and $b_i$
Otherwise : clash
End Case
```

Figure 22: Common Part Computation Algorithm

t_n) having a common part t_c.

```
Frontier = $\emptyset$
Loop over leaves of $t_c$
      If leaf is variable x then
         Let $s_1$ ... $s_n$ be the terms (sub-term) of $t_i$
         corresponding to the position of this leaf in $t_c$
         Let S be the set of variable terms in { $s_1$ ... $s_n$ }
            and M be the set of non-variable terms in { $s_1$ ... $s_n$ }
         Add S=M to frontier
```

Figure 23: Frontier Computation Algorithm

4.2.3 Data Structures

The major data structures used are briefly discussed in this section.

1. Term is a structure:

   - Name: a string. An assumption is made regarding the name of the term: If it starts with a capital letter or the underscore character then it is a variable. If it starts with a small letter it is a constant. If its arguments field is not null (it has arguments), and starts with any letter or the underscore it is a function. Name is the variable name, the constant name, or the composite term root name for example $f(x)$ has f as its name.
   - Arguments: a pointer to a term. For composite terms, arguments is the argument list. For variable and constant terms arguments is null.
   - Next: a pointer to a term. Next allows the construction of a list of arguments, where each argument is a term and its next part points to the term following it, the last term has a null next.

2. Multi-Equation is a structure:

   - Counter: integer. Counter indicates the number of occurrences of the variables in the S part, of a multi-equation S=M, in the M parts of all multi-equations in the U part of the system.

- S part: a list of pointers to variables. It is the LHS of a multi-equation and it holds the variables.
- M part: a list of pointers to terms. It is the RHS of a multi-equation and it holds all the non-variable terms.

3. Unsolved Part is a structure:
   - Number of multi-equations: integer
   - Multi-Equations with counters equal zero: list of multi-equations. The multi-equations in this list can be selected to be processed without resulting in a cycle.
   - Multi-Equations with non-zero counters: list of multi-equations. The multi-equations in this list cannot be selected for process. Processing them will result in a cycle.
4. System is a structure:
   - Unsolved Part: a structure where the multi-equations not yet processed are stored.
   - Triangular Part: list of multi-equations. Multi-equations in T part are in solved form. When the algorithm terminates with success, T part is the unifier.

4.2.4 Data Dictionary

- **System Class**
  - **Description**
    It is a way of representing the two terms as a list of multi-equations in order to compute their most general unifier (mgu).
  - **Methods**
    1. SolveSystem: solves the system and stores the solved multi-equations in its triangular part. It indicates if there is no unifier.
    2. Reduce: implements multi-equation reduction.

3. ComputeCounters: scans the whole system and computes the zero counters of all the multi-equations.
4. Rearrange: scans the whole system and moves the multi-equations to the zero or non-zero lists depending on the values of their counters.
5. GetT and GetU: return the triangular part and the unsolved part of the system respectively.
6. AddMultEq: Adds a multi equation to the system

- **Solved Part Class**
  - **Description**

    It is the part of the system that holds the solved multi-equations. When the algorithm terminates, the solved part contains a substitution that unifies the two input terms.
  - **Methods**

    The only methods of this class are to add and remove multi-equations, and to print itself.

- **Unsolved Part Class**
  - **Description**

    It is the part of the system that holds the unsolved multi-equations. The multi-equations are arranged into two lists depending on the value of their counters.
  - **Methods**

    1. MoreZeroMultEq: checks whether there are still multi-equations with zero counter.
    2. RemoveMultEq: removes a multi-equation.
    3. GetZeroMltEq and GetMltEq: return the list of multi-equations with zero counter and with non-zero counter respectively.
    4. AddMltEqInMultEqs: adds a multi-equation to the unsolved part and compactifies by merging multi-equations as necessary.
    5. AddMltEqInZeroMultEq: inserts a multi-equation in the ZeroMultEq list of the unsolved part.

- **Multi-Equation Class**
  - **Description**

    A multi-equation is the generalisation of an equation. It allows us to group together many terms which should be unified. A Multi-Equation has two parts: S and M.
  - **Methods**
    1. ComputeCounter: computes the value of the counter of a multi-equation. The counter is the number of occurrences of the variables in S in all the M parts of the multi-equations in U.
    2. ComputeCommonPart: computes the common part of the multi-equation.
    3. ComputeFrontier: computes the frontier of the multi equation.
    4. GetZeroCounter: returns the value of the counter of a multi-equation.
    5. GetS: returns the S part of a multi-equation.
    6. GetM· returns the M part of a multi-equation.
- **S Part Class**
  - **Description**

    S is the left hand side of a multi-equation. It is a nonempty set of variable terms.
- **M Part Class**
  - **Description**

    M is the right hand side of a multi-equation. It is a multi-set of constant and composite terms. Unlike the S part, the M part might be empty for some multi-equations.
- **Term Class**
  - **Description**

    A term is either a constant, a variable, or a function $f(t_1, t_2, ..., t_n)$ where each t_i is a term. TERM is a base class with a derived class for each type of term: variable, constant, or composite.
  - **Methods**

1. Type: returns the type of the term. An assumption is made for this application:
   * If it is a variable then the name starts with either capital letter or underscore letter.
   * If it is a constant then the name starts [illegible] lower case letter.
   * If it has arguments and the name starts [illegible] any letter or the underscore symbol then it is a composite.
2. Unify: is implemented using double dispatching. It calls the appropriate version of UnifyWith depending on its first argument type.
3. Match: is implemented using double dispatching. It calls the appropriate version of MatchWith depending on its first argument type. It returns whether the two terms match or not. It is used in computing the common part and in initialising the system, where a non-match indicates a clash and the unification fails.
4. UnifyWith, UnifyWithConstant, UnifyWithVariable, UnifyWithCompositeTerm: are virtual in the base class allowing each subclass to define its own version.
5. MatchWith, MatchWithConstant, MatchWithVariable, MatchWithCompositeTerm: are virtual in the base class allowing each subclass to define its own version.

- **Variable Class**
  - **Description**

    A variable may be substituted or instantiated to a term. Variable is a subclass of term.
  - **Methods**

    1. UnifyWith, UnifyWithConstant, UnifyWithVariable, UnifyWithCompositeTerm: are the variable implementations of the virtual versions defined in the base class. They all deal with the first step of unification where the system is initialised, then either SolveSystem is called to return the solution in the system solved part or without calling SolveSystem the system is already solved or unsolvable. For instance

in the case of two variables, the system would be solved without calling SolveSystem.

2. MatchWith, MatchWithConstant, MatchWithVariable, MatchWithCompositeTerm: are the variable implementations of the virtual versions defined in the base class. They return either true or false depending on the combination of the two terms and their actual syntax. For example if we have a variable and a constant then we return true (Variable::MatchWithC).

- **Constant Class**
  - **Description**

    A constant is an atomic, primitive value. It does not change value. Constant is a subclass of term.
  - **Methods**
    1. UnifyWith, UnifyWithConstant, UnifyWithVariable, UnifyWithCompositeTerm: are the constant implementations of the virtual versions defined in the base class. They all deal with the first step of unification where the system is initialised. In the case of constant either the solution is in the system solved part (Constant::UnifyWithV) or there is no unification (two different constants or Constant::UnifyWithT).
    2. MatchWith, MatchWithConstant, MatchWithVariable, MatchWithCompositeTerm: are the constant implementations of the virtual versions defined in the base class. They return either true or false depending on the combination of the two terms and their actual syntax. For example if we have two different constant terms then we return false (Constant::MatchWithC).

- **Composite Term Class**
  - **Description**

    The composite term is a function or predicate. It may have arguments. It could have constants, variables, or even other composite terms as its arguments. Composite term is also a subclass of term.
  - **Methods**

1. UnifyWith, UnifyWithConstant, UnifyWithVariable, UnifyWithCompositeTerm: are the composite term implementations of the virtual versions defined in the base class. They all deal with the first step of unification where the system is initialised, then either SolveSystem is called to return the solution in the system solved part or without calling SolveSystem the system is already solved or unsolvable. For instance, the case of two composite terms needs to call SolveSystem, while a composite term and a variable is already solved, and a composite term with a constant is unsolvable

2. MatchWith, MatchWithConstant, MatchWithVariable, MatchWithCompositeTerm: are the composite term implementations of the virtual versions defined in the base class. They return either true or false depending on the combination of the two terms and their actual syntax. For example, a composite term and a constant term will return true (Composite Term::MatchWithC).

4.3 A Complete Unification Example

In this section, a complete unification example is first described. Then an event trace shows how the actual steps of this example correspond to the different method calls of the implementation. The two input terms are:

$$T1{:}f(x_1, g(x_2, x_3), x_2, b)$$
$$T2{:}f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)$$

They are transformed into the system S.
Each multi-equation has a counter to indicate the number of occurrences of the variables in the LHS of the multi-equation in the RHS of all multi-equations:
U:

Counter	Multi-equation
0	$\{x\} = (f(x_1, g(x_2, x_3), x_2, b),\ f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4))$
2	$\{x_1\} = ()$
3	$\{x_2\} = ()$
1	$\{x_3\} = ()$
2	$\{x_4\} = ()$
1	$\{x_5\} = ()$

T:()
Figure 24 shows the event trace for the initial transformation of the terms into a system.

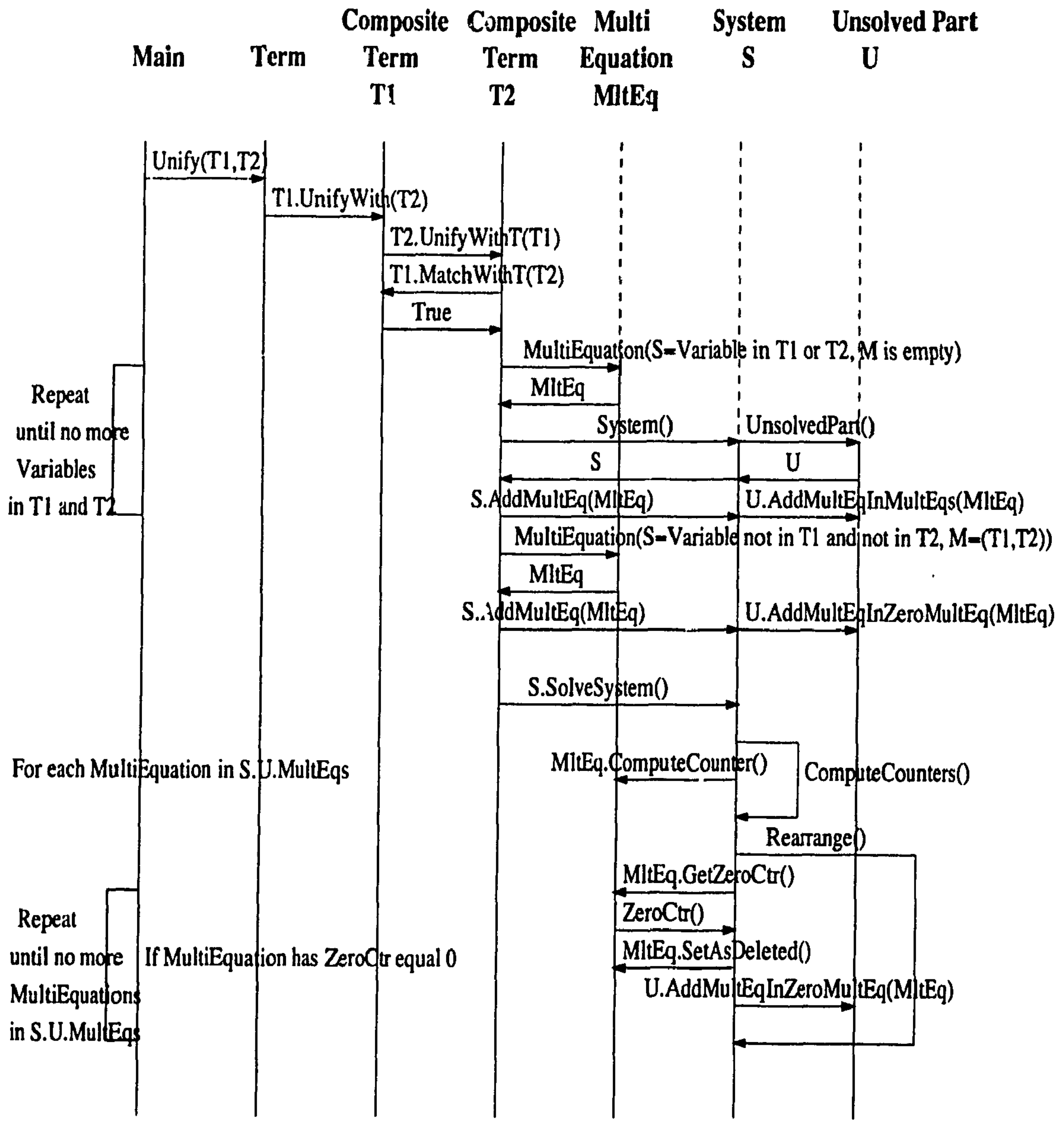

Main
Term
Composite Term T1
Composite Term T2
Multi Equation MltEq
System S
Unsolved Part U
Unify(T1,T2)
T1.UnifyWith(T2)
T2.UnifyWithT(T1)
T1.MatchWithT(T2)
True
MultiEquation(S=Variable in T1 or T2, M is empty)
MltEq
System()
UnsolvedPart()
S
U
S.AddMultEq(MltEq)
U.AddMultEqInMultEqs(MltEq)
MultiEquation(S=Variable not in T1 and not in T2, M=(T1,T2))
MltEq
S.AddMultEq(MltEq)
U.AddMultEqInZeroMultEq(MltEq)
Repeat until no more Variables in T1 and T2
S.SolveSystem()
For each MultiEquation in S.U.MultEqs
MltEq.ComputeCounter()
ComputeCounters()
Rearrange()
MltEq.GetZeroCtr()
ZeroCtr()
MltEq.SetAsDeleted()
U.AddMultEqInZeroMultEq(MltEq)
Repeat until no more MultiEquations in S.U.MultEqs
If MultiEquation has ZeroCtr equal 0

Figure 24: Event Trace for Transforming the Two Terms into a System

For the **first Iteration** of the unification algorithm the selected multi-equation is

$$\text{Mlt:}\{x\} = (f(x_1, g(x_2, x_3), x_2, b),\ f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4))$$

The computation of the common part is shown in Figure 25, yields

$$f(x_1, x_1, x_2, x_4)$$

The frontier computation is shown in Figure 26, using the notation of Figure 27 to identify sub-terms. The result is

$$\begin{aligned}\{Mlt1 : \{x_1\} &= (g(h(a, x_5), x_2)) \\ Mlt2 : \{x_1\} &= (g(x_2, x_3)) \\ Mlt3 : \{x_2\} &= (h(a, x_4)) \\ Mlt4 : \{x_4\} &= (b)\}\end{aligned}$$

Then the system is reduced and compacted as shown in Figure 28, yielding
U:

Counter	Multi-equation
0	$\{x_1\} = (g(h(a, x_5), x_2), g(x_2, x_3))$
2	$\{x_2\} = (h(a, x_4))$
1	$\{x_3\} = ()$
1	$\{x_4\} = (b)$
1	$\{x_5\} = ()$

T:

$$(\{x\} = f(x_1, x_1, x_2, x_4))$$

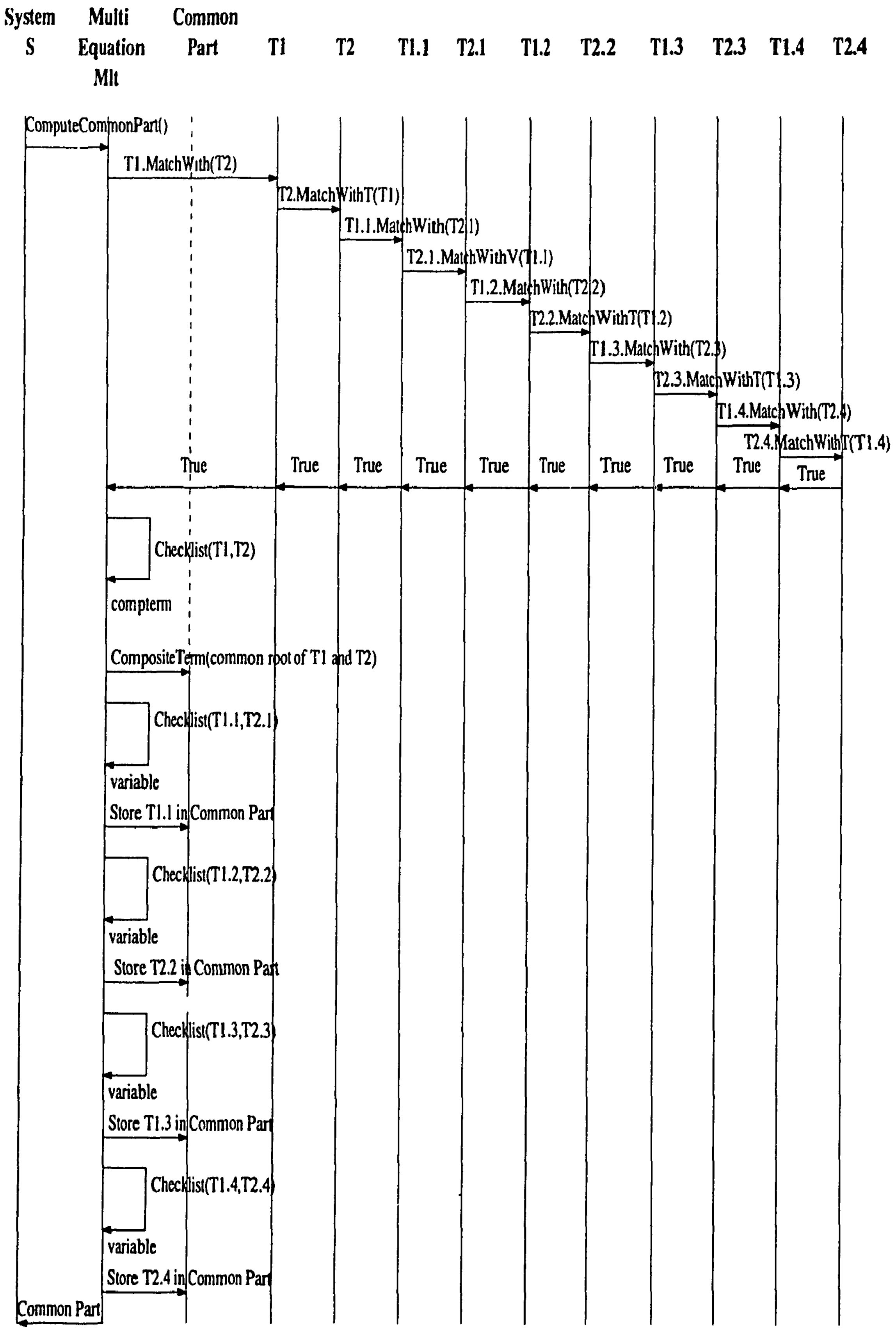


Figure 25: Event Trace for Common Part Computation 1st Iteration

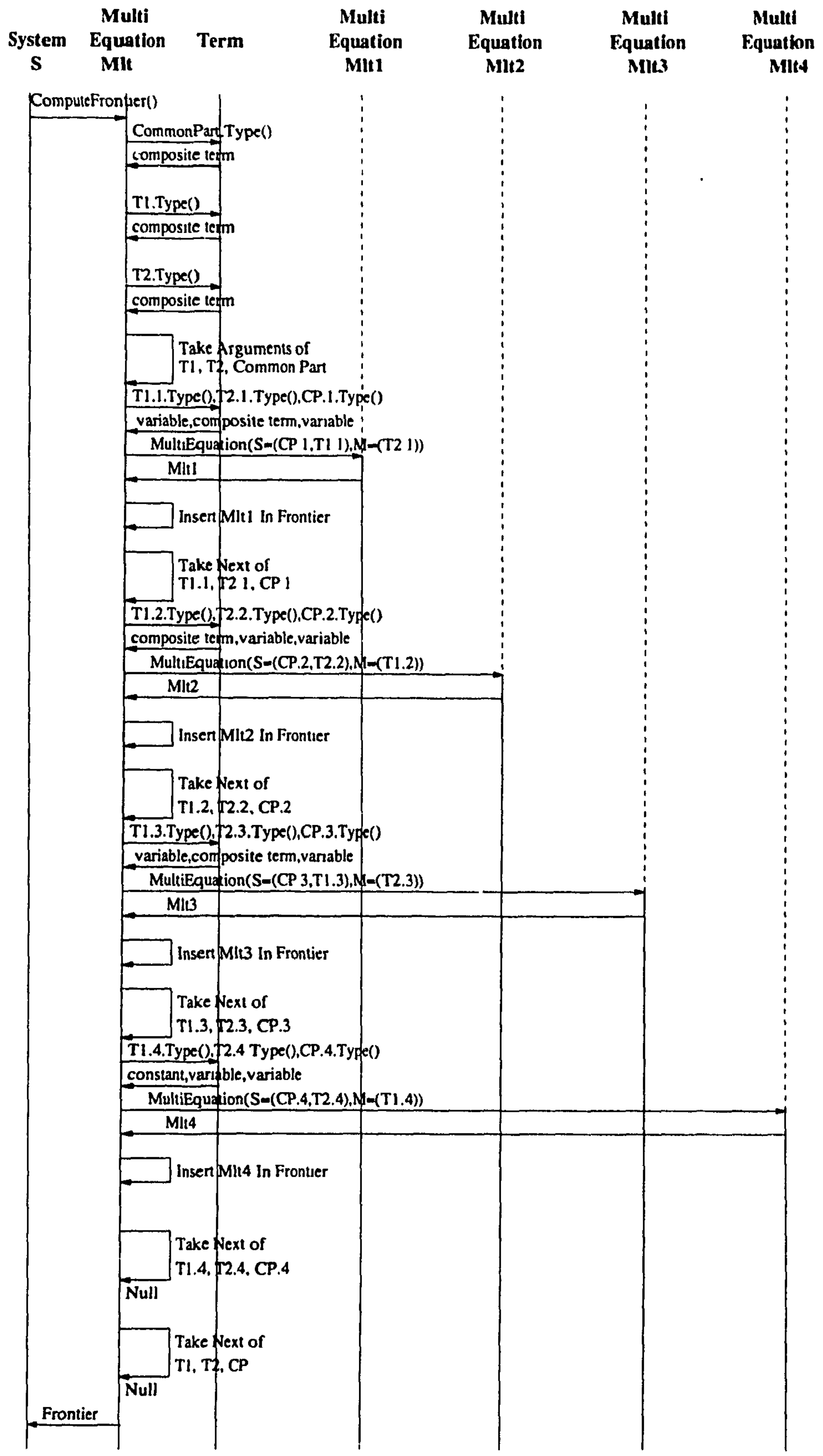

System S
Multi Equation Mlt
Term
Multi Equation Mlt1
Multi Equation Mlt2
Multi Equation Mlt3
Multi Equation Mlt4
ComputeFrontier()
CommonPart.Type()
composite term
T1.Type()
composite term
T2.Type()
composite term
Take Arguments of T1, T2, Common Part
T1.1.Type(),T2.1.Type(),CP.1.Type()
variable,composite term,variable
MultiEquation(S=(CP 1,T1 1),M=(T2 1))
Mlt1
Insert Mlt1 In Frontier
Take Next of T1.1, T2 1, CP 1
T1.2.Type(),T2.2.Type(),CP.2.Type()
composite term,variable,variable
MultiEquation(S=(CP.2,T2.2),M=(T1.2))
Mlt2
Insert Mlt2 In Frontier
Take Next of T1.2, T2.2, CP.2
T1.3.Type(),T2.3.Type(),CP.3.Type()
variable,composite term,variable
MultiEquation(S=(CP 3,T1.3),M=(T2.3))
Mlt3
Insert Mlt3 In Frontier
Take Next of T1.3, T2.3, CP.3
T1.4.Type(),T2.4 Type(),CP.4.Type()
constant,variable,variable
MultiEquation(S=(CP.4,T2.4),M=(T1.4))
Mlt4
Insert Mlt4 In Frontier
Take Next of T1.4, T2.4, CP.4
Null
Take Next of T1, T2, CP
Null
Frontier

Figure 26: Event Trace for Frontier Computation 1st Iteration

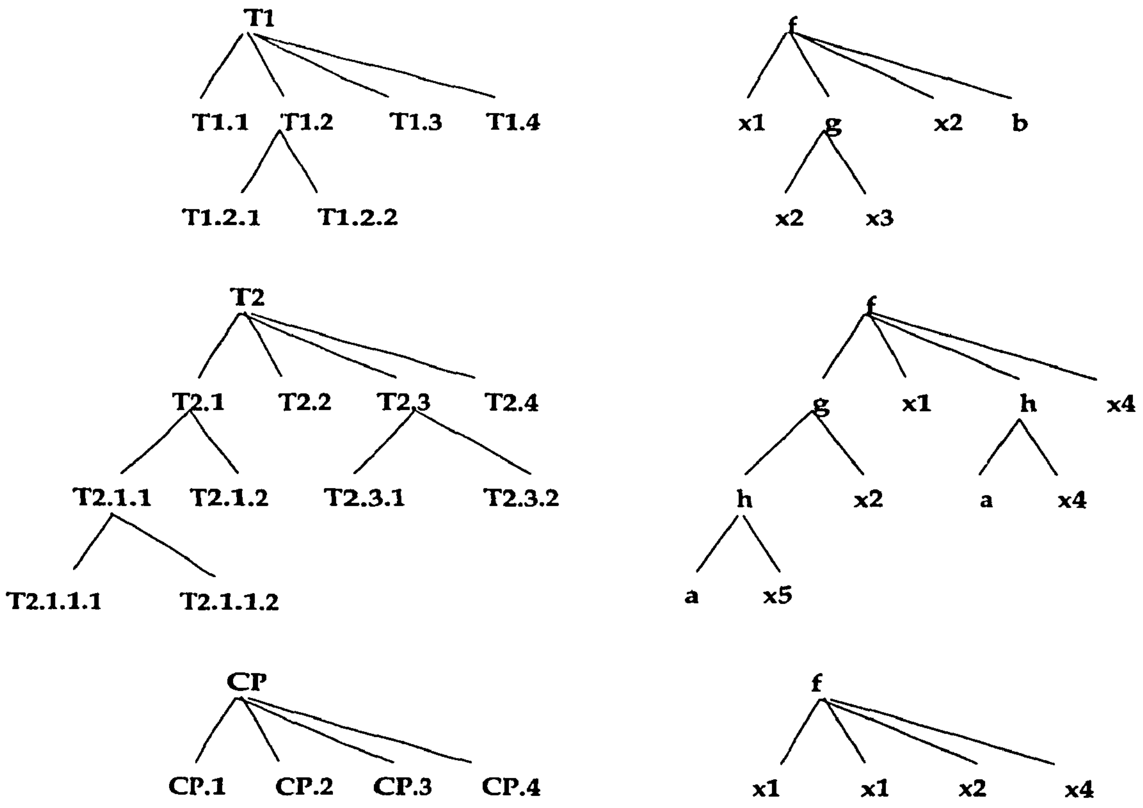


Figure 27: First Iteration: Common Part and Frontier Computation

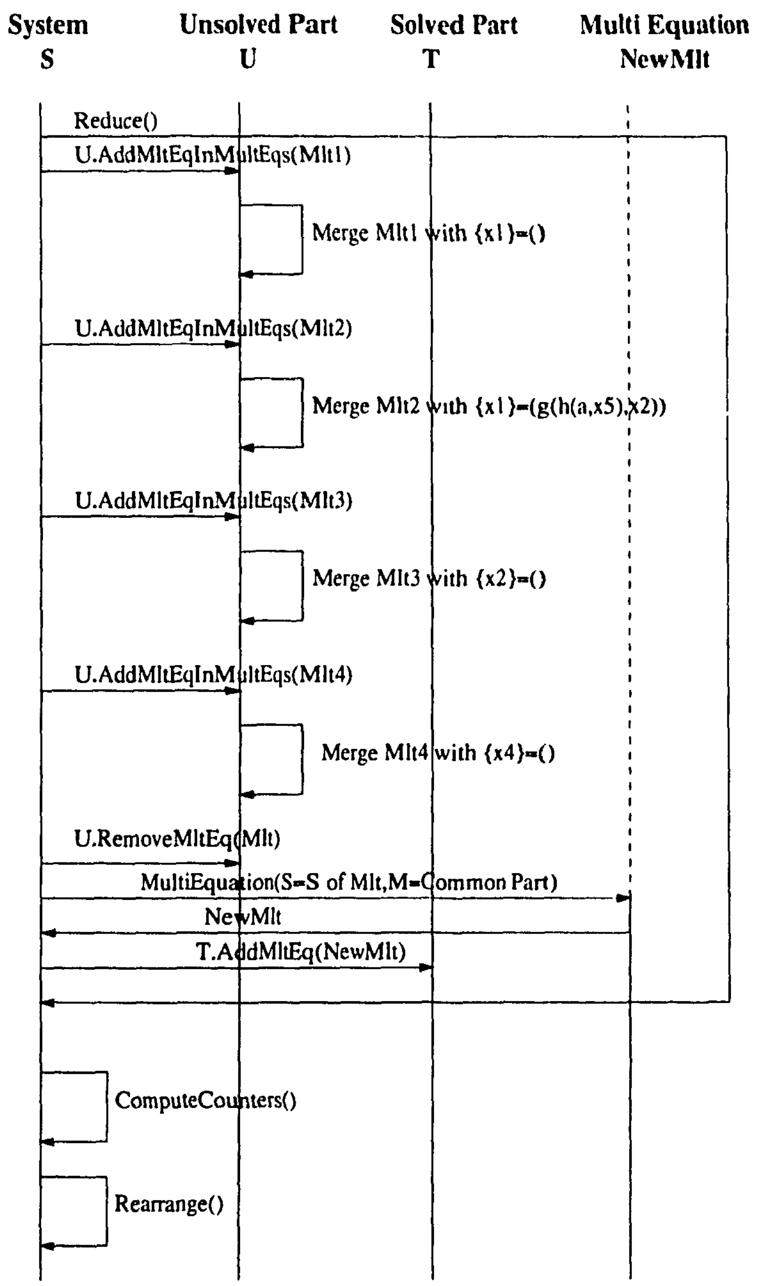


Figure 28: Event Trace for System Reduction and Compactification 1st Iteration

For the **second iteration** of the unification algorithm the selected multi-equation is

$$\text{Mlt: } \{x_1\} = (g(h(a, x_5), x_2), g(x_2, x_3))$$

The computation of the common part yields

$$g(x_2, x_3)$$

The frontier computation, using the notation of Figure 29 to identify sub-terms, results in

$$\begin{aligned} \{Mlt1 : \{x_2\} &= (h(a, x_5)) \\ Mlt2 : \{x_2\} &= (x_3)\} \end{aligned}$$

Then the system is reduced and compacted yielding
U:

Counter	Multi-equation
0	$\{x_2, x_3\} = (h(a, x_4), h(a, x_5))$
1	$\{x_4\} = (b)$
1	$\{x_5\} = ()$

T:

$$\begin{aligned} &(\{x\} = f(x_1, x_1, x_2, x_4) \\ &\{x_1\} = g(x_2, x_3)) \end{aligned}$$

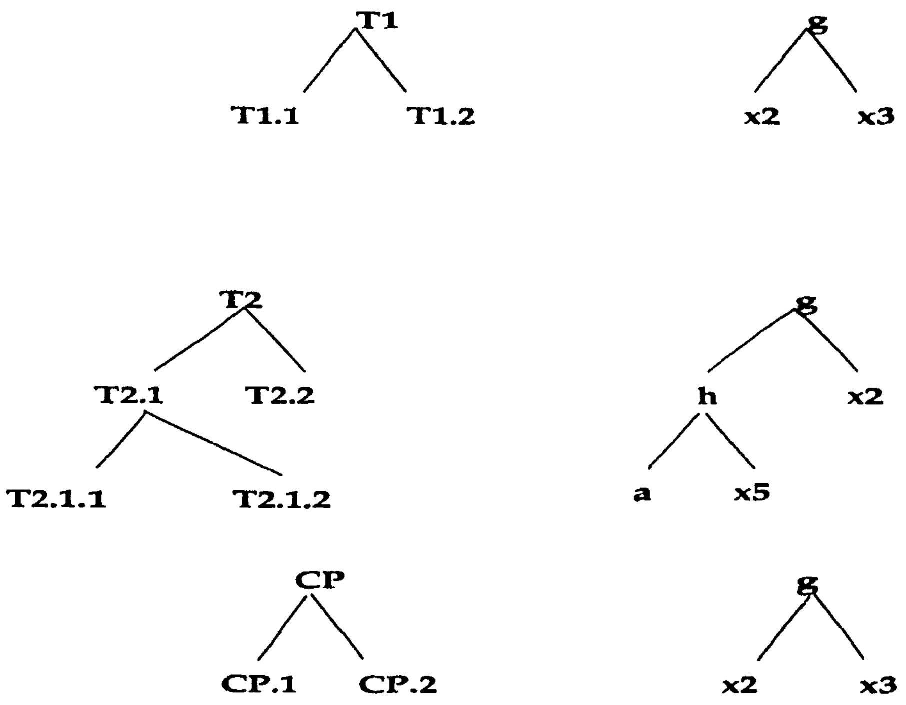


Figure 29: Second Iteration: Common Part and Frontier Computation

For the **third iteration** of the unification algorithm the selected multi-equation is

$$\text{Mlt: } \{x_2, x_3\} = (h(a, x_4), h(a, x_5))$$

The computation of the common part yields

$$h(a, x_4)$$

The frontier computation, using the notation of Figure 30 to identify sub-terms, results in

$$\text{Mlt1: } \{\{x_4\} = (x_5)\}$$

Then the system is reduced and compacted yielding
U:

Counter	Multi-equation
0	$\{x_4, x_5\} = (b)$

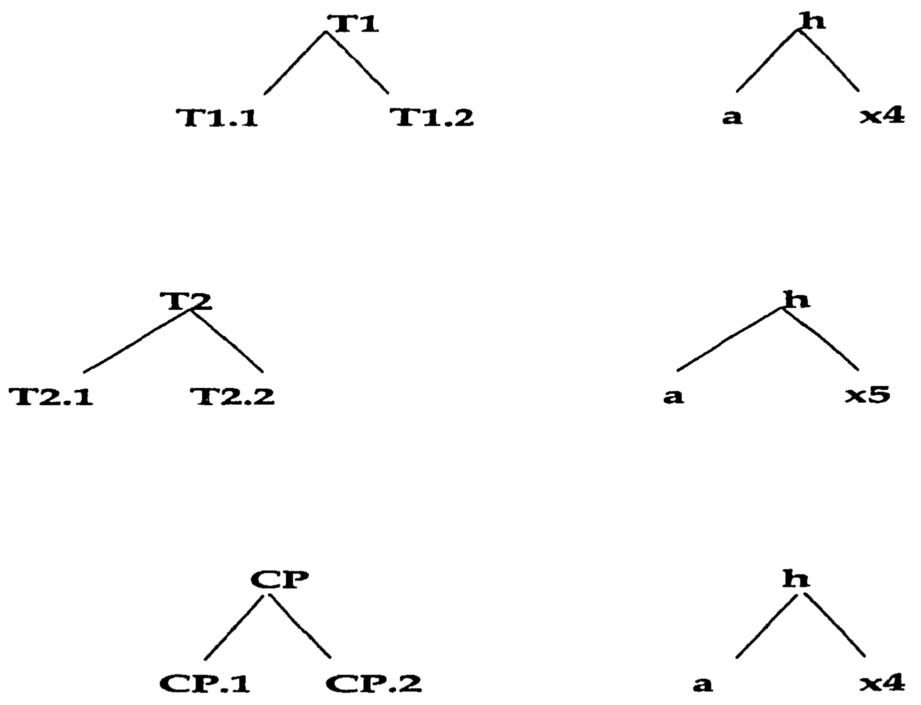


Figure 30: Third Iteration: Common Part and Frontier Computation

T:

$$
\begin{aligned}
(\{x\} &= f(x_1, x_1, x_2, x_4) \\
\{x_1\} &= g(x_2, x_3) \\
\{x_2, x_3\} &= h(a, x_4))
\end{aligned}
$$

For the **fourth iteration** of the unification algorithm the last multi-equation is selected

$$\text{Mlt: } \{x_4, x_5\} = (b)$$

The computation of the common part yields

$$b$$

No Frontier since the common part is a constant.
Then the system is reduced and compacted yielding
U:()
T:

$$\begin{aligned}
&(\{x\} = f(x_1, x_1, x_2, x_4) \\
&\{x_1\} = g(x_2, x_3) \\
&\{x_2, x_3\} = h(a, x_4) \\
&\{x_4, x_5\} = (b))
\end{aligned}$$

U is empty. The two terms can be unified by applying the set of substitutions in the T part of the system in a bottom up way. The most general unifier is:
$\{[x_1/g(h(a,b), h(a,b)]$, $[x_2/h(a,b)]$, $[x_3/h(a,b)]$, $[x_4/b]$, $[x_5/b]\}$

Chapter 5

Inference Mechanism

A language for knowledge representation should possess a decidable and fast procedure for deciding whether statements follow from a knowledge base. It should be expressively powerful and its semantics should reflect its constructs — what you see is what you get. The four-valued logic, described in section 2.1.4, has the last two properties since it is based on the language of standard first-order logic. In this section, a decidable inference mechanism [31] based on t-entailment is described in detail. This satisfies the first two properties, making four-valued logic a good candidate for knowledge representation.

5.1 Inference Using t-entailment

t-entailment emphasises conjunctions and universal quantifiers which tend to be used in knowledge representation systems, so it seems a better choice than f-entailment or tf-entailment for an inference mechanism for Mantra. What is needed is a decidable algorithm to compute the t-entailment —to decide whether $\alpha \rightarrow_t \beta$ or $\alpha \not\rightarrow_t \beta$. Such an algorithm was described by Patel-Schneider:

t-entailment Algorithm Theorem 1 *[31, page 378] Let α be a formula in conjunctive normal form with no existentially quantified variables, $\alpha = \forall \vec{z} \wedge \alpha_j$. Let β be a formula in t-quantifier normal form. Let λ be a substitution that maps each universal variable, y of β into $f_y(E(y))$, where $E(y)$ is a sequence of the existentially bound variables of β dominating y, and f_y is a unique new funciion symbol called the Skolem function for y. Let β' be the matrix of β in conjunctive normal form, $\beta' = \wedge\beta_i$. Let $\vec{x}$*

be some ordering of the existentially quantified variables in β. *Then* $\alpha \rightarrow_t \beta$ *iff there exists* θ, *a substitution of ground terms for* $\vec{x}$, *such that for each* β_i *there exists some* α_j *and* ψ, *a substitution of ground terms for* $\vec{z}$, *such that* $\alpha_j\psi \subseteq \beta_i\lambda\theta$, *and moreover, if* x , *an existentially bound variable of* β, *and* y, *a universally bound variable of* β, *are bound in different branches of some disjunct in* β *(independent in* β*), then* f_y *does not occur in* $\theta(x)$.

How does t-entailment apply to Mantra? Any stored fact is an α formula and any question is a β formula. If $\alpha \rightarrow_t \beta$ then the question (β) follows from the stored facts (α). Hence, the answer for the question β is "yes". If, for all facts α, $\alpha \not\rightarrow_t \beta$, then the answer is "no".
Certain equivalences which are valid in standard first order logic, are not valid in four-valued logic. Firstly, quantifiers cannot be moved around and combined in all the ways of standard first order logic. For example, $\exists x Px \vee \exists y Qy$ is not equivalent to $\exists z(Pz \vee Qz)$ it is only equivalent to $\exists x \exists y(Px \vee Qy)$, the reason behind this is the fact that formulae are evaluated in compatible sets of situations. A set S$=\{s_1,s_2\}$ of compatible situations with domain D$=\{d_1,d_2\}$ might support the truth of $\exists x Px \vee \exists y Qy$ and $\exists x \exists y(Px \vee Qy)$ (s_1 supports the truth of $P(d_1)$ and s_2 supports the truth of $Q(d_2)$) but S might not support the truth of $\exists z(Pz \vee Qz)$. Secondly, in four-valued logic there are formulae which cannot be converted into an equivalent prenex normal form. The reason behind this is that by just moving quantifiers, the scope changes and so does the interpretation of the formula. For example, if $\forall y \exists x Pyx \vee \forall y' \exists x' Qy'x'$ is transformed into its prenex form $\forall y \exists x \forall y' \exists x'(Pyx \vee Qy'x')$ then the two are not equivalent because the prenex form will be true in more compatible sets of situations than the original formula. To overcome this problem, the formula is skolemized first then it is prenexed.
The transformations and the normal forms considered for the t-entailment algorithm retain t-entailment and are defined as follows:

- α is transformed into a conjunctive normal form with only universal variables: $\forall \vec{z} \wedge \alpha_j$ where $\vec{z}$ is some ordering of all the universally bound variables in α. This tranformation is done similarly to classical logic.

- β is transformed into t-quantifier normal form with only existential variables and its matrix is in CNF: $\exists \vec{x} \wedge \beta_i$ where $\vec{x}$ is some ordering of all the existentially bound variables in β.

The steps of the t-entailment algorithm follow directly from the theorem:

1. Transform α into CNF: $\forall\vec{z} \wedge \alpha_j$
2. Transform β into t-quantifier normal form: $\exists\vec{x} \wedge \beta_i$
3. For each β_i find a suitable α_i and find substitutions θ_i of variables in $\vec{x}$ and substitutions ψ_i of variables in $\vec{z}$. Such that $\alpha_i\psi_i \subseteq \beta_i\theta_i$ when considered as sets of literals.
4. The substitutions θ_i must satisfy a technical condition: for all $x \in \vec{x}$, $\theta(x)$ does not contain the skolem function f_y of any variable y independent of x.
5. $\alpha \rightarrow_t \beta$ iff, for each i, one can find α_i, θ_i, ψ_i satisfying 3. and 4.

We present two examples to illustrate t-entailment.
Example 1: Let $\alpha = \forall t(shape(t, square) \wedge colour(t, red))$ and $\beta = \exists x(shape(box, x))$. Choose $\psi = [t/box]$ and $\theta = [x/square]$. Then $\{shape(t, square)\}\psi \subseteq \{shape(box, x)\}\theta$. Hence, $\alpha \rightarrow_t \beta$.
Example 2: Consider the fact $\forall z \forall z' \exists w \exists w'(Pzw \vee Qz'w')$ and the query $\forall y \exists x Pyx \vee \forall y' \exists x' Qy'x'$. In normal form this gives:
$\alpha = \forall z \forall z'(Pzf_w(z, z') \vee Qz'f_{w'}(z, z'))$
$\beta = \exists x \exists x'(Pf_y x \vee Qf_{y'}x')$
The candidate substitutions are
$\psi = \{[z/f_y], [z'/f_{y'}]\}$ and $\theta = \{[x/f_w(f_y, f_{y'})], [x'/f_{w'}(f_y, f_{y'})]\}$. However, x and y' are independent, and $f_{y'}$ occurs in $\theta(x)$. This violates the technical condition. Hence, $\alpha \not\rightarrow_t \beta$.
In an inference mechanism for a knowledge base, the question may contain (existential) variables and the inference should return all possible instantiations of the variables that satisfy the question. So the implementation of t-entailment should enumerate all possible choices of ψ and θ.
Example 3: Consider the facts:
Bird(tweety) $\wedge$ Size(tweety,small) $\wedge$ $\forall C$ (Colour(tweety,C))

Bird(birdy) $\wedge$ Size(birdy,small) $\wedge$ $\forall C$ (Colour(birdy,C))
and the question:
$\exists X$ (Bird(X) $\wedge$ Size(X,small) $\wedge$ (Colour(X,red) $\vee$ Colour(X,blue))
This will give the following four solutions:
$Solution_1 = \{[X/tweety], [C/red]\}$
$Solution_2 = \{[X/tweety], [C/blue]\}$
$Solution_3 = \{[X/birdy], [C/red]\}$
$Solution_4 = \{[X/birdy], [C/blue]\}$

5.2 Implementation of the Inference Mechanism

5.2.1 Object Model

Figure 31 shows all the classes involved in the inference mechanism implementation. Most of the classes and associations described on this figure were already defined in the separate sections of CNF and unification implementations. The new classes introduced are: INFER and SUBSTITUTION. And the new associations are: INFER verifies the t-entailment of formulae and uses the UNIFICATION ALGORITHM to compute this entailment, the UNIFICATION ALGORITHM produces the SUBSTITUTION necessary to unify two terms, and SUBSTITUTION generated is the answer given by INFER whenever a fact (α) t-entails a question (β). The new classes and their associations are shown separately with their attributes and methods in Figure 32.

5.2.2 Data Structures

The major data structures used are briefly described below :

1. Substitution is a structure:

   - Substitutions: a pointer to a List of multi-equations. Each multi-equation is in a solved form hence, it defines a substitution of the variables in its S part to the term in its M part.
   - Status: enumeration {Unsolved, Solved-yes, Solved-no}. The status of a substitution indicates whether the substitution is unsolved —is still a choice point, or solved giving a "yes" or "no" answer —can be reported as a solution.

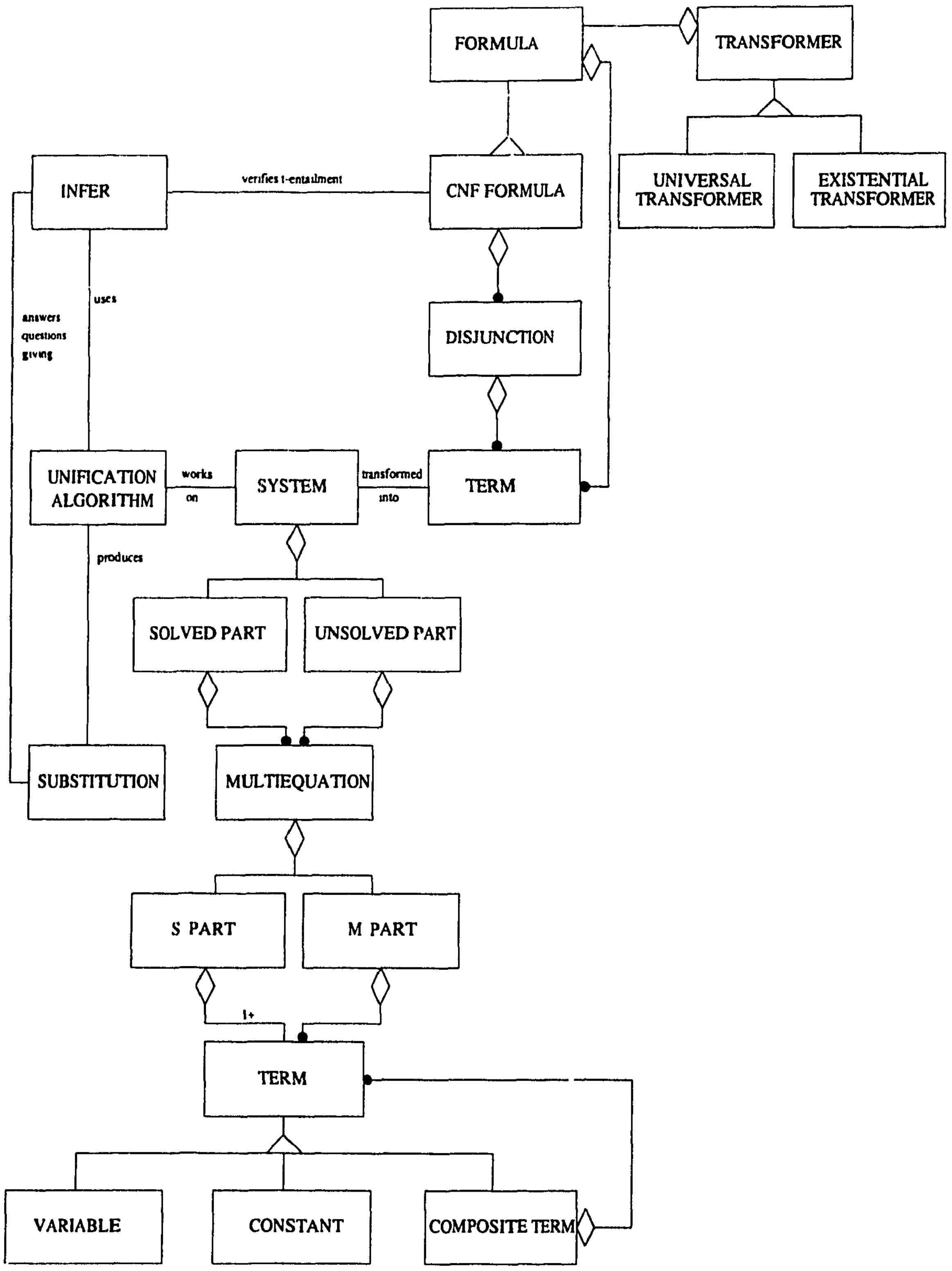


Figure 31: Global Object Model

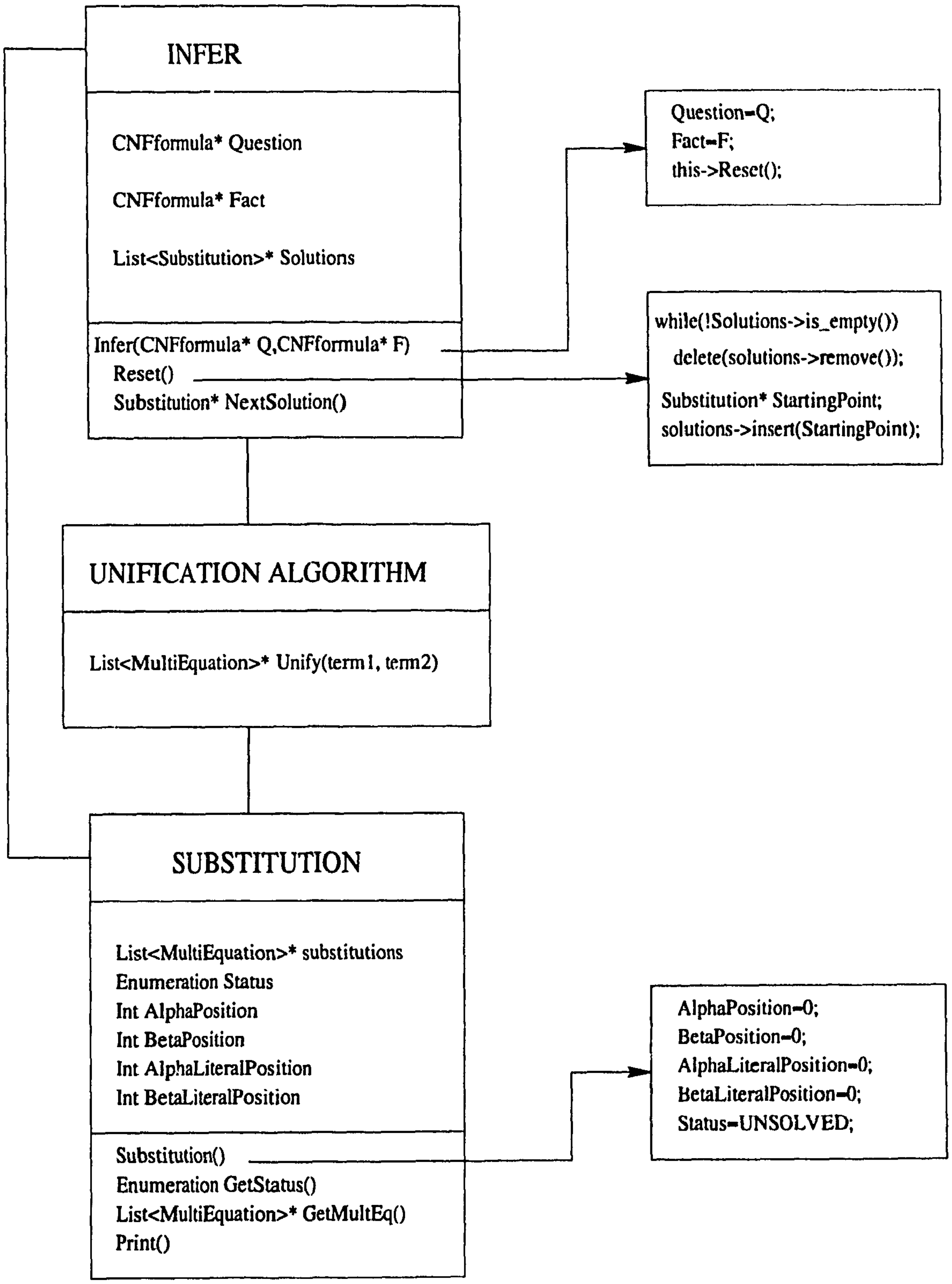


Figure 32: Inference Object Model Details

- BetaPosition: integer. When substitution is a choice point, BetaPosition indicates the position, i, of β_i where the choice occurred —the position of the disjunction in the query.
- AlphaPosition: integer. When substitution is a choice point, AlphaPosition indicates the position, j, of α_j where the choice occurred —the position of the disjunction in the fact.
- AlphaLiteralPosition: integer. When substitution is a choice point, AlphaLiteralPosition indicates the position, n, of $literal_{jn}$ where the choice occurred —the position of the literal in a disjunction α_j in the fact .
- BetaLiteralPosition: integer When substitution is a choice point, BetaLiteralPosition indicates the position, m, of $literal_{im}$ where the choice occurred —the position of the literal in a disjunction β_i in the query.

2. Infer is a structure:

   - Question: a pointer to a CNFFormula
   - Fact: a pointer to a CNFFormula
   - Solutions: a pointer to a list of Substitutions. Each substitution is either an answer or is a choice point that needs to be solved.

5.2.3 Data Dictionary

- **Unification Algorithm**
  - **Description**

    It takes two terms and returns either null if the two terms cannot be unified, or a list of multi-equations (unifier) where each multi-equation is a substitution of a term (M part of a solved multi-equation) for a variable (S part of the multi-equation), to make the two terms equal. If the list is empty, the terms are equal.

- **Infer**
  - **Description**

    Defines the inference mechanism which takes a fact and a question and returns an answer to the question. In the case of more than one answer for a given fact and question, Infer keeps track of all possible answers and returns all of them if it is asked to.

– **Attributes**

1. Question: the question for which an Infer object is instantiated and on which the inference mechanism is currently working. A question is a formula in t-quantifier normal form, given to the inference mechanism to be answered based on the facts stored in the knowledge base. The answer to the question could be either no or yes with the necessary substitutions. For the t-entailment algorithm, question is β.
2. Fact: the fact or facts used to answer a particular question during the current instantiation of an Infer object. A fact is a formula stored as a conjunction of disjunctions (CNF). It could be a collection of facts relevant to the question asked and grouped under one fact. Each fact being a conjunction of disjunctions, one global fact could hold all the disjunctions. For the t-entailment algorithm, fact is α.
3. Solutions: whenever for a given fact and question there is more than one answer, Infer keeps track of the choice points in Solutions (a list), then each choice is handled one at a time to check whether it is a potential answer or not. In Solutions, a new choice point is added at the end, and removed from the top in order to be handled (FIFO strategy).

   A choice point holds all the information required to get to that choice:

   (a) The choice of fact α_j

   (b) The choice of query β_i

   (c) The choice of literal in α_j

   (d) The choice of literal in β_j

   (e) The substitutions determined so far

   (see Section 5.2.4 for details on choice points).

   When Infer continues handling a choice point, it will either add the substitutions necessary so the choice point is a complete answer, or if it fails then the choice point is not a correct answer. A choice point is a reference to a substitution, and when it is solved it could be given as an answer.

- **Methods**

  1. Infer: given a question and a fact, Infer is the constructor to set the question and fact attributes equal to those passed to Infer, then a first empty choice point (starting point) is inserted into solutions in order to be able to start the inference mechanism. Infer initialises the inference mechanism with a given fact and question.
  2. Reset: resets the inference mechanism to start its search for answers from the beginning. For the same question and fact, it sets the choice points list (solutions) to empty.
  3. NextSolution: is the most important method. It returns the next answer (whether it is no or yes) or null if there aren't any choice points left in solutions list. NextSolution implements the t-entailment algorithm theorem. Each time it is called, it works on the current choice point on top of the solutions list. If during its call new choice points are found, they are inserted into the solutions list to be solved by the subsequent NextSolution calls. NextSolution traverses the proof-tree in breadth-first order.

- **Substitution**

  - **Description**

    A Substitution instance is a choice point as well as a substitution. It is either solved and could be given as an answer (yes or no), or partially solved. A substitution instance is a unifier which holds a list V of substitutions. $V = ([x_1/t_1], ..., [x_n/t_n])$ where a variable x_i is substituted by a term t_i. Substitutions are produced by the unification algorithm (list of solved multi-equations) whenever two terms are unified.

  - **Attributes**

    1. Substitutions: list of solved multi-equations, each multi-equation defines a substitution of variables in its S part to a term in its M part. It is one unifier.

2. Status: the choice point is either unsolved, or solved giving a yes answer, or solved giving a no answer.
3. BetaPosition, AlphaPosition, AlphaLiteralPosition, BetaLiteralPosition: define the exact position where the choice point occurred. When NextSolution solves this particular choice point, it knows exactly where to continue.

– **Methods**

1. Print: prints a substitution. If it is solved with a yes answer then it prints the unifier. Otherwise, it only prints a no answer.
2. Substitution: constructs a substitution instance with an empty list of multi-equations, all positions set to zero and an unsolved status.
3. GetStatus and GetMultEq: return the status and the unifier respectively.

5.2.4 Pseudocode

The algorithm to compute $\alpha \rightarrow_t \beta$ is shown in Figure 33.

The notation used is as follows:

- α is in CNF, $\alpha = \alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge \ldots \wedge \alpha_n$ where $n \geq 1$, and where each $\alpha_j = s_{j1} \vee s_{j2} \vee s_{j3} \vee \ldots \vee s_{jm_j}$ for $m_j \geq 1$
- β is in t-quantifier normal form, $\beta = \beta_1 \wedge \beta_2 \wedge \beta_3 \wedge \ldots \wedge \beta_k$ where $k \geq 1$ and where each $\beta_i = t_{i1} \vee t_{i2} \vee t_{i3} \vee \ldots \vee t_{il_i}$ for $l_i \geq 1$

The pseudocode enumerates all solutions. A solution records the choice of α_j for each β_i, and a substitution Θ such that

$$\{\ s_{j1}, s_{j2}, \ldots, s_{jm_j}\ \}\ \Theta \subseteq \{\ t_{i1}, t_{i2}, \ldots, t_{il_i}\ \}\ \Theta$$

The information returned to the user is the substitution Θ.
By recording choice points, the algorithm can be modified to return solutions one at a time, either in breadth-first or depth-first order.
Choice points occur at two levels: At the alpha level, a choice occurs whenever more

```
For each $\beta_i$ of $\beta$
      For each $\alpha_j$ of $\alpha$
            For each literal $s_{jp}$ of $\alpha_j$
                  For each literal $t_{iq}$ of $\beta_i$
                     Find $\Theta_{i.q,j.p}$ which unifies $t_{iq}\Theta$ with $s_{jp}\Theta$
                        where $\Theta$ is the composition of previous $\Theta_{ij}$
                  EndFor
                  If no $\Theta_{i.q,j.p}$ is found then
                     $\alpha_j \not\subseteq \beta_i$
                     move to the next $\alpha_j$
            EndFor
            $\Theta_{ij}$ is the composition of $\Theta_{i.q,j.p}$
            and $\alpha_j\Theta_{ij} \subseteq \beta_i\Theta_{ij}$
      EndFor
      If no $\Theta_{ij}$ is found then
         no $\alpha_j$ is $\subseteq$ $\beta_i$
         $\alpha \not\to_t \beta$
EndFor
Solution is a composition of $\Theta_{ij}$
```

Figure 33: t-entailment Algorithm

than one α_j is found for the same β_i such as, after unification, $\alpha_j \subseteq \beta_i$. At the literal level, a choice occurs whenever more than one literal t_{il_i} in β_i matches the same literal s_{jm_j} in α_j. In terms of the algorithm in Figure 33, the match of s_{jp} to t_{iq} is made by the substitution $\Theta_{i,q,j,p}$.

5.3 Inference Mechanism Examples

Two examples are described to illustrate how the inference mechanism works given a question to answer based on a set of facts:

Example 1 illustrates the occurence and the handling of choice points on both levels. The following two facts are given:

Fact 1 describes tweety as a bird whose size is small and whose colour could be anything.

Fact 1: Bird(tweety) $\wedge$ Size(tweety,small) $\wedge$ $\forall C$ (Colour(tweety,C))

Fact 2 describes birdy as a bird whose size is small and whose colour could be anything.

Fact 2: Bird(birdy) $\wedge$ Size(birdy,small) $\wedge$ $\forall C$ (Colour(birdy,C))

The question is whether there exists a bird whose size is small and whose colour is either red or blue.

Question: $\exists X$ (Bird(X) $\wedge$ Size(X,small) $\wedge$ (Colour(X,red) $\vee$ Colour(X,blue))

Tranformed into:

Facts:

1. α_1: s_{11}: Bird(tweety)

2. α_2: s_{21}: Size(tweety,small)

3. α_3: s_{31}: Colour(tweety,C)

4. α_4: s_{41}: Bird(birdy)

5. α_5: s_{51}: Size(birdy,small)

6. α_6: s_{61}: Colour(birdy,C)

Question:

1. β_1: t_{11}: Bird(X)

2. β_2: t_{21}: Size(X,small)

3. β_3: t_{31}: Colour(X,red), t_{32}: Colour(X,blue)

1. Choice Point0:

   (a) β_1

      i. β_1 and α_1

         s_{11} = Bird(tweety) unified with t_{11} = Bird(X) gives $\Theta_{1.1,1.1}$: X=tweety

      ii. β_1 and α_2

         s_{21} = Size(tweety,small) unified with t_{11} = Bird(X) gives $\Theta_{1.1,2.1}$ Null

      iii. β_1 and α_3

         s_{31} = Colour(tweety,C) unified with t_{11} = Bird(X) gives $\Theta_{1.1,3.1}$ Null

      iv. β_1 and α_4

         s_{41} = Bird(birdy) unified with t_{11} = Bird(X) gives $\Theta_{1.1,4.1}$: X=birdy
         Choice Point1 on α level because β_1 already found a match with α_1

      v. β_1 and α_5

         s_{51} = Size(birdy,small) unified with t_{11} = Bird(X) gives $\Theta_{1.1,5.1}$ Null

      vi. β_1 and α_6

         s_{61} = Colour(birdy,C) unified with t_{11} = Bird(X) gives $\Theta_{1.1,6.1}$ Null

      β_1 matched α_1 and α_3

   (b) β_2

      i. β_2 and α_1

         s_{11} = Bird(tweety) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,1.1}$ Null

      ii. β_2 and α_2

         s_{21} = Size(tweety,small) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,2.1}$ Empty

      iii. β_2 and α_3

         s_{31} = Colour(tweety,C) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,3.1}$ Null

      iv. β_2 and α_4

         s_{41} = Bird(birdy) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,4.1}$ Null

v. β_2 and α_5

s_{51} = Size(birdy,small) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,5.1}$ Null

vi. β_2 and α_6

s_{61} = Colour(birdy,C) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,6.1}$ Null

β_2 matched α_2

(c) β_3

i. β_3 and α_1

s_{11} = Bird(tweety) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,1.1}$ Null

s_{11} = Bird(tweety) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,1.1}$ Null

ii. β_3 and α_2

s_{21} = Size(tweety,small) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,2.1}$ Null

s_{21} = Size(tweety,small) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,2.1}$ Null

iii. β_3 and α_3

s_{31} = Colour(tweety,C) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,3.1}$: C=red

s_{31} = Colour(tweety,C) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,3.1}$: C=blue Choice Point2 on literal level because s_{31} already found a match with t_{31}

iv. β_3 and α_4

s_{41} = Bird(birdy) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,4.1}$ Null

s_{41} = Bird(birdy) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,4.1}$ Null

v. β_3 and α_5

s_{51} = Size(birdy,small) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,5.1}$ Null

s_{51} = Size(birdy,small) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,5.1}$ Null

vi. β_3 and α_6

s_{61} = Colour(birdy,C) unified with t_{31} = Colour(tweety,red) gives $\Theta_{3.1,6.1}$ Null

s_{61} = Colour(birdy,C) unified with t_{32} = Colour(tweety,blue) gives $\Theta_{3.2,6.1}$ Null

β_3 matched α_3 on t_{31} and t_{32}

Solution1 is: $\Theta_{1.1,1.1}$: X=tweety and $\Theta_{2.1,2.1}$ Empty and $\Theta_{3.1,3.1}$: C=red
$\Theta_1 = \{[X/tweety],[C/red]\}$

2. Choice Point1:

(a) β_2

i. β_2 and α_1

s_{11} = Bird(tweety) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,1.1}$ Null

ii. β_2 and α_2

s_{21} = Size(tweety,small) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,2.1}$ Null

iii. β_2 and α_3

s_{31} = Colour(tweety,C) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,3.1}$ Null

iv. β_2 and α_4

s_{41} = Bird(birdy) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,4.1}$ Null

v. β_2 and α_5

s_{51} = Size(birdy,small) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,5.1}$ Empty

vi. β_2 and α_6

s_{61} = Colour(birdy,C) unified with t_{21} = Size(birdy,small) gives $\Theta_{2.1,6.1}$ Null

β_2 matched α_5

(b) β_3

i. β_3 and α_1

s_{11} = Bird(tweety) unified with t_{31} = Colour(birdy,red) gives $\Theta_{3.1,1.1}$ Null

s_{11} = Bird(tweety) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,1.1}$ Null

ii. β_3 and α_2

s_{21} = Size(tweety,small) unified with t_{31} = Colour(birdy,red) gives $\Theta_{3.1,2.1}$ Null

s_{21} = Size(tweety,small) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,2.1}$ Null

iii. β_3 and α_3

s_{31} = Colour(tweety,C) unified with t_{31} = Colour(birdy,red) gives $\Theta_{3.1,3.1}$ Null

s_{31} = Colour(tweety,C) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,3.1}$ Null

iv. β_3 and α_4

s_{41} = Bird(birdy) unified w[illegible]$_{31}$ = Colour(birdy,red) gives $\Theta_{3.1,4.1}$ Null

s_{41} = Bird(birdy) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,4.1}$ Null

v. β_3 and α_5

s_{51} = Size(birdy,small) unified with t_{31} = Colour(birdy,red) gives $\Theta_{3.1,5.1}$ Null

s_{51} = Size(birdy,small) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,5.1}$ Null

vi. β_3 and α_6

s_{61} = Colour(birdy,C) unified with t_{31} = Colour(birdy,red) gives $\Theta_{3.1,6.1}$: C=red

s_{61} = Colour(birdy,C) unified with t_{32} = Colour(birdy,blue) gives $\Theta_{3.2,6.1}$: C=blue Choice Point3 on t level because s_{61} already found a match with t_{31}

β_3 matched α_6 on t_{31} and t_{32}

Solution2 Θ is: $\Theta_{1.1,4.1}$: X=birdy and $\Theta_{2.1,5.1}$ Empty and $\Theta_{3.1,6.1}$: C=red
$\Theta_2 = \{[X/birdy], [C/red]\}$

3. Choice Point2:
   Solution3 Θ is: $\Theta_{1.1,1.1}$: X=tweety and $\Theta_{2.1,2.1}$ Empty and $\Theta_{3.2,3.1}$: C=blue
   $\Theta_3 = \{[X/tweety], [C/blue]\}$

4. Choice Point3:
   Solution4 Θ is: $\Theta_{1.1,4.1}$: X=birdy and $\Theta_{2.1,5.1}$ Empty and $\Theta_{3.2,6.1}$: C=blue
   $\Theta_4 = \{[X/birdy], [C/blue]\}$

Example 2 shows the possibility of having a no answer. The following fact and question are given:
The fact is that tweety is a bird whose size is small.
Fact: Bird(tweety) $\wedge$ Size(tweety,small)
The question is whether there exist a bird whose size is large.
Question: $\exists X$ (Bird(X) $\wedge$ Size(X,large))
Tranformed into:
Facts:

1. α_1: s_{11} : Bird(tweety)

2. α_2: s_{21} : Size(tweety,small)

Question:

1. β_1: t_{11} : Bird(X)

2. β_2: t_{21} : Size(X,big)

1. β_1

   (a) β_1 and α_1
   s_{11} = Bird(tweety) unified with t_{11} = Bird(X) gives $\Theta_{1.1,1.1}$: X=tweety

   (b) β_1 and α_2
   s_{21} = Size(tweety,small) unified with t_{11} = Bird(X) gives $\Theta_{1.1,2.1}$ Null

   β_1 matched α_1

2. β_2

   (a) β_2 and α_1

   s_{11} = Bird(tweety) unified with t_{21} = Size(tweety,small) gives $\Theta_{2.1,1.1}$ Null

   (b) β_2 and α_2

   s_{21} = Size(tweety,small) unified with t_{21} = Size(tweety,large) gives $\Theta_{2.1,2.1}$ Null

   β_2 did not match any α_i so there is no solution

5.4 Evaluation of the Inference Mechanism

The evaluation of the inference mechanism is summarized as follows: It is decidable, the t-entailment algorithm described above always terminates either with a yes or no answer for any fact and question. It is sound and complete with respect to the four-valued semantics. It automatically gives answers which are the substitutions, rather than just yes or no. If more than one answer or the complete set of answers is required, all different substitutions are searched for. In terms of implementation, this is done by calling NextSolution() to get the next possible answer.
A drawback of this inference mechanism is that its worst case running time is exponential in $|\alpha| * |\beta|$ where the magnitude of α and β is the number of literals in each [31, page 380]. However, in most cases, a knowledge representation system will have the following reasonable assumptions [31, page 380]:

Facts are already in CNF. If they are not, converting them to normal form will not increase their size significantly.

Knowledge base has different predicates, thus only a small number will require unification.

Queries are small even when converted to CNF.

Facts are stored as short clauses (disjunctions).

Under these assumptions, running time is in order $\ln k * k^{\ln \ln k}$ where k is the size of the knowledge base.

Being based on four-valued logic, the inference mechanism does not admit the disjunctive syllogism nor any type of chaining, such as modus ponens. It also does not do reasoning by contradiction or reasoning by cases. A knowledge representation using only t-entailment is weak. One way to make it stonger is to have a stronger domain specific method that could be used whenever entailment fails. The advantage of this is that entailment can explain what its failure to produce an answer means, since it is a semantically motivated and well-defined notion. This explanation could be used by the next stronger method which could be first order logic for instance [29, page 458].

Chapter 6

Conclusion

In this thesis, a unification module and an inference mechanism based on t-entailment were designed in the Object Oriented paradigm making use of design patterns, and then implemented in C++.

Some important topics had to be studied first in order to gather the necessary information required in the actual work of the thesis. The second step, was the design and implementation of the transformation of a formula into CNF. The design uses design patterns which make it reusable in other programs or applications. The third step, was the understanding of the unification concept and the design and implementation of the unification module of Mantra. The unification module is at the heart of the inference mechanism and it is used by the three formalisms in Mantra: logic, semantic nets, and frames. It was designed using the iterator pattern, and a combination of the composite and visitor patterns. Its implementation is general to accept any two terms to be unified. Testing was done using examples from Martelli and Montanari paper [25], some logic books [16, 17, 19, 37] and inputs prepared by the author and her supervisor. The final step was the building of the inference mechanism based on t-entailment, which used the CNF transformation and the unification module. This mechanism is specific to the logic formalism, it provides a decidable way to answer any given question based on the facts stored in the logic formalism of Mantra. We are confident about the correctness of this mechanism, we have tested it with all the examples in Mantra manual [8], some examples from artificial intelligence books [28, 34], and some tests prepared by the author and her supervisor.

The importance of this work lies in some major aspects: The selection of the algorithms and the use the Object Oriented paradigm in the design and implementation phases.
The unification algorithm selected, based on Martelli and Montanari work, has been shown to have a better performance than many well-known algorithms [25, page 44]. It has a linear-time complexity [25, page 44]. The inference mechanism based on t-entailment is sound, complete and decidable. Its performance in the circumstances we are working in is typically in the order of $\ln k * k^{\ln \ln k}$, where k is the size of the knowledge base. Besides the good performance of both selected algorithms, their design and implementation being Object Oriented makes them easier to modify and extend, and reusable in different applications. By using the OMT standard notations in describing the design and by using design patterns, understanding the work and using it by others becomes easier. There is a kind of common design vocabulary and documentation within the patterns, and the OMT notations are simple, standard, and they clearly illustrate the most common Object Oriented features.

No work is achieved without encountering difficulties. In the design phase, the selection of the design pattern that will best fit into each situation was sometimes not easy, especially when identifying the objects and the roles they play in the pattern. Besides, changing my concept of programming from the conventional programming where data structure and behaviour are only loosely connected and where finding a solution to a problem was in terms of functions, to Object Oriented programming where objects combine both data structures and behaviour and all the thinking is in terms of objects rather than functions, was difficult at the beginning.

Although the implementation is robust, it has some limitations and drawbacks:
The unification algorithm complexity, when there is a very high probability of stopping with success, is nonlinear. However, when unifying small terms, the complexity tends to become linear.
The worst-case complexity of the t-entailment algorithm is very bad. It is exponential in $|\alpha| * |\beta|$ when computing $\alpha \rightarrow_t \beta$.
The t-entailment algorithm is weak being based on four-valued logic, which does not support all the classical logic rules. For example: If the following two facts are stored: $\neg student(maurice) \vee father(tania, maurice)$ and $student(maurice)$, then

asked $father(tania, maurice)$, the implemented inference mechanism answers "no", while in classical logic, by applying modus ponens, $father(tania, maurice)$ could be derived.

Mantra provided us with the context for this work. The future work will be to reengineer the semantic-nets and frame formalisms at the epistemological level and their inference mechanisms at the logical level. This could be done either by converting the semantic-nets and frame knowledge into logic and using the implemented inference mechanism, or by implementing a separate mechanism for each formalism based on the four-valued semantics.
I endorse an important suggestion given by Patel-Schneider in [29, page 458]: "Any knowledge representation system based on four-valued logic is going to be quite weak. One way of making such a system stronger is to make entailment be only the first method for answering questions. If entailment fails to produce an answer then a stronger method could be used, perhaps some domain specific method", which could also be considered for future work.

Bibliography

[1] R.P. Abelson and R.C. Schank. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Assoc., 1977.

[2] R. Ackermann. *Introduction to Many Valued Logics.* Dover Publications Inc., 1967.

[3] M. Arbib, A.J. Kfoury, and R. Moll. *A Programming Approach to Computability.* Springer Verlag, 1982.

[4] N.D Belnap. How a computer should think. In G. Ryle, editor, *Contemporary Aspects of Philosophy Proceedings of the Oxford International Symposium*, pages 30–56. Oriel Press, 1975.

[5] Wolfgang Bibel. *Automated Theorem Proving.* Vieweg & Sohn, 1987.

[6] G. Bittencourt. A hybrid system architecture and its unified semantics. In *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 150–158, Charlotte North Carolina USA, October 12–14 1989.

[7] G. Bittencourt. The integration of terminological and logical knowledge representation languages. In *Proceedings of the 5th International Symposium on Methodologies for Intelligent Systems*, pages 226–234, Knoxville Tennessee, October 25–27 1990.

[8] G. Bittencourt. *The Mantra Reference Manual.* Universität Karlsruhe, Germany, 12 January 1990.

[9] G. Bittencourt, J. Calmet, and I.A. Tjandra. Mantra: A shell for hybrid knowledge representation. In *Tools for Artificial Intelligence.* IEEE Computer Society Press, 1991.

[10] M. Blaha, F. Eddy, W. Lorensen, W. Premerlani, and J. Rumbaugh. *Object Oriented Modeling and Design.* Prentice Hall, 1991.

[11] Michael A. Carrico, John E. Girard, and Jennifer P. Jones. *Building Knowledge Systems.* Mc Graw Hill Book Company, 1989.

[12] Donald N. Cohen. *Knowledge Based Theorem Proving and Learning.* UMI Research Press, 1981.

[13] M. Davis. Eliminating the irrelevant from mechanical proofs. In *Proceedings of Symposia in Applied Mathematics*, volume 15. American Math. Society, 1963.

[14] M. Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80, 1973.

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[16] David Duffy. *Principles of Automated Theorem Proving.* John Wiley & Sons, 1991.

[17] Melvin Fitting. *First Order Logic and Automated Theorem Proving.* Springer Verlag, 1990.

[18] D.M Gabbay, C.J. Hogger, J.A. Robinson, and J. Siekmann. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Clarendon Press, 1994.

[19] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row Publishers, 1986.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, 1995.

[21] G. Gentzen. Investigation into logical deduction (english translation). In M.E. Szabo, editor, *Collected Papers of Gerhrad Gentzen.* North-Holland Publishing Co., 1969.

[22] Frank Van Harmelen, Peter Jackson, and Han Reichgelt. *Logic-Based Knowledge Representation.* The MIT Press, 1989.

[23] J. Herbrand. *Recherches sur la Theorie de la Demonstration.* PhD thesis, Sorbonne Paris, 1930.

[24] H. Levesque and R. Brachman. *Readings in Knowledge Representation.* Morgan Kaufmann, 1985.

[25] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[26] Marvin Minsky. A framework for representing knowledge. In Patrick H. Winston, editor, *The Psychology of Computer Vision.* Mc Graw Hill Book Company, 1975.

[27] R. Moore. The role of logic in knowledge representation and common-sense reasoning. In *Proceedings of American Association for Artificial Intelligence 82*, pages 428–433, Pittsburg, Pa, August 1982.

[28] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[29] Peter F. Patel-Schneider. A decidable first order logic for knowledge representation. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, August 18–23, 1985.

[30] Peter F. Patel-Schneider. A four-valued semantics for frame-based description languages. In *Proceedings of AAAI-86 5th International Conference on AI*, Philadelphia, USA, August 11–15 1986. Springer-Verlag.

[31] Peter F. Patel-Schneider. A decidable first order logic for knowledge representation. *Journal of Automated Reasoning*, 6:361–388, 1990.

[32] D. Prawitz. An improved proof procedure. In *Theoria*, 1960.

[33] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison Wesley Publishing Company, 1995.

[34] Elaine Rich. *Artificial Intelligence.* Mc Graw Hill Book Company, 1983.

[35] David C. Rine. *Computer Science and Multiple Valued Logic Theory and Applications.* North-Holland Publishing Company, 1977.

[36] J.A Robinson. A machine-oriented logic based on resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23-41, January 1965.

[37] Uwe Schoning. *Logic for Computer Scientists.* Birkhauser, 1989.

[38] J.H Siekmann and P. Szabo. Universal unification. In Wolfgang Wahlster, editor, *6th German Workshop on Artificial Intelligence*, pages 102–141. Springer-Verlag, September 27-October 1, 1982.

[39] Jorg H. Siekmann. Universal unification. In R.E. Shostak, editor, *7th International Conference on Automated Deduction*, Napa California USA, May 14–16, 1984. Springer-Verlag.

[40] P. Trum and G. Winterstein. Description, implementation, and practical comparison of unification algorithms. Internal Rep. 6/78, Fachbereich Informatik, Univ. of Kaiserlautern, Germany.

[41] T. Winograd. *Understanding Natural Language.* Academic Press, 1972.

[42] T. Winograd. A procedural model of language understanding. In K.M. Colby and R.C. Schank, editors, *Computer Models of Tought and Language.* Freeman, 1973.

[43] A.A. Zinov'ev. *Philosophical Problems of Many Valued Logic.* D. Reidel Publishing Company, 1963.