QUERY PROOF STRUCTURE CACHING FOR INCREMENTAL EVALUATION OF TABLED PROLOG PROGRAMS

TAHER MUHAMMAD ALI

THESIS SUBMITTED IN FULFILLMENT

OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

UNIVERSITY OF MALAYA

KUALA LAMPUR

2013

Abstract

PROLOG is the most well known, widely used programming language for logic programming. PROLOG is a programming language that uses a small set of basic mechanisms to create surprisingly powerful programs. These mechanisms are pattern-matching, tree-based data structuring and backtracking. PROLOG is used for the development of scheduling systems, knowledge systems, expert systems and many other applications. However, being goal-driven (query driven) PROLOG'S query engine suffers from some well-known problems such as susceptibility to infinite looping, repeated subcomputation and unsatisfactory semantics of negation. These limitations have been addressed by the tabled extensions (TLP) to PROLOG evaluation. The main idea of tabling is to cache the answers for computations and then reuse them when a repeated computation appears. The TLP approach assumes that the database of facts/rules, of which the query results drawn from, remains constant through time and therefore the tabulated results are assumed to remain valid. This prohibits the use of TLP in non-monotonic logic where such an assumption cannot be guaranteed. To overcome this problem the idea of incremental tabulation has been introduced. An incrementally maintained table is one that continually contains the correct answers in the presence of updates to underlying predicates on which the tabled predicate depends. The critical challenges for incremental evaluation are how to detect and update the tabled answers due to the changes in the database of facts and rules associated with the tabled answers.

This thesis presents an alternative approach to incremental tabulation that is capable of working in non-monotonic situations. The basic idea of our approach is that, as the PROLOG inference engine evaluates the query for the first time, the answers returned by the engine for the query are converted into a justification-based truth-maintenance system (JTMS) network. Every successful branch is translated into a JTMS network that links the facts used in the proof branch to the answer generated by that branch. When the same query is re-evaluated, the query engine collects the valid answers for the query at that moment and returns them by using the cached proof structure for the query. When the state of database is changed, the JTMS component propagates the effect of the change through its network to maintain the consistency of old proofs. At the same time, the database monitor keeps watching the addition of the new data (facts/rules) to the system and triggers the resumption of previously proven queries in order to update their proof structure.

The outcome of this thesis is JLOG. JLOG is a sub-system integrated with PROLOG inference engine which supports incremental tabled extensions to PROLOG evaluation. The thesis presents the design, necessary data structures, algorithms and implementation details of JLOG. The results of comparing the performance of JLOG to regular PROLOG and Tabled PROLOG systems are also presented.

Abstrak

Prolog adalah satu bahasa pengaturcaraan yang diketahui ramai dan digunakan secara meluas untuk program logik. Prolog menggunakan satu set kecil mekanisme asas untuk menghasilkan program yang hebat. Mekanisme ini adalah pemadanan corak (pattern-matching), penstrukturan data berasaskan pepohon (tree-based data structuring) dan pengulangan semula (backtracking) Prolog digunakan untuk pembangunan sistem penjadualan, sistem pengetahuan, sistem pakar dan lain-lain aplikasi-aplikasi. Walau bagaimanapun, sebagai program didorongi oleh matlamat (goal-driven) enjin pertanyaan Prolog menimbulkan beberapa masalah yang biasa seperti kecenderungan kepada gelung tidak terhingga (infinite looping), sub pengiraan (subcomputation) yang berulang dan penafian semantik yang tidak memuaskan (unsatisfactory semantics of negation). Kelemahan-kelemahan ini telah diperbaiki oleh lanjutan berjadual (TLP) kepada penilaian Prolog. Tujuan utama penjadualan ialah untuk menyimpan jawapan bagi perhitungan yang akan diguna kembali bila perhitungan yang sama diperlukan. TLP adalah satu teknik pelaksanaan di mana jawapan bagi sub pengiraan dikekalkan dan digunakan semula apabila muncul pengiraan yang berulang-ulang. Pendekatan TLP mengandaikan bahawa pangkalan data fakta / peraturan, dari mana keputusan pertanyaan diperolehi, kekal sepanjang masa dan oleh itu keputusan berjadual diandaikan kekal sah. Tetapi, andaian ini tidak semestinya benar, maka adalah tidak sesuai LTP digunakan dalam logik bukan monotonik (non-monotonic logic). Bagi mengatasi masalah ini satu ide berkenaan penjadualan tambahan telah diperkenalkan. Jadual yang dikekalkan secara berperingkat adalah suatu yang sentiasa mengandungi jawapan yang betul dalam kehadiran kemaskini kepada predikat

asas bergantung kepada predikat mana yang dibentangkan Cabaran utama bagi penilaian menokok ialah bagaimana mengesan dan mengemaskini jadual jawapan disebabkan oleh perubahan di dalam pangkalan data untuk penyataan dan petua berkaitan dengan jadual jawapan tersebut. Tesis ini membentangkan satu pendekatan alternatif untuk penjadualan menokok bagi keadaan bukan monotonik. Di dalam pendekatan ini, bila enjin inferens PROLOG menilai pertanyaan bagi kali pertama, jawapan yang berikan oleh enjin itu ditukar kepada satu rangkaian sistem pengekalan kebenaran berdasarkan justifikasi (JTMS). Setiap cadang yang berjaya diterjemahkan ke dalam satu rangakaian JTMS yang menghubungkan penyataan yang diguna di dalam cabang bukti dengan jawapan yang dihasilkan dari cabang tersebut. Bila pertanyaan yang sama dinilai semula, enjin pertanyaan itu kumpulkan jawapan yang sah bagi pertanyaan ketika itu dan kembalikannya dengan menggunakan struktur bukti yang disimpan bagi pertanyaan tersebut. Bila kandunagan pangkalan data bertukar, komponen JTMS kesan pertukaran tersebut dijana melalui rangkaian bagi mengekalkan konsistensi bukti yang asal. Pada masa yang sama, data yang ditambah di dalam pangkalan data oleh sistem dan memberi isyarat bagi memulakan semula pertanyaan yang telah dibukti bagi mengemaskini struktur buktinya. JLOG adalah satu subsistem yang dihasilkan dari tesis ini. Ia digabungkan dengan enjin inferens PROLOG yang menyokong penambahan berjadual secara menokok untuk penilaian PROLOG. Tesis ini membentangkan reka bentuk, struktur data yang berkaitan, algoritma dan butiran implimentasi untuk JLOG. Penemuan dari perbandingan pencapaian JLOG dengan PROLOG biasa serta sistem PROLOG berjadual juga dibincang.

Acknowledgments

First and foremost, I would like to thank GOD for giving me the patience during the hard times, and for guiding me to the right way in case of any doubts. I would like to express my sincerest gratitude to my supervisor, Prof. Dr. Mohd Sapiyan Bin Baba, for his continuous encouragement and guidance. His constructive support, patience, suggestions and ideas were very helpful in completing this work successfully. I thank him for making this experience a memorable one. Special thanks goes to Dr. Diptikalyan Saha for his valuable feedback in guiding the contribution of this thesis. Furthermore, I would like to thank Dr. Mohammad Omran, Mr. Fahed Al-Rifidi, Mr. Fadi Deeb, Ms. Sarah Bargal, Ms. Sara Elghandour, Ms. Mariam Kinawi, Mr. Hussain Mostafavi, Mr. Hakeem Tass and Mr. Hareesh Buddah for their support.

Last but not least, I am extremely grateful to my family members and friends who have supported and inspired me in all my endeavors making my parents' dream come true. Without their emotional support, prayers, and love, I could have never made it this far. Many thanks to all of them for always being there whenever I needed them most. Deep gratitude to my elder sister, my wife, and my daughters Haneen and Areej (who were encouraging me to finish the thesis so they can go Disneyland with their father).

Contents

List of Figures

List of Algorithms

List of Symbols and Abbreviations

- PROLOG: Programming in Logic
- SLD: Selected Literal Definite Clause
- SLDNF: Selected Literal Definite Clause with Negation-by-Failure
- SLG: An extension of SLDNF which is a table-oriented resolution.
- TLP: Tabled Logic Programming
- IE: Inference Engine
- KB: knowledge base
- TMS: Justification-based truth-maintenance system
- JTMS: Justification-based truth-maintenance system
- API: Application programming interface
- SIS: Student Information System
- BI: Business Intelligence
- NP: Normal Query Evaluation
- TP: Tabled Query Evaluation
- ITP: Incremental Tabled Query Evaluation

Chapter 1

Introduction

PROLOG is a logic programming language associated with artificial intelligence and computational linguistics [Clocksin & Mellish, 1984, Covington, 1994, Bratko, 2000]. It is a programming language based on Horn clause[Lloyd, 1987]. A Horn clause has the form

$$H :- B_1, B_2, \ldots, B_n.$$

where H is a first-order atom and each B_i is a first-order literal. H is called the head of the clause and $B_1, B_2, \ldots, B_n$ is called the body of the clause. A unit clause, or fact, is a clause whose body is empty, i.e. n=0. A rule is a clause with non-empty body. A PROLOG program consists of facts and rules. The following is a simple PROLOG program:

$man(socrates).$

$mortal(X) :- man(X).$

The first line can be read, "Socrates is a man." It is a base clause, which represents a simple fact. The second line can be read, "X is mortal if X is a man;" in other words, "All men are mortal." This is a clause, or rule, for determining when its input X is "mortal." (The symbol ":-", sometimes called a turnstile, is pronounced "if") We can test the program by asking the question:

$?- mortal(socrates).$

that is, “Is Socrates mortal?” (The “?-” is the computer’s prompt for a question(PROLOG query)). PROLOG will respond “yes”. Another question we may ask is:

$$?-mortal(X).$$

that is, “Who (X) is mortal?”. PROLOG will respond “X = socrates”.

Since PROLOG programs are collections of facts and rules, PROLOG has a feature of including new facts and rules or to remove existing facts and rues while the program is being executed. This is performed in PROLOG using two meta-predicates: *assert* and *retract*. Based on these two operations, there are two kinds of logic systems:

1. A monotonic logic system [Nilsson & Maluszynski, 1995] in which base facts remain constants. In other words, this system does not allow *assert* and *retract* operations to be performed. This is the default logic in PROLOG.

2. A non-monotonic logic system [Boella & van der Torre, 2005, McDermott, 1982] where base facts can be either added or deleted. *assert* operation is used to add base facts to the system while *retract* operation is used to delete them from the system. To be able to use non-monotonic logic, the programmer must inform the PROLOG engine that certain predicates are non-monotonic by using the Prolog predicate *dynamic*.

Tabled extension for Logic Programming (TLP) [Tamaki & Sato, 1986, Warren, 1992, Swift, 2000] evaluation enhances the performance of the PROLOG query engine and allows the termination of certain computations that do not terminate under the normal PROLOG query evaluation. The concept of adding tabled evaluation to PROLOG is simple: during execution, maintain a table of query calls and the values they return; if the same query is made later in the computation, do not re-execute the query, instead, use the saved answer to update the current state as if the query has been called and has returned that value. This mechanism assumes that the set of facts and/or rules participated in generating the answers of a certain query are static.

The incremental evaluation [Saha & Ramakrishnan, 2003] of tabled PROLOG programs allows to maintain the correctness (soundness) and completeness of the tabled answers under the dynamic state. This evaluation strategy allows the system to update the tabled answers when the set of facts and/or rules participated in generating the answers of a certain query are either retracted from or asserted to the PROLOG program.

1.1 Problem Statement

Tabled resolution for logic programs mitigate some of the well-known problems of PROLOG, including the tendency to fall into infinite loops, repeating sub-computations, and the unsatisfactory semantics of negation [Chen & Warren, 1996]. Tabled based PROLOG implementations are able to reduce the search space, avoid looping, and have better termination properties than normal PROLOG implementations.

The default tabling implementations like XSB [Swift & Warren, 2012], YAP-Prolog [Rocha et al., 2000], and B-Prolog [Zhou et al., 1996] assume a monotonic logic system in which base facts/rules remain unchanged. Based on this assumption, answers collected for a certain query (sub-goal) remain the same when the same query (subgoal) is re-evaluated at a later time. Traditional memoing (tabling) implementations fail to handle non-monotonic logic where the base facts/rules can be either added (asserted) or deleted (retracted). Under these circumstances, the set of collected answers for a query may become incomplete or even erroneous when the same query is re-evaluated. In order to overcome this problem, the usual solution for a programmer is to explicitly abolish the query cached answers (table) whenever changes occurred to the predicates related to the tabled answers. This inefficient solution avoids getting the wrong answers if the query is re-evaluated. When the query is re-evaluated after table abolishment, the system has to recompute the query from scratch and table it again. The incremental evaluation of tabled PROLOG programs [Saha, 2006] can handle the non-monotonic logic in a better way. The basic concept behind the incremental evaluation is that when some facts or rules are

changed in a program, the system re-computes only the results affected by the change, instead of re-evaluating the query from scratch. The critical challenges for incremental tabled evaluation strategies that stores the final answers of the query can be categorized as follows:

1. How to detect which table entries need to be changed, and how to compute the changes.

2. How to avoid the re-computation which is required to update the tabled answers due to the changes in the related database of facts and rules.

3. How to minimize the computation needed to update the tabled answers due to the addition of new related facts and rules.

4. How to filter the answers of the query to return the answers of related sub-queries.

Among the major PROLOG implementations like (SWI-Prolog [Wielemaker et al., 2012], Gnu-Prolog [Diaz et al., 2012], B-Prolog [Zhou, 2012], XSB [Sagonas et al., 1993b], Ciao [Hermenegildo et al., 2011] and SICStus-Prolog [Carlsson & Mildner, 2010]). XSB is the only system that supports incremental tabulation. XSB is a dialect of the PROLOG programming language. When a query is proved for the first time, XSB saves the final results of the query in a table space. Along with the table space, the system keeps the dynamic call dependency graph to track the call dependency between the predicates. When the base facts/rules that are related to the tabled answers get asserted or retracted, XSB uses the call dependency graph to determine which tabled answers should be removed from the table, and which new answers should be computed and added to the table space. The above approach [Saha & Ramakrishnan, 2006] used by XSB is capable of handling the non-monotonic logic. However the approach suffers from some limitations and performance issues which are addressed by this research. This research (thesis) presents an alternative approach for incremental evaluation of tabled PROLOG programs that suits the non-monotonic logic and tries to avoid the existing issues in the previous approaches.

1.2 Research Objectives

The main aim of this research is to develop a system that is capable of caching and maintaining the correct answers of the query under the non-monotonic logic in an efficient way. The research aim has been modularized into smaller measurable objectives. The objectives are:

1. To investigate the current implementations of systems that are capable of evaluating tabled PROLOG programs incrementally. Achieving this objective requires a detailed study and analysis of the current PROLOG implementations that supports tabulation. This objective can be achieved by answering the following questions:

   (a) How does a PROLOG inference engine prove a query?

   (b) How normal and incremental tabulation are incorporated into different PROLOG dialects?

   (c) What are the strengths and weaknesses of the current approaches to support incremental evaluation of tabled PROLOG programs?

2. To design a system that is capable of evaluating tabled PROLOG programs incrementally. The design should avoid, as mush as possible, the limitations and disadvantages in the existing approaches to support incremental evaluation of tabled PROLOG programs. To do that, the following sub-objectives are set:

   (a) To investigate an approach to cache the query answers which fits under the non-monotonic logic. The investigation should answer the following questions:

      i. How to cache the query answers when the query is proven for the first time?

      ii. How to return the cached answers when the query is re-evaluated or a related sub-query is evaluated (re-evaluated)?

iii. How to guarantee that the cached answers for the previously proven query remain correct regardless of any changes happens to the database of facts and rules (program clauses).

(b) To investigate the required data structures that suits the selected approach.

(c) To formulate the necessary algorithms required to implement the system.

3. To implement the system. The system implementation must take into consideration the following performance factors:

   (a) Minimize the overhead of caching the query answers when the query is proven for the first time.

   (b) Minimize the time and space needed by the system to ensure that the cached answers of the previously proven query remain correct regardless of any changes happens to the database of facts and rules (program clauses).

4. To evaluate the performance of the system. This objective can be achieved by the following sub-objectives :

   (a) To set evaluation criteria.

   (b) To decide the system benchmarking.

   (c) To generate the evaluation results and then analyze them.

1.3 Research Methodology

In accordance with the research objectives, we have divided the research into phases where each phase is linked with one of the objectives.

1.3.1 Investigate the current approaches for incremental evaluation of tabled PROLOG programs

The objective of this phase is to investigate the advantages and the limitations of the current approaches for incremental evaluation of tabled PROLOG programs. The limitations are the derived factors for designing and implementing the new proposed solution by this research. Our investigation is focused on XSB since it is the only PROLOG implementation so far that supports incremental tabulation. The incremental evaluation approach used by XSB suffers from the following limitations:

1. Evaluating a sub-query related to a previously tabled query

   For example, if the answers of the query $?-parent(X,Y)$ are already tabled, then the answers of the sub-query $?-parent(a,Y)$ can be filtered from the answers of the query $?-parent(X,Y)$ without computing it from the scratch. XSB (version 3.3.6, at the time of writing this thesis) considers the evaluation of sub-queries related to the perviously tabled queries as a new case rather than filtering the tabled answers of the parent query.

2. Maintaing the cached answers up-to-date in response to the change in the related predicates without re-evaluating the query from the scratch.

   The incremental tabulation in XSB is not supported for certain types of rules in the program. For example, the system can not handle the incremental tabulation for rules that are not defined in terms of other rules. If such rule is defined as dynamic and incrementally tabled, then the system re-evaluates the related queries from the scratch.

3. Repeating computations (PROLOG inference work).

   When certain facts are retracted, then system checks if these facts are related to any tabled query answers. If yes, then any invalid tabled answers must be deleted from the table to preserve the consistency of the cached answers. Let's consider now a typical scenario of non-monotonic logic. When a retracted fact is asserted back into the database, all answers that were deleted from the table when this fact got retracted should be reverted. To revert the answers, PROLOG inference work is needed. Under the dynamic systems, such kind of events are frequent, thus many entries will be deleted and added back to the tabled answers.

1.3.2 System Design

The objective of this phase is to design a system that is capable of evaluating tabled PROLOG programs incrementally. This thesis presents a novel approach to incremental tabulation that is capable of working in non-monotonic situations. The main idea is to cache the proof generated by the deductive inference engine rather than the end results. In order to be able to efficiently maintain the proof to be updated, the proof structure is converted into a justification-based truth-maintenance (JTMS) network. JTMS [Dung, 1996, Brewka et al., 1991] is a domain-independent belief revision system. It is usually coupled with an inference engine that does the actual inference work. JTMS operates on propositional objects and is used to record and maintain dependencies between deductive inferences. This can be done by representing deductive dependencies as a JTMS network. JTMS saves the dependency between deduced facts and the facts used to make the deduction in order to be able to efficiently cache the proof structure. The system translates every successful branch of a query into a JTMS network that links the facts and rule used in the branch to the answer generated by that branch.

```
:-dynamic married/2.
married(taher,hunaiza). %F1
married(shoib,shazia). %F2
married(naser,mariam). %F3
```

Figure 1.1: A simple PROLOG program representing marriage predicate.

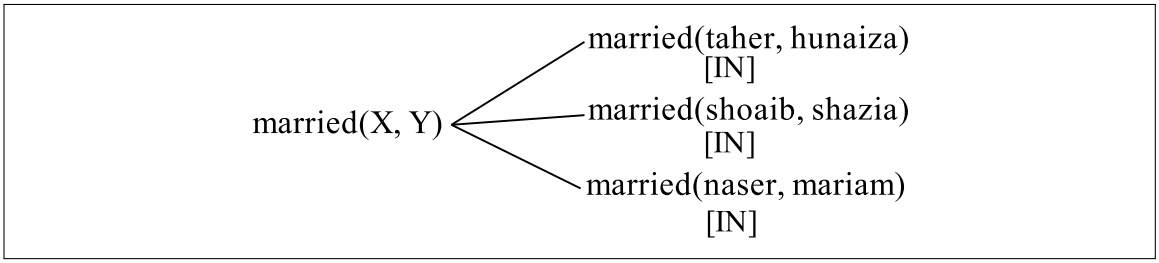

Figure 1.2: JTMS network for the query $?-married(X,Y)$ with respect to the Prolog Program of Figure 1.1.

Query Evaluation and Tabulation

When the system starts evaluating a query, it checks first if the query has been previously proven or not. If yes, then the query engine should collect the valid answers at that moment and return them. If the query has not been previously proven, then it is evaluated using the PROLOG inference engine (SWIPROLOG, XSB, or YAP-PROLOG) attached to the system. At the same time, JTMS network for the query is installed to cache the proof structure.

Example 1.1

Figure 1.1 shows an example of a small PROLOG representing a database of married couples. Figure 1.2 shows the JTMS network for the query $?-married(X,Y)$ installed by the system when the query is proven for the first time. Each answer is labeled *IN* indicating that it is a valid answer currently. If the query is re-evaluated then no inference work is needed from PROLOG engine to return the answers. To return the answers of the query, the system collects all the valid(*IN*) answers from the JTMS network and returns them.

Soundness and Completeness of the Cached Proof Structure

The system has to ensure that at any time, regardless of the number of assertions or retractions, the cached proof structure for a previously proven query is both sound and complete.

1. Soundness of the cached proof structure

   The proof soundness means that the cached proof structure is consistent. When the database changes, the TMS propagates the effect of the change through its network so as to maintain the consistency of old proofs. In other words, when a query is re-evaluated after any changes in the related facts/rules to the query answers, the system should report only those answers, from the JTMS network, that are valid due to the changes:

   - Let $J1$ be the set of branches associated with $Q1$ in the JTMS network. $J1$ represents the cached proof structure for $Q1$.
   - Let $T1$ be the set of valid answers for the query $Q1$ when the query is evaluated for the first time. $T1$ is extracted from $J1$.
   - When the related facts/rules of $J1$ are changed, the system updates $J1$ branches according to new changes.
   - Let $T2$ be the set of invalid answers for the query $Q1$ due to the changes in database of related facts/rules.
   - When the query $Q1$ is re-evaluated after the changes, then the system should report $T3$. $T3$ is the set of valid answers for the query $Q1$ at that moment. This means that $T3$ is $T1 - T2$. Actually $T3$ is extracted from $J1$.

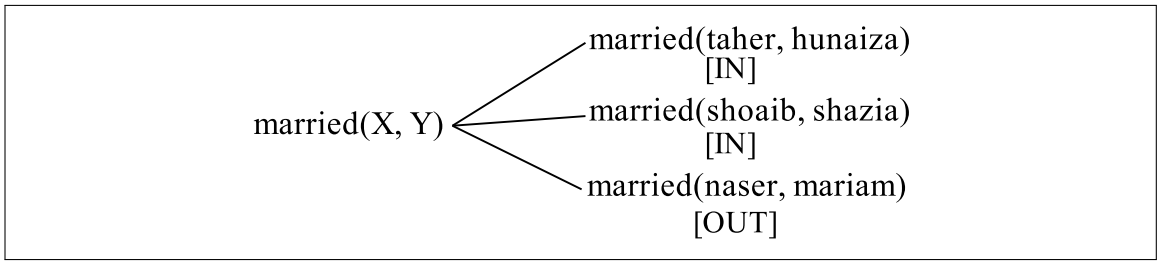

Figure 1.3: JTMS network Figure 1.2 after retracting the fact $?-married(naser,mariam)$.

Example 1.2

Consider the PROLOG program of Figure 1.1. Assume that the third married couple decided to divorce. This applies that the fact $married(naser,mariam)$ is retracted from the database of facts in Figure 1.1. Figure 1.3 shows the JTMS network for the query $?-married(X,Y)$ after retracting the fact $?-married(naser,mariam)$. The OUT label indicates that the answer in not valid. Later, if the query is re-evaluated then the system will return only two answers for this query since the third answer is not valid anymore.

2. Completeness of the cached proof structure

   The proof completeness is concerned with the assertion of new data that was not available when the query was evaluated for the first time. This is important since asserting new data to the database may add some results to those already available for the query, or even remove some of the results. When the query is re-evaluated after the addition of related new facts/rules, then the system should report all previous valid answers, from the JTMS network, along with any new answers generated due to the additions of new predicates:

   - Let $J1$ be the set of branches associated with $Q1$ in the JTMS network. $J1$ represents the cached proof structure for $Q1$.
   - Let $T1$ be the set of valid answers for the query $Q1$ when the query is evaluated for the first time. $T1$ is extracted from $J1$. Let $J2$ be the set of new branches added to the JTMS network due to the insertion of new data related to $J1$. As a

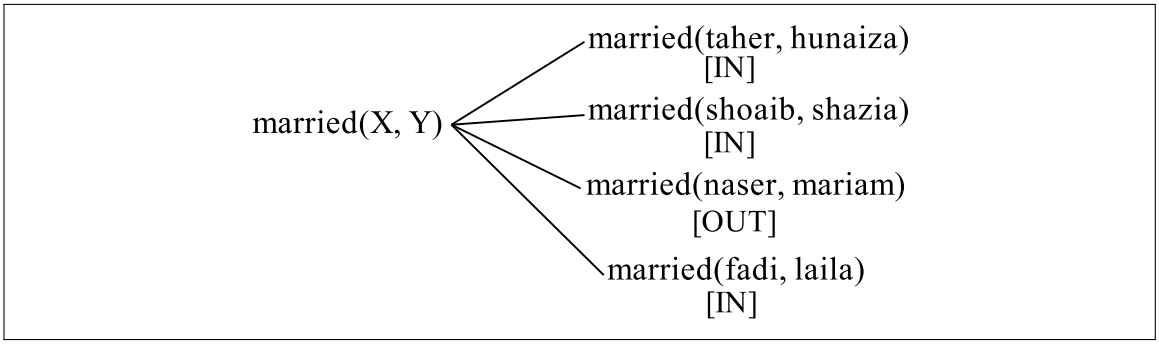

Figure 1.4: JTMS network Figure 1.3 after asserting the fact $?-married(fadi, laila)$.

result, the cached proof structure of $Q1$ is updated to $J3$. $J3$ is simply $J1+J2$.

- When the query $Q1$ is re-evaluated after the changes, then the system should report $T2$. $T2$ is the set of valid answers for the query $Q1$ at that moment. $T2$ is extracted from $J3$.

In order to achieve proof completeness, the system uses a forward-driven rule-based system [Fickas, 1985] (database monitor) that has to ensure the completeness of the proof structure as the database changes. Whenever a new fact/rule is asserted in the database, the system checks to see if the new TMS node associated with the fact/rule can contribute to a previously proven query. If the TMS node can contribute to a previously proven query, then the system resumes the query to update its cached proof structure with respect to the new data.

Example 1.3

Consider the PROLOG program of Figure 1.1. Assume that a new couple(fadi, laila) got married. This applies that the fact $married(fadi, laila)$ is asserted to the database of facts in Figure 1.1. Figure 1.4 shows the JTMS network for the query $?-married(X,Y)$ after asserting the fact $married(fadi, laila)$. Later, if the query is re-evaluated then the system will return the three answers for this query which are valid.

1.3.3 System Implementation

The system implementation itself is an another research problem addressed by this thesis. This is due to the fact that our system caches the query proof structure rather than caching the final query answers. This approach requires a deep understanding of how PROLOG inference engine works and how to extract the query proof from the engine. In order to cache the proof structure of a query, the system requests the PROLOG inference engine to execute the query in a special way that allows caching the query proof structure. For each query answer, we need the following information to cache the proof structure:

1. The rule participated as antecedent in generating the deduction.

2. The set of facts participated as antecedents in generating the deduction.

3. The answer itself which represents the consequence of the deduction.

For each query answer, the system creates a branch in the JTMS network and associates it with the query proof structure. This mechanism is quite complicated comparing to tabling the final answers of the query.

The second challenge in implementing the system is to develop the JTMS component and integrate it with the PROLOG inference engine. The problem is that the main operations of JTMS must operate quickly to avoid system performance issues. The implementation of the system is based on applying the source level transformations to a tabled program [Rocha et al., 2007b]. This gives us the advantage to incorporate the idea of incremental tabulation into different PROLOG systems using the same general framework.

1.3.4 Performance Evaluation

To assess the performance of our system, it is compared with the regular, traditional, and incremental tabled PROLOG systems. Our benchmark is the XSB system since it is the only PROLOG implementation so far that supports incremental tabulation and it is also considered as one of the efficient PROLOG implementations for normal tabulation [Swift & Warren, 1993, Sagonas et al., 1994]. We tested the performance of our system by implementing a small business intelligent system (reporting tool) based on real data extracted from the student information system of a mid-sized university. We have chosen this approach rather than the standard benchmark dataset to test the system on real data. The objective is to observe if the system is able to work used under real applications.

The main assessment factors for testing the performance of system are categorized into the following:

1. Evaluating a query for the first time

   This measurement gives the indication of how fast the system is behaving compared to normal, tabled and incremental tabled implementations of PROLOG when the query is evaluated for the first time. Also it indicates the overhead of building (caching the prof structure) the JTMS network.

2. Re-evaluation of a query

   This measurement is important as to know how fast the system is capable of extracting the valid answers of the query from the JTMS network and report them back to the user.

3. Evaluating a sub-query related to a previously proven query

   This assessment helps to know if the system is capable of extracting the answers of sub-computations from the cached proof structure rather than asking the PROLOG inference engine to evaluate the sub-query.

4. The cost of maintaining the soundness and completeness of the cashed proof structure

   These are the main assessment factors of the system. We would like to know how the system is efficient in maintaining the cached proof structure up-to-date, in correspondence to the changes in the database of related predicates. The results can be compared only with PROLOG systems that support incremental tabulation.

1.4 Scope of Research

The main aim of this research is to develop a system that is capable of evaluating tabled PROLOG programs incrementally. In this research we will focus on the following PROLOG programs and queries:

1. Normal (definite) programs that do not contain any negated and list terms.

2. Stratified and non-stratified programs which extend the definite programs to include the default negation operator.

3. Queries that generate finite answers.

The following are the current system limitations :

1. The system needs to keep track of all proof structure branches that generate the answers of the query. This is achieved by retrieving all answers of the query at once when it is evaluated for the first time. The drawback of this approach is that the system can not handle queries with infinite answers.

2. If the query generates large number of answers, then system performance is degraded due to the time spent in building the JTMS network due to the large number of query answers.

1.5 Significance of Research

In today's environment of mobility and interactive systems, the need of non-monotonic systems is increased. There is technological gap in supporting incremental evaluation of tabled PROLOG programs compared to normal tabled evaluation. This research opens the door for logic based systems that depend on dynamic facts and rules to benefit in their performance from the idea of incremental evaluation of tabled PROLOG programs. More precisely our approach favors the dynamic rules based logic systems. Below are examples of few potential systems that can get benefits from the concept of tabulation with existence of non-monotonic evaluators like the one provided as an outcome for this research

1. Belief Revision Systems [Shapiro, 1998]

   Belief revision is a typical example of non-monotonic logic. It is the process of changing beliefs to accommodate a new belief that might be inconsistent with the old ones. In the assumption that the new belief is correct, some of the old ones have to be retracted in order to maintain consistency. Plenty of daily needs expert systems can fall under this category, e.g. currency exchange rates, weather forecast,...,etc. The typical nature of these systems is that the base knowledge is changing so frequently, adding the incremental tabulation support would enhance the response time of the query engine in those systems.

2. Security Analysis

   The main idea behind these systems is to represent the security policies, vulnerabilities and their interaction as logical rules. The host and network configuration are expressed as facts. Applying the set of rules on set of current facts determines if a single or multistage attacks are possible in the system. Incorporating the incremental tabulation features into those systems can enhance the functionality of changing the configurations of the network system and can quickly view the effect of such changes.

3. Static analysis

   Static analysis, also called static code analysis, is a method of examining the source code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. The key idea of static analysis tools is to query patterns in the code and relate the patterns together. Adding the incremental tabled evaluation to the query engine in such systems would be useful.

1.6 Thesis Outline

This thesis consists of six chapters including this introduction chapter. Chapter 2 provides a brief overview of the research areas covered by this thesis, highlighting the main ideas behind the key aspects of each area. The chapter starts with a brief introduction to logic programming (PROLOG) and to PROLOG proofs, then the chapter introduces the idea of tabulation (memoing) and the incremental tabulation for logic programming. Then the chapter explores the current approaches for incremental evaluation of tabled PROLOG programs and their limitations. Finally the chapter summarizes our finding related to the tabled evaluation of PROLOG programs.

Chapter 3 introduces JLOG (Justification-Based Logic), the outcome of this thesis. This chapter presents the design of JLOG and how it works in details. The relation between the PROLOG inference engine and JLOG is also highlighted in this chapter .

Chapter 4 describes the implementation details of JLOG. It starts with the implementation approach used to build the system, then it shows how JLOG converts the PROLOG program rules into a format that allows the system later to get the query answers in a shape that is suitable for building the JTMS network for the query. After that, the chapter describes the necessary algorithms and required data structures to implement the main components of the system.

Chapter 5 discusses the performance of JLOG. The chapter starts with describing the testing methodology used to assess the performance of JLOG, followed by an idea about the basic data set (facts); the PROLOG programs and queries used in the assessment process. Then it compares the results of evaluating and re-evaluating the PROLOG queries and subqueries in JLOG with evaluation of these queries under the regular, tabled and incremental tabled PROLOG implementations. The chapter also covers the cost of maintaining the completeness and soundness of solution for the previously proven queries as the dynamic state changes. The chapter finally provides a detailed comparison between JLOG and XSB based on the above performance factors.

Finally Chapter 6 presents the thesis summary, contribution, current limitations, future work and conclusion. . The thesis also includes four appendices . Appendix A contains the system code which is implemented in JAVA. Appendix B, C, and D contains the data samples from the dataset used to test the performance of the system. The complete dataset is not included in appendices due to huge size. However the soft copy of this thesis contains the complete dataset.

Chapter 2

Literature Review: Incremental Evaluation of Tabled PROLOG Programs

The aim of this chapter is to provide a brief overview of the research areas covered by this thesis, highlighting the main ideas behind the key aspects of each area. We start with a brief introduction to logic programming and PROLOG. Then we explore the idea of tabulation (memoing) for logic programs followed by the incremental evaluation of tabled PROLOG programs. After that we explore the current approaches for incremental evaluation of tabled PROLOG programs. Finally we summarize our finding related to the tabled evaluation of PROLOG programs.

2.1 Logic Programming

Programming that uses a form of symbolic logic as a programming language is usually called logic programming, whereas languages based on symbolic logic are called logic programming languages or declarative languages. Declarative languages [Lloyd, 1994], also called non-procedural or very high level, are programming languages that express

the logic of a computation without describing its control flow. These languages are non-procedural because programs in such languages specify what is to be done rather than how to do it. This is different from imperative languages like C and Java in which a program defines a series of steps that must be performed.

Logic programming [Lloyd, 1987] systems allow programmers to state a collection of axioms from which theorems can be proven. The user of a logic program states a theorem, or goal, and the language implementation attempts to find a collection of axioms and inference steps that together imply the goal. In other words, given a theory (or program) and a goal (query), the theorem prover uses the theory to search for alternative ways to satisfy the goal.

Axioms are written in a standard form known as Horn clause [Lloyd, 1987], a subset of First Order Logic [Smullyan, 1995]. A Horn clause statement has the form:

$$H : -B_1, B_2, ..., B_n.$$

where H is a first-order atom and each B_i is a first-order literal. H is called the head of the clause and $B_1, B_2, ..., B_n$ is the body of the clause. The semantic of this statement are that when B_i all are true, we can deduce that H is true as well. The above clause can be read "$H, if\, B_1, B_2, \ldots, B_n$".

Logic programming systems combine existing statements, by canceling like terms, to drive a new statement. This process is known as resolution. Let's assume that A implies B, for example, and B implies C. we can conclude that A implies C.

$$\frac{\begin{array}{c} B \leftarrow A \\ C \leftarrow B \end{array}}{C \leftarrow A}$$

In general, terms like A, B, C may consist of constants:

- "$kualalumpur\,is\,rainy$"

and predicates applied to atoms or to variables:

- $rainy(kualalumpur)$.
- $rainy(islamabad)$.
- $rainy(X)$.

During the process of resolution, free variables may acquire values through unification with expressions in matching terms:

$$\frac{agricultural(X) \leftarrow rainy(X) \qquad rainy(kualalumpur)}{agricultural(kualalumpur)}$$

2.2 PROLOG and Proving Logical Queries

PROLOG is one of the main programming languages for logic programming. PROLOG is a programming language that uses a small set of basic mechanisms to create surprisingly powerful programs [O'Keefe, 1990]. These mechanisms are pattern matching, tree-based data structuring and backtracking. PROLOG is used for the development of scheduling systems, knowledge systems, expert systems and many other applications [Bratko, 2000].

2.2.1 A Quick Look at PROLOG

A PROLOG engine runs in context of a database of Horn clauses. The clauses in a PROLOG database can be classified as facts or rules. A fact is a Horn clause without a right hand side. It looks like a single term:

$rainy(kualalumpur).$

The first line can be read, "*kualalumpur is rainy*" it is a base clause, which represents a simple fact. A rule has a right hand side:

$snowy(X) :- rainy(X), cold(X).$

The token $:-$ is the implication symbol; the comma indicates "*and*"; X an is uninstantiated variable. The above line can be read, *"X is snowy if X is rainy and X is cold*". PROLOG program logic is expressed as a database of facts and rules. A computation is initiated by running a query, or a goal, over the database of facts and rules. A query in PROLOG is clause with an empty left-hand side. In most implementations of PROLOG, queries are entered with a special $?-$ version of the implication symbol. For example:

$?- rainy(C).$

that is "*which (C) is rainy?*", if we were to type the following:

$rainy(kualalumpur).$

$rainy(islamabad).$

$?- rainy(C).$

the PROLOG query engine would respond with

$C = kualalumpur$

Of course, $C = islamabad$ would also be a valid answer, but PROLOG query engine will find *kualalumpur* first, because it comes first in the database. The second answer will be displayed if we continue by typing a semicolon as follows:

$C = kualalumpur;$

$C = islamabad$

if we type another semicolon, the PROLOG query engine will indicate that no further answer exists:

$C = kualalumpur;$

$C = islamabad;$

no

likewise, given

$rainy(kualalumpur).$

$rainy(islamabad).$

$cold(islamabad).$

$snowy(X) :- rainy(X), cold(X).$

the query

$?- snowy(C).$

will return only one solution:

$C = islamabad;$

2.2.2 Monotonic vs. Non-Monotonic Logic

Since PROLOG programs are databases of facts and rules, PROLOG has a feature to include new facts and rules or to remove existing facts and rules while the program is been executed. This is performed in PROLOG using two meta-predicates: *assert* and *retract*. The syntax of *assert* is:

- $assert(predicatename(terms))$.

  Assert records the predicate $name(terms)$ in the program and becomes available for use immediately. For example, If we type the following after typing the examples in previous section:

  $assert(rainy(langkawi)).$

  $?- rainy(C).$

  the PROLOG query engine would respond with three answers:

  $C = kualalumpur;$

  $C = islamabad;$

  $C = langkawi;$

  no

The syntax of *retract* is:

- $retract(predicatename(terms))$.

  Retract eliminates the predicate $name(terms)$ from the set of the facts. If we type the following:

  $retract(cold(islamabad))$.

  $?-snowy(C)$.

  then the query engine will respond with no answer.

Based on the these two(assert/retract) operations there are two kind of logic systems:

- A *monotonic logic* [Nilsson & Maluszynski, 1995] system in which base facts/rules remain constant. In other words we do not allow assert and retract operations to be performed in these kind of systems.

- A non-monotonic logic [Boella & van der Torre, 2005, McDermott, 1982] system where base facts/rules can be either added or deleted. We add base facts/rules to the system by using the assert operation and we delete them from the system by using the retract operation.

2.2.3 Proving Logical Queries in PROLOG

A computation in PROLOG is initiated by running a query, or a goal, over the database of facts and rules. In the realm of formal logic, when it comes to query evaluation, there are two principal search strategies:

- Forward chaining(bottom-up)

  Start with existing clauses and work forward, attempting to derive the goal.

- Backward chaining(top-down)

  Start with goal and work backward, attempting to unresolved it into set of preexisting clauses.

Many debates exist on the comparative advantages of forward and backward chaining query evaluation techniques [Brass, 1995, Perea, 2010]. If the number of existing rules is very large, but number of facts is small, it is possible for forward chaining to find a solution more faster than backward chaining. However, backwards chaining tends to be more efficient in other cases because backward evaluation is goal oriented and hence it generally computes less redundant facts. PROLOG is defined to use backward chaining [Gallier, 1985].

The proof method used, for proving logical queries, in PROLOG is SLD-resolution [Brass, 1995] applied to Horn clauses. It applies a backtracking search through the tree of SLD refutations using top-to-bottom rule selection strategy, and left to right subgoal selection in the body of the rule. The power of PROLOG as a programming language comes from the fact that when SLD-resolution is used as the evaluation technique, a horn clause can be understood as defining a procedure (in the ordinary sense of procedural languages, e.g. a C function), and evaluation can be understood as executing such a procedural program. One complication is that in PROLOG programs, a procedure may have multiple definitions which are interpreted as alternate definitions of the procedure. So procedure-wise, PROLOG is a non-deterministic language where execution carried out as a depth-first search through the tree of possible alternative executions.

The logical foundation of PROLOG is apparent in the reading of each clause of a PROLOG program as an implication from a conjunction of conditions (goals) to a conclusion (the clause's head). Operationally, a predicate $p(...)$ is interpreted as a procedure, a goal $p(...)$ as a call of that procedure, and the clause with head $p(...)$ as alternative ways to executing the call. These clauses are considered in textual order. A pattern-matching procedure, unification, is used to try to find the first clause head that can be made identical to the goal by substituting appropriate terms for the variables in the goal and the clause. If this succeeds, the substitutions found are added to the current substitution set. If no matching clause is found, the goal fails, and PROLOG backtracks to the most recent successful goal. Its clause choice and unification substitution are undone, and the next

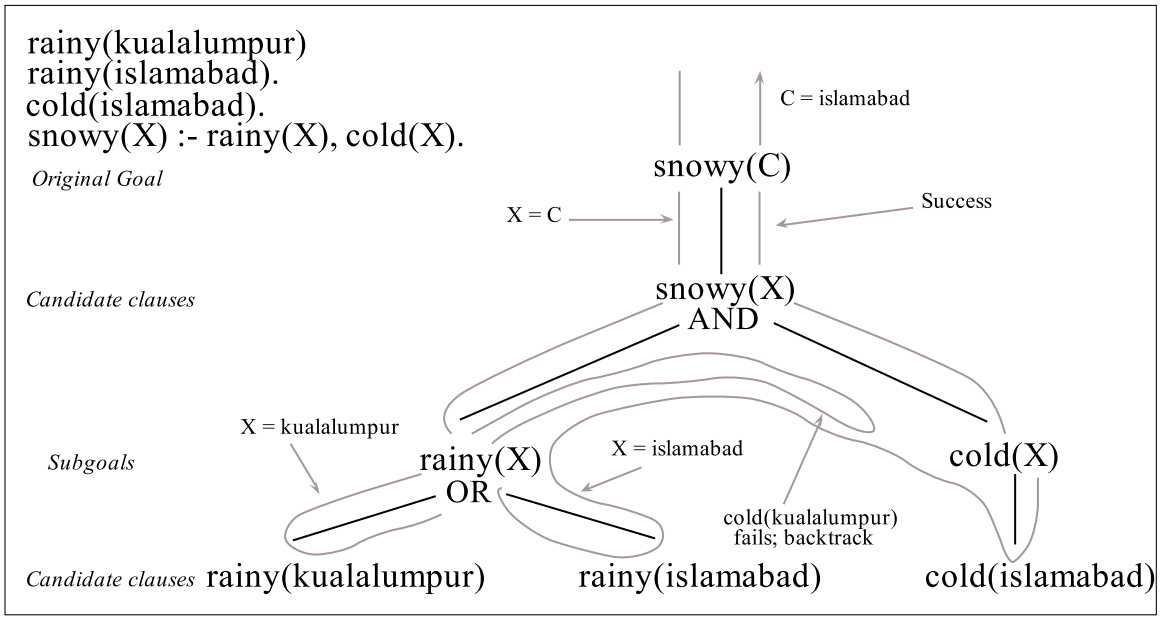

Figure 2.1: Backtracking Search in PROLOG

remaining clause for that goal is considered. If no more clauses match, that goal in turn fails and PROLOG backtrack again.

Example 2.1

Figure 2.1 shows an example of SLD-resolution. The SLD-derivation tree is for the query $? - snowy(C)$ with respect to the PROLOG program of Figure 2.1(top left corner). The proof structure of a query is a tree whose head is the query itself (i.e. $? - snowy(C)$) and each branch in the tree is an SLD-derivation. For example, the proof structure in Figure 2.1 has two branches. Each arrow in the figure is a derivation step. The tree of possible resolutions contains alternative *AND* and *OR* levels. An *AND* level consists of subgoal from the right-hand side of a rule, all of which must be satisfied. An *OR* level consists of alternative database clause whose head will unify with the subgoal above to it.

The PROLOG query engine explores the tree in Figure 2.1 using depth first, from left to right. It starts at the beginning of the database, searching for a rule R or fact F whose head can be unified with the top-level goal . It then considers the terms in the body or R(in case of rule) as sub goals, and attempts to satisfy them, recursively, left to right. If at any point a subgoal fails, the query engine returns to previous subgoal and attempts to

satisfy it in a different way. The process of returning back to previous goal is known as *backtracking*.

Coming back to the example of Figure 2.1. Query execution starts with searching for a fact or a rule whose head can be unified with top-goal $snowy(C)$. The only rule satisfying the condition is $snowy(X) :- rainy(X), cold(X)$. Variable C is associated with variable X in such way that if either receives a value in the future it will be shared by both. The next step is executing subgoals $rainy(X)$ and $cold(X)$ respectively from left to right, $rainy(X)$ is unified first with the fact $rainy(kualalumpur)$. X is bind to $kualalumpur$. The query engine then tries to execute the next subgoal $cold(kualalumpur)$. This subgoal fails because there exist no rule or fact head that can be unified with $cold(kualalumpur)$. The system backtracks to $rainy(X)$. The *OR* rule is applied. This time $rainy(X)$ is unified with the fact $rainy(islamabad)$. X is bind to $islamabad$. Now the system attempts to find a rule or fact head that matches the subgoal $cold(islamabad)$ and it succeeds. *The AND* rule is satisfied hence this generates first solution for the top-level goal . Solutions to the query are generated from successful branches only. Each successful branch generates exactly one answer which is obtained from the original query by applying all variable substitutions attached to the branch. Therefore, the only solution for the query $?- snowy(C)$ is $islamabad$.

2.2.4 Negation and the Closed World Assumption

PROLOG program logic is expressed as a database of Horn clauses(facts and rules). The database of Horn clauses represents a list of things assumed to be true. It does not include any things assumed to be false. PROLOG is relying on purely positive logic. This is the reason that PROLOG has the *not* predicate which is different from logical negation. PROLOG includes a negation-by-failure operator. Negation-by-failure [Lloyd, 1987] is based on the closed-world assumption. The closed-world assumption assumes that the PROLOG database is complete; contains everything that is true. The goal $not(G)$ can succeed simply because our current knowledge is insufficient to prove G. Another important point, to

be highlighted, is that negation in PROLOG occurs outside any implicit existential quantifiers on the right-hand side of a rule. Therefore

$?-not(takes(X,csc122).$

where X is uninstantiated, means

$$\neg\exists X[takes(X, csc122)]$$

instead of

$$\exists X[\neg takes(X, csc122)]$$

The SLD-derivation method presented in previous section is extended into SLDNF-derivation [Warren, 1983] in order to handle the negation operator. Briefly, when a negative literal $\neg G$ is selected for evaluation, the query engine suspends proving the original query in order to prove G. If the system fails in proving G, then $\neg G$ is assumed true according to the closed-world assumption. Otherwise, $\neg G$ is assumed false. The system resumes the suspended query once the value of the negative literal is determined.

Example 2.2

Figure 2.2 shows an example of an SLDNF-resolution. The SLD-derivation tree is for the query $?-bachelor(X)$ with respect to the PROLOG program of Figure 2.2. There are two branches in the main proof structure with only one being successful. When each branch reaches the part of the proof where a negated subgoal is selected, it stops its work and starts proving the non-negated subgoal.

The query $?-bachelor(X)$ execution starts with unifying the top-goal $bachelor(X)$ with the head of rule $bachelor(P):-male(P),not(married(X))$. Variable X is associated with variable P. The subgoals $male(X)$ and $not(married(X))$ are executed respectively from left to right, $male(X)$ is unified first with the fact $male(taher)$. X is bound to $taher$, then the negated subgoal $not(married(taher))$ should be executed. The PROLOG query engine suspends execution of $not(married(taher))$ and starts execution of the non-negated subgoal $married(taher)$. Since the database of facts states that $taher$ is married,

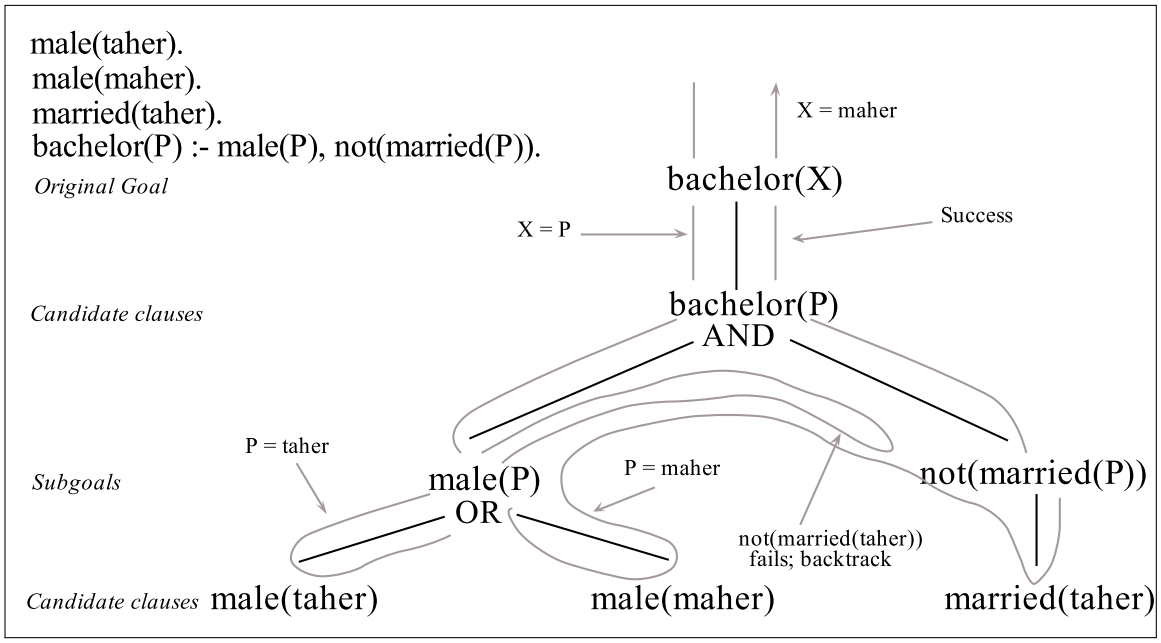

Figure 2.2: An example of SLDNF-dervation tree.

the subgoal $married(taher)$ is considered to be $true$. Now the query engine resumes the negated subgoal $not(married(taher))$ which is assumed to be $false$ since $married(taher)$ is $true$, this branch of the proof is failed and system backtracks. The *OR* rule is applied. $male(X)$ is unified with the fact $male(maher)$. X is bound to *maher* and the negated subgoal $not(married(maher))$ is executed and yields as $true$ since $married(maher)$ is $false$. The only answer generated for the query $?-bachelor(X)$ is coming from the second branch.

In the context of logic programming related to existence of negated terms or not, the PROLOG programs can be classified into three groups:

1. Definite Programs

   Definite programs, also called Horn Clause Programs, are those programs that do not contain any negated terms. For example, the program of Figure 2.3(a) is an example of a definite PROLOG program.

(a)

$path(X,Y) :- path(X,Z), edge(Z,Y).$
$path(X,Y) :- edge(X,Y).$
$edge(a,b).$
$edge(a,c).$
$edge(b,d).$

(b)

$shaves(barber, Person) :- person(Person), not(shaves(Person, Person)).$
$person(barber).$
$person(taher).$

Figure 2.3: Example of a definite, stratified and Non-Stratified PROLOG program.

2. Stratified Normal Programs

   Normal programs extend definite programs to include the default negation operator, which assumes a fact to be false if all attempts to prove it fails. Allegedly, a program uses stratified negation whenever there is no recursion through negation. Practically, most of the time, the majority of programmers use stratified negation. The prolog program of Figure 2.2 is an example of a stratified normal PROLOG program.

3. Non-Stratified Programs

   In stratified programs, facts are either true or false while in non-stratified programs facts may also be undefined. A program uses non-stratified negation whenever there is a recursion through the negation or the negation is implicit. The non-stratified logic is also know as well-founded semantics. Figure 2.3(b) shows an example non-stratified program, which presented one of Russell's paradoxes as a logic program. Note that the program in Figure 2.3(b) will suffer from infinite loop if we try to execute the query $?- shaves(X,Y)$ under normal PROLOG implementations. We will see in Section 2.3.3 how this problem can be resolved using the Tabled Logic Programming.

2.3 Tabled Logic Programming (TLP)

The paradigm of Tabled Logic Programming (TLP) is now supported by a number of PROLOG systems, including XSB [Swift & Warren, 2012], YAP Prolog [Costa et al., 2012], B-Prolog [Zhou, 2012] and Mercury [Somogyi & Sagonas, 2006]. Tabled resolution for logic programs palliate some of the well-known problems of PROLOG including susceptibility to infinite looping, repeated subcomputations, and unsatisfactory semantics for negation [Chen & Warren, 1996]. These limitations make PROLOG unsuitable to important applications such as Deductive Databases [Sagonas et al., 1994].

2.3.1 Tabled PROLOG systems

As seen in section 2.2, PROLOG query execution carries out a depth-first search through the tree of possible alternative executions. One of the main disadvantages of PROLOG's search strategy is its susceptibility to infinite looping. Another disadvantage of PROLOG's computational strategy is its tendency to recompute the same answers. The solution to avoid these problems is based on SLG resolution [Chen & Warren, 1996], also called tabled resolution. The SLG resolution uses memorization or tabling to overcome the problem of infinite looping, repeated subcomputation, and unsatisfactory semantics for negation.

Tabling is an implementation technique where answers for subcomputations are stored and then reused when a repeated computation appears. Tabled resolution-based systems uses a database to memorize subgoals that were proved and to use them later if they were needed, instead of re-computing them. This is simply done by storing all the answers of subgoals that can be proved while answering a given query. Therefore, memoing evaluation of PROLOG programs is more efficient than the standard PROLOG'S SLD resolution evaluation [Swift & Warren, 1993]. When given a query, a TLP system first checks if there is an answer in the table for the query to avoid re-computation. If not, it evaluates the call and stores the results that can be known from evaluating the query.

The basic execution model of adding tabulation to PROLOG is simple:

1. Tabled subgoals are evaluated by storing their answers in an appropriate data space called the table space.

2. Variant calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries.

3. When a variant call exhausts the set of available answers, the computation state for the variant call is suspended.

4. During this process and as further new answers are found, they are stored in their tables and returned to all variant calls.

2.3.2 Traditional Tabulation in the Context of PROLOG

Traditional memoing implementations assume a monotonic logic system in which base facts remain constant. Based on this assumption, answers collected for a certain query remain the same when the same query is re-evaluated at a later time. Traditional memoing implementations fail to handle non-monotonic logic where base facts can be either added (asserted) or deleted (retracted). Under these circumstances, the set of collected answers for a query may become incomplete or even erroneous when the same query is re-evaluated [Saha & Ramakrishnan, 2003].

Example 2.3

Consider the evaluation of the query: $?-connected(b,Y)$ with respect to the PROLOG program of Figure 2.4(a). The table in Figure 2.4(b) shows the results saved by a traditional memoing implementation after the execution of the query and the sub queries involved in proving the main query. Now if the same query is re-evaluated, then the answers saved in the table will be used instead of recomputing the query. This method works

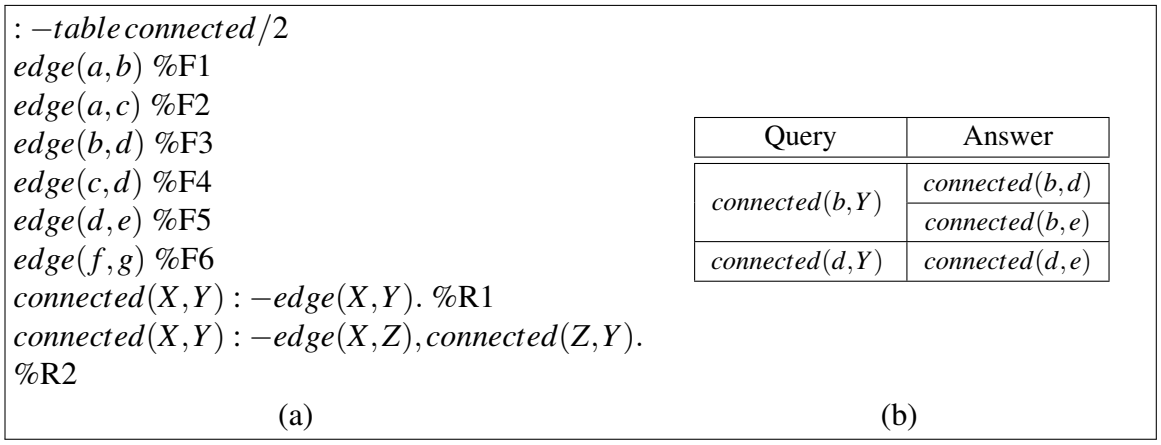

Figure 2.4: Translative closure program of the directed edge relationship with tabled answers after evaluating the query $?-connected(b,Y)$ for first time.

fine as long as no data assertion or retraction occurred. But in non-monotonic logic, that assumption can not be guaranteed. Consider the retraction of the base fact $edge(b,d)$ from database of Figure 2.4(a). The answers for the query $?-connected(b,Y)$ saved in the table are no longer valid(sound) because node b is not connected anymore to any node in the graph after retraction of the fact $edge(b,d)$. Also, if the fact $edge(b,f)$ is asserted to the database, then the answers in the table are incomplete since adding this fact will yield new answers $connected(b,f)$ and $connected(b,g)$. The new answers for the query should be added to the answers already in the table.

In order to overcome this problem, the usual solution for a traditional memoing implementations programmer is to explicitly abolish a table whenever changing(with assert or retract) a predicate on which the table depends. When the query is re-evaluated after table abolishment, the system has to recompute the query from scratch and table it again. This may work if changes in the database are not very often. But if changes in the database are very high, then a lot of query re-computations will take place and the system will lose the benefits of tabling while still paying its overhead.

2.3.3 Tabled Logic Programming and Well-Founded Semantic

The Prolog program of Figure 2.3(b) is an example of well-founded Semantic. Logically speaking, the meaning of this program should be that the barber shaves persons who do not shave themselves, but the case of the barber is trickier. If we conclude that the barber does not shave himself then our meaning does not reflect the first rule in the program. If we conclude that the barber does shave himself, we have reached that conclusion using information beyond what is provided in the program. The well-founded semantics, does not treat $shaves(barber, barber)$ as either true or false, but as undefined. PROLOG, of course, would enter an infinite loop for any query related to the barber, for example $?-shaves(barbar, Y)$. Using the tabled logic programming, this program can be converted into a working one using the concept of conditional answers. The tabled PROLOG systems treat undefined facts as conditional answers. See [Rao et al., 1997] for more details about handling well-founded semantic using tabled logic programming.

2.4 Incremental Tabulation

As described in Section 2.3, tables are created when tabled goals are called and are used until they are abolished. But if a tabled predicate depends on a dynamic predicate and the dynamic predicate changes, the table becomes out of date. To overcome the above problem, the idea of incremental tabulation was introduced [Saha & Ramakrishnan, 2003]. An incrementally maintained table is one that continually contains the correct answers in the presence of updates to underlying predicates on which the tabled predicate depends. If tables are thought of as database views, then incremental tabulation enables what is known in the database community as the incremental view maintenance [Mohania et al., 1997, Lee et al., 2001]. The basic idea behind incremental tabulation is that when some facts or rules change in a program, the system recomputes only the results affected by the changes, instead of reevaluating the query from scratch. The Incremental evaluation technique computes the changes in the results based on changes in the input. It is a good way

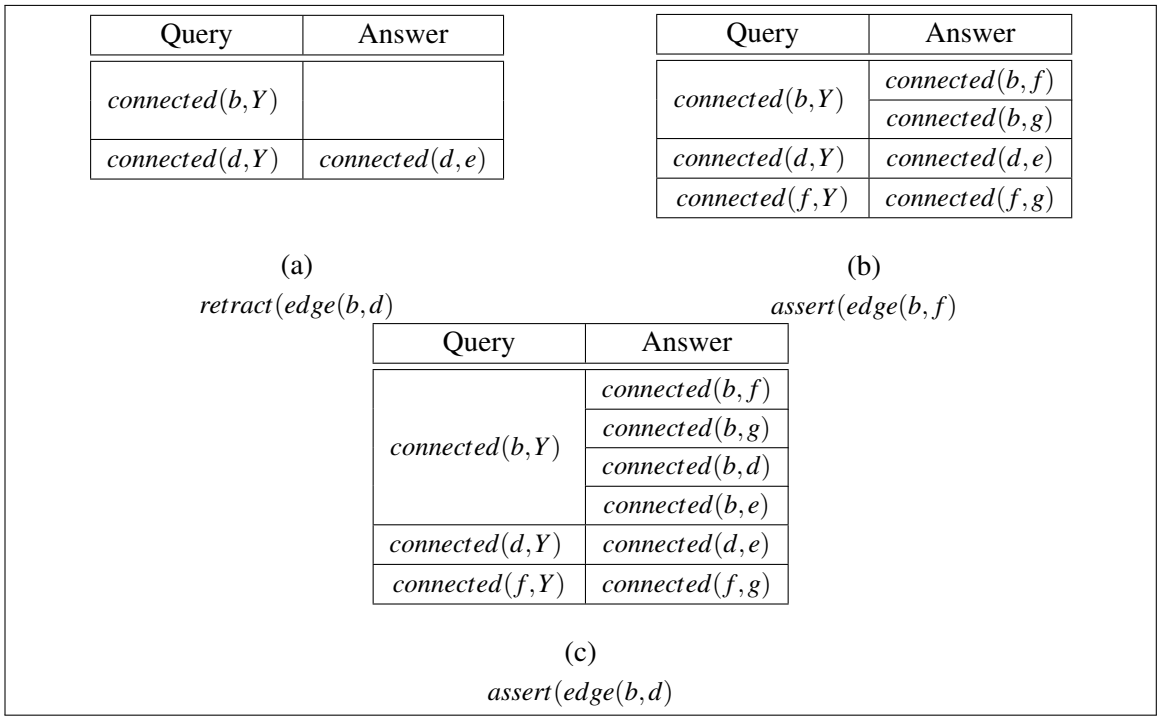

Figure 2.5: Tabled results of the query $?-connected(b,Y)$ after deliberating few changes to the base facts of Figure 2.4(a).

of keeping the cached results of a query sound and complete by computing the changes in the tables in response to changes in the database of facts and rules. In most cases, small changes in database of facts/rules lead to small changes in the cached tables, and in those small changes incremental evaluation should be faster than from-scratch evaluation strategy.

Example 2.4

Going back to Example 2.3, take into account few changes to the base facts of Figure 2.4(a) after tabling the results of query ?-connected(b,Y) for the first time. Figure 2.5 shows how the incremental evaluation strategy handles the tabled results when dealing with few changes in base facts. Three changes(in order) to the base facts of Figure 2.4(a) are placed in this example:

1. Retract the fact $edge(b,d)$

   Since node b is not connected anymore to any node in the graph after retraction

of the above fact, the two entries $connected(b,d)$ and $connected(b,e)$ are deleted from table in Figure 2.4(b) as shown in Figure 2.5(a). Now the tabled answers in Figure 2.5(a) are sound which means that no invalid results are stored in the table.

2. Assert the fact $edge(b,f)$

   Adding this fact yields new answers $connected(b,f)$ and $connected(b,g)$. These two answers should be added to the tabled results of Figure 2.5(a) resulting in Figure 2.5(b). By applying these actions, the incremental evaluation strategy makes sure that tabled answers are complete.

3. Assert the fact $edge(b,d)$

   This is a typical scenario of non-monotonic logic. A retracted fact is asserted back into the database, this means that all answers that were deleted from the table when this fact got retracted should be reverted back. The two entries $connected(b,d)$ and $connected(b,e)$ are added to Figure 2.5(c). Note that if such kind of events are frequent, then many entries will be deleted and added back to the tabled answers and that will be a waste of time.

The critical challenges for incremental tabled evaluation strategies that stores the final answers of the query, as shown in the above example, can be categorized as follows:

1. How to detect which table entries need to be changed, and how to compute the changes.

2. How to avoid the re-computation which is required to update the tabled answers due to the changes in the related database of facts and rules.

3. How to minimize the computation needed to update the tabled answers due to the addition of new related facts and rules.

4. How to filter the answers of the query to return the answers of related sub-queries.

The problem with traditional or incremental evaluation strategies, when it comes to non-monotonic logic, is that they store the final answers of the query. An alternative approach, which is the basis of this thesis, is that instead of saving the final answers for the query, the system stores the proof structure that was generated when the query was proved for the first time.

2.5 Incremental Tabualtion: Current Approaches

Incorporation of tabulation in the execution model of PROLOG leads to a more powerful and flexible paradigm. On the other hand, tabling introduces some extra complications in the standard implementation for PROLOG. Most of the complications are attributed to the more flexible control that tabling requires: i.e. the need to suspend and resume computations, is a main issue in a tabling implementation, because some subgoals, called producers, produce answers that go into the tables, while other subgoals, called consumers, consume answers from the tables. The situation is more complicated when incremental tabulation is needed [Saha & Ramakrishnan, 2006].

Among the major PROLOG implementations, XSB is the only one which supports incremental tabulation. The problem of incremental tabulation was addressed first in [Saha & Ramakrishnan, 2003]. The current version of XSB supports incremental tabulation based on algorithms described in [Saha, 2006, Saha & Ramakrishnan, 2005]. In this section, we summarize our findings related to the current approaches for incremental evaluation of tabled PROLOG programs and their limitations. We categorize our findings according to the following factors:

1. Tabulation approach

   XSB saves the end results of the query in a table space. Along with the table space, the system keeps the dynamic call dependency graph to track the call dependency between the predicates. When the base facts/rules related to the tabled answers get asserted or retracted, XSB uses the call dependency graph to determine which

tabled answers should be removed from the table, and which new answers should be computed and added to the table space. The drawback of this approach is that it requires two data structures to support incremental tabulation: the table and the call dependancy. Also the system might keep repeating computations (PROLOG inference work) if some entries in the table are added or removed due to the changes in the database of facts and rules.

2. Implementation approach

   Incremental tabulation support in XSB is implemented by modifying and extending the low-level PROLOG engine. The advantage of this approach is the run-time efficiency. However, the drawback is that it is not efficiently portable to other PROLOG systems because the engine level modifications are slightly more complex and time consuming.

3. Types of supported PROLOG programs

   In the current version of XSB, the incremental tabling works for definite and stratified programs that do not involve conditional answers.

4. Evaluating the query for the first time

   The time it takes for evaluating the query for the first time depends on the number of answers that must be tabled and the overhead of maintaing the call dependency to support the incremental tabulation in XSB. The overhead varies and depends on the program nature. For some cases the overhead is reasonable (10-15%), and for some other cases it starts getting worse.

5. Re-evaluating the query

   This is an advantage area for XSB, the system returns the answers of previously tabled query in a time which is almost similar to the time it takes from normal tabled evaluation. The system is capable of returning the answers in an average of 90-95% faster than the time it takes to evaluate the query for the first time.

6. Subqueries evaluation

   This point is one of the limitations in XSB, the current version of XSB considers the evaluation of subqueries related to a tabled query, which has been already evaluated, as new query. The system recomputes the whole inference to generate the answers which are already tabled for the main query.

7. Keeping the soundness and completeness of the cached answers (proof structure) for a query

   We measure the performance of maintaining the soundness and completeness of the tabled answers by asserting and retracting facts/rules that affect the tabled answers.

   (a) Retracting/Asserting old facts

       This case is related to maintaing the soundness of the cached answers. XSB deals with this case as a two-step process. First, it needs to look at the dependency graph to know which answers should be removed from the table space or/and the required computation (inference work) needed to generate the new answers and add them to the table space. The limitation of this approach is that it requires repeated inference work from the system if the deleted answers from the table needed to be added back to the table due to changes in the database of facts.

(b) Asserting new facts

This case is related to maintaing the completeness of the cached answers. The system have to compute the new answers for the query that might be generated due to the assertion of the new facts related to the query answers. The time it takes from the system to update the table space depends on the nature of program and how much inference work is required to generate the new answers.

(c) Retracting/Asserting old rules

This case is related to maintaing the soundness of the cached answers. In order to support assertion/retraction of rules on a tabled predicate, the tabled predicate must be defined as incrementally tabled and dynamic in XSB. The incremental tabulation in XSB is not supported for certain types of rules in the program. For example, the system can not handle the incremental tabulation for rules that are not defined in terms of other rules. If such rule is defined as dynamic and incrementally tabled, then the system re-evaluates the related queries from the scratch.

(d) Asserting new rules

This case is related to maintaing the completeness of the cached answers. XSB suffers in this case from the same problems when asserting/retracing the old rules.

2.6 Summary

In this section we summarize our findings related to the current approaches to support tabled evaluation of PROLOG programs. Tabled based PROLOG implementations are able to reduce the search space, avoid looping, and have better termination properties than normal PROLOG implementations. In fact, it has been proven that using the tabulation guarantees the termination for all PROLOG programs with the bounded term-

size property [Chen & Warren, 1996]. The current implementations of tabling have become stable and efficient for dealing with the monotonic logic [Sagonas et al., 1994, Swift & Warren, 1993].

The incremental evaluation of tabled PROLOG programs can handle the non-monotonic logic. The basic idea behind incremental tabulation is that when some facts or rules change in a program, the system recomputes only the results affected by the change, instead of reevaluating the query from scratch. The incremental evaluation approach used by current approaches [Saha, 2006, Saha & Ramakrishnan, 2005], implemented in XSB, suffers from the following problems:

1. Evaluating the sub-query related to a previously tabled query from the scratch instead of filtering the answers of the parent query.

2. Maintaing the cached answers up-to-date in response to the change in the related predicates without re-evaluating the query from the scratch.

3. Repeating computations (PROLOG inference work).

Chapter 3

JLOG Design

The objective of this research is to design and implement a system that supports tabled evaluation of PROLOG programs under non-monotonic environments. This chapter describes the purposed solution in details. The purposed solution tries, as far as possible, to avoid the existing limitations in the current approaches of incremental tabulation. In section 2.6 we summarized our findings related to the current approaches to support tabled evaluation of PROLOG programs. In order to achieve the objectives of this research we need to:

1. Set the system targeted goals.
2. Investigate the required data structures that suits the targeted design.
3. Formulate the necessary algorithms required to implement the system.

3.1 System Targeted Goals

Table 3.1 shows the issues related to the current approach to support incremental evaluation of tabled PROLOG programs and what is required to resolve or minimize these issues. The requirements to resolve the current issues in Table 3.1 are representing the targeted goals for the system:

	Current Issues	What is required?
1	System requires two data structures to support incremental tabulation.	Maintaing the information in one single data structure.
2	Repeating computations (Prolog inference work).	Avoid repetition of computations.
3	The cost of maintaining the correctness of the tabled answers.	Maintaing the correctness of tabled answers with minimal or no inference work from PROLOG side.
4	Handling the assertion/retraction of rules.	Handling the assertion/retraction of rules in efficient manner.
5	Evaluation of sub-queries.	A way to filter the answers of parent rather than re-evaluating the sub query from scratch.

Table 3.1: The issues related to the current approach to support incremental evaluation of tabled PROLOG programs and what is required to resolve these issues.

$:-table\, connected/2$
$edge(a,b)$ %F1
$edge(a,c)$ %F2
$edge(b,d)$ %F3
$edge(c,d)$ %F4
$edge(d,e)$ %F5
$edge(f,g)$ %F6
$connected(X,Y) :- edge(X,Y).$ %R1
$connected(X,Y) :- edge(X,Z), connected(Z,Y).$ %R2

Figure 3.1: Translative closure program of the directed edge relationship.

1. Maintaing the information in one single data structure

   The current approach caches the final answers (PROLOG deduced facts) of the query in a table space. Along with the table space, the system keeps the dynamic call dependency graph to track the call dependency between the predicates in order to maintain the correctness and completeness of the cached answers. An alternative approach would be that the system caches the PROLOG facts and rule that participates in generating the answer along with the answer itself. Consider the evaluation of the query $?-connected(b,Y)$ with respect to the PROLOG program of Figure 3.1. $connected(b,e)$ is one the query answers. The base facts that are participating to generate this answer are $connected(b,d)$ and $connected(d,e)$. If the system stores

all these three linked facts together in a single data structure then there will be no need to track the call dependency between the PROLOG predicates in the program.

2. Avoid repetition of computations

   Whenever a tabled answer is not valid anymore, the answer is removed from the table. If the same answer becomes valid later due the changes in the database of facts and rules, the current approach needs to re-compute the answer and add it to the table. An alternative approach would be that instead of removing the answer from the table, we label the answer as invalid (OUT). Later, if the answer becomes valid all what we have to do is to change the label to valid (IN).

3. Maintaing the correctness of tabled answers with minimal or no inference work from PROLOG side

   If we cache the PROLOG facts and rule that participate in generating the answer along with the answer and label them, then maintaing the correctness of cached answers would require no efforts from the PROLOG inference engine.

4. Handling the assertion/retraction of rules in efficient manner

   To handle the assertion/retraction of rules, the system needs to track the rule that causes the generation of each cached answer, when the rule is asserted/retracted from the program the cached answers label should be updated accordingly.

5. A way to filter the answers of parent query rather than re-evaluating the sub query from scratch

   To resolve this issue, each cached answer of a query should be linked to the query node. When a subquery is evaluated, the system refines the parent query answers that can be unified with the head of the subquery and link them to the subquery node.

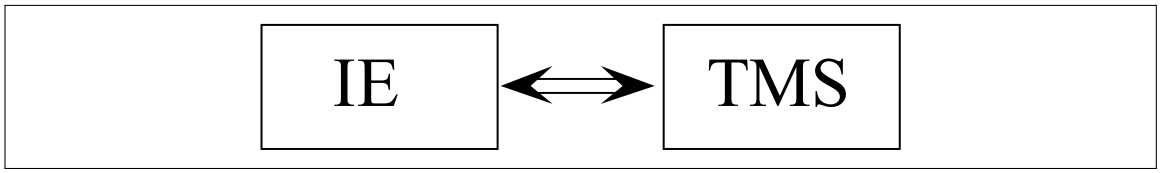

Figure 3.2: An overview of Problem Solving Systems

To summarize our needs, we are looking for a sub-system that can be integrated with PROLOG engine to support incremental evaluation of tabled Prolog programs. The sub-system should be able to do the following:

1. Represent the PROLOG predicates (facts, rules, queries) as nodes and link them together in a dependency network style. The objective is to link the facts and rule participating as antecedents to generate the answer of the query(consequence).

2. Record the current believes (query answers) and maintain (correct) the believes as the database of facts and rules are changed in the program.

3. Add new believes as the database of facts and rules are changed in the program.

Looking into the above requirements, we found that the Justification Based Truth Maintenance Systems (JTMS) [Forbus & de Kleer, 1993, Brewka et al., 1991] can help us to achieve our goals. A brief description of Truth Maintenance Systems (TMS) and Justification Based Truth Maintenance Systems (JTMS) is provided in the next section before we dig into the design of our system.

3.2 Truth Maintenance Systems

The aim of this section is to give a brief introduction about the Truth Maintenance Systems, also called Reason Maintenance Systems. Truth Maintenance Systems [Doyle, 1979], or TMS, are used within Problem Solving Systems [Forbus & de Kleer, 1993], in conjunction with Inference Engines (IE) such as rule-based inference systems like PROLOG, to manage the inference engine's beliefs in given sentences as a Dependency Network.

Figure 3.2 gives an an overview of Problem Solving Systems that uses TMS along with IE.

A TMS is a knowledge representation method for representing both beliefs and their dependencies. A TMS is intended to satisfy a number of goals. One of these goals is the ability to remember derivations computed previously. It may happen that the same question is being asked from the problem solver over and over. If the previous knowledge is not cached when the question was answered for the first time, then the IE needs to re-compute the knowledge again and again. But if the previous knowledge was in the knowledge base, then there is no need for retracing the same knowledge. The use of TMS can avoid such retracing. Note that this feature of the truth maintenance systems can be used to support tabulation in PROLOG. When the query is proved for the first time, the answers can be cached as a truth maintenance network.

3.2.1 Justification-Based Truth-Maintenance Systems

JTMS is the simplest type of TMS where one can examine the consequences of the current set of assumptions. JTMS is a domain-independent belief revision system [Shapiro, 1998] which is usually coupled with an inference engine that does the actual inference work. JTMS operates on propositional objects and is used to record and maintain dependencies between deductive inferences. This can be done by representing deductive dependencies as a JTMS network.

3.2.1.1 JTMS Nodes and Justifications

The basic JTMS specification can be given in terms of two sets: the set of enabled assumptions (domain propositions) and the set of justifications. Propositions of the domain are mapped into nodes where each node is labeled either *IN* or *OUT* depending on whether or not it is currently assumed. Nodes corresponding to propositions that the system currently believes in are labeled *IN* while currently disbelieved propositions are labeled *OUT*. A node is labeled *OUT* by default but JTMS may label a node as *IN* in exactly two cases:

either by a request from the inference engine or if there exists an active justification that supports the node.

In order to form a JTMS network the nodes are linked by justifications. A justification is a structure that is responsible for recording a single inference. A justification has two sets of nodes, the $in-list$ and the $out-list$ as its antecedent and a single node as its consequent. A justification is said to support its consequent node. Note that it is possible to have multiple justifications supporting the same node. An active justification is a justification where all the nodes in the $in-list$ are labeled IN and all the nodes in $out-list$ are labeled OUT. The consequent node of an active justification will be labeled IN while the consequent node of an inactive justification will be labeled OUT unless it has another active justification supporting it.

3.2.1.2 JTMS and Non-Monotonic Logic

Search is one of the most important needs of problem solvers. Usually the problem solvers suffer from retracing conclusions. If a problem solver cached its inference, then it would not need to retrace conclusions that it had already derived earlier in the search. By caching the inferences, the problem solver avoid throwing away useful results and avoid wasting effort rediscovering the same things over and over. One of the JTMS goals, inherited from the TMS, is that JTMS is able to remember derivations computed previously. JTMS can do this by caching the inferences. This effort of JTMS can help in implementing incremental tabling features for PROLOG that will work with non-monotonic logic programs. The idea is that instead of remembering the end results as traditional memoing implementations does, the PROLOG inference engine caches its inferences by the help of JTMS. By caching the inferences, JTMS will reflect any change in data through its network to keep the inferences updated. The responsibility of JTMS is answering queries correctly with respect to the contents of JTMS nodes and justifications at the moment the query is made.

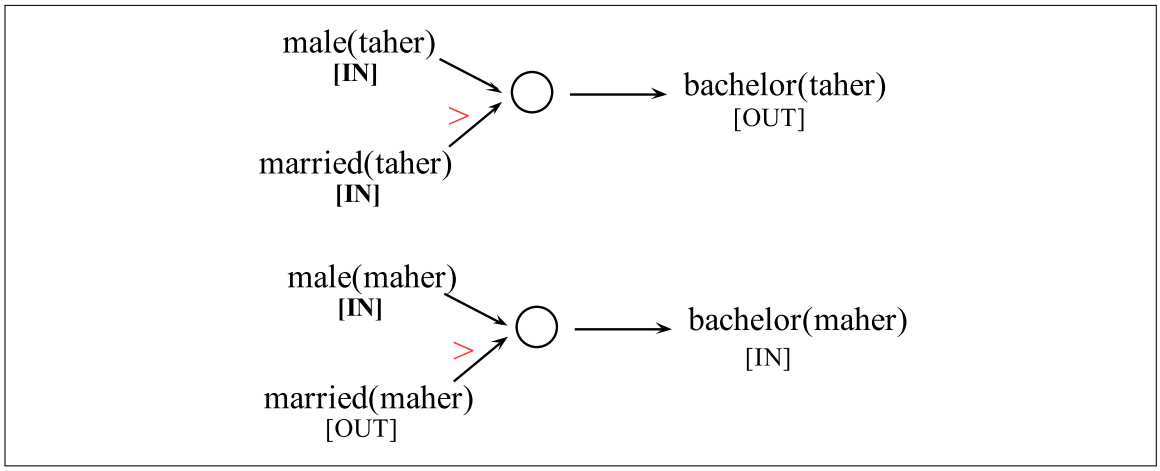

Figure 3.3: JTMS network for proof tree of Figure 2.2

Example 3.1

Figure 3.3 shows an example of a justification network for the proof tree in Figure 2.2. Nodes are shown by printing their corresponding domain atoms. Justifications are shown as circles. The antecedents of a justification are identified by arrows pointing towards the justification while the consequent is pointing away from the justification. A negative literal in the antecedent (i.e. a member of the justifications $out - list$) is identified by placing a $\neg$ sign on top of the arrow pointing to the justification. Figure 3.3 also shows the current labeling of the nodes. Labels that are printed in bold are specified by the inference engine while the rest are assigned by the JTMS. Note that JTMS network of Figure 3.3 has two justifications that correspond to the two proof branches of Figure 2.2 proof tree, whether or not the proof branch was successful. This will allow the JTMS network to reflect any changes in data. (i.e. the query results are always correct and updated).

Coming back to the example of Figure 3.3, consider that $married(maher)$ is asserted to the database of PROLOG facts in Figure 2.2. JTMS reflect this change through it's network in order to keep the network updated. JTMS is capable of updating (revising) its belief $bachelor(maher)$ without invoking the inference engine by propagating the changed value through the network. The resultant network is shown in Figure 3.4.

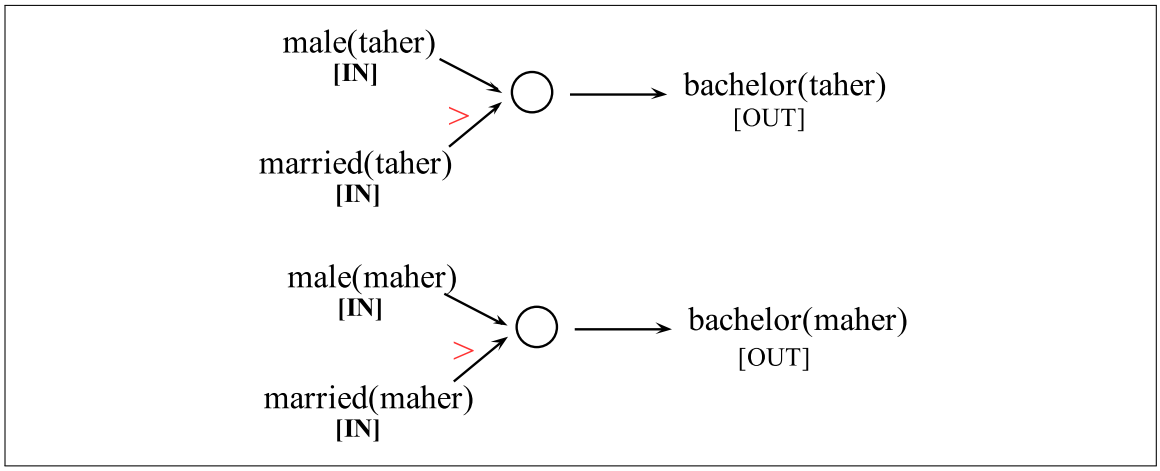

Figure 3.4: JTMS network of Figure 3.3 after asserting *married*(*maher*) to the database of PROLOG facts in Figure 2.2.

3.3 JLOG: Incremental Evaluation of Tabled PROLOG Programs

This thesis presents a novel approach to incremental tabulation that is capable of working in non-monotonic situations. *The main idea is to cache the proof generated by the deductive inference engine rather than the end results.* In order to be able to efficiently maintain the proof up-to-date, the proof structure is converted into a justification-based truth-maintenance (JTMS) network. JTMS saves the dependency between deduced facts and the facts used to make the deduction in order to be able to efficiently cache the proof structure. The system translates every successful branch of a query into a JTMS network that links the facts and the rule used in the branch to the answer generated by that branch.

We named our system as JLOG (Justification-based Logic), the idea of this name came from the word PROLOG (Programming in logic). Another possible name that can suit out system would be JLP (Justification-based logic programming). Before describing details of the design of JLOG along with algorithms needed to handle incremental tabulation, an illustrative example will follow which shows how JLOG handles soundness and completeness of query proof structure when the database is changed.

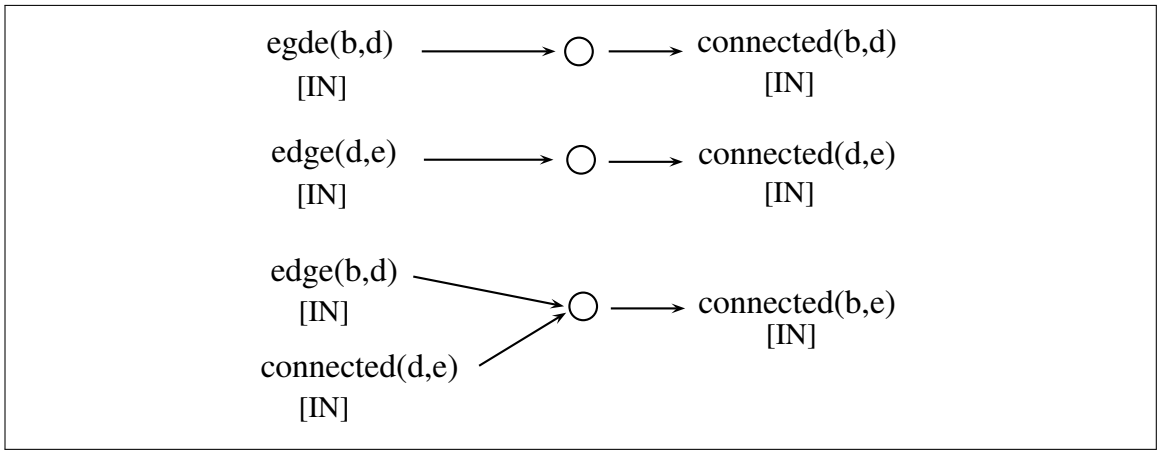

Figure 3.5: JTMS network installed by JLOG after proving query $?-connected(b,Y)$ for the first time.

Example 3.2

Figure 3.5 shows the justifications installed by JLOG when it proves the query $?-connected(b,Y)$ with respect to the PROLOG program of Figure 3.1(a). These justifications represent the proof structure of the query $?-connected(b,Y)$. A justification is installed for each complete branch of the SLD-tree. When a query is re-evaluated, JLOG returns the answers of the query by collecting the IN consequences of each query's Jtms justification. When changes in database take place, JLOG has to ensure that the proof structure is both sound and complete.

Example 2.4 shows how incremental evaluation deals with tabled answers of the query $?-connected(b,Y)$ when the base facts get asserted/retracted to/from the database of facts with respect to the PROLOG program of Figure 3.1. We will consider the same changes of Example 2.4 in the base facts and show how those events are handled by JLOG to maintain soundness and completeness of query's cached proof structure:

1. Retract the fact $edge(b,d)$

   JLOG has to ensure that whenever base facts which participate as antecedents in any justification are asserted/retracted, the effect of this assertion/retraction should be propagated through the JTMS justifications in order to keep the proof structure sound. Achieving this is not difficult since changing the state of any antecedent

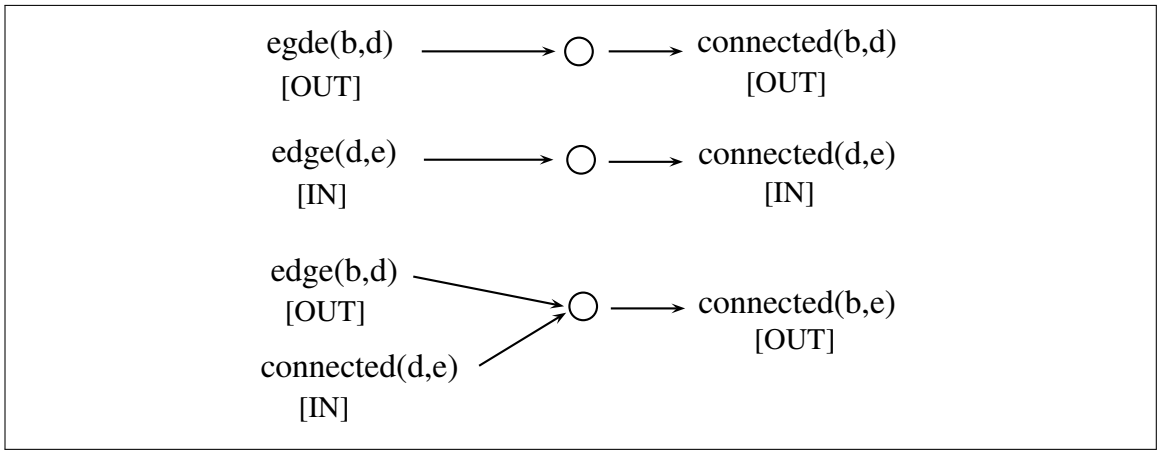

Figure 3.6: JTMS network of Figure 3.5 after retracting the fact $edge(b,d)$ to the database of Figure 3.1.

that is asserted/retracted to/from the database requires marking the label from IN-/OUT or vice versa, and after that, propagating the effect of this change through the whole network. Figure 3.6 shows the effect of retracting the fact $edge(b,d)$ from the database of the Figure 3.1. The first effect of this retraction is on the first justification since $edge(b,d)$ is in the antecedent list of that justification. This results in marking $connected(b,d)$ from IN to OUT. Since $edge(b,d)$ is in the antecedent list of the 3*rd* justification, the result of outness propagation marks $connected(b,e)$ from IN to OUT. This method of propagating inness/outness ensures the returned results by JLOG are valid answers regardless of changes in the database. Note that ensuring the soundness of the proof structure does not require any PROLOG inference work.

2. Assert the fact $edge(b,f)$

   Here the situation is more complicated. JLOG has to take care about the effect of asserting new data that was not available when a query was evaluated for the first time. This is important since asserting new data to the database may add to the set of results already available for the query or even remove some of the results. JLOG handles this problem by monitoring nodes that may contribute to some new results of a query. Whenever a new fact that is related to a monitored node is asserted, query

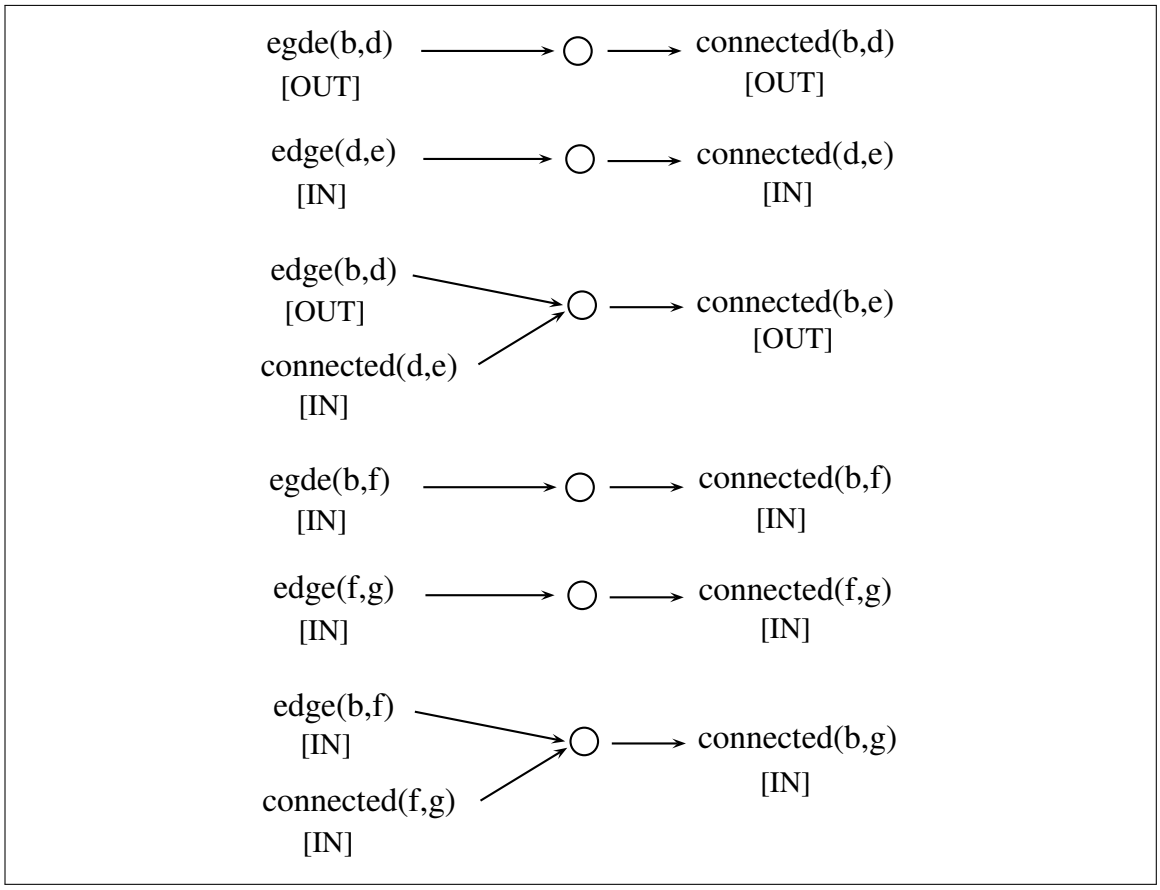

Figure 3.7: JTMS network of Figure 3.6 after asserting the fact $edge(b,f)$ to the database of Figure 3.1.

resumption takes place to update the query's cached proof structure. Referring back to the example of Figure 3.1, when JLOG proves the query $?-connected(b,Y)$ for the first time, it marks the nodes that may participate in resuming this query when new data is asserted. Those nodes come from the right hand side of program rules, i.e. $edge(X,Y)$ and $connected(Z,Y)$. Whenever new data that is related to the marked nodes is asserted, the query $?-connected(b,Y)$ resumes its work to update the proof structure of the query. Figure 3.7 shows the effect of asserting the fact $edge(b,f)$ to the database of Figure 3.1 on the JTMS network of Figure 3.6. Three new justifications have been installed upon resuming the query after the assertion of $edge(b,f)$. An important point that should be mentioned here is that, in order to keep the proof structure complete, JLOG has to use the help of the PROLOG inference engine.

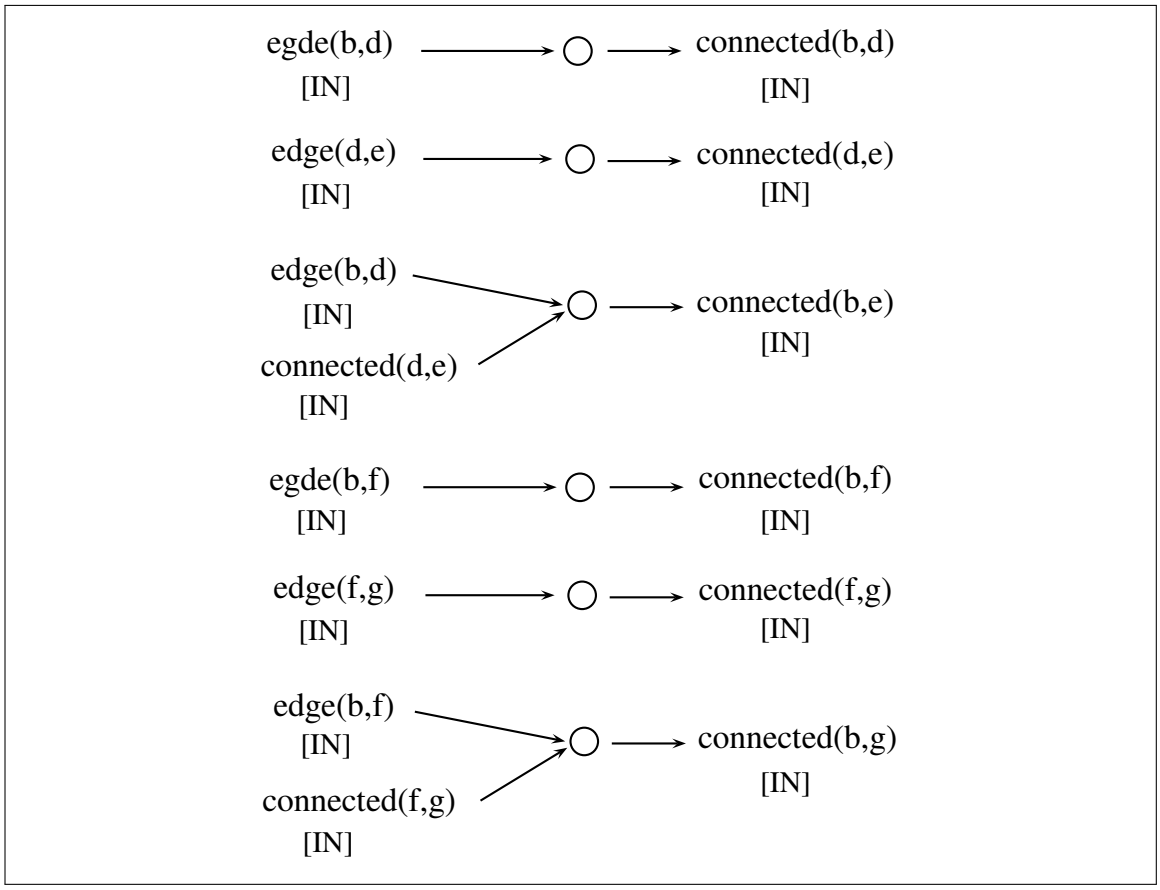

Figure 3.8: JTMS network of Figure 3.7 after asserting back the fact $edge(b,d)$ to the database of Figure 3.1.

3. Assert the fact $edge(b,d)$

   The retracted fact $edge(b,d)$ is asserted back to the base facts of Figure 3.1. JLOG is going to change the label of the TMS node attached to this fact from OUT to IN and then propagates the effect of this change in label through the JTMS network. Figure 3.8 shows the effect of asserting back the fact $edge(b,d)$ to the database of Figure 3.1 on the JTMS network of Figure 3.7.

```
:-dynamic likes/3, mayVote/2, votes/3.
likes(ali, p1, educationPlan). %F1
likes(ali, p2, educationPlan). %F2
likes(ali, p2, healthPlan). %F3
likes(baba, p2, healthPlan). %F4
likes(baba, p1, foreignPolicyPlan). %F5
mayVote(X,Y) :- likes(X,Y,educationPlan). %R1
mayVote(X,Y) :- likes(X,Y,healthPlan). %R2
votes(X,Y) :- likes(X,Y,educationPlan), likes(X,Y,healthPlan). %R3
```

Figure 3.9: Voting Decisions Program: an example of a Non-Monotonic logic.

3.4 An Introductory Description of JLOG

The main idea of JLOG is to cache the proof generated by the deductive inference procedure (PROLOG) rather than just the end results. In order to be able to efficiently maintain the proof up-to-date, the proof is converted into a justification-based truth-maintenance (JTMS) network. This idea of caching the proof structure supports non-monotonic logic systems where base facts or rules can be either added or deleted.

Example 3.3

Figure 3.9 shows an example of a small PROLOG program that can be used during election campaigns to have ideas about voters preferences for the selection of candidates, or a political party. The program in Figure 3.9 is a good example of non-monotonic logic where base facts ($likes/3$) and rules ($mayVote/2, votes/2$) may change, get asserted or retracted, as the election campaigns are moving on. The main idea of JLOG is to cache the proof generated by the deductive inference engine rather than the end results. In order to be able to efficiently maintain the proof up-to-date, the proof is converted into a justification-based truth-maintenance (JTMS) network.

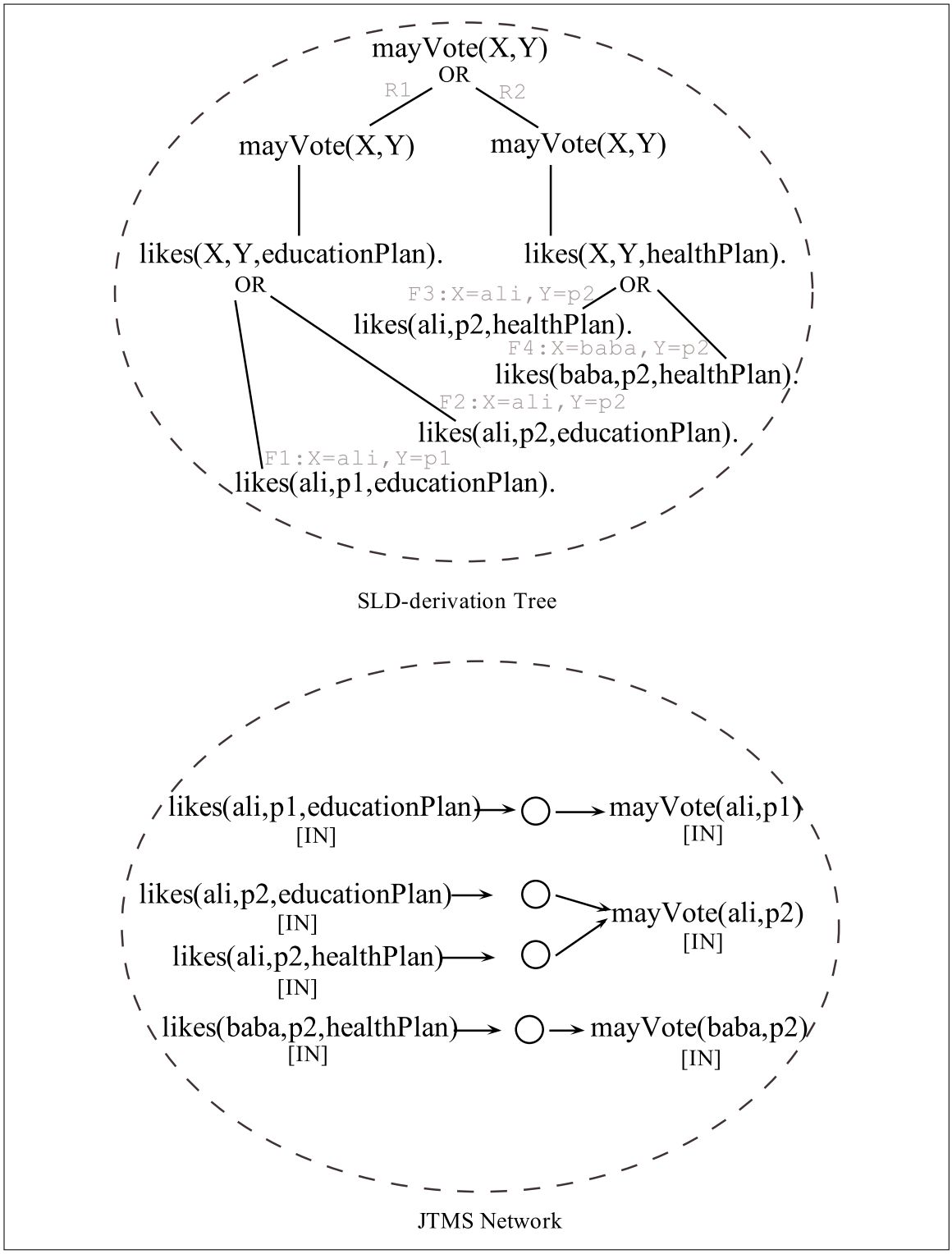

Figure 3.10: SLD-derivation tree for the query $?-mayVote(X,Y)$ with respect to the PROLOG program of Figure 3.9 and the justifications installed by JLOG when the query is proved for the first time.

Proving the Query for First Time

Consider the evaluation of a query $?-mayVote(X,Y)$ with respect to the PROLOG program of Figure 3.9. Figure 3.10 (upper oval) shows the successful branches of the SLD-derivation tree for the query $?-mayVote(X,Y)$ when the query is evaluated by the PROLOG query engine. JLOG returns the answers generated by the query engine and meanwhile the proof structure is cached for future evaluations of the same query. Note that JLOG reports repeated answers only once, e.g. $mayVote(ali,p2)$ is reported once although it is reported twice in a normal PROLOG implementation. Figure 3.10 (right oval) shows the justifications installed by JLOG when the query $?-mayVote(X,Y)$ is proved for the first time. These justifications represent the proof structure of the query $?-mayVote(X,Y)$. A justification is installed for each successful branch of the SLD-derivation tree. JLOG translates every complete branch of the SLD-tree into a TMS network that links the facts used in the branch to the answer generated by that branch. Note that there is a possibility of having two or more different branches that lead to the same consequence TMS node, the node $mayVote(ali,p2)$ in Figure 3.10 has two supported justifications.

Re-evaluating the Query

When a query is re-evaluated, JLOG has to ensure that no inference work is in place to generate the answers for the query. Further more, the system must support detection of call similarity for the cached query via two methods:

1. Determining Call Similarity via Variance

   JLOG determines that a subgoal S is similar to a tabled subgoal S_{tab} if S is a variant of S_{tab}, that is if S and S_{tab} are identical up to variable renaming. As an example $votes(X,Y)$ is a variant of $votes(A,B)$, but not of $votes(A,A)$. Under the call similarity via Variance, when a tabled subgoal S is encountered, a search for a table entry containing a variant subgoal S_{tab} is performed. If S_{tab} exists, then all of its

answers are also answers to S, and therefore will be resolved against it.

2. Determining Call Similarity via Subsumption

   Call similarity can also be based on call subsumption. A term $T1$ subsumes a term $T2$ if $T2$ is more specific than $T1$. Furthermore, we say that $T1$ properly subsumes $T2$ if $T2$ subsumes $T1$, but is not a variant of $T1$. For example, the term $votes(X,Y)$ subsumes $votes(X,p2)$ and $votes(ali,Y)$. Under call subsumption, when a tabled subgoal S is encountered, a search is performed for a table entry containing a subsuming subgoal S_{tab}. Notice that, if such an entry exists, then its answer set A of S_{tab} logically contains all the solutions to satisfy S. The subset of answers $A' \subseteq A$ which unify with S are said to be relevant to S.

In order to support the above strategy the JTMS network is expanded to include information that will help to detect call similarity via variance and subsumption. This is achieved by installing general terms for each deduction. Figure 3.11 shows the full expanded JTMS network of Figure 3.10. Consider the re-evaluation of the query $?-mayVote(X,Y)$, its variances or subsumptions. JLOG tries to locate the TMS node corresponding to the query $?-mayVote(X,Y)$ or its special cases (variances/subsumptions). If the node is located in the JTMS network then JLOG returns the answers of the query by collecting the IN consequence leaves of the query's corresponding TMS node . If the corresponding TMS node is not found in the JTMS network then JLOG assumes that the query is being proved for first time which requires inference work from PROLOG query engine.

The main problem with the above approach of expanding the JTMS network, that it can lead to exponential space overhead. For example if a PROLOG fact contains four arguments, then in order to build the JTMS network using full expanded scheme we need 2^4 nodes. An alternative solution is that each term points only to it's most general term. Figure 3.12 shows the partial expanded JTMS network of Figure 3.10. Now lets examine again the re-evaluation of the query $?-mayVote(X,Y)$, its variances or subsumptions. JLOG treats the query $?-mayVote(X,Y)$ or its variances in similar fashion of full ex-

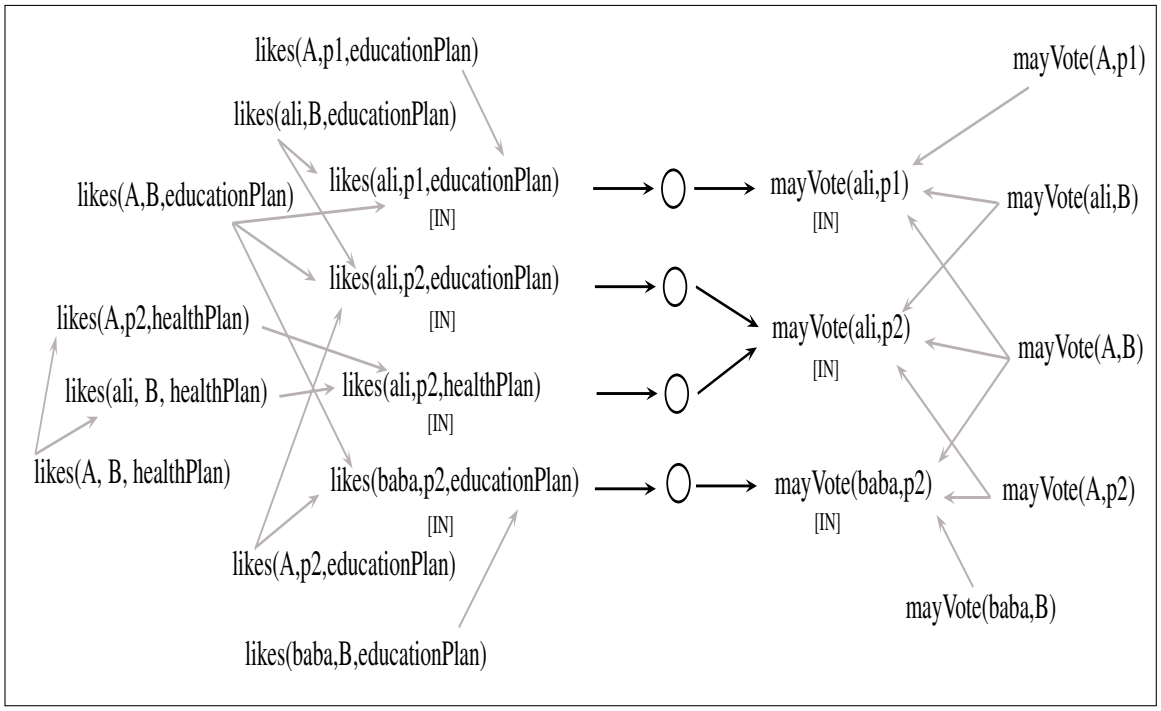

Figure 3.11: The full expanded JTMS network installed by JLOG after proving the query $?-mayVote(X,Y)$ for the first time.

panded network, however the treatment of subsumptions is different. When a subsumption S of the query $?-mayVote(X,Y)$ is evaluated, JLOG starts searching for the most general term S' for S. If the node S' is located in the JTMS network then JLOG returns the answers of the query by collecting the IN consequence leaves of the query's corresponding TMS node that are unified with S. The drawback of this approach is that any subquery S related to a cashed query S' requires extra efforts from the system to refine the matched results.

The above two approaches of expanding the JTMS network depicts the typical problem of a space–time [Avoine et al., 2008], or time–memory, tradeoff in computer science field. The first approach of expanding the full network gives advantage to subsumption queries but the payoff is space overhead. This scheme allows to return the answers of a sub-query without the need of filtering the parent query answers. The partial network approach saves space but needs extra time for refining the results for subsumption queries . When the sub-query is evaluated for the first time under this scheme, the system must filter the answers of the parent query and then link them to the TMS node of the sub-query.

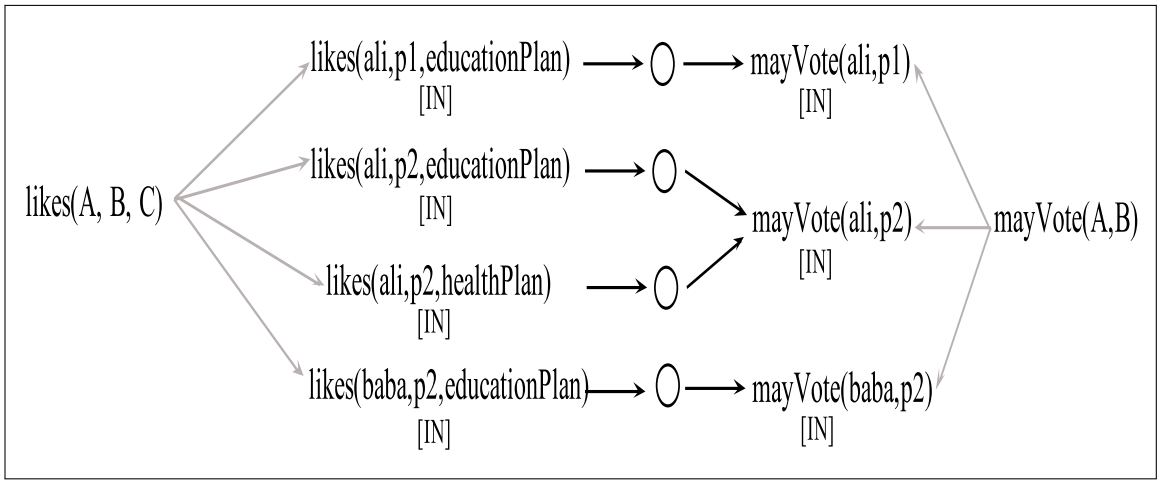

Figure 3.12: The partial expanded JTMS network installed by JLOG after proving the query $?-mayVote(X,Y)$ for the first time.

Change that Affects the Cashed Query Proof

Going back to example of Figure 3.9, the usual scenario is that preferences of voters might change as the election campaign are moving on. This is going to change the database of facts and rules, that is new facts/rules are asserted or existing facts/rules are retracted. As the database changes, two things may happen to the proof structure: successful branches can no longer succeed, or failed branches can become successful. There are four cases that can affect the cached proof structure of a particular query:

1. Assertion/Retraction of base facts/rules participated as antecedents in any justification.

2. The right hand side of an existing rule participated as an antecedent in any justification is updated.

3. Assertion of new facts/rules such that they do not conflict with existing facts/rules.

4. Assertion of a new rule which conflicts with an existing rule participated as antecedent in any justification.

When changes in database take place, JLOG has to ensure that the proof structure is both sound and complete. The next two sections focuses on how proof soundness and completeness of a perviously proven query is achieved by JLOG.

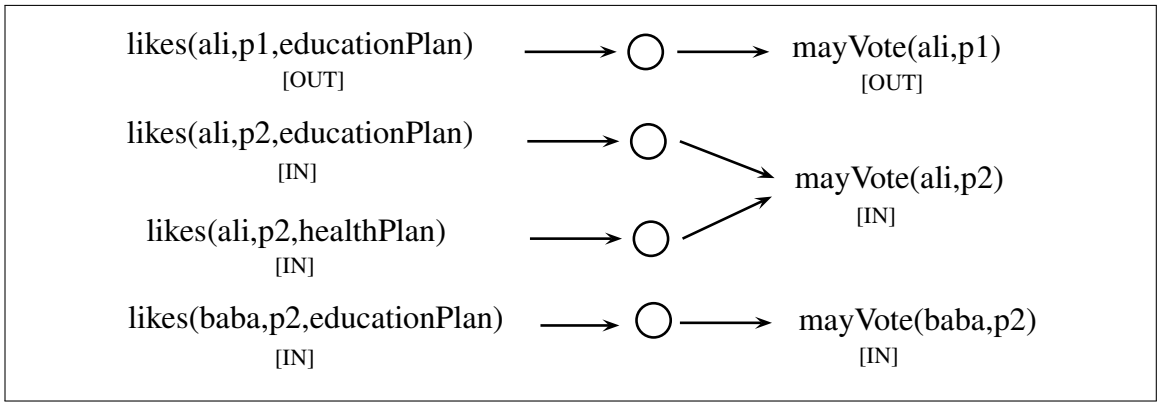

Figure 3.13: The JTMS network of Figure 3.10 after retracting the fact $likes(ali, p1, educationPlan)$ from the database of facts in Figure 3.9.

3.4.1 Maintaining Soundness of the Proof Structure

JLOG has to ensure that whenever base facts/rules participating as antecedents in any justification are asserted/retracted, the effect of this assertion/retraction should be propagated through the JTMS justifications in order to keep the proof structure sound. An important point to be highlighted is that soundness of proof structure does not deal with assertion/retraction of new facts/rules that do not exist when the query is proved for the first time. First let's consider the case of asserting/retracting of facts and then asserting/retracting of rules.

Assertion/Retraction of Exisiting Facts

In this case, achieving soundness of the query's proof structure is not difficult since changing the state of any antecedent that is asserted or retracted to/from the database of facts requires marking the label from IN/OUT or vice versa, and after that, propagating the effect of this change through the whole network. Figure 3.13 shows the effect of retracting the fact $likes(ali, p1, educationPlan)$ from the database of the Figure 3.9. The first effect of this retraction is on the first justification since $likes(ali, p1, educationPlan)$ is in the antecedent list of that justification. This results in marking $likes(ali, p1, educationPlan)$ from IN to OUT. Since mayVote(ali,p1) is in the consequence list of the same justification, the result of outness propagation marks mayVote(ali,p1) from IN to OUT. This method of propagating inness/outness ensures that whenever the query is re-evaluated, the re-

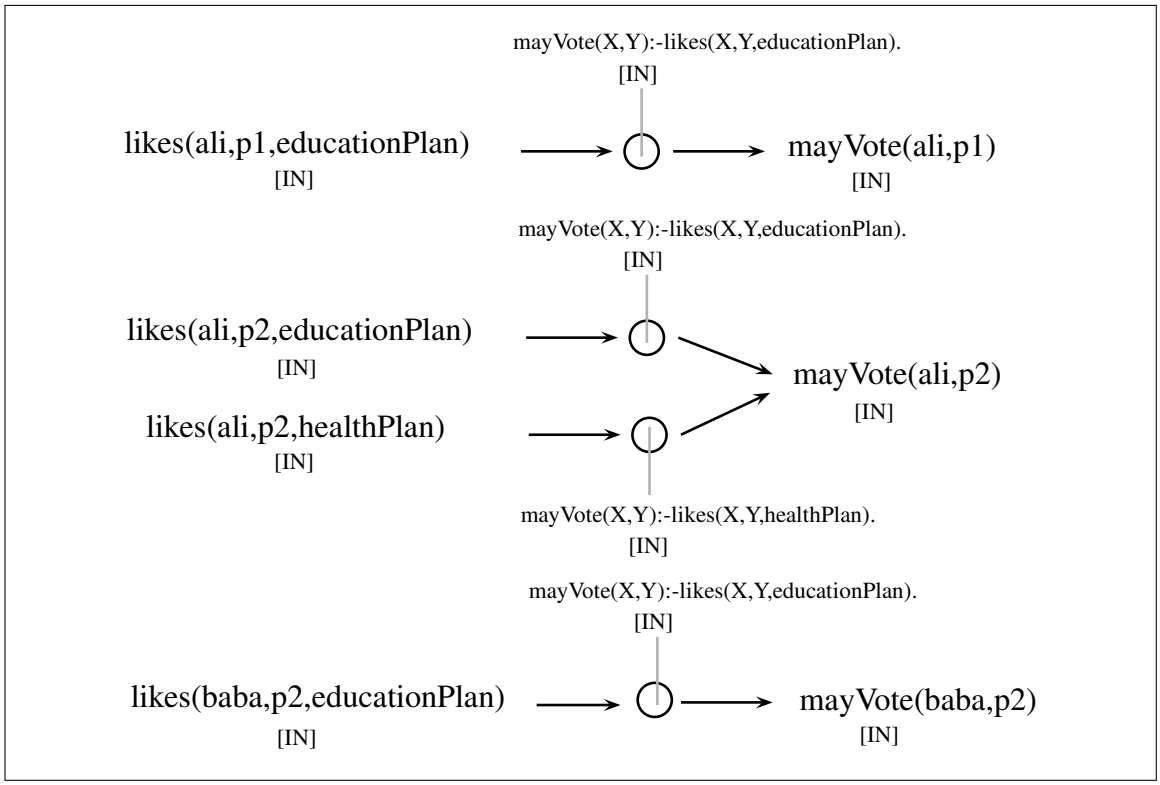

Figure 3.14: The JTMS network of Figure 3.10 after incorporating rules information that is related to each justification.

sults returned by JLOG are valid answers whether or not the database has been changed. A significant point is that, the above strategy does not require any inference work from PROLOG side.

Assertion/Retraction of Existing Rules

In this case the situation is more complicated. In order to handle the assertion/retraction of rules, each justification of query's JTMS network must remember information about the rule(s) involving in it. Figure 3.14 shows the JTMS network of Figure 3.10 after incorporating rules information that is related to each justification. Initially all rules are marked as IN when the justification related to the rule is installed in the system for the first time. Then JLOG handles assertion/retraction of rules according to the following:

1. When a rule gets retracted from the database of rules, JLOG marks the label attached to the rule from IN to OUT, then it finds all justifications in the JTMS network re-

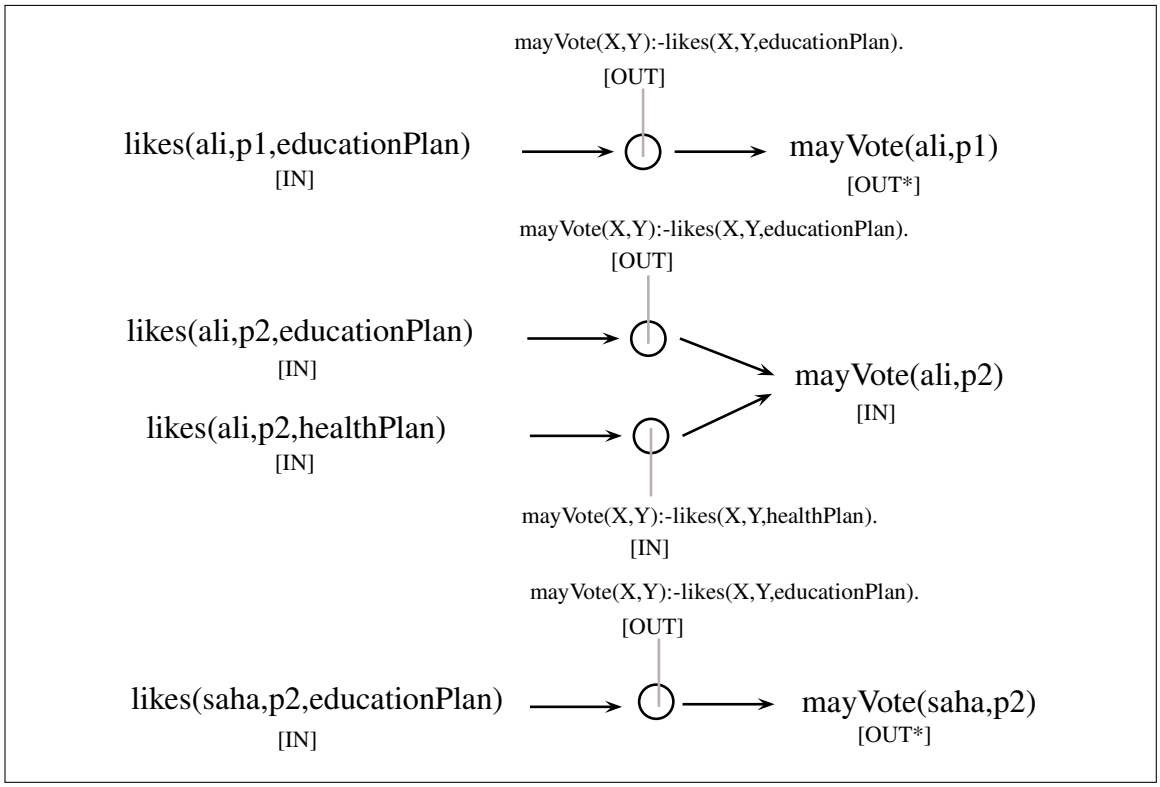

Figure 3.15: The JTMS network of Figure 3.14 after retracting the rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$ from the database of the Figure 3.9.

lated to the retracted rule. For each such justification JLOG marks the consequence of the justification to OUT* indicating that the consequence is false, not because of its antecedent list, but because the rule that is behind this consequence is retracted from the database. Figure 3.15 shows the JTMS network after retracting the rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$ from the database of the Figure 3.9. The first effect of this retraction is the change of rule's label form IN to OUT. Since the rule is participating in three justifications, the consequences of all these three justifications are marked from IN/OUT to OUT* except if a consequence has another supporting branch. The result of these actions is shown in Figure 3.15, the labels of $mayVote(ali,p1)$ and $mayVote(baba,p2)$ are marked as OUT* while the label of $mayVote(ali,p2)$ is not changed since it has another justification that supports it.

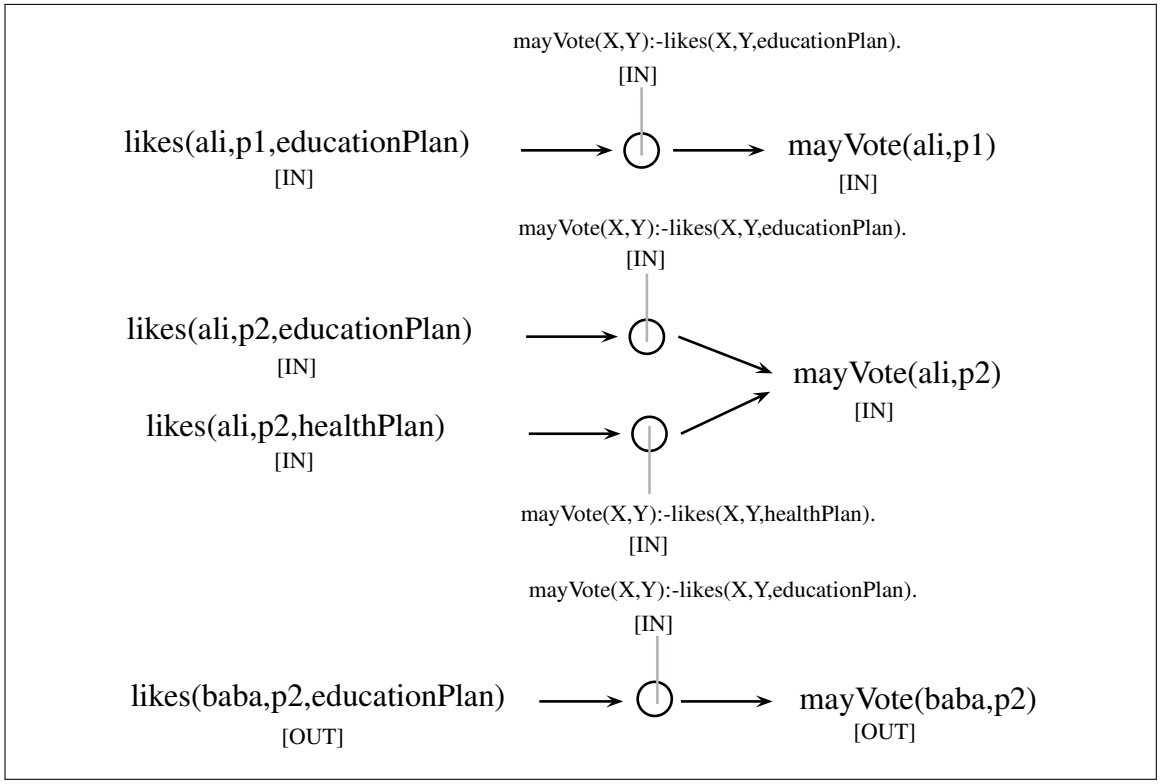

Figure 3.16: The JTMS network of Figure 3.15 after retracting the fact $likes(baba, p2, educationPlan)$ and asserting the rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$.

2. When a rule gets asserted back to the database of rules, JLOG marks the label attached to the rule from OUT to IN, then it finds all justifications in the JTMS network related to the asserted rule, for each such justification JLOG marks the consequence of the justification to IN if all justification's indecent list elements are labeled IN. Figure 3.16 shows the JTMS network of Figure 3.15 after retracting the fact $likes(baba, p2, educationPlan)$ and asserting the rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$. The rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$ is marked from OUT to IN in 3.16. The label of $mayVote(ali, p1)$ is marked as IN while label of $mayVote(baba, p2)$ is marked OUT because it is not supported by the justification incident list.

3.4.2 Maintaining Completeness of the Proof Structure

Ensuring the completeness of the proof structure is the responsibility of the database monitor. The database monitor checks whether an assertion of new facts/rules (TMS node) triggers the resumption of a cached proof. A new fact can be created by an external system (through assertion) or internally by the query engine (intermediate result), whereas a new rule can only be created by an external system.

Assertion of New Facts

Coming back to the PROLOG example of Figure 3.9, consider the situation where the fact $likes(haneen, p1, educationPlan)$ is asserted into the database of PROLOG facts. The database monitor checks if this fact can resume one of the previously proven queries. A fact can resume one of the previously proven queries if the fact or one of its general cases is appearing on the right hand side of a rule involved in proving a previously cashed query. In this example, the query $?-mayVote(X,Y)$ can be resumed or extended. Since the query is already proven, the resumption of the query should be only based on the new data, i.e. JLOG does not resume the general case of the query $?-mayVote(X,Y)$. The database monitor looks for the query rules in the program where this fact or one of its most general node exist, the rule is $mayVote(X,Y):-likes(X,Y,educationPlan)$. JLOG unifies the value of the new fact into the rule, the rule is made more specific into $mayVote(haneen, p1):-likes(haneen, p1, educationPlan)$. After that, the database monitor asks the query engine to execute the right hand side of the above rule, i.e. $likes(haneen, p1, educationPlan)$ which results in adding a new justification and two TMS nodes to the JTMS network of Figure 3.10 as shown in Figure 3.17. See Chapter 4 for more details about how this is implemented in JLOG.

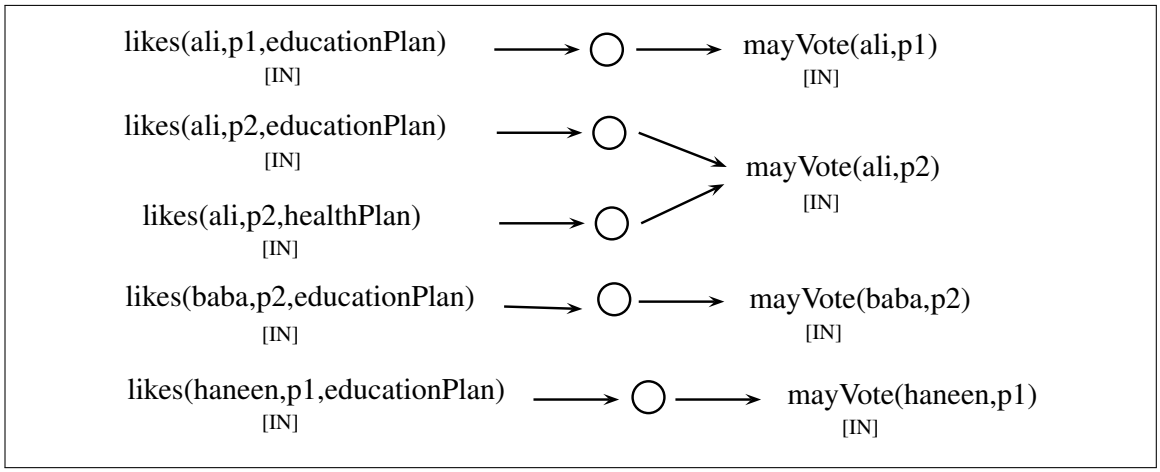

Figure 3.17: JTMS network of Figure 3.10 after asserting the fact $likes(haneen, p1, educationPlan)$. to the database Figure 3.9.

Assertion of New Rules

In this case the database monitor has to deal with three different cases related to rule assertion:

1. A new rule is asserted which does not conflict with any existing rule.
2. An existing rule participated as antecedent in any justification is updated.
3. A new rule is inserted which conflicts with an existing rule that participates as antecedent in any justification.

The rest of this section will cover each of above points in details along with illustrative examples.

A New Rule is Asserted which does not Conflict with any Existing Rule

Referring back to the PROLOG example of Figure 3.9. Let's consider the case where preferences of the voters are changed such that the candidate or the political party with good foreign policy is voter's choice. This change of voter's preference requires assertion of the new rule: $mayVote(X,Y) : -likes(X,Y,foreignPolicyPlan)$ to the database of Figure 3.9. The newly asserted rule does not conflict with any previous rule participated in the JTMS network of Figure 3.14. Any new answers for the query $?-$

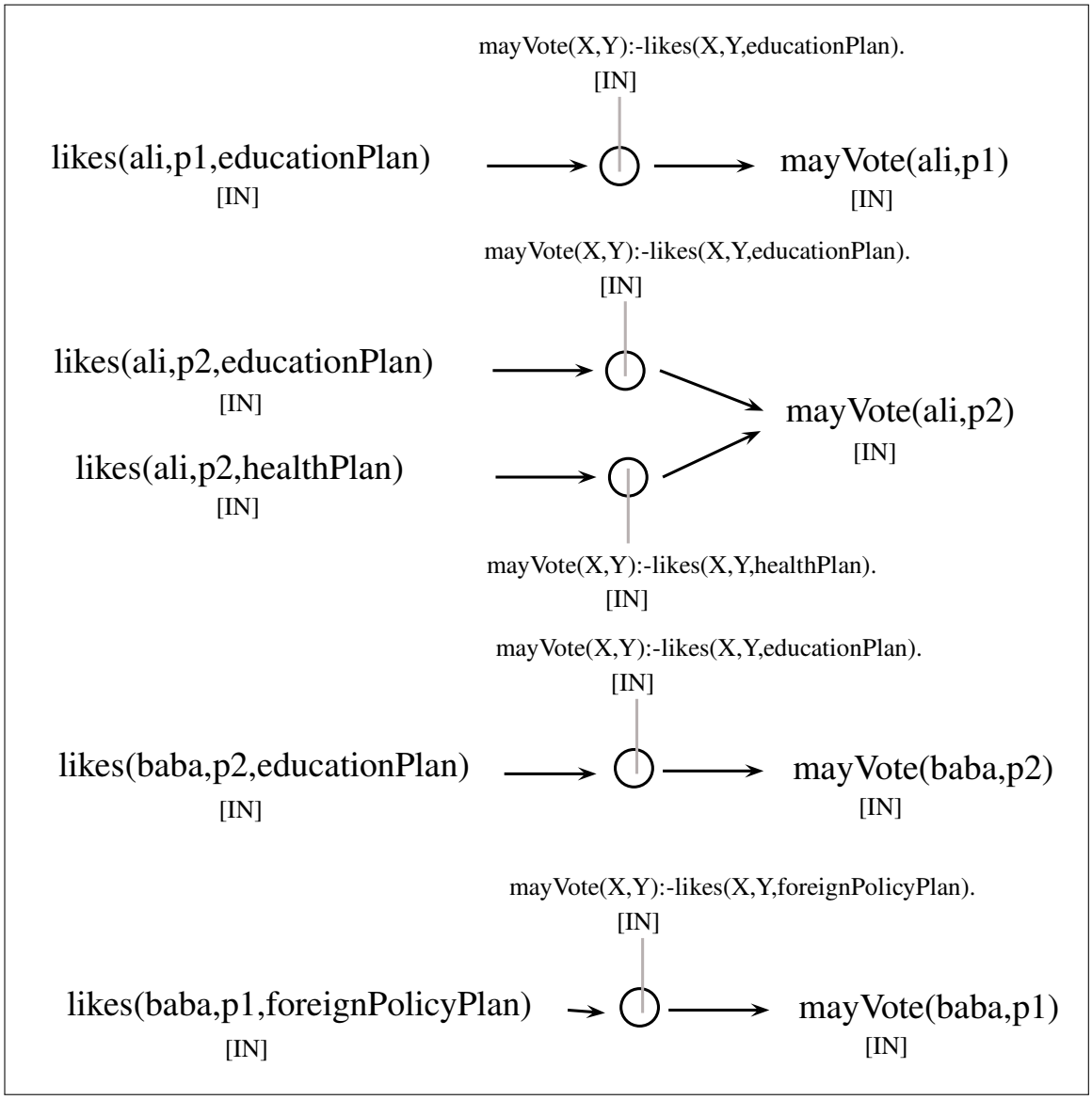

Figure 3.18: JTMS network of Figure 3.14 after asserting the rule $mayVote(X,Y) :- likes(X,Y,foreignPolicyPlan)$. to the database Figure 3.9.

$mayVote(X,Y)$ will come from the right hand side of the new the asserted rule, i.e. $likes(X,Y,foreignPolicyPlan)$. The database monitor requests the query engine to execute $likes(X,Y,foreignPolicyPlan)$ in such away that it returns all the necessary information for installing the new justifications coming out because of this new rule (See Chapter 4 for implementation details). Figure 3.18 shows the JTMS network of Figure 3.14 after asserting the rule $mayVote(X,Y) :- likes(X,Y,foreignPolicyPlan)$. to the PROLOG program of Figure 3.9. A new justification is installed to the cached JTMS network of previously proven query $?- mayVote(X,Y)$.

An Existing Rule Participating as Antecedent in any Justification is Updated

The program in Figure 3.9 states that voters may vote for candidates or parties based on their health or education plan. Let's deal with the case where voter's preferences are changed such that the voters are more concerned now with candidates/parties health or foreign policy plan. This means that the second rule from database of of Figure 3.9 should be changed from:

$mayVote(X,Y) :- likes(X,Y,educationPlan).$

to

$mayVote(X,Y) :- likes(X,Y,foreignPolicyPlan).$

The new situation can be handled in JLOG as two steps process. The first step in the process is to retract the rule:

$mayVote(X,Y) :- likes(X,Y,educationPlan).$

from the database of Figure 3.9 and then assert the rule:

$mayVote(X,Y) :- likes(X,Y,foreignPolicyPlan).$

to the database of Figure 3.9. These two steps are related to both soundness and completeness of a cashed proof of a perviously proven query. The first step is a typical case of maintaining soundness of the proof structure while the second step deals with a normal case of maintaining completeness of the proof structure. Figure 3.19 shows JTMS network of Figure 3.14 after retracting the rule:

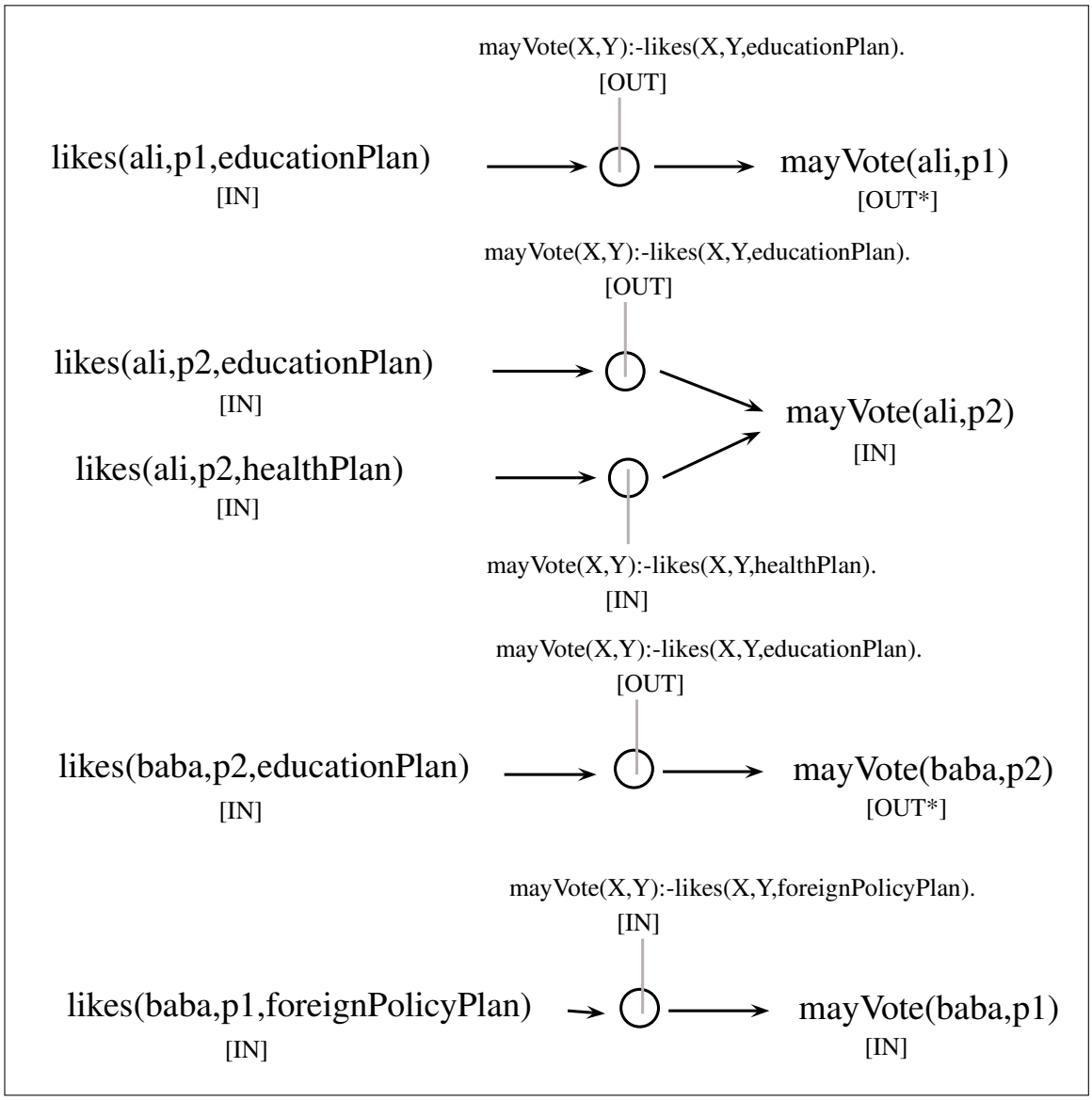

Figure 3.19: JTMS network of Figure 3.14 after retracting the rule $mayVote(X,Y) : -likes(X,Y,educationPlan)$ and asserting the rule $mayVote(X,Y) : -likes(X,Y,foreignPolicyPlan)$ to the database Figure 3.9.

```
:-dynamic dish/1,indian/1,chinese/1,italian/1,iran/1,likes/2.
likes(taher,Food):-dish(Food),indian(Food).
likes(taher,Food):-dish(Food),chinese(Food).
dish(dahl). indian(dahl).
dish(kurma). indian(kurma).
dish(pizza). italian(pizza).
dish(chow_mein). chinese(chow_mein).
dish(chelo_kbab). iran(chelo_kbab).
```

Figure 3.20: A Food Preferences PROLOG Program.

$mayVote(X,Y):-likes(X,Y,educationPlan).$

and asserting the rule:

$mayVote(X,Y):-likes(X,Y,foreignPolicyPlan).$

to the database of Figure 3.9.

A New Rule is Inserted which Conflicts with an Existing Rule that Participates as an Antecedent in any Justification.

This situation is complicated and must be handled carefully. For example look at the food preferences PROLOG program of Figure 3.20. The two rules in the program of Figure 3.20 states that Taher likes all dishes that are either Indian or Chinese. Figure 3.21 shows the JTMS Network installed by JLOG after proving the query:

$?-likes(taher,X)$

for the first time with respect to the PROLOG program of Figure 3.20. The food preferences of people changes from time to time and even on each weekend the food preferences are different. For example, Taher changed his preferences stating that he likes any dish which is not chinese. This statement is asserted to any PROLOG engine as the following rule:

$likes(taher,Food):-dish(Food),not(chinese(Food)).$

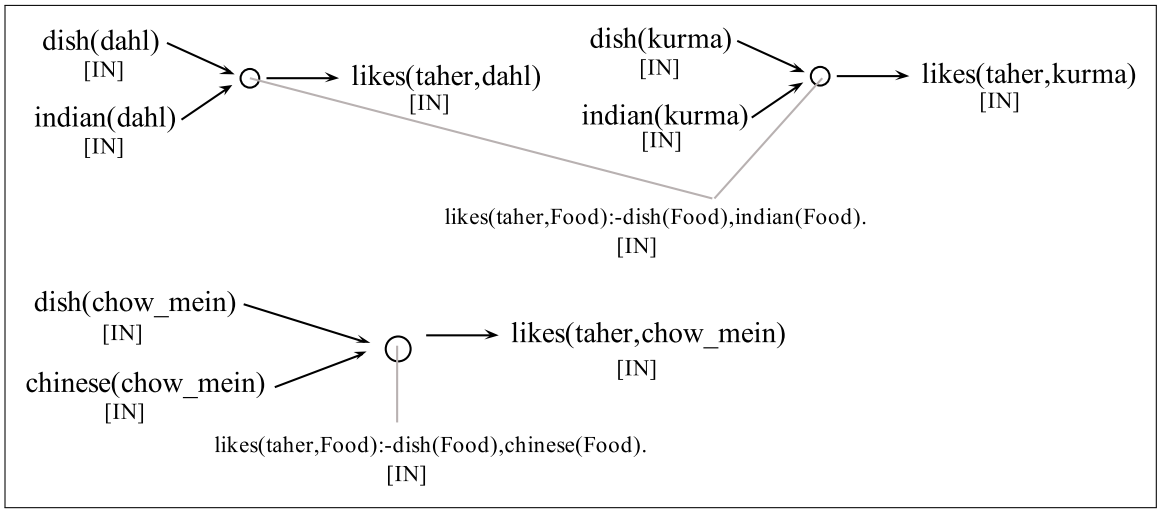

Figure 3.21: JTMS Network installed by JLOG after proving the query $?-likes(taher,X)$ for the first time with respect to the PROLOG program of Figure 3.20.

The new asserted rule conflicts with one of the existing rules:

$likes(taher,Food):-dish(Food),chinese(Food).$

In normal PROLOG systems, this conflicted situation is discarded. If the query:

$?-likes(taher,X)$

is executed after the insertion of above rule which states that Taher does like Chinese dishes, the PROLOG query engine will still respond that Taher likes Chow_mein which is a chinese dish because of the rule:

$likes(taher,Food):-dish(Food),chinese(Food).$

JLOG however can handle this situation in a different way. If the new inserted rule is conflicting with an existing rule that participates as antecedent in any justification, JLOG treats this case by first marking the existing rule OUT and propagate the action throughout the JTMS network, followed by the normal insertion of the new rule. Figure 3.22 shows the JTMS network of Figure 3.21 after asserting the rule:

$likes(taher,Food):-dish(Food),not(chinese(Food))$

into the database of the PROLOG program of FIGURE 3.20.

The main issue of the above mechanism is how to detect the conflicted rules. If the rules are conflicted through the explicit negation then detecting such a conflict is straight forward. For example, the rule:

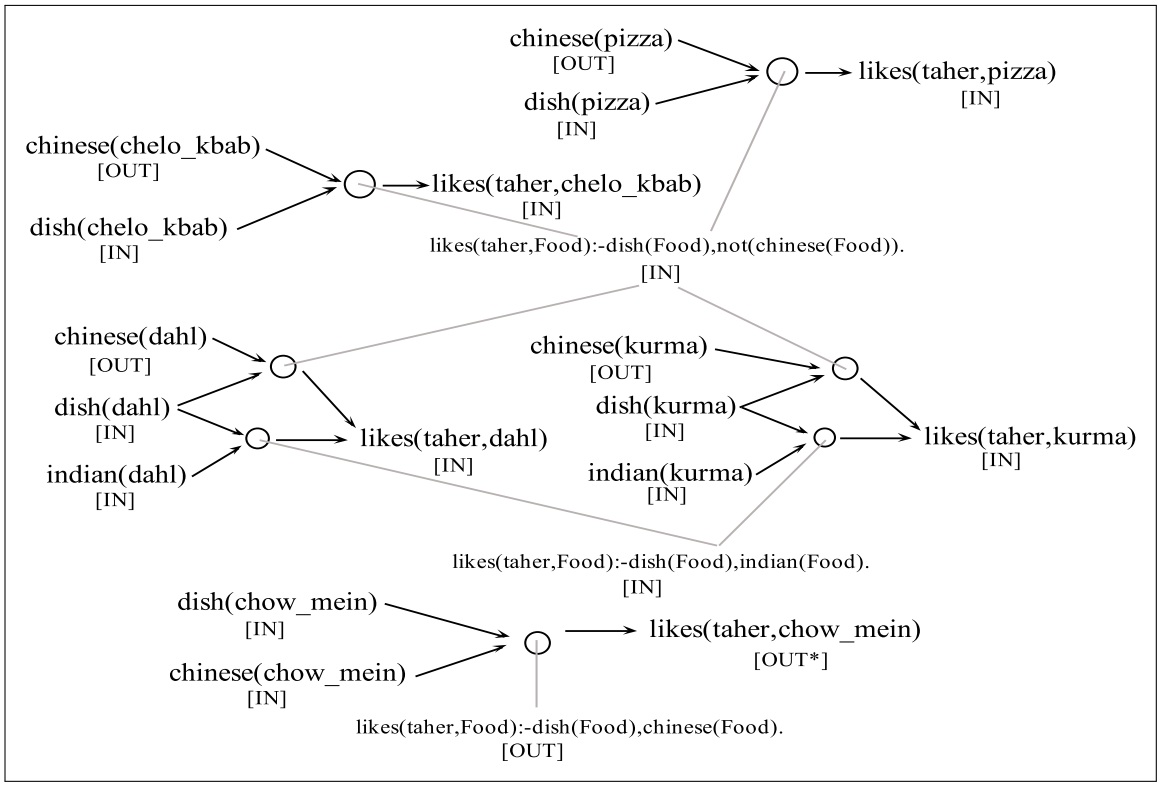

Figure 3.22: The JTMS Network of Figure 3.21 after asserting the rule $likes(taher, Food) : -dish(Food), not(chinese(Food))$ into database of PROLOG program of FIGURE 3.20.

$$likes(taher, Food) : -dish(Food), chinese(Food)$$

and the rule:

$$likes(taher, Food) : -dish(Food), not(chinese(Food))$$

are conflicted through the explicit negation. The difficult situation arises when rules are conflicted through the implicit negation. Currently JLOG supports detection of clashed rules via explicit negation only.

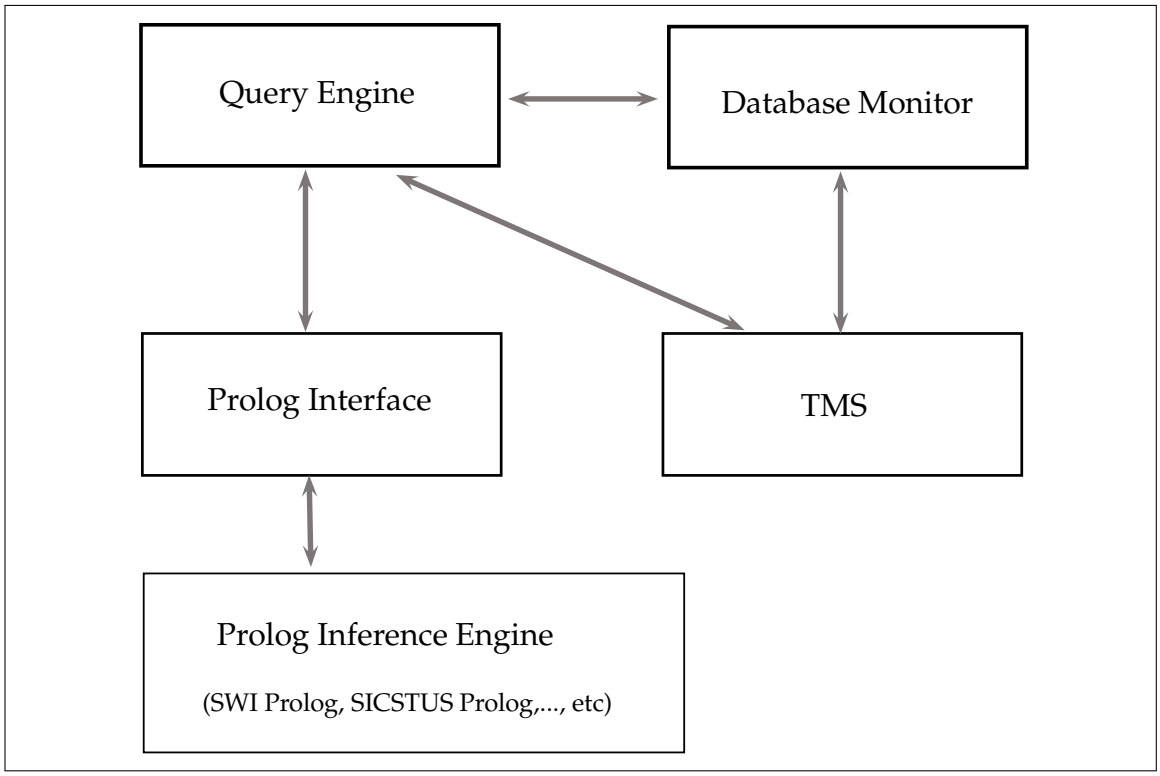

Figure 3.23: An overview of JLOG .

3.5 Detailed Design of JLOG

This section describes how JLOG engine works to support incremental evaluation to PROLOG programs. The section is separated into four subsections, each subsection depicts the features of one of JLOG's main components. The overall design of JLOG is categorized into four main components. The main four component of JLOG are:

1. The PROLOG interface to any well known PROLOG implementation.
2. A TMS component that keeps the proof structure consistent.
3. The database monitor system to keep the proof structure complete when new data is added to the system.
4. The query engine.

The first three components interact through JLOG's query engine to provide the full functionality of the system. Figure 3.23 shows an overview of the JLOG system. JLOG is

mainly designed to handle systems where queries (subgoals) are re-evaluated frequently.

3.5.1 TMS and Justifications Nodes

A computation in PROLOG is initiated by running a query, or a goal, over the database of facts and rules. The proof structure of a query is a set of branches, some are successful branches and others have failed. For every successful branch in a proof, a TMS justification is constructed by JLOG. JLOG translates every complete branch of the SLD-tree into a TMS network that links the facts and rules used in the branch to the answer generated by that branch. Basically, all the clauses that were involved in the derivation should be in the antecedent of the justification, but in JLOG the situation is a little bit different:

- Rather than having a single justification representing the complete derivation branch, JLOG constructs a justification for each of the intermediate goals. The advantage of doing this is that justifications can be shared among proofs.

A TMS node is constructed for every PROLOG fact/rule. There are two alternative approaches regarding the creation time of a TMS node that corresponds to a PROLOG fact/rule:

1. Program load time

   The system constructs a TMS node for every PROLOG fact/rule at program load time. This approach has two advantages. The first advantage is that the overhead of creating the TMS nodes is not going to be part of query execution time. The other benefit is that any query related to PROLOG atoms does not require any PROLOG inference work.

2. Query evaluation time

   Rather than building the TMS network from PROLOG facts/rules before any query is evaluated, the system waits until a query is evaluated. Then the nodes are constructed incrementally as facts/rules get referenced by the query engine. The ad-

vantage of this approach is that a TMS node is created only if it has been referenced by the query engine.

Both approaches have their advantages. The choice is an implementation stage decision. In JLOG, each TMS node is linked to the nodes corresponding the most-general of its atom (See Figure 3.12), so that JLOG maintains the nodes in a generalization hierarchy. Nodes are unique, i.e. equivalent PROLOG atoms are associated with the same TMS node. When a PROLOG fact/rule is asserted, its corresponding node is labeled IN. When a PROLOG atom is retracted, the node is labeled OUT.

3.5.2 Query Engine

As discussed in previous sections the main objective of JLOG is provide a tabled PROLOG system that supports non-monotonic logic. When JLOG proves any query for the first time it takes the help of PROLOG inference engine to prove the query and cache the proof structure of the query in a way that allows JLOG later to maintain the soundness and completeness of the proof structure. Theoretically JLOG can be integrated with any PROLOG engine/system as long the engine supports an interface to any external programming language. The current version of JLOG is implemented using Java platform. JLOG is integrated with XSB, SWI-Prolog and Yap-Prolog through Inter-Prolog. Inter-Prolog provides Java with the ability to call any Prolog goal through a PrologEngine object, and for Prolog to invoke any Java method through a javaMessage predicate.

The Query engine of JLOG is similar to the PROLOG'S standard query engine, however, there are some minor differences:

1. In normal PROLOG, the answers for a query are generated one by one while this is not the case in JLOG, where the set of all solutions is always generated for a query. JLOG needs to keep the track of investigated branches of the proof structure. This is achieved in JLOG by retrieving all the answers for a query at once. The drawback to this is that JLOG can't handle queries with infinite answers.

2. In PROLOG, the order of the rules, facts, and subgoals within a clause affects the order of which answers are generated, but in JLOG there is no specific order of how answers are reported back.

3. The the last difference between PROLOG and JLOG is that each different answer for a query is reported only once in JLOG. However, for each branch generating the same answer reported already, a different justification is still installed. This alternative justification is used to support the answer when the current justification supporting the answer becomes inactive.

When JLOG starts evaluating a query, it first checks if the TMS node associated with this query is tagged as previously proven or not. If yes, then the query engine should collect the valid answers at that moment and return them. The answers of the query are either equal to or more specific than the query itself. Since JLOG builds the general hierarchy of the TMS nodes, the query engine can make use of it in order to locate the answers for the query. Thus the query engine collects all the nodes that are descendant of the query node using the child link of the TMS node. These collected nodes are all the answers for the query and only those that, are labeled IN, are the valid answers that will be reported by the query engine. If the query is not tagged as previously proven, then a query is evaluated using PROLOG inference engine, and at the same time, JTMS network for the query is installed to cache the proof structure.

JLOG's Query Engine Utilizes Support of Normal Tabulation

As seen in the above description, when a query is evaluated for the first time, JLOG calls the PROLOG inference engine to prove the query. If the PROLOG inference engine attached to JLOG supports normal tabulation, then JLOG utilizes this feature to get benefits of the normal tabulation as described is Section 3.5.2. When JLOG gets the results of the query from the PROLOG inference engine, it requests the engine to abolish all tabled answers to avoid any conflicts. If the PROLOG inference engine, attached to JLOG, does not support tabulation, then the advantages of normal tabulation cannot be utilized.

3.6 How to Handle Negation in JLOG

The objective of this section is to show how negation is handled in JLOG. JLOG uses the support of traditional Selected Literal Definite Clause with Negation-by-Failure (SLDNF-derivation) used by PROLOG to handle the negation operator. The Negation-by-Failure can be formalized as follows:

$\neg Goal :- Goal, !, fail.$

$\neg Goal.$

When a negative literal $\neg$ is selected for evaluation, the query engine suspends proving the original query in order to prove G . If the system fails in proving G , then $\neg G$ is assumed true(Negation-by-Failure). Otherwise, $\neg G$ is assumed false and the original query resumes its work. Generally, JLOG treats any PROLOG program that contains negation(Stratified/Non-Stratified) in similar fashion to the definite programs. However there are some minor differences that are reported below:

1. When JLOG installs a justification for a proof branch, the antecedents of the justifications are categorized into two sets, those that should be IN (true) and those that should be OUT(false) in order to handle negation.

2. The rules soundness and completeness are revered:

   (a) When data is asserted to the database, JLOG should ensure the soundness of the proof structure.

   (b) When data is retracted from the database, JLOG should check the completeness of the saved proof structure.

```
:-table p/1,r/1,q/1,s/1,w/1    as incremental
p(X):-q(X),not(r(X)). %R1
r(X):-w(X),not(s(X)). %R2
q(a).q(b).q(c).s(a).s(c).w(a).w(b). %F1..F7
```

Figure 3.24: A Stratified PROLOG Program.

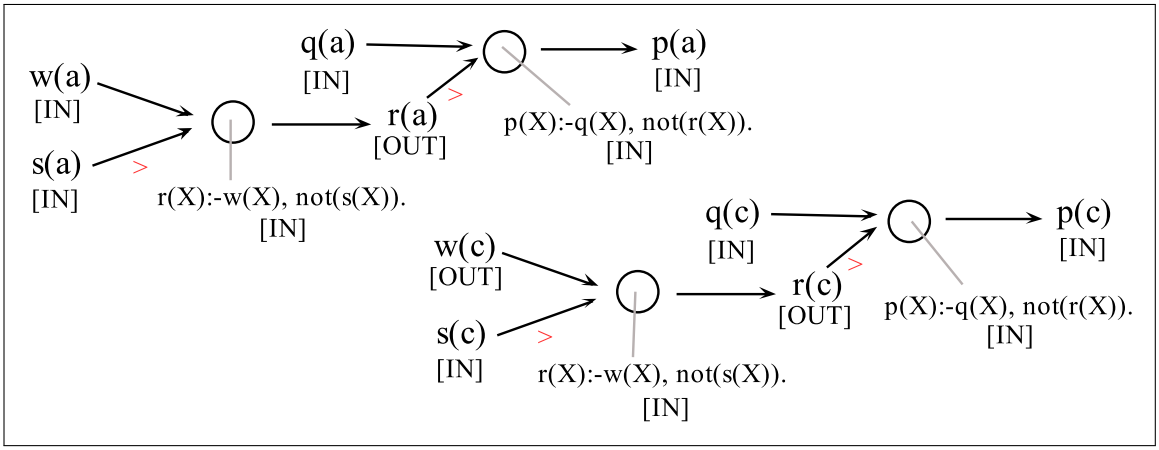

Figure 3.25: JTMS network installed by JLOG after proving the query $?-p(X)$ for the first time with respect to the PROLOG program of Figure 3.24.

3.6.1 Stratified versus Non-Stratified Negation

A PROLOG program uses stratified negation whenever there is no recursion through negation whereas a program uses non-stratified negation whenever there is a recursion through the negation or the negation is implicit. In JLOG, The behavior of the program depends totally on the PROLOG inference engine, attached to JLOG, and whether the engine supports normal tabulation or not. Basically, if the negated program terminates in the PROLOG inference engine independently then it will also terminate in JLOG. The following two examples of PROLOG programs illustrates how does JLOG handles negation.

An example of Stratified Program

Figure 3.24 shows an example of a stratified PROLOG program. Consider the evaluation of the query $?-p(X)$ for the first time with respect to the PROLOG program of Figure 3.24. The program in the Figure 3.24 contains two rules with negated functors. Figure 3.25 shows the JTMS network installed by JLOG after proving the query $?-p(X)$ for the first

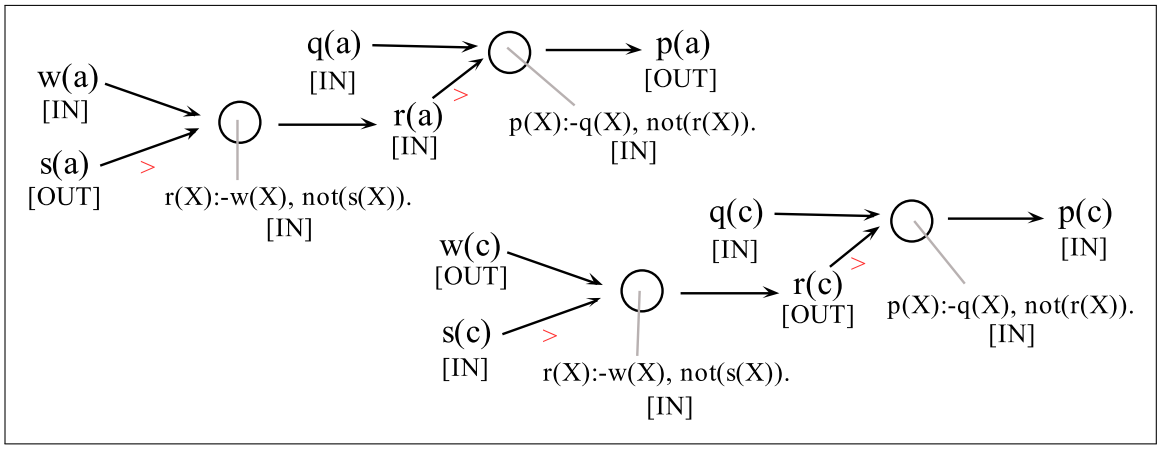

Figure 3.26: JTMS network of Figure 3.25 after retracting the fact $s(a)$.

time. A justification is installed for each complete branch of the SLD-tree. The difference in this case comparing to definite programs is that the antecedent list is categorized into two groups. The first group is the list of antecedents that should be IN(true) to support the consequence of the justification. The other group is the list of antecedents that should be OUT(false) to support the consequence of the justification.

Once the JTMS network is cached when the query is proved for the first time, JLOG's TMS component and the database monitor maintains the completeness and soundness of the proof structure. Let's take into account couple of changes to the database of facts/rules in Figure 3.24 that affects the completeness and soundness of the cashed proof structure of Figure 3.25:

1. $retract(s(a))$.

   This case is related to maintaining the soundness of the cashed proof structure of Figure 3.25. Figure 3.26 shows the JTMS network of Figure 3.25 after retracting the fact $s(a)$. The label of the TMS node $s(a)$ is changed from IN to OUT. Then the TMS component of JLOG propagates the effect of this change throughout the JTMS network; The TMS node $s(a)$ participates as antecedent in the out list of one of the justifications. The label of the consequence $r(a)$ of this justification is shifted from OUT to IN. This case is different from the definite programs where usually retraction of any antecedent list data of a particular justification results is retraction

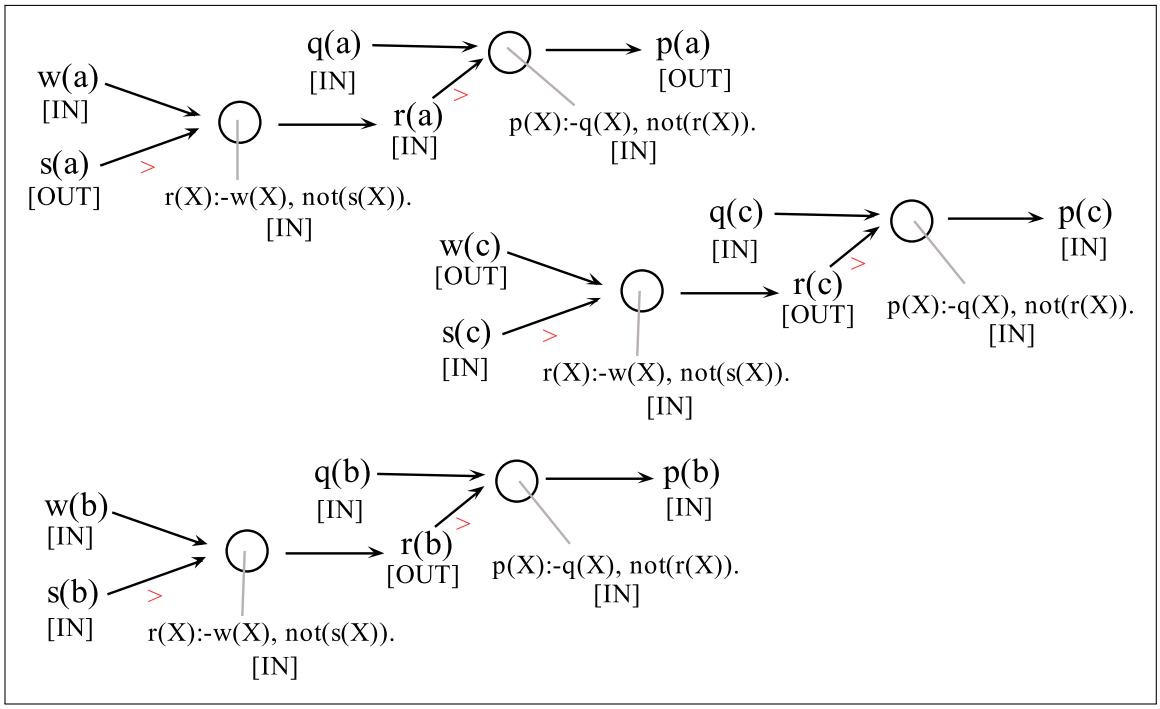

Figure 3.27: JTMS network of Figure 3.26 after asserting the fact $s(b)$.

of the consequence of that justification. Continuing the scenario, the TMS node $r(a)$ participates as antecedent in the out list of one of the justifications. Since the label of the node $r(a)$ is changed from OUT to IN, the label of the consequence $q(a)$ of this justification is changed from IN to OUT

2. $assert(s(b))$.

The node $s(b)$ was not presented when the query $?-q(a)$ was proved for the first time which yields that this assert statement is related to completeness of the cashed proof structure of Figure 3.26. The database monitor keeps track of the general forms of facts and rules that might add some new justifications to the JTMS network of perviously proven queries. Figure 3.27 shows the JTMS network of Figure 3.26 after asserting the fact $s(b)$. The general TMS node $s(X)$ of the newly asserted fact $s(b)$ is appearing at the right hand side of the rule:

$$r(X) : -w(X), not(s(X)).$$

the database monitor resumes the execution of the query $?-q(X)$ to accumulate

```
:-table shave/2 as incremental
shaves(barbar,P) :- person(P), not(shaves(P,P)). %R1
person(taher). person(maher). person(barbar).%F1..F3
```

Figure 3.28: A Non-Stratified PROLOG Program.

its proof structure based on the new fact $s(b)$. This results in installing a new justification in the JTMS network for the query $?-r(X)$. The consequence of the new justification in the new TMS node $r(b)$. The database monitor propagates the effect of new data though out the whole JTMS network. Since The general TMS node $r(X)$ is appearing at the right hand side of the rule:

$p(X) :- q(X), not(r(X))$.

the database monitor resumes also the execution of the query $?-p(X)$ to update its proof structure. A new justification is added to the JTMS network of the query $?-p(X)$.

An example of Non-Stratified Program

The PROLOG program of Figure 3.28 shows an example of a non-stratified negation. The rule $R1$ in the program states that the barber shaves everyone who does not shave himself. The question is who shaves the barber? Consider the evaluation of the query $?-shaves(barbar,P)$ for the first time with respect to the PROLOG program of Figure 3.28. In normal PROLOG implementations, this query engine returns first two answers $P = taher$ and $P = maher$ for the the query $?-shaves(barbar,P)$ and, then the query engine suffers from infinite loop because of the subquery $?-shaves(barbar,barbar)$. In JLOG the query $?-shaves(barbar,P)$ is not going to terminate and return any answer if the PROLOG engine attached to JLOG does not support tabulation.

In stratified programs, facts are either true or false, while in non-stratified programs facts may also be undefined. True answers in the well-founded semantics are those that have a tabled derivation while False answers are those for which all possible derivations

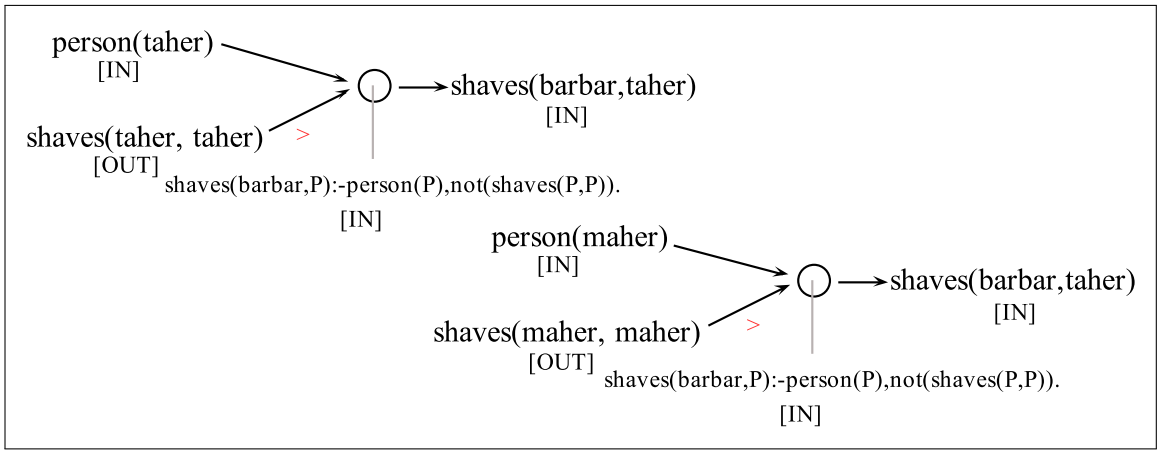

Figure 3.29: JTMS network installed by JLOG after proving the query $?-shaves(barbar,Y)$ for the first time with respect to the PROLOG program of Figure 3.28.

fail. The derivations of False answers fails either finitely as the case in PROLOG or by failing in positive loops. Using the normal tabulation, the query $?-shaves(barbar,barbar)$ returns False by failing in positive loop. If the PROLOG engine attached to JLOG does support tabulation, then the query $?-shaves(barbar,P)$ terminates and returns all answers($P=taher$ and $P=maher$) for the above query. Figure 3.29 shows the JTMS network installed by JLOG after proving the query $?-shaves(barbar,Y)$ for the first time with respect to the PROLOG program of Figure 3.28.

3.7 The relation between PROLOG inference engine and JLOG

As seen in the previous sections, the main four components of JLOG are the PROLOG inference engine (SWI-Prolog, XSB orYAP-Prolog), the TMS component, the database monitor, and the query engine. The first three components interact through JLOG's query engine to provide the full functionality of the system. When a query a proved for the first time, JLOG requests the PROLOG inference engine to prove the query and return its proof structure. The cashed proof structure is stored by JLOG in a way that allows the system to keep it complete and sound when the database of facts/rules related to the query is

changed.

The main objective of JLOG is to provide **Tabled Prolog System** that can work under **Non-Monotonic Logic**. One important point to be highlighted here is that the behavior of any query in JLOG depends on prolog engine attached to JLOG. If the query terminates by the engine, it also terminates in JLOG. The performance and behavior of any query in PROLOG is better when JLOG is attached to a PROLOG inference engine that supports normal tabulation.

Chapter 4

JLOG Implementation

In Chapter 3 we presented the detailed design of PROLOG. In this chapter we will focus on the on the system implementation. The current version of JLOG is implemented using the JAVA [Gosling et al., 2005] Programming language. The system implementation must take into consideration the following performance factors:

1. Minimizing the overhead of caching the query proof structure.

2. Minimizing the time and space needed to maintain soundness and completeness of the cached proof structure.

Another factor that we have to take into consideration before starting the system implementation is the portability of integrating JLOG with more than one PROLOG inference engine.

4.1 JLOG Implementation Approach

Tabling is an implementation technique where answers for subcomputations are stored and then reused when a repeated computation appears. There are two approaches to integrate tabling support into existing PROLOG systems:

1. Modify and extend the low-level engine

   This is the common approach used by most of systems to support tabled evaluation. This approach modifies and extends the low-level engine [Sagonas et al., 1993a]. The advantage of this approach is the run-time efficiency, however, the drawback is that it is not efficiently portable [Wielemaker & Costa, 2010] to other PROLOG systems because the engine level modifications are slightly more complex and time consuming.

2. Apply source level transformations

   The second approach to incorporate tabled evaluation into existing PROLOG systems is to apply the source level transformations to a tabled program, and then use external tabling primitives to provide direct control over the search strategy. This idea was first explored by Fan and Dietrich [Fan & Dietrich, 1992] and later used by Rocha, Silva and Lopes [Rocha et al., 2007a] to implement tabled PROLOG systems. The main advantage of this approach is the portability to apply the approach to different PROLOG systems. The drawback of course is the efficiency, since the implementation is not at a low level.

JLOG implementation is based on applying the source level transformations to a tabled program. This will allow to incorporate the idea of incremental tabulation into different PROLOG systems using the same general framework. Figure 4.1 gives an overview of JLOG and its components, meanwhile, Figure 4.2 depicts the system class diagram which gives a general idea of how elements of JLOG interact together to provide the overall functionality of the required system.

In order to integrate JLOG with PROLOG inference engine we are using INTERPROLOG. INTERPROLOG [Calejo, 2004] is an PROLOG-JAVA application programming interface that supports multiple PROLOG systems through the same API. The current version (2.1.2a at the time of writing this thesis) of INTERPROLOG supports three Prolog implementations (XSB [Swift & Warren, 2012], SWI-PROLOG [Wielemaker et al., 2012] and

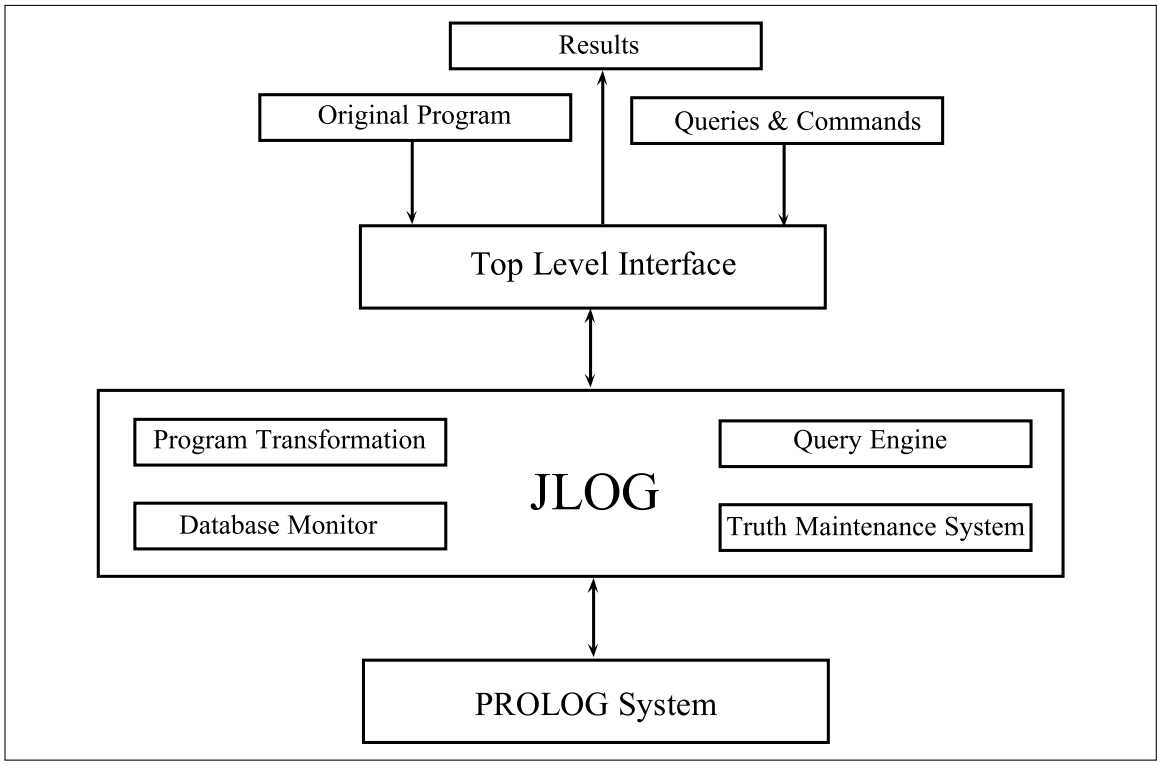

Figure 4.1: An overview of JLOG

YAP-PROLOG [Costa et al., 2012]) on different platforms (Windows, Linux and Mac OS X). It promotes coarse-grained integration between logic and object-oriented layers, by providing the ability to bidirectionally map any class data structure to a PROLOG term. This basic idea of INTERPROLOG is allowing PROLOG to call any JAVA method, and for JAVA to invoke PROLOG goals. This is achieved by using a communication layer to pass object/term data among both processes. INTERPROLOG is used in the implementation of JLOG to provide an interface from JLOG to all PROLOG inference engines supported by INTERPROLOG.

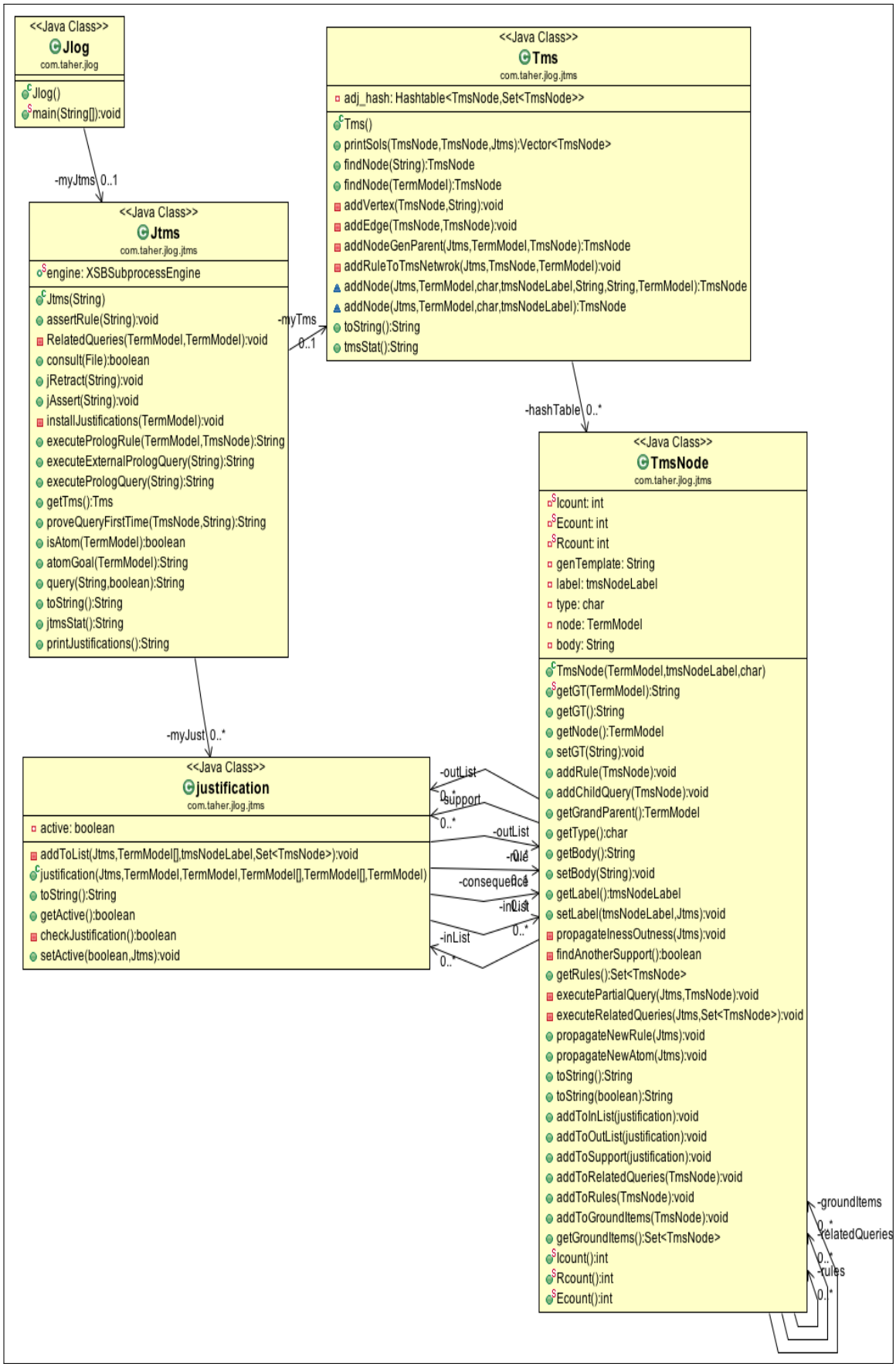

Figure 4.2: JLOG class diagram

Algorithm 4.1 Program Transformation Algorithm

Input: R, original Program rules list
Output: R_t, transferred rules List

```
assign each rule in R a unique id
for each rule Ri in R
        h = head of Ri, b = body of  Ri;
        ht= h with extra unique var argument Jc;
        bt = b+",";
        bt = bt + "append(["+id(Ri)+"],[";
        bt = bt + "["+all non-negated terms in b+"],";
        bt = bt + "["+all negated terms in b+"],["+h+"]";
        bt = bt + "],"+ Jc + "  )";
        Rti = ht + ":-" + bt+".";
```

4.2 Program Transformation

As described in Chapter 2, PROLOG rules are written in a standard form known as Horn clauses. A Horn clause statement has the form:

$$H : -B_1, B_2, ..., B_n.$$

where H is a first-order atom and each B_i is a first-order literal. H is called the head of the clause and $B_1, B_2, ..., B_n$ is the body of the clause. The semantic of this statement is that when B_i all are true, we can deduce that H is true as well. The above clause can be read "$H, if B_1, B_2, \ldots, B_n$". This basic concept of logic programming is used by JLOG to do the program transformation. When a PROLOG programmer executes the command $consult/1$ to read a PROLOG source file through the JLOG's top level interface, JLOG executes the program transformation method to convert the PROLOG predicates (rules) into a format that allows the system later to table the query answers as a JTMS network according to the mechanisms described in the previous chapter. Recall, from Chapter 3, that JLOG translates every complete branch of the SLD-tree into a TMS network that links the facts used in the branch to the answer generated by that branch.

JLOG applies program transformation only on clauses that are selected by the PROLOG programmer to be as incremental tabled predicates. Algorithm 4.1 illustrates the program transformation process. The process starts with assigning each rule in the original program a unique ID. For example, The first rule in the translative closure program of

Figure 4.3 is uniquely identified as $r1$, while the other rule is defined as $r2$. The next step in the algorithm is to convert each rule in the original program into a format that allows the query engine later to link the facts which are participated as antecedents to produce the consequence (answer) of the query. This is achieved by applying two major changes in each program rule:

1. Add an extra unique var argument to the rule head, this var points to a list which contains all the necessary information needed by JLOG to build the JTMS network of any query answer generated from this rule. The list is to be composed of the following items:

   (a) The rule *ID*.

   (b) List of facts (antecedents), coming from the rule body, that must be true (IN) to support the rule consequence.

   (c) List of facts (antecedents), that must be false (OUT) to support the rule consequence. This case is used when the body contains negated terms.

   (d) The consequence of the rule, which is basically the head of the rule.

2. Add an extra term to the rule body that generates the above list and stores it in the extra var added to rule head.

Once the rule is in its final format using the above conversion method, a TMs node is created for the rule and it is linked to the TMS node of the original rule head. This will help the query engine later to identify all rules related to a certain query entered by the user.

Figure 4.3 shows how a $connected/2$ tabled predicate that defines the translative closure program of the directed edge relationship is transformed to support JLOG's tabulation strategy. Each rule head, in Figure 4.3, is converted from $connected(X,Y)$ to $connected(X,Y,J0)$ by adding an extra var argument $J0$. In the same manner each rule body is changed by appending the required code that originates the list in the var $J0$. For example the second rule body is changed from:

%Original Tabled Predicates
$connected(X,Y) :- edge(X,Y).$ %r1
$connected(X,Y) :- edge(X,Z), connected(Z,Y).$ %r2

%Transformed Predicates by JLOG
$connected(X,Y,J0) :- edge(X,Y), append([r1],[[edge(X,Y)],[],connected(X,Y)],J0).$
$connected(X,Y,J0) :$
$-edge(X,Z), connected(Z,Y,J1), append([r2],[[edge(X,Z),connected(Z,Y)],[],connected(X,Y)],J0).$

Figure 4.3: Original tabled and transformed Predicates by JLOG for the translative closure program of the directed edge relationship.

Algorithm 4.2 Query Engine

Input: Q, a PROLOG query.
Output: List of query answers.

```
queryTmsNode = tmsNode assocaited with Q
if ( queryTmsNode != full ){
        tn = getGrandParent(Query);
        gt = getRules(tn);
        while (gt != null ){
                justifications=executePrologQuery(gt.element.rhs);
                installJustifications(justifications);
        }
        justifications=executePrologQuery(Query);
        installJustifications(justifications);
    }
}
showQuerySolutions(queryTmsNode);
```

$edge(X,Z), connected(Z,Y)$

to:

$edge(X,Z), connected(Z,Y,J1), append([r2],[[edge(X,Z),connected(Z,Y)],[],connected(X,Y)],J0).$

Both of these two rules are linked to the TMs node of the original rules head, i.e. $connected(X,Y)$.

4.3 Query Engine

Query engine of JLOG responds to end user queries through the system top interface. The necessary methods needed to implement the logic of the query engine are part of the JTMS class. There are two scenarios for the query execution module that are described in Algorithm 4.2. The first scenario checks if the TMS node associated with the user query already exists in the JTMS network and is marked as fully proved. When this condition is

true the query engine simply shows all the valid(IN) answers of the query from the JTMS network linked to the query's TMS node. The second scenario deals with the case when the query is executed for the first time. JLOG deals with this case as a three step process:

1. The system locates the general TMS node associated with the query. The desired TMS node is used to locate the set of PROLOG rules associated with the query. If the set is not empty, the query engine requests the PROLOG abstract engine attached to it to execute the right hand side of each rule. The answers (justifications) coming from the PROLOG abstract engine execution are installed as justifications in the JTMS network.

2. The query engine tries to find more solutions for the query that might come from the database of facts which were not discovered in the first step. If such answers exist, then justifications related to these answers are also installed in the JTMS network.

3. After the previous two steps, the query answers are ready. The query engine simply shows all the answers of the query from the JTMS network linked to the query's TMS node.

4.3.1 Installing the Justifications

As seen in Section 4.3, the PROLOG abstract engine, attached to JLOG, executes PROLOG queries requested by the query engine. The results of these queries must be installed as justifications in the JTMS network associated with the query. Figure 4.4 represents the list of answers (b) returned by the PROLOG abstract engine for the query $connected(X,Y)$ with respect to the translative closure PROLOG program (a). The list of results are in a format that allows the system to convert them easily into justifications. The list item contains the rule participated as antecedent in generating the deduction. The set of facts participated as antecedents in generating the deduction.

The answer itself which represents the consequence of the deduction. Below are the details of each list item:

1. The rule *ID* that is behind the generation of this result.

2. Set of PROLOG items that must be IN to support the consequence of the result. This case is related to non-negated subgoals in the right hand side of the rule.

3. Set of PROLOG items that must be OUT to support the consequence of the result. This case is related to negated subgoals in the right hand side of the rule.

4. The consequence of this result.

These four items are one to one mappings to the data attributes for the Justification class. The class contains five attributes. The first attribute is used as a flag to indicate whether the justification is active or not. The rest of the attributes are used to store the above information for the justification.

Going back to the example of Figure 4.4(b), the first answer:

$[r2, [edge(a,b), connected(b,d)], [], connected(a,d)]$

for the query $connected(X,Y)$ states the following:

1. The rule $r2$ is in the antecedent list of the answer $connected(a,d)$.

2. The atoms $edge(a,b)$ and $connected(b,d)$ must be IN to support the consequence of the the answer $connected(a,d)$.

3. None of the atoms must be OUT to support the consequence of the answer since there are no negated subgoals in the program of Figure 4.4(a).

4. The consequence of this answer is the PROLOG atom $connected(a,d)$.

For each of the answers of Figure 4.4(b), JLOG installs a justification for it and makes the justification active. When the data related to any justification is changed, the system maintains the consistency of the justification according to the changes in the set of rules/-facts related to the justification. An important point that must be highlighted is that each

```
connected(X,Y) :- edge(X,Y). %r1
connected(X,Y) :- edge(X,Z), connected(Z,Y). %r2
edge(a,b). edge(a,c). edge(b,d). edge(c,d). edge(d,e). %f1..f5
```

(a)

```
[r2, [edge(a,b), connected(b,d)], [], connected(a,d)]
[r2, [edge(a,c), connected(c,d)], [], connected(a,d)]
[r2, [edge(a,b), connected(b,e)], [], connected(a,e)]
[r2, [edge(a,c), connected(c,e)], [], connected(a,e)]
[r1, [edge(a,b)], [], connected(a,b)]
[r1, [edge(a,c)], [], connected(a,c)]
[r2, [edge(b,d), connected(d,e)], [], connected(b,e)]
[r1, [edge(b,d)], [], connected(b,d)]
[r2, [edge(c,d), connected(d,e)], [], connected(c,e)]
[r1, [edge(c,d)], [], connected(c,d)], [r1, [edge(d,e)], [], connected(d,e)]
```

(b)

Figure 4.4: List of answers(b) returned by the PROLOG abstract engine for the query $connected(X,Y)$ with respect to the translative closure PROLOG program(a).

justification element is stored as a TMS node. The details of the TMS node data structure is given in next section.

4.4 TMS Nodes

A TMS node is the basic data structure used in JLOG to cache the proof structure of a PROLOG query. The set of TMS nodes linked together through justifications represents the JTMS network for the query. The TMS node is a complicated data structure, it must store all the information that allows the system to maintain the consistency and completeness of the cached proof structure for a perviously proven query. The following subsections describe the main most important attributes and operations for this crucial data structure.

4.4.1 Attributes

Label

The label attribute stores the current status of the node. At any time, the TMS node is labeled one of the following:

1. *IN* or *OUT*

   One of these two labels are used to represent the status of the TMS node when the TMS node that is pointing to a PROLOG predicate, i.e. a fact or a rule. When the system believes that the PROLOG predicate is true or active, then the label of the TMS node is *IN*. The TMS node label is *OUT* when the predicate is false or inactive.

2. *Full*, *Partial* or *New*

   One of these three labels are used when the TMS node is pointing to a PROLOG query. The label *Full* indicates that the query associated with the TMS node is fully proved and when the user re-evaluates the query no additional work would be required from the PROLOG inference engine attached to JLOG in generating the answers of the query. The label *Partial* means that the query attached to the TMS node is partially proved. If a partially proven query is going to be evaluated then the system must complete the query evaluation before returning its answers. The *New* label is used when the query is proved for the first time and a new TMS node is created for the query.

Type

This attribute stores the type of the TMS node. JLOG uses three types of TMS nodes. The type of the TMS node is either a PROLOG fact, rule or a query.

Node

The Node attribute points to the actual PROLOG term.

Support

This object represents the set of justifications that support the PROLOG fact attached to this TMS node. Recall from chapter 3 that a TMS node may have more than one justification

supporting it.

In-List

The set of justifications where the PROLOG predicate, associated with this TMS node, is participating as an $IN(true)$ antecedent in the justification.

Out-List

The set of justifications where the PROLOG atom, associated with the this TMS node, is participating as an $OUT(false)$ antecedent in the justification.

Related Queries

This set points to the list of other cached queries, in the system, which are effected whenever there is a change in the proof structure of the PROLOG query attached to this TMS node.

Rules

If the TMS node is associated with a PROLOG query where the query term matches the head of certain rules in the PROLOG program, then this attribute points to the query related rules from the program. Note that some of the above attributes might have a *Null* value depending on the type of the TMS node. For example, if the TMS node is associated with a PROLOG query, then the values of support, in-list and out-list is *Null* since they are used with TMS nodes that point to a PROLOG fact.

4.4.2 Operations

Set Label

The "Set Label" is the most critical operation of JLOG. The operation is maintaining the consistency and the completeness for the queries cached proof structure. Algorithm 4.3

Algorithm 4.3 Set label operation's algorithm for the TMS node

Input: *newLabel*, the new label
Output: *void*

```
if (this.label != newLabel) {
        boolean active = false;
        if ( this.type == Fact && newLabel == OUT ){
                active = this.findAnotherSupport();
                if (!active)     {
                        this.label = x;
                        if (this.type == 'Fact' || this.type == 'Rule')
                                this.propagateInnessOutness(myJtms);
                }
        }
}
```

describes the set label process for a TMS node. The algorithm starts with checking if the new label is different from the current label of the TMS node. This check is required to avoid any redundant computations when there is no change in the label. The next step in the algorithm is to check if the label of the TMS node is being changed from *IN* to *OUT* and if that particular node is pointing to a PROLOG fact. If this is true, then the system tries to find an active justification that can support this fact, and hence, avoid changing its label from *IN* to *OUT*. If no such active justification is found, or the TMS node type is not a fact, or the new label is being changed from *OUT* to *IN*, then the system changes the label of the TMS node to its new value. Once the label value is changed, JLOG propagates the effect of this change throughout the JTMS network by calling the *propagateInnessOutness* method.

Propagate Iness/Outness of Existing Facts/Rules

The objective of this operation is to maintain the consistency of the JTMS network whenever there is a change in the label of a TMS node. The algorithm for propagating the change of a TMS node label throughout the JTMS network is explicated in Algorithm 4.4. The logic of the algorithm is straightforward. For each justification in the in-list of the current TMS node, the system tries to set the justification active when the label of TMS node is *IN*; otherwise the justification is set as inactive. Also, for each justification in the

Algorithm 4.4 Propagating the change of a TMS node label throughout the JTMS network.

Input: *this*, the node with changed label.
Output: *void*

```
for each justification in this.inList {
        if (this.label == IN)
                try to make the justification active;
        else
                make the justfication inactive;
}
for each justification in this.outList {
        if (this.label == OUT)
                try to make the justification active;
        else
                make the justfication inactive;
}
```

out-list of the current TMS node, the system tries to set the justification active when the label of the TMS node is OUT, otherwise the justification is set as inactive.

- Activating a justification

  To activate a justification, the system must ensure that all antecedents of the In-List are labeled as IN, and all antecedents of the Out-List are labeled as OUT. If these two conditions are satisfied, then the justification's consequent TMS node label is changed to IN, and the justification is marked as active.

- Inactivating a justification

  To mark a justification as inactive, the system changes the justification's consequent TMS node label to OUT. Then the the justification is marked as inactive.

Propagate the Asserting of a New Fact

This method is called when a new PROLOG fact is asserted through the JLOG interface. Figure 4.5 outlines the algorithm for propagating the effect of inserting a new PROLOG fact. The system locates the TMS query nodes that might get affected from the insertion of the new fact. For each such query, if the query has been fully proved previously, the algorithm locates the set of rules attached to the query's TMS node. Every rule is unified

Algorithm 4.5 Propagating the effect of asserting a new PROLOG fact.

Input: *this*, the new PROLOG fact's TMS node.
Output: *void*

```
RQ= this.relatedQueries;
for (each Query in RQ) {
        if (Query.label == Full) {
                Rules= Query.getRules();
                for (each Rule in Rules) {
                        Query.executePartialQuery(unify(Rule, this));

                }
        }
}
```

Algorithm 4.6 Propagating the effect of asserting a new PROLOG rule.

Input: *this*, the new PROLOG rule's TMS node.
Output: *void*

```
ruleQueryNode=tms.find(this.head);
if (ruleQuery != null) {
        if (ruleQuery.getLabel() == Full) {
                Jtms.executePrologRule(head, this);
        }
}
```

with the new fact, then the system executes the partial PROLOG query (which is coming from the right hand side of the rule) to update the cached proof structure.

Propagate the Asserting of a New Rule

This method is called when a new PROLOG rule is asserted through the JLOG interface. Figure 4.6 outlines the algorithm for propagating the effect of inserting a new PROLOG rule. The process starts with locating the TMS query node that matches the new rule head. If such query node exists and it is marked as fully proved, then the system requests the PROLOG inference engine attached to JLOG to execute the right hand side of the new rule to update the cached proof structure. This is achieved by invoking the JTMS method *executePrologRule*. Recall from Chapter 3 that any new answers for the cached proof structure will come only from the right hand side of the newly asserted rule.

4.5 The JTMS Class

The JTMS class is the main component of JLOG. It interacts with the system's top interface (JLOG class) to provide the desired functionality (see Figure 4.3). The main data component and operations of this class are described briefly below.

4.5.1 Attributes

PROLOG Engine

The PROLOG engine points to the PROLOG inference engine. The integration between the PROLOG inference engine and JLOG is done by using INTERPROLOG package. See Section 4.1 for more details about INTERPROLOG.

TMS Object

The object points to an instance of the TMS class. See Section 4.6 for more details about this component.

Justifications Table

The justification object of JTMS class points to the list of all current justifications in the system. Justifications are maintained as a hash table to allow fast retrieval of information related to them. See Section 4.3.1 for more details about how the justifications are installed and maintained by JLOG.

4.5.2 Operations

Executing PROLOG Queries

This method allows the JTMS class to execute the PROLOG queries with the help of the PROLOG inference engine object. Usually this method is called by the query engine when JLOG's end user requests to execute the query for the first time. Later, the method is

Algorithm 4.7 Executing a PROLOG Query

Input: *query*, the PROLOG query(goal).
Output: *void*,

```
solutionVars=engine.deterministicGoal(query);
for (each solution in solutionVars ){
        this.installJustifications(solution);
}
```

Algorithm 4.8 Handling the the assertion of a fact/rule to the PROLOG program.

Input: term, the asserted PROLOG fact or rule.
Output: *void*,

```
engine.deterministicGoal(assert(term));
TmsNode t1 = myTms.findNode(term);
if (t != Null ) {
        t1.setLabel(IN)
} else {
        if (term == Fact ) {
                Tms newNode = myTms.addNode(term);
                newNode.propagateNewAtom();
        } else {
                Term newRule = convert(term);
                TmsNode newNode = myTms.addNode(newRule);
                newNode.propagateNewRule();
        }
}
```

invoked by the TMS node objects to maintain the correctness and completeness of the cached proof structure for the same query. Algorithm 4.7 outlines the execution of a PROLOG query. This is a two step process. First the incoming query is executed by invoking the method *deterministicGoal* which returns the set of solutions for this particular query. In the next step, the algorithm takes each solution returned by the PROLOG inference engine and calls the method *installJustifications* to install the justification in the JTMS network of JLOG.

jAssert

The method *jAssert* is invoked when the end user initiates the PROLOG command *assert*/1, through JLOG's user interface to insert a fact/rule into a PROLOG program. Algorithm 4.8 describes the steps to handle the the assertion of a fact/rule to the PROLOG program. The first step in the algorithm is to inform the PROLOG engine about this change in the set

of facts/rules related to the associated program. This is achieved by executing the assert command by the PROLOG engine. In the next step, the algorithm tries to locate the TMS node associated with the asserted fact/rule. If the TMS node already exists in the database of TMS nodes then the method *setLabel* (See Section 4.4.2) is invoked to propagate the effect of this assertion through out the JTMS network.

If the TMS node, associated with the asserted fact/rule, does not exist in the database of TMS nodes, then this case is related to the insertion of a new PROLOG fact/rule to the PROLOG program which was not presented at the program consult time. The system handles the insertion of new PROLOG predicates according to the following scenarios:

1. Asserting a new fact

   A new TMS node is created for the new fact. The algorithm 4.8 then invokes the method *propagateNewAtom* (See Section 4.4.2) to update the JTMS network in response to this change in data.

2. Asserting a new rule

   A new TMS node is created for the new rule. The rule is converted according to the format described in Section 4.2. Then the algorithm invokes the method *propagateNewRule* (See Section 4.4.2) to update the JTMS network in response to this change in the set of rules.

Algorithm 4.9 Handling the the retraction of a fact/rule from the PROLOG program.

Input: term, the retracted PROLOG fact or rule.
Output: *void*,

```
engine.deterministicGoal(retract(term));
if (t != Null ) {
        TmsNode t1 = myTms.findNode(term);
        t1.setLabel(OUT)
}
```

jRetract

The inverse logic of jAssert. This method is invoked when the end-user initiates the PROLOG command *retract*/1, through JLOG's user interface, to remove a fact/rule from the PROLOG program. Algorithm 4.9 shows the three required steps to handle the retraction of a fact/rule from the PROLOG program. In the first step, the system informs the PROLOG engine about this change in the database of facts/rules by executing the retract command on the PROLOG engine. Then the algorithm tries to locate the TMS node associated with the asserted fact/rule. If the TMS node already exist in the database of TMS nodes, then the algorithm invokes the *setLabel* method to propagate the effect of this retraction throughout the JTMS network. If the TMS node associated with the retracted fact/rule does not exist in the database of TMS nodes, then no action is required because the retracted fact/rule is not part of the current JTMS network.

4.6 TMS Class

The TMS class is used in JLOG to link all TMS nodes together in a parent/child relationship. Basically, the TMS class stores and links the TMS Nodes using the Graph data structure. In the rest of this section, a brief description is given about the main components of the TMS class.

Algorithm 4.10 Adding a PROLOG term to the TMS Network.

Input: term, the PROLOG term; type, the type of the PROLOG term; label, the label of the PROLOG term.
Output: t, the TMS node associated with the PROLOG term.

```
t1 = new TmsNode(node,label, type);
this.addVertex(t1, null);
TmsNode t2=this.addNode( t1.getGrandParent(),'I',New);
this.addEdge(t2, t1);
switch (nodeType) {
        case Fact:
                t1.propagateNewAtom();
        break;
        case Rule:
                t1.propagateNewRule();
        break;
}
return t1;
```

Attributes

There are two main attributes in this class. The first attribute is the *hashTable* that keeps the track of Prolog terms (facts, rules, and queries) with their associated TMS nodes in the HashTable data structure to allow fast retrieval of the terms whenever required by other system components. The other attribute of the TMS class is the *ad jhash* which represents the adjacency list of each TMS node. The adjacency list of each TMS node links it to other TMS nodes in the JTMS network of JLOG.

Operations

The operations of the TMS node are the typical ones of any Graph or HashTable abstract data type. Mainly the most important operations are:

Add a TMS node

The algorithm for adding a PROLOG term to the TMS Network is presented in 4.10. The algorithm starts with creating a new TMS node for the PROLOG term. The newly created TMS node is added to the Graph as a vertex. Then the algorithm links the TMS node to its most general term. See Section 3.5.1 for more details about TMS nodes hierarchy. Once this setup is done, the algorithm propagates the effect of this new PROLOG term

throughout the JTMS network.

Find a TMS Node

Given a PROLOG term (fact, rule, or query), the find operation returns a pointer to the TMS node object associated with the PROLOG Term. If such TMS node does not exist in the HashTable, then the method returns *null*.

Chapter 5

JLOG Perfromance

The main objective of JLOG is to provide a PROLOG system which supports incremental tabulation. This is what JLOG achieves. JLOG evaluates the query only once, maintaining enough information to ensure both consistency and completeness of the collected solution as the dynamic state changes. When the query is re-evaluated, JLOG returns cashed answers which are always up to date. This chapter discusses the performance of JLOG. JLOG is compared with the regular, traditional, and incremental tabled PROLOG systems. Our benchmark is the XSB [Swift & Warren, 2012] system since it is the only PROLOG implementation so far that supports incremental tabulation. XSB supports all flavors of the query executions that we would like to compare with JLOG. The XSB system supports the query execution with no tabulation, as any normal PROLOG system. It also supports the traditional tabled queries that assumes monotonic logic. Furthermore, the XSB is the first system that allows tabling of queries incrementally in order to support the non-monotonic logic. The main intension for assessing the performance of JLOG is to compare it with:

1. Normal PROLOG implementations [Wielemaker et al., 2012] that do not support tabulation.

2. Tabled PROLOG implementations [Costa et al., 2012, Swift & Warren, 2012] that support monotonic logic systems. Note that JLOG is not designed to be an alternative for tabled PROLOG implementations. Actually JLOG utilizes the normal tab-

ulation evaluation, through the inference engine attached to it, in order to provide support for incremental tabulation. See section 3.5.2

3. Incremental tabled PROLOG implementations [Swift & Warren, 2012] that supports monotonic logic systems. This is considered to be the main assessment factor since the main objective of this research is to support incremental tabulation.

The current version of JLOG can use three PROLOG implementations (XSB , SWI-PROLOG and YAP-PROLOG) as a PROLOG inference engine attached to it. See Section 4.1 for more information about JLOG implementation approach. We can use any of the three inference engines to assess the above goals. The experiments presented in this chapter are based on the fact that the inference attached to JLOG is the XSB with the support of normal tabulation only. The reason behind this selection is that when JLOG evaluates the query for the first time to cache the proof structure, it will evaluate it under same environment when the query is evaluated for the first time in XSB. The incremental tabulation strategy used in XSB utilizes the normal tabled support when the query is proven for the first time [Saha, 2006]. The same strategy is used by JLOG; when the query is proven for the first time, JLOG requests the inference engine to execute the query with normal tabled support (See Section 3.5.2).

We tested the performance of JLOG by implementing a small business intelligence [Grabova et al., 2010], or a reporting tool for a mid-size University (Gulf University for Science and Technology, Student Information System). We have chosen this approach rather than the standard benchmark dataset to test the system on real data. The objective is to observe if the system is able to work under real applications. The time shown for the experiments in the subsequent sections is presented in milliseconds, all tests are done on a MacBook Pro machine(Processor 2.4 GHz Intel Core i7, Memory 6 GB 1333 MHz DDR3 and the Software Mac OS X Lion 10.7.5 (11G63))

5.1 Testing Methodology and Testing Dataset

The main objective of this chapter is to test the performance of JLOG using empirical testing method. The main assessment factors for testing the performance of JLOG are categorized into the following:

1. Evaluating the query for the first time

   We execute PROLOG queries on normal, tabled, incremental tabled PROLOG and JLOG. The execution time of these queries is analyzed and compared among the four systems. This measurement gives the indication of how fast the system is behaving compared to normal, tabled and incremental tabled implementations of PROLOG when the query is evaluated for the first time. Also it indicates the overhead of building (caching the prof structure) the JTMS network.

2. Re-evaluation of a query

   Once a query is evaluated for the first time, the same query is re-evaluated again on normal, tabled, incremental tabled PROLOG and JLOG. The execution time of re-evaluating of this the query is analyzed and compared among the four systems. This measurement is important as to know how fast the system is capable of extracting the valid answers of the query from the JTMS network and report them back to the user.

3. Evaluating a subquery related to a previously proven query

   We compare the time it takes to evaluate subqueries related to some previously proven queries on normal, tabled, incremental tabled PROLOG and JLOG. This assessment helps knowing if the system is capable of extracting the answers of sub-computations from the cached proof structure rather than asking the PROLOG inference engine to evaluate the sub-query.

4. The cost of maintaining the soundness and completeness of the cashed proof structure for a previously proven query

   After the query is proven the first time, we assert/retract PROLOG facts or rules to/from the PROLOG program that change the state of cached answers for the query. The effect of these changes in the cached proof structure is compared with XSB in terms of time and correctness of the cached answers. These are the main assessment factors of the system. We would like to know how the system is efficient in maintaining the cached proof structure up-to-date, in correspondence to the changes in the database of related predicates. The results can be compared only with PROLOG systems that support incremental tabulation.

5.1.1 Testing Dataset

The basic dataset to test the system is collected from a student information systems (SIS) of a mid-sized University. A variety of PROLOG programs are written (on the top of the dataset taken from the SIS). A variety of queries are designed to asses the performance of the system. The queries are varying based on the following factors:

1. The amount of inference work needed from PROLOG engine to generate the answers of the query.

2. Number of answers generated for the query.

3. Whether the answer set contains duplicated (redundant) answers.

Note that we do not take into consideration the complexity of the query since we are not assessing the performance of PROLOG inference engine. Our objective is to examine the performance of JLOG to support increment tabulation. A brief description of the dataset, the PROLOG programs and its related queries is given below.

```
schedule(0201,000003,1004).
schedule(0201,000003,1005).
schedule(0201,000003,1006).
schedule(0201,000003,1007).
schedule(0201,000003,1008).
schedule(0201,000003,1009).
schedule(0201,000003,1010).
schedule(0201,000022,1011).
schedule(0201,000022,1012).
```

Figure 5.1: Set of PROLOG facts representing the class schedule.

5.1.1.1 Facts

The set of facts used to test the system performance is based on the basic information that is stored in any student information system (SIS). We can categorize the database of facts into four PROLOG predicates:

1. Scheduled classes

   Figure 5.1 depicts the set of PROLOG facts representing the class schedule. The set represents the list of scheduled classes. For example, the PROLOG fact $schedule(0201, 000022, 1012)$ in Figure 5.1 states that the course number 22, section 1012 is offered in semester 0201. Appendix B contains the list of scheduled classes in the academic year 2002/2003 (We did not include the scheduled classes for all 10 years due to the huge size of data).

2. Class enrollments

   Figure 5.2 shows the set of PROLOG facts representing the class enrollments. In this set we collect the classes enrollment data. For example the PROLOG fact $reg(0201, 000253, 0000476, 1035)$ in Figure 5.2 states that during semester 0201 the student number 476 is registered in course 253, section 1035. Appendix C contains a sample enrollment data for the academic year 2002/2003 (We just include sample because the enrollment data for one academic year contains thousands of records).

```
reg(0201,000253,0000247,1035).
reg(0201,000253,0000252,1035).
reg(0201,000253,0000333,1035).
reg(0201,000253,0000361,1035).
reg(0201,000253,0000389,1035).
reg(0201,000253,0000397,1035).
reg(0201,000253,0000435,1035).
reg(0201,000253,0000450,1035).
reg(0201,000253,0000476,1035).
```

Figure 5.2: Set of PROLOG facts representing the class enrollments.

```
grade(0201,000253,0000467,'F' ).
grade(0201,000253,0000292,'C' ).
grade(0201,000253,0000293,'C-').
grade(0201,000253,0000071,'C+').
grade(0201,000253,0000363,'B-').
grade(0201,000253,0000358,'B' ).
grade(0201,000253,0000519,'C' ).
grade(0201,000253,0000511,'C' ).
grade(0201,000253,0000502,'B-').
```

Figure 5.3: Set of PROLOG facts representing the class grades.

3. Grades

   Figure 5.3 depicts a snapshot for the set of PROLOG facts representing the class grades. For example, the PROLOG fact $grade(0201, 000253, 0000502, 'B-')$ in Figure 5.3 states that the student number 253 got a $B-$ grade in class number 502 which is offered during the semester 0201. Appendix D contains a sample list of grades for the Academic Year 2002/2003.

From the point of view of logic programming, the interesting point about the above data is that it can fit in both monotonic and non-monotonic logic. Data related to the previous semesters is considered as monotonic because it represents historical data that can not be changed. If the data related to the previous semesters can be changed then it is considered as non-monotonic. The data related to the current semester falls under the non-monotonic logic since it can be changed throughout the semester.

For example:

1. The set of facts representing the list of scheduled classes and the enrollments in these classes in the previous semesters are considered monotonic.

2. The set of facts corresponding to the student grades in the previous semester classes can not be considered pure monotonic because there is a possibility that the grades can be changed through the grade appeal process.

3. The set of facts representing the list of scheduled classes in the current semester are considered as non-monotonic until the end of the add/drop period. After the add/-drop period they are monotonic since the classes schedule is not going to change.

4. The set of facts that imitates the current semester enrollments are categorized purely under non-monotonic logic because the students enrolled in these classes can drop from the class at any time or at least until a certain point/date in the semester.

The concept of incremental tabulation supported by JLOG and XSB suits the queries that are searching for certain information related to the current semester data. The normal tabulation strategies can be used for the queries that are related to the historical data.

5.1.1.2 Programs and Queries

We implemented a set of simple/straight forward PROLOG programs to test the performance of JLOG using the above database of facts. The programs are varying in their complexity; some of the implemented programs do not require much work from the PROLOG inference engine, while others need a lot of inference work until the answers are generated.

Searching the Base Facts

Figure 5.4 lists the set of PROLOG rules that allows the end-user to search/filter the base facts imported from the SIS system (see Figures 5.1, 5.2 and 5.3) . Different questions (Prolog queries) can be answered through system using these rules.

```
class_schedule(Sem, Cid, Sid):- schedule(Sem, Cid, Sid).
class_enrollment(Sem, Cid, Sid, Stid):- reg(Sem, Cid, Stid, Sid).
class_grade(Sem, Stid, Cid, Grade):- grade(Sem, Stid, Cid, Grade).
```

Figure 5.4: Set of PROLOG rules to search the base facts imported from the SIS system.

For example:

- The query $class\_enrollment(1201, Cid, Stid, Sid)$ requests the PROLOG inference engine to retrieve the enrollment data of all classes scheduled in semester 1201.
- The query $class\_enrollment(1201, Cid, 555, Sid)$ requests the PROLOG inference engine to retrieve the list of classes where the student number 555 is enrolled during the semester 1201.
- The query $class\_grade(Sem, Stid, Cid, 'A')$ requests the PROLOG inference engine to retrieve grades data for all semesters, students and classes where the assigned grade is an A.
- The query $class\_grade(1201, Stid, 1220, 'F')$ requests the PROLOG inference engine to retrieve the list of students who failed in the course 1220 in semester 1201.

Conflicing Classes

Many events in the University require the ability to know the list of scheduled classes that conflict with each other to avoid scheduling of certain events related to these classes at the same time. For example, an extra class is required to be scheduled for a certain course (CSC122, Section1). In order to select the best time for this the extra class, we have to get the list of classes that conflict with our class (CSC122, Section1) and then we choose a time for the extra class that does not overlap with any of the classes in the conflicting list.

```
conflict(Sem,C1,S1,C2,S2):- reg(Sem,C1,St,S1),
                            reg(Sem,C2,St,S2),
                            C1\=C2.
goTogether(Sem,C1,S1,C2,S2):- schedule(Sem,C1,S1),
                              schedule(Sem,C2,S2),
                              C1\=C2, S1\=S2,
                              not(conflict(Sem,C1,S1,C2,S2)).
```

Figure 5.5: PROLOG program to find classes that conflict with each other and the classes that do not conflict.

In this same manner, some events require knowledge of the list of classes that do not conflict with a certain class, or set of classes, which is the inverse of the conflict predicate. For example, a slot is allocated for the final exam schedule with a capacity of 400 students. We can get the list of classes that are not yet scheduled in any of the previous slots, and not conflicting with each other; we schedule each class in the above slot until we reach the maximum capacity of the slot.

Figure 5.5 describes the PROLOG program to find the classes that conflict with each other and the classes that can be scheduled together because they do not conflict with each other. The first PROLOG rule in the program states that two classes conflict with each other if they have at least one common student enrolled in both classes. $C1$ points to the first course, $C2$ points to the second course, St points to the student that is enrolled in both sections. $C1\backslash= C2$ and is used to avoid answers that will state that the same course conflicts with itself. The second rule in the PROLOG program of Figure 5.5 locates every couple of scheduled classes in a certain term; if these two classes do not conflict with each other, then the PROLOG inference engine concludes that these two classes can be scheduled together.

Reachability Problem

Graph reachability is a classic problem with many applications in the real-world. The graph reachability problem has been used as a benchmark in any PROLOG tabled implementation. We mapped the graph reachability to the student information database using the following scenarios:

- Picking a certain student in an academic semester, we would like to know the set of students that can be reached from, connected to, this particular student. We used the assumption that all students registered in the same class are connected to each other, i.e. we add an edge between each couple of students registered in the same class (There are so many scenarios in the student information system that can be mapped to the reachability problem, we just picked one example). Then we apply the transitive closure; if student X is connected to Y, Y is connected to Z; we conclude that X is connected to Z.

- In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph. In a student information database, we would like to know how many connected components of students exist in a certain semester. Each student registered in the current semester is represented as a vertex in the graph. Whenever two students are registered in the same class, we add an edge between these two students (vertices) in the graph.

Figure 5.6 shows the translative closure PROLOG program to find the connected students in a certain semester. This is a two step process. The first rule in the program connects each couple of students registered in the same class. The second rule uses the transitive relation to connect students indirectly.

```
edge(Sem,S1,S2):-    reg(Sem,Course,S1,Section),
                     reg(Sem,Course,S2,Section),
                     S1 < S2.
connected(Sem,X,Y) :-  edge(Sem,X, Y).
connected(Sem,X,Y) :-  edge(Sem,X, M), connected(Sem,M, Y).
```

Figure 5.6: Translative closure PROLOG program to find the connected students in a certain semester.

Repeated Courses

The university would like to study the behavior of the students repeating certain courses. For each academic year, which consists of three semesters (Fall:01, Spring:02, Summer:03), we would like to generate a list of students repeating any course in the given academic year. The list should generate six tuple answers; the course Id, the student Id, the previous semester(s) id when the course was taken by the student, the grade that was assigned to the student in that semester, the current semester id when the course is repeated and the new assigned grade. Figure 5.7 represents the PROLOG program to find the list of students repeating certain classes in a given academic year. The first rule in the program generates the repeat list for the Fall semester, the second and third rules generate the repeat list for the Spring and Summer semesters respectively.

5.2 Query Evaluation

The objective of this section is to examine the performance of JLOG in terms of query evaluation. The test queries are built around the data and programs presented in the previous section. During the test period, JLOG is using the XSB as the PROLOG inference engine attached to it, see Figure 4.1 and Chapter 4 for more details.

```
repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG):-
                                NewS is Year * 100 + 1,
                                grade(OldS, Sid,Cid, OldG),
                                grade(NewS, Sid,Cid, NewG),
                                OldS  < NewS.
repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG):-
                                NewS is Year * 100 + 2
                                grade(OldS, Sid,Cid, OldG),
                                grade(NewS, Sid,Cid, NewG),
                                OldS  < NewS.
repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG):-
                                NewS is Year * 100 + 3,
                                grade(OldS, Sid,Cid, OldG),
                                grade(NewS, Sid,Cid, NewG),
                                OldS  < NewS.
```

Figure 5.7: PROLOG program to find the list of students repeating certain classes in given academic year.

Each test query is evaluated four times on Xsb and JLOG using the following scenarios:

1. Evaluate the query on XSB as a normal PROLOG query with no tabulation support. We refer to this run as NP.

2. Evaluate the query on XSB using the traditional tabulation method. We refer to this run as TP.

3. Evaluate the query on XSB using the incremental tabulation method. We refer to this run as ITP.

4. Evaluate the query on JLOG using the incremental tabulation method. We refer to this run as JLOG.

We compare the statistics of evaluating the test queries using the above scenarios. Each query is evaluated four times according to the above scheme. For example the first run

evaluates the query using tabulation support (NP). Then, the next run evaluates the query with normal tabulation support (TP). Within each run, we repeat the query evaluation 20 times and record the time for each repetition. Then, we take the mean value for recorded repetition times. Repeating the runs and taking the mean value minimize the inconsistency of CPU times. The difference in times between the repetitions is not exceeding, in any case, more than 5-10%. This is due to the fact that PROLOG is a deterministic system. Being a deterministic system means that usually the time taken to repeat the same computation is consistent.

The query timings reported in the following subsequent sections is based on the mean value of evaluating the same query 20 times. We compare the results of evaluating the query for the first time, then the results of re-evaluating the same query and subqueries related to it. We skip the statistics of re-evaluation or subquery evaluation whenever there are no significant performance differences between the four runs (NP, TP, ITP and JLOG), or whenever the figures are similar to one of the previous examples.

5.2.1 Querying the Base Facts

Figure 5.4 lists the set of PROLOG rules that allows the end-user to search/filter the base facts imported from the SIS system. The university would like to keep track of students taking an $'A'$ grade for all classes scheduled in any semester.

The query $courseGrades(S,C,Y,'A')$ filters the database of grades based on the assigned grade which must be an $'A'$. This is a very simple query because it just filters the base facts which do not require much inference work from the PROLOG engine to generate the results of the query.

Figure 5.8 shows the statistics of evaluating the query $courseGrades(S,C,Y,'A')$. The file size for the database of facts is varying from $12K$ to $175K$. The results are divided into two categories:

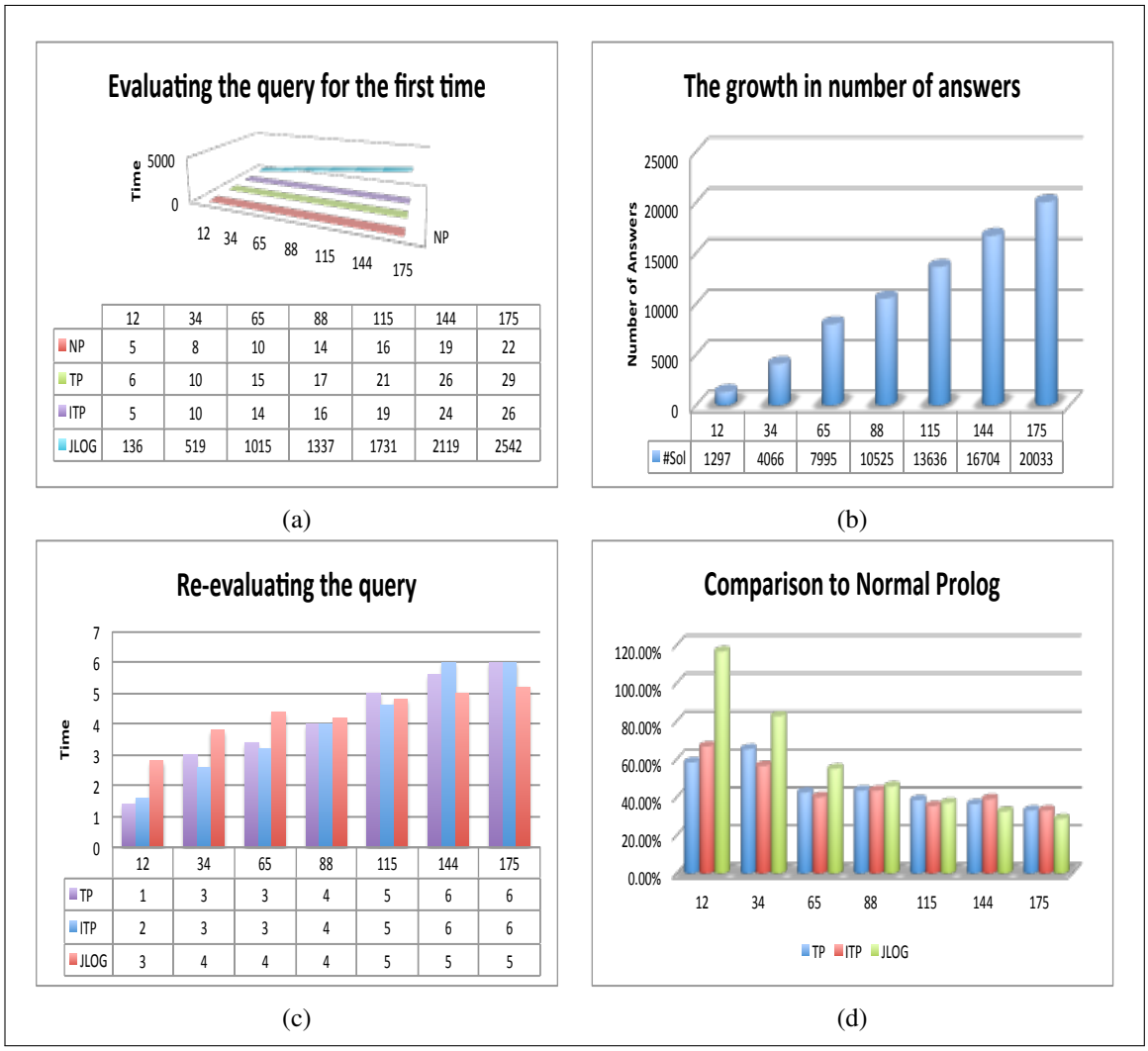

Figure 5.8: Statistics of evaluating the query $courseGrades(S, C, Y, 'A')$.

1. Evaluating the query for the first time

   The table in the figure 5.8 (a) shows that the Normal PROLOG evaluation is faster than all tabled PROLOG systems (TP, ITP and JLOG) since the query does not require much inference work while the tabled systems are paying the cost (overhead) of caching the results of the query. JLOG is paying the highest overhead since it caches the proof structure of the query rather than the end results as the case in TP and ITP. JLOG suffers from significant overhead for this kind of query when it is evaluated for the first time because there is no heavy inference work required from the PROLOG engine to generate the answers for the query, and the query has a large amount (Figure 5.8 (b)) of answers that must be converted into a JTMS network; However, the worst case difference between JLOG and the other tabled runs is 2 to 3 seconds.

2. Re-evaluating the query

   When the query $courseGrades(S,C,Y,'A')$ is re-evaluated, the results in Figure 5.8 (c and d) proves that all tabled systems are getting the benefits of tabulation. On average, tabled evaluations are 45% to 55% faster than the normal PROLOG evaluation. The average enhancement in tabled runs is not that high compared to NP, as we will see in next examples, due to the fact that the query does not require high inference work from the PROLOG engine. JLOG is slower than TP and ITP for the small size of the database of facts, but as the size is growing JLOG is enhancing its performance in a very stable manner. The results show that JLOG is performing better than TP and ITP when the size of the database is growing by 10-15%. In this particular example, the behavior of re-evaluating the query in all tabled runs is almost the same since the worst case difference in the performance is very small.

5.2.2 Repeating Courses

Figure 5.7 represents the PROLOG program to find the list of students repeating certain classes in a given academic year. The general queries related to the this program requires a lot of inference work from the PROLOG engine to generate the answers and usually the size of the answer set is small. Also, the data(answers) related to the repeated courses rarely contain redundant values which means that when the query is proven for the first time there will be no much difference in the execution time between the tabled and non-tabled runs. Figure 5.9 shows the statistics of evaluating the query $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$. We evaluate the query by assigning different academic year (2003..2011) values to the variable *Year*. For example, when the query is generating the repeated courses in 2007, the database of PROLOG facts contains all the grades stored in the SIS up to the academic year 2007. The file size for the database of facts is varying from $12K$ to $175K$. The results are divided into three categories:

1. Evaluating the query for the first time

   The table in Figure 5.9 (a) shows the required time to execute the query

   $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$ for different academic years. As explained earlier, it is expected that the query takes a long time to be proved by the PROLOG inference engine, and this is what the figures are proving in the table. Also, the differences in timing between tabled (JLOG, TP, and ITP) and NP is relatively small. The values in the table shows that JLOG and TP are faster than the NP and ITP by almost 10% when the query is proved for the first time. Also, in this example, JLOG is the fastest among all others (NP,TP, and ITP). The chart in Figure 5.9 (b) shows the number of answers generated for the selected academic years. It is clear from the chart that the number of answers for the query is comparatively small related to the number facts (grades).

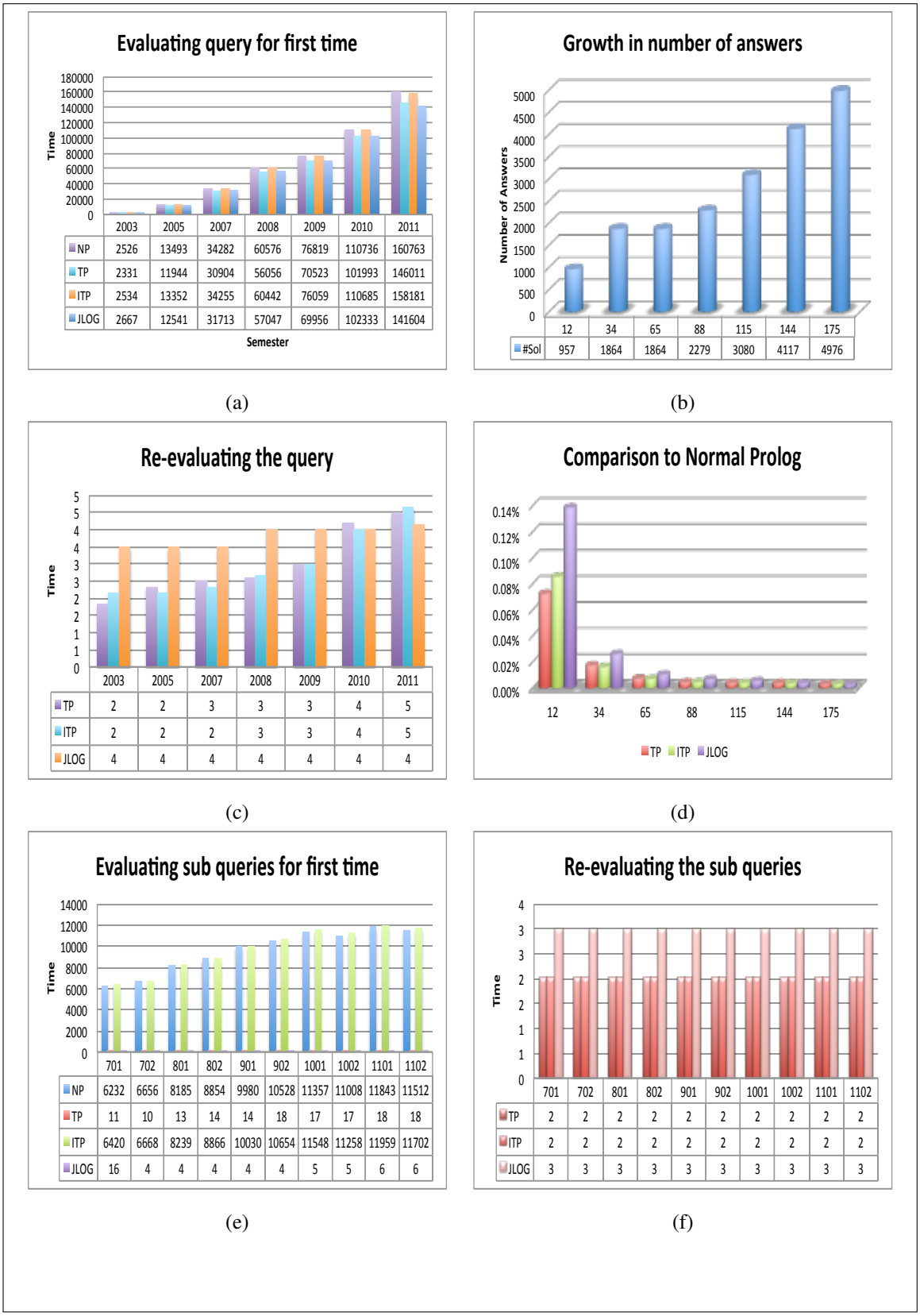

Figure 5.9: Statistics of evaluating the query $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$.

These kinds of Prolog programs and queries are ideal for JLOG for two reasons:

(a) When proving the query for the first time, tremendous inference work is needed from the PROLOG engine. When the database of facts is changed after proving the query, JLOG minimizes the inference work required to update the cached proof structure for the query. Actually JLOG will only ask for the help of the inference engine when we assert new data related to the query. Any change in the existing data related to the query does not require inference work from PROLOG side and will be handled by propagating the effect of new changes throughout the JTMS network.

(b) The number of answers generated for the query is small. This means that the overhead of building the JTMS network for the query at JLOG side will be relatively small.

2. Re-evaluating the query

   Figure 5.9 (c and d) shows the required time to re-evaluate the query $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$ for different academic years. This is clearly a winning situation for all tabled (JLOG, TP, and ITP) runs. The time required by tabled runs to re-evaluate the query is on average 1% of the time needed by normal PROLOG to do the same work.

3. Evaluating the sub queries

   Once the query is proven for the first time, and it is tabled using the normal or the incremental tabulation strategies used by XSB or the proof structure for the query is cached as the case in JLOG, then evaluating the sub queries related to the tabled query should not take significant time since it is a process of filtering the cached query answers that can be unified with the new sub query. The current version of XSB (incremental tabulation) considers the evaluation of subqueries related to a tabled query, which has been already evaluated, as a new query. The system recomputes the whole inference work to generate the answers which are already

tabled for the main query. This is not the case with JLOG where minimal inference work is required from PROLOG engine to generate the answers for the subquery that is related to a previously proven query. Going back the Prolog program of Figure 5.7, consider the following cases:

(a) We evaluate the query $repeatedCourses(11, Cid, Sid, OldS, OldG, NewS, NewG)$ that generates the list of repeated course during the academic year $2011/2012$.

(b) We would like to evaluate some special case subqueries related to the above query. For example, we would like to know the list of repeated courses in the academic year $2011/2012$ that were taken previously by the student in semester 0701. To retrieve such list, we evaluate the subquery $repeatedCourses(11, Cid, Sid, 0701, OldG, NewS, NewG)$.

Figure 5.9 (e and f) shows the statistics for evaluating and re-evaluating the subqueries based on filtering the answers of the main query per semester. When the subquery is evaluated for the first time, the performance of JLOG and TP is faster than ITP AND NP. Comparing JLOG with TP we find that JLOG is slower than TP for the first subquery and after that the performance of JLOG is slightly better than TP. This is due to the extra setup needed by JLOG in the JTMS network to support the evaluation of subqueries.

5.2.3 Conflicting Classes

Figure 5.5 depicts the PROLOG program used to find the classes that conflict and the classes that can be scheduled together because they do not conflict. The general query related to the first rule in the program is $conflict(Sem, C1, S1, C2, S2)$. The query asks the system to find all tuples of classes scheduled in a certain semester (Sem) such that class1 $(C1, S1)$ conflicts with class2 $(C2, S2)$. These kinds of queries generates lot of redundant answers. For example, lets us assume that the number of common students between two classes $CSC122/1$ and $CSC130/1$ is 10 students. To state that these two classes are

conflicting with each other it is enough for the system to find a student that is enrolled in both classes and report the answer once, however the PROLOG search will report that these two classes conflict with each other at least 10 times which represents the number of common students in both classes. The tabled system can enhance the performance of this query by memorizing the first answer which admits that the two classes $CSC122/1$ and $CSC130/1$ conflict with each other. Later, any attempt by the Prolog inference engine to find the other nine solutions can be avoided. The above strategy saves time and memory space required to table the answers and works perfectly under the monotonic logic. If the query has to work under the non-monotonic logic, then some sort of inference work is needed to maintain the completeness and consistency of the tabled answers. Lets assume the Id's of the common students between the two classes $CSC122/1$ and $CSC130/1$ are numbered from $[1..10]$. When the query is proven for the first time by the PROLOG engine, we assume that the first answer stating that these two classes are conflicting is coming from the facts that the student number 1 is registered in both classes. What happens, if the student number 1 dropped one of these classes, shall we remove the tabled answer or not? The answer is no because there are still nine common students in both classes. XSB deals with this situation by maintaining a call dependance graph to be able to check if the tabled answer should be removed or not after any change in the data related to the tabled answers. This means that any change in the database of facts requires inference work from the PROLOG engine to maintain the table space.

JLOG uses another approach to handle the above situation, the system requests the PROLOG engine to generate all answers for the query including the redundant ones. For each answer, a justification is installed in JTMS network attached to the query. Since there are ten common students between the two classes $CSC122/1$ and $CSC130/1$, JLOG installs ten justifications, one justification for each answer. The consequence of each justification is the same; they differ in the antecedents. When student number 1 drops one of these classes, the related antecedent of the justification is marked as OUT but the consequence of the justification is not marked OUT because there still are nine more

Figure 5.10: Statistics of evaluating the query $conflict(Sem, C1, S1, C2, S2)$.

justifications that are supporting this consequence. If all the ten students dropped from one of the classes, then in this case, the system believes that classes $CSC122/1$ and $CSC130/1$ do not conflict. The strategy to handle the above situation does not require any work from the PROLOG inference engine. The disadvantage of this approach is the space and time overhead that JLOG has to pay for building the JTMS network when proving the query for the first time.

Figure 5.10 shows the statistics of evaluating the query $conflict(Sem, C1, S1, C2, S2)$. We evaluate the query on the data of one academic year data. The file size for the database of facts is varying from $2K$ to $16K$. The charts(a and b) in the Figure 5.10 represent the time taken and number of answers generated by the PROLOG engine when evaluating the query for the first time. NP reports all answers of the query. As an average, 55% of the query answers are redundant that is why these answers are not reported by TP and ITP. JLOG receives all the answers of the query from the PROLOG engine and installs a justification for each answer, however, it reports to the end user answers that are non-redundant only. This is the reason why JLOG is the slowest when the query is evaluated for the first time (in comparison with NP, TP and ITP).

The charts (c and d) in Figure 5.10 show the statistics of re-evaluating the query $conflict(Sem, C1, S1, C2, S2)$. It is clear from the figures, in the charts, that performance of JLOG is better than NP, TP and ITP for larger sizes of the database of facts. The last two charts (e and f) in Figure 5.10 present the statistics of evaluating and re-evaluating the subqueries of $conflict(Sem, C1, S1, C2, S2)$ based on filtering the answers for a particular semester. For example, the query $conflict(201, C1, S1, C2, S2)$ returns the list of conflicting courses in Fall 2001. As explained in the previous section, JLOG's performance is better than ITp when evaluating the subquery for the first time. JLOG is slower than TP in the first evaluation of each subquery because we run each subquery on a different database which requires extra setup in the JTMS network (to support the subqueries). This situation is in contrast to the example of Section 5.2.2 where each subquery is evaluated on the same database.

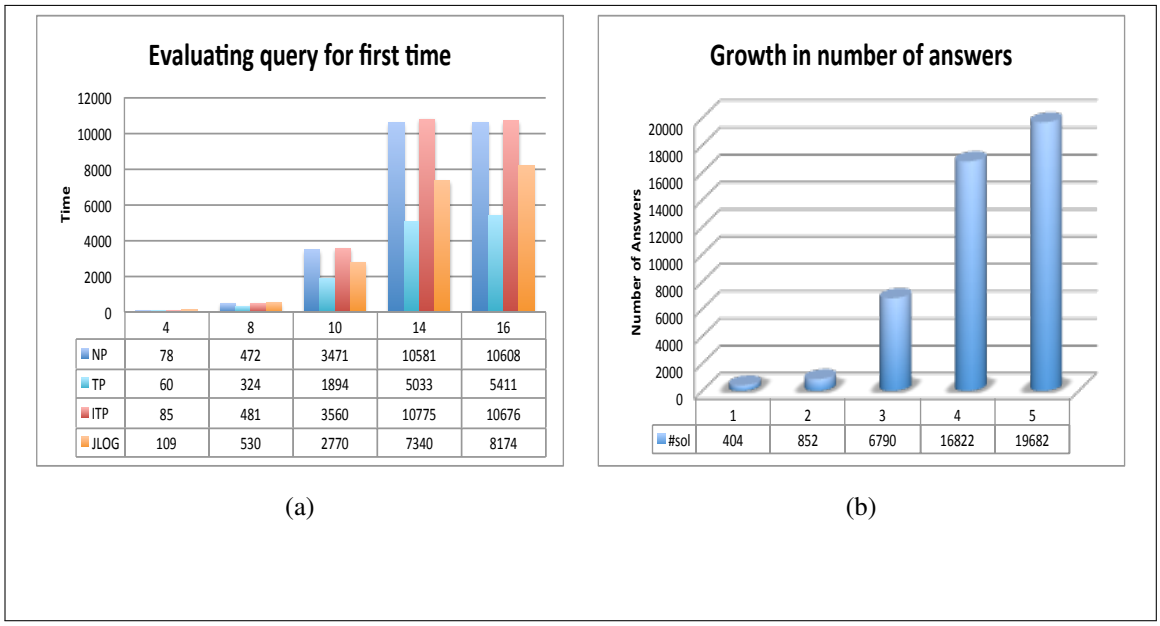

Figure 5.11: Statistics of evaluating the query $goTogether(Sem, C1, S1, C2, S2)$.

5.2.3.1 Non Conflicting Classes

The second rule of the PROLOG program in Figure 5.5 allows us to find all tuples of classes that do not conflict with each other. In fact, the second rule is the inverse of the first rule in the program. Figure 5.11 shows the statistics of evaluating the query $goTogether(Sem, C1, S1, C2, S2)$ for the first time. This is a query which requires heavy inference work, no redundant answers exist and the values are reasonable. In other words an ideal situation for JLOG. Hence, JLOG is faster than ITP which is almost running at the speed of NP. TP is faster than JLOG, which is not the case in the example of Section 5.2.2, because the number of answers in this example is relatively larger.

5.2.4 Students Connectivity

Figure 5.6 shows the translative closure PROLOG program used to find the connected students in a certain semester. Given the enrollment data up to a certain academic year, we would like to list all the students connected to a particular student. For example, the query $connected(Sem, 946, Y)$ finds the list of students connected to the student number 946 in all the semesters that exist in the database of facts. Figure 5.12 presents the statistics of

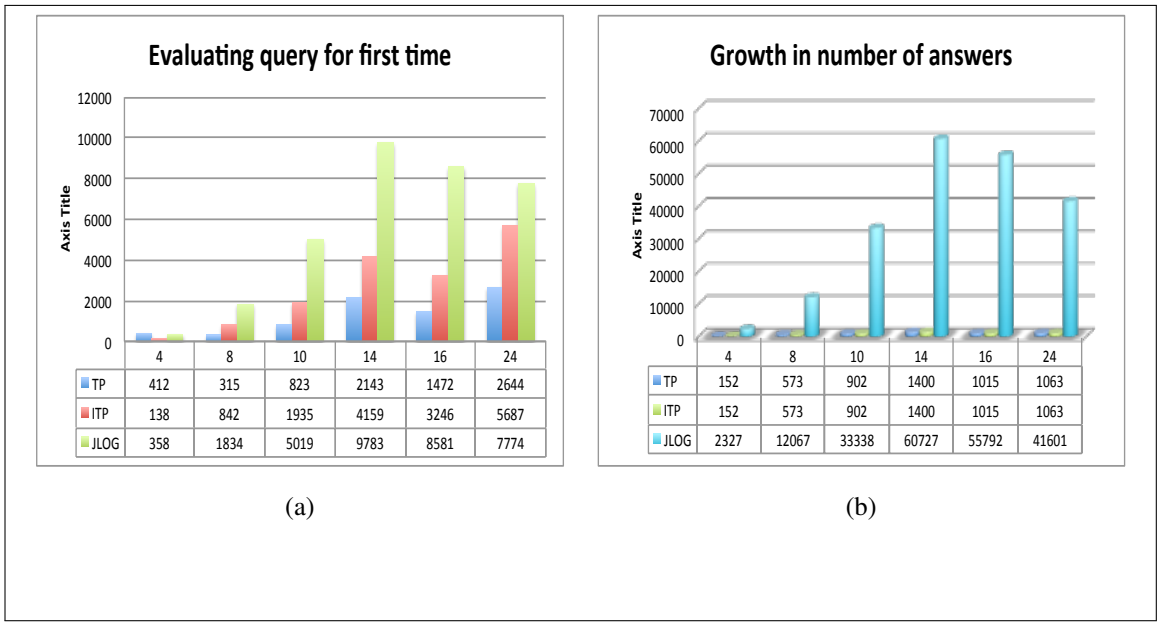

Figure 5.12: Statistics of evaluating the query $connected(Sem, 946, Y)$.

evaluating the query $connected(Sem, 946, Y)$ for the first time. The graph that is going to be constructed from the relation $edge/3$ contains cycles which yields that this query suffers from infinite loop in NP while it terminates successfully in all tabled (TP, ITP, JLOG) runs. The query generates a lot of redundant answers which are neglected by TP and ITP. These answers are not neglected by JLOG, hence it is suffering from overhead when the query is proved for the first time. We will propose a solution in Chapter 6 (future directions) that should resolve this overhead problem.

5.3 Cost of Maintaining the Completeness and Soundness

JLOG has to ensure that whenever the base facts/rules participating as antecedents in any justification are asserted/retracted, the effect of this assertion/retraction should be propagated through the JTMS justifications in order to keep the proof structure sound. This mechanism ensures the soundness of the cached proof for any query. At the same time, JLOG must ensure the completeness of the proof structure which is the responsibility of

the database monitor. The database monitor checks whether an assertion of new facts/rules (TMS node) triggers the resumption of a cached proof to add a new justification to the proof structure. We are picking few examples from the previous section queries to show the cost of maintaining the soundness and completeness of the cached proof structure for these queries. In this section we compare the performance of JLOG with XSB (ITP) only because it is the only module that is capable of maintaining the soundness and completeness of the query answers when the database of facts/rules is changed.

5.3.1 Course Grades

Figure 5.8 shows the statistics of evaluating the query $courseGrades(S,C,Y,'A')$. This is a basic query which is simply filtering the database of facts. The performance of NP is quite fast for this query. ITP is faster than JLOG when evaluating the query for the first time. Now let's consider the general case of the query which is $courseGrades(S,C,ST,G)$. We evaluate the query for the first time, and then to test the soundness of the cached query we retract 20-30% of existing grades that were presented when the query was proved for the first time. In the same manner we assert 20-30% of the original grades as new grades to generate new answers (justifications) for the query. Figure 5.13 shows the statistics of maintaining the soundness and completeness of the cached query $courseGrades(S,C,ST,G)$.

The first two charts(a and b) in the Figure 5.13 show the cost of maintaining the soundness and completeness of the query $courseGrades(S,C,ST,G)$ based on a different number of answers(justifications) for proving the query for the first time. It is clear from the charts that ITP is slightly faster than JLOG in keeping the query answers complete. This is expected because JLOG has to refer to the PROLOG engine through an external library whenever it wants to execute a PROLOG command, whereas ITP is implemented at low level in XSB so the communication is faster with the PROLOG engine, however, the performance is very close and as the database gets larger, JLOG is performing better.

On the other hand, JLOG is performing much better than ITp when the case comes to keeping the proof structure (tabled answers in XSB) sound. The reason behind this

Figure 5.13: Statistics of maintaining the soundness and completeness of the cached query answers and then re-evaluating the query $courseGrades(S, C, ST, G)$.

difference is that JLOG does not require any PROLOG inference work to update the JTMS network as the case for ITp. In JLOG, any assert/retract related to existing TMS nodes is a matter of changing the label from *IN* to *OUT* or vice versa and propagate the effect of this change throughout the JTMS network. Still, the performance of JLOG can be enhanced because every time a rule or a fact is asserted/retracted by the end user, JLOG informs the PROLOG engine about this change for future use, and then it propagates the effect (the assertion/retraction) in the JTMS network. A better way to handle this situation is to keep all the assertions/retractions at the JLOG side. Whenever there is a real need for some inference work at that time, JLOG informs the PROLOG engine about these changes.

The third and fourth charts(c and d) in Figure 5.13 show the comparison between maintaining the completeness and soundness of each system. In general, JLOG handle soundness of the proof structure faster than completeness due to less or no inference that is required from the PROLOG side. When we look at ITP, it handles completeness of tabled answers of the query faster than soundness. This might happen because dealing with completeness adds new answers to the table which is faster, while dealing with soundness needs searching for and deleting the invalid answers from the table (which is a two phase operation).

The last two charts(e and f) in the Figure 5.13 present the figures of re-evaluating the query $courseGrades(S,C,ST,G)$ after updating the answers of the query. JLOG and ITP are almost behaving the same way with a slightly better performance of JLOG. The query $courseGrades(S,C,ST,G)$ is handled by normal Prolog engine (non0tabled module) in a quite reasonable time. Our recommendation for simple queries which do not require much work from the PROLOG inference engine, is that the query should not be tabled incrementally.

5.3.2 Repeating Courses

Figure 5.7 represents the PROLOG program to find the list of students repeating certain classes in a given academic year. Figure5.9 shows the statistics of evaluating the query $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$. The query requires a lot of inference work from the PROLOG engine to generate the answers. The number of the answers is small. This is the ideal situation for JLOG. When the query is proved for the first time, JLOG and ITP are almost running with the same speed with a slight faster response from JLOG. Let's consider the following two scenarios that will allow us to test the completeness and soundness of the cached proof structure for the above query:

1. Testing the soundness

   We would like to filter the results of the query to include only a certain semester in the academic year. For example, we are only interested in finding the repeat list of the Fall (01) semester. This can be achieved by retracting the second and third rule of the PROLOG program of Figure 5.7. To assess the system performance on the example, we compute the cost of retracting the rules followed by re-evaluating the query and then asserting the rules back to be on the original state.

2. Testing the completeness

   To asses the completeness, first we retract the second and third rules in the program of Figure 5.7. Then we evaluate the query for the first time. The query contains the repeated list for the first semester Fall (01). Later, we assert the second rule in the program of Figure 5.7. This should update the query results to include the Fall (01) and Spring (02) repeated list.

To achieve the above scenarios, the PROLOG predicate $repeatedCourses/6$ in the program of Figure 5.7 must be defined as dynamic in order to be able to assert and retract the program rules. This is different from the situation in example 5.2.2 where the predicate $repeatedCourses/6$ is defined in case of ITP as incrementally tabled but not dynamic. In our case, we define $repeatedCourses/6$ as incrementally tabled and dynamic.

Once *repeatedCourses*/6 is defined as dynamic, ITP is behaving in an unusual manner, the time it takes to evaluate the query for the first time is almost the same when the query is re-evaluated. Both times are similar to the time taken by NP. This means that the incremental tabulation is not supported on a top level predicate which is defined as dynamic, however it is supported on the below level predicates. When the predicate *repeatedCourses*/6 is not defined as dynamic, it is still depending on the predicate *reg*/4 which is defined as incrementally tabled and dynamic as well, see Figure 5.7.

Coming to the testing of soundness and completeness of the query answers, Figure 5.14 presents the statistics of maintaining the soundness and completeness of the cached query answers and then re-evaluating the query
repeatedCourses(*Year*,*Cid*,*Sid*,*OldS*,*OldG*,*NewS*,*NewG*). It is clear from the the Figure 5.14 that ITP is following the 'from scratch evaluation' strategy for this query regardless to whether the query is evaluated before or after any assert/retract command. The first four charts (a, b, c and d) in Figure 5.14 show the time it takes for JLOG to keep the query proof structure sound. When the second and third rules of the PROLOG program of Figure 5.7 are retracted, JLOG takes a few milliseconds to update the Jtms network, and then, re-evaluating the query takes the same time as it was taking when the query was re-evaluated before the retract operation takes place, see Figure 5.8. When the same rules are asserted back, the maintenance of the JTMS network attached to the query takes again a few milliseconds. Keeping the proof structure sound, in this example, is done in a very short time because all the changes in the JTMS network are based on the retraction of two rules only (two TMS nodes). The situation in the example of Section 5.3.1 is different since the maintenance of query soundness is based on plenty of base facts that are either retracted/asserted from/to the PROLOG program and that is why it takes more time to keep the proof structure sound.

The last two charts (e and f) in Figure 5.14 shows the time it takes from JLOG to keep the query proof structure complete when we assert the second rule of the PROLOG program of Figure 5.7. Note that the query was evaluated for the first time where only

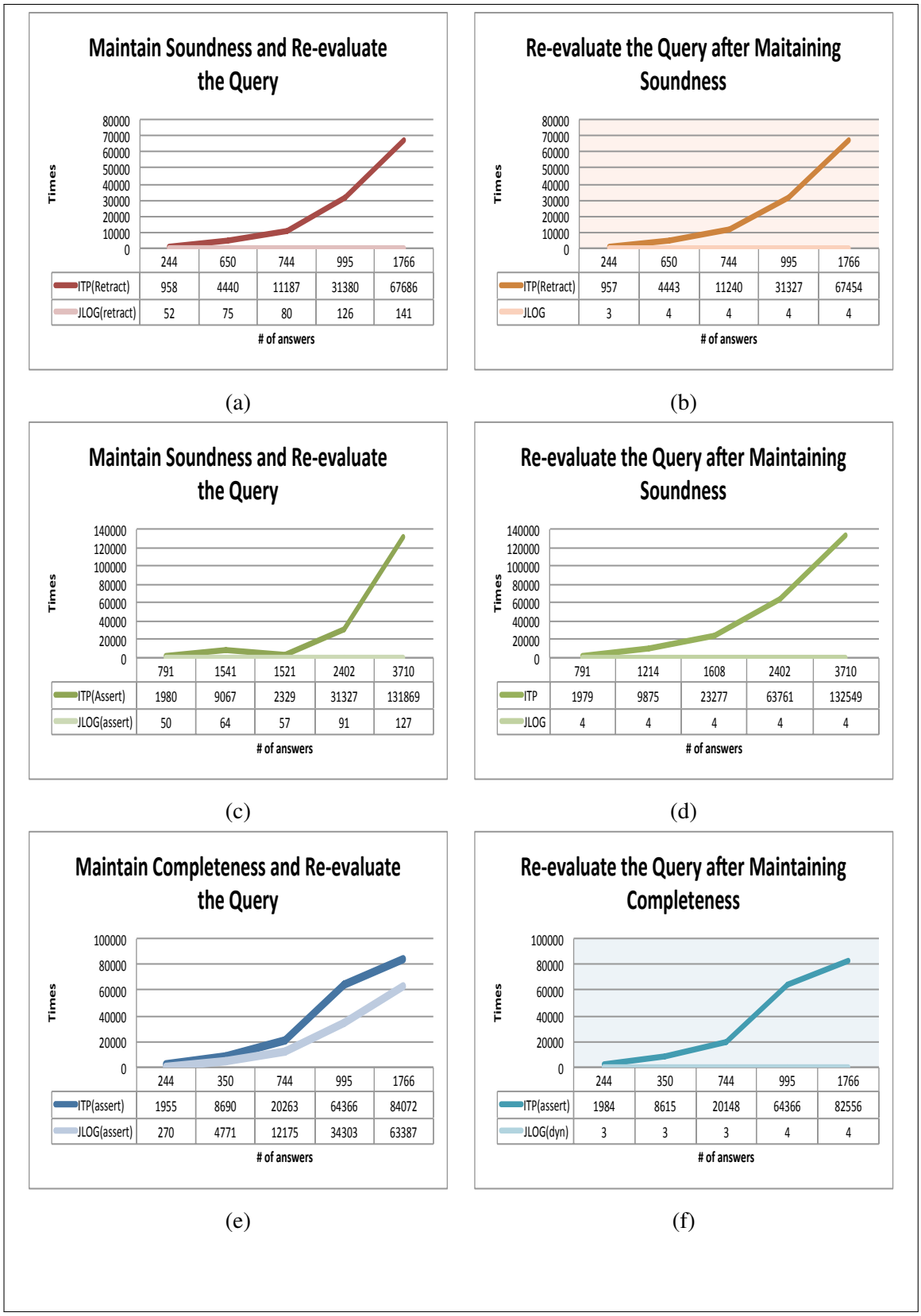

Figure 5.14: Statistics of maintaining the soundness and completeness of the cached query answers and then re-evaluating the query $repeatedCourses(Year, Cid, Sid, OldS, OldG, NewS, NewG)$.

the first rule of the program existed. This action adds a new rule to the program which requires inference work from PROLOG engine to generate new answers(justifications) for the query, and then adds the new justifications to the query's JTMS network. Jlog handles the execution of the query related to the new assert rule in time which is less than the time it takes to evaluate the whole query from scratch as the case with ITP. When the query is re-evaluated after maintaing the completeness of cached proof structure, JLOG speed is faster than ITP.

5.3.3 Student Connectivity

Figure 5.6 shows the translative closure PROLOG program used to find the connected students in a certain semester. To test the soundness and completeness, we picked samples of the data such that the number of edges (facts), coming from the $reg/4$ predicate, is between 2 to $4k$. The sample takes snapshot of the data before the first day of classes, i.e. start of add/drop period in the university. First we evaluate the general query related to this program which is $connected(Sem, X, Y)$. We pass the semester values for which we are looking the list of connected students.

Then, we use the following scenarios to test the soundness and completeness of the cached proof structure:

1. We track all the changes that take place on the predicate $reg/4$ starting from the 1st day of add drop period until the end of semester. When a student drops (soundness) a class, the related PROLOG fact is retracted. When a student adds a new class, then the fact is asserted. This can be a new fact (completeness) if the student is adding the class for the first time, or it can be an old fact (soundness) because the student dropped the class after registering it for the first time and then decided to reenroll back in the class. The current version of JLOG updates the JTMS network, attached to the cached query, whenever the assert/retract command is executed. This means that the query proof structure is always updated and returns the correct answers.

The incremental support of XSB (ITP) offers two ways to handle the assertion and retraction of facts/rules related to the incrementally tabled predicates:

- *incr_assert*$(+Clause)$

  This command adds the clause to the database of fact/rules after any other clauses for the same predicate currently in the database. Then it updates all incrementally maintained tables that depend on this predicate. This command must be used if we would like to keep the query answers updated after each single change in the database.

- *incr_assert_nval*$(+Clause)$

  This command is similar to *incr_assert*$/1$ except that it does not update the incrementally maintained tables, but only marks them as invalid. The tables should be updated by an explicit call to *incr_table_update*$/0$. This separation of functionality gives XSB better efficiency for processing the table maintenance after a batch of operations. This approach can be used if we have no problems updating the query answer through batch processing. For example, we update the query results by the end of the business day, which means that cached answers are correct for the previous day events.

Figure 5.15 shows the statistics of maintaing the soundness and completeness of the query $connected(Sem,X,Y)$ based on the changes in the predicate $reg/4$. For the same add/drop events ITP (XSB) is faster that JLOG. The reason behind this difference in performance is coming from the fact that JLOG is updating the JTMS network after each assert or retract command, while ITP is handling the situation through batch processing using the *incr_assert_nval*$/1$ and *incr_table_update*$/0$. When the tables were updated after each assert or retract command in ITP, the performance of the system was degraded. For example, for the semester 1101, ITP takes 47608 milliseconds to update the query answers through batch processing, see Figure 5.15. This time jumps to 12,972,825 milliseconds when we tried to update

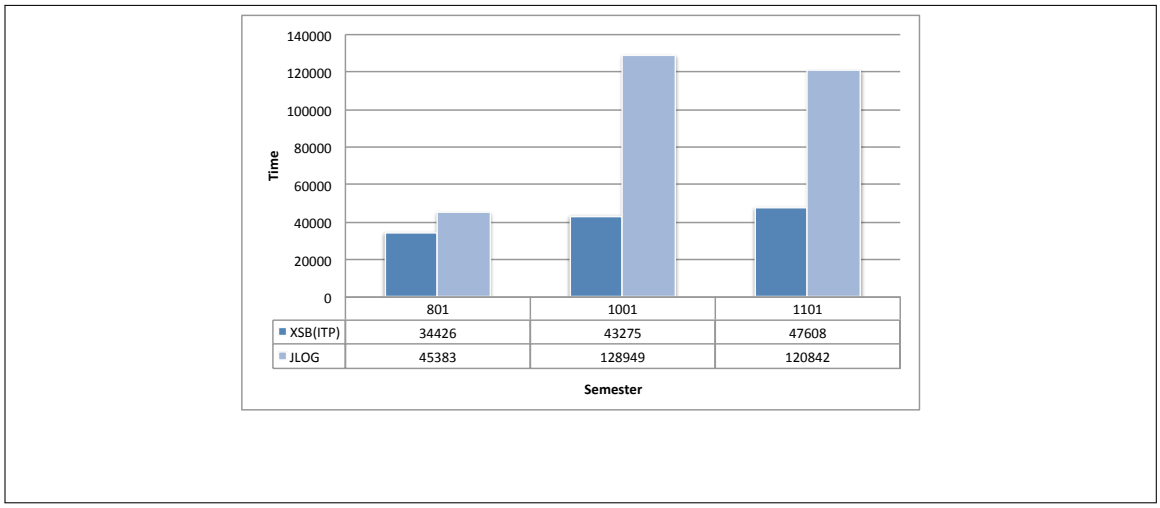

Figure 5.15: Statistics of maintaing the soundness and completeness of the query $connected(Sem,X,Y)$ based on the changes in the predicate $reg/4$.

the tables after each assert/retract command. JLOG updates the JTMS network after each assert/retract in 120,842 milliseconds which is significantly lower than the time taken by ITP to handle the events one by one.

2. The second scenario is used to test soundness of the query which is related to assertion/retraction of rules in the program of Figure 5.6. Consider the case where we would like to know the list of students who are connected directly and we want to exclude the tuple of students who are connected indirectly. This can be achieved by retracting the third rule in the program of Figure 5.6 which connects the students indirectly. In order to be able to retract the rule, the predicate $connected/2$ must be defined as incrementally dynamic. Once $connected/2$ is defined as dynamic predicate, the query $connected(Sem,X,Y)$ does not terminate in ITP. ITP fails to handle the query due to an infinite loop. Recall from Section 5.1.1.2 that this query does not terminate also under NP. JLOG handles the situation smoothly. Figure 5.16 shows the time taken by JLOG for maintaing the soundness of the query $connected(Sem,X,Y)$ after retracting of the rule

   $connected(X,Y) :- edge(X,M), connected(M,Y)$ from the PROLOG program of Figure 5.6. JLOG handles the retraction easily because it is a single event used to

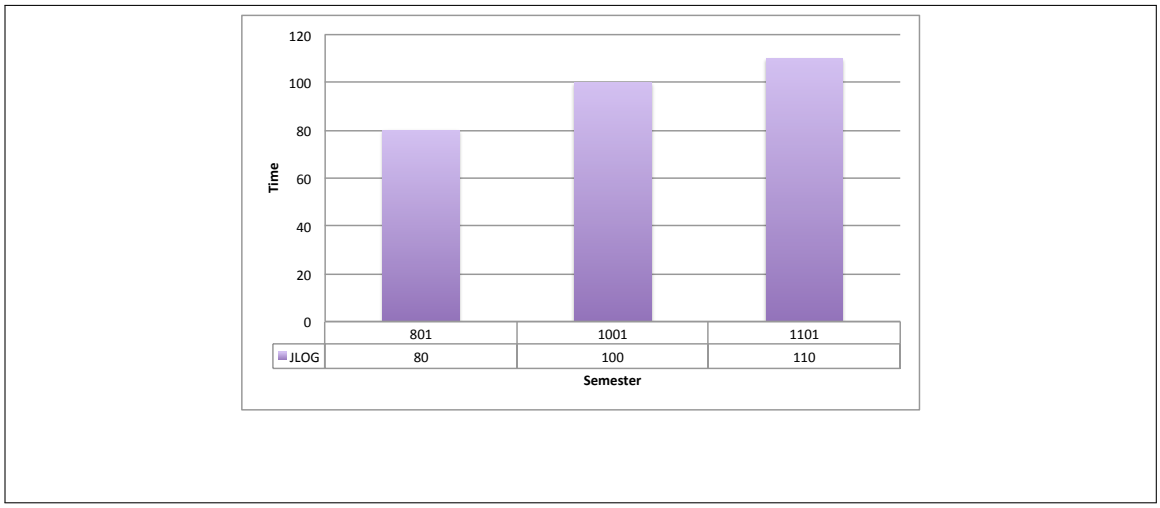

Figure 5.16: Maintaing the soundness of the query $connected(Sem,X,Y)$ after retracting of the rule $connected(X,Y) :- edge(X,M), connected(M,Y)$ from the PROLOG program of Figure 5.6.

update the Jtms network and does not require any inference work from the PROLOG side.

5.4 JLOG Versus XSB

In this section, we summarize our findings that differentiate between JLOG and XSB. When we talk about XSB we mean the incremental tabulation support for tabled predicates that is available in XSB. The comparison between JLOG and XSB is based on the following factors:

1. Tabulation approach.
2. Supported PROLOG programs.
3. Evaluating the query for the first time.
4. Re-evaluating the query.
5. Subqueries evaluation.
6. Keeping the soundness and completeness of the cached answers (proof structure) for a query.

5.4.1 Tabualtion Implementation Approach

JLOG and XSB allow the user to declare that the system should incrementally maintain soundness and completeness of the answers that are tabled (cached) for queries related to certain predicates in the program. An incrementally maintained table is one that continually contain the correct answers in the presence of updates to underlying predicates on which the tabled predicate depends [Saha & Ramakrishnan, 2003]. If tables are thought of as database views, then this subsystem enables what is known in the database community as incremental view maintenance [Mohania et al., 1997, Lee et al., 2001].

The core difference between JLOG and XSB is the approach used to support incremental tabulation. XSB saves the end results of the query in a table space. Along with the table space, the system keeps the dynamic call dependency graph to track the call dependency between the predicates. The system keeps track of which tabled goals depend on which other tabled goals and (incremental) dynamic goals. When the base facts/rules related to the tabled answers get asserted or retracted, XSB uses the call dependency graph to determine which tabled answers should be removed from the table, and which new answers should be computed and added to the table space.

JLOG uses alternative approach to support the incremental tabulation, which is the basis of this thesis. Instead of saving the final answers for the query, the system stores the proof structure that was generated when the query was proved for the first time. The proof structure is stored as a JTMS network. Two components are used to update the proof structure when the related facts/rules get asserted or retracted:

1. The TMS component that keeps the proof structure consistent.

2. The database monitor system that keeps the proof structure complete when new data is added to the system.

5.4.2 Implementation Approach

Incremental tabulation support in XSB is implemented by modifying and extending the low-level PROLOG engine. JLOG is using the source level transformation approach to support the incremental tabulation. It applies the source level transformations to a tabled program and then uses external tabling primitives (JTMS network) to provide direct control over the search strategy. JLOG implementation approach gives the advantage of incorporating incremental tabulation support to any PROLOG systems that provide external library interface.

5.4.3 Types of Supported PROLOG Programs

JLOG supports definite and stratified PROLOG programs. The behavior of the programs in JLOG depends on the PROLOG engine (system) attached to JLOG. If the program terminates on the PROLOG engine then it works fine with JLOG. The current version of JLOG can be integrated with YAPPROLOG, SWI-PROLOG and XSB. We recommend to integrate JLOG with YAPPROLOG and XSB since both of them support normal tabulation. When JLOG is attached to a PROLOG engine that supports tabulation, then it can handle non-stratified programs that terminate on the PROLOG engine. In the current version of XSB, the incremental tabling works for definite and stratified programs that do not involve conditional answers.

5.4.4 Evaluating the Query for the First Time

The time it takes to execute a query for the first time on JLOG and XSB depends on the nature of the query. A query that requires heavy inference work, and generates reasonable number of answers is the ideal one for JLOG. JLOG handles such queries faster than XSB. XSB on the other hand performs better than JLOG when the query generates a large number of answers, a lot of the answers are redundant so they are not added to the table space, and/or the query does not require heavy inference work.

When the query contains a large number of answers, JLOG has to convert each answer to a justification even if the final answer is repeated already because the antecedents of the justification are different for each answer (consequence). This is a time and memory consumption problem. A solution to enhance the performance of JLOG with such queries is described in chapter 6.

5.4.5 Re-evaluating the Query

Once the query is evaluated for the first time and tabled, both systems are getting benefits from the tabulation and return the answers of the query in very fast manner. We did not point out major differences in the performance of JLOG and XSB when it comes to re-evaluating a query. However, there is a slightly faster response from JLOG when the number of answers for the query is large.

5.4.6 Evaluation of Subqueries

This point is related to one of the contributions of this thesis. The current version of XSB considers the evaluation of subqueries related to a tabled query, which has been already evaluated, as new query. The system recomputes the whole inference to generate the answers which are already tabled for the main query. This is not the case with JLOG where minimal inference work is required from the PROLOG engine to filter the answers of the main query that can be unified with the subquery. The timing differences for evaluating the subqueries for the first time between the two systems clearly goes in favor of JLOG. When the subqueries are re-evaluated, both systems are giving almost the same timing.

5.4.7 Maintaining the Soundness and Completeness of the Cached Answers

We measure the performance of maintaining the soundness and completeness of the tabled answers by asserting and retracting facts/rules that affect the tabled answers. Our findings are based on handling the assert/retract commands (events) live (one by one) as they arrive in JLOG and update the JTMS network accordingly. The same events are treated on the Xsb side as batch events since it is more efficient than the live method.

1. Retracting/Asserting old facts

   This case is related to maintaing the soundness of the cached answers. JLOG treats the assertion/retraction of facts faster than XSB since it does not require inference work from the PROLOG engine. JLOG propagates the effect of this change throughout the JTMS network in a reasonable time. However, it executes the assertion/retraction command on the PROLOG engine for future use and this adds an overhead to the response time of JLOG. XSB on the other side deals with this case as a two-step process. First, it needs to look at the dependency graph to know which answers should be removed from the table space or/and the required computation (inference work) needed to generate the new answers and add them to the table space.

2. Asserting new facts

   This case is related to maintaing the completeness of the cached answers. Both systems have to compute the new answers for the query that might be generated due to the assertion of the new facts related to the query answers. XSB handles this situation better than JLOG since it needs to compute (XSB is implemented at low level) the new answers and add them to the table space. JLOG deals with the case as a three-step process. First, it builds the proper subquery that generates the new answers for the query. Then, it requests the PROLOG engine to execute the subquery and return the desired answers(justifications). Finally, JLOG has to install the new justifications in the JTMS network attached to the query.

3. Retracting/Asserting old rules

   Another contribution of the thesis. This case is related to maintaing the soundness of the cached answers. In order to support assertion/retraction of rules on a tabled predicate, the tabled predicate must be defined as incrementally dynamic in XSB. If the predicate is defined as dynamic, then XSb is behaving strangely. The time it takes to evaluate the query for the first time is almost the same when the query is re-evaluated. This means that the incremental tabulation in XSB is not supported on a top level predicate which is both tabled and dynamic. JLOG deals with the case of retracting/asserting of old rules by marking the label of the rule from *IN* to *OUT* or vice versa, then propagating the effect of this change throughout the JTMS network.

4. Asserting new rules

   This case is related to maintaing the completeness of the cached answers. XSB is also suffering here with the same problem it suffers when asserting/retracing the old rules. JLOG deals with the case as a four-step process. First, it transforms the rule into a format which allows JLOG to get the answers generated from this rule as justifications, see Chapter 5 for more details. The next step is to build the proper subquery that generates the new answers for the query. Then, JLOG requests the PROLOG engine to execute the subquery and returns the desired answers(justifications). Finally, JLOG has to install the new justifications in the JTMS network attached to the query. The time it takes JLOG to deal with these four steps depends mainly on the time needed by PROLOG engine to execute the query and the time needed by JLOG to install the justifications which depends on the number of new answers.

Chapter 6

Summary, Contribution and Future Work

In this chapter we present the thesis summary, contribution, current limitations, future work and conclusion.

6.1 Thesis Summary

This thesis proposed JLOG which is a subsystem integrated with PROLOG inference engine. Currently the system can be integrated with the following PROLOG dialects:

1. SWI-PROLOG
2. YAP-PROLOG
3. XSB

JLOG supports the incremental tabulation for PROLOG query evaluation effectively under the non-monotonic logic. JLOG evaluates a query only once, maintaining enough information to ensure both consistency and completeness of the collected solution as the dynamic state changes. The main idea of JLOG is to cache the proof generated by the PROLOG inference engine rather than saving the end results as it is the case for most tabling systems.

In order to be able to efficiently store and maintain the up-to-date proof structure, JLOG uses two components, namely, a justification-based truth-maintenance (JTMS) system and a database monitor (forward-driven rule-based system).

The basic idea of JLOG is that, as the PROLOG inference engine evaluates the query for the first time, the answers (justifications) returned by the engine for the query is converted into a JTMS network. JLOG translates every successful branch into a TMS network that links the facts used in the proof branch to the answer generated by that branch.

When the same query is re-evaluated, the query engine of JLOG collects the valid answers for the query at that moment and returns them by using the cached proof structure for the query. When the state of database is changed, TMS propagates the effect of the change through its network to maintain the consistency of old proofs. At the same time, the database monitor keeps watching the addition of the new data to the system and triggers the resumption of previously proven queries in order to update their proof structure. In other words, the database monitor system ensures completeness of previously evaluated queries.

JLOG is suitable when a query that depends on dynamic information is to be evaluated repeatedly as the dynamic state changes, which is the case with dynamic databases. The detailed analysis of JLOG performance is presented in this thesis. We presented the performance in terms of query evaluation timings comparing to regular, tabled and incremental tabled Prolog implementations. Our benchmark to test the JLOG performance is the XSB system since it is the only PROLOG implementation so far that supports incremental tabulation. We have shown the advantages and disadvantages of the incremental tabulation approach used in JLOG comparing to the approach used by XSB.

6.2 Thesis Contributions

The outcome of this thesis is JLOG. JLOG presents a novel approach to support incremental tabulation that is capable of working in non-monotonic situations. JLOG caches the proof structure that the deductive PROLOG used to find results rather than caching the final results. JLOG then uses data driven techniques that are normally used in truth-maintenance and rule-based systems to ensure that the cached proof structures remain correct regardless of any changes to the database of facts and rules (program clauses).

The key idea that differentiates our approach from the approach used in XSB is that JLOG caches the proof structure rather than the end results of the query. The advantages of our approach can be categorized into the following:

1. Caching the query answers in one consolidate subsystem (JTMS Network). JLOG does not require extra data structure(s) to keep the cached proof structure sound and complete.

2. The cost of maintaing the soundness.

3. Evaluation of subqueries.

4. Handling the assertion/retraction of rules. The idea of caching the query proof structure rather than the end result is not a trivial one to implement (See Chapters 4 and 5 for more details). This approach requires a deep understanding of how PROLOG inference engine works and how to extract the query proof from the engine. The implementation is based on applying the source level transformations to a tabled program. Theoretically any PROLOG implementation which offers external primitives to call PROLOG commands can be integrated with JLOG. Practically, we have tested the system integration with three PROLOG dialects.

6.3 Limitations of the Current Approach

This section describes the limitations and performance issues of the current version of JLOG.

6.3.1 Inability to Handle Queries with Infinite Answers

JLOG needs to keep track of investigated branches of the proof structure for a previously proven query. This is achieved in JLOG by retrieving all answers for the query at once when it is evaluated for the first time. The drawback to this approach is that JLOG can not handle queries with infinite answers.
Consider for example, the following PROLOG program:

$natural(0).$

$natural(s(X)) : -natural(X).$

There are two kind of queries related to the above program.

1. Finite queries.

   The number of answers for this kind of queries is finite. For example the query $? - natural(s(s(0)))$ has one answer (yes). JLOG is able to handle this query because the number of answers for the query is finite.

2. Infinite queries.

   These queries generate infinite number of answers. For example the query $? - natural(X)$ has infinite number of answers. The standard PROLOG will return the answers for the query one by one until the caller decides to quit. JLOG fails to handle this query since it tries to find all the answers for the query which are unlimited.

6.3.2 Caching the Proof Structure

When a certain ground expression, a fact or rule, that is participated as an antecedent/-consequence in any justification is currently been retracted from the database, JLOG keep caching the justifications related to retracted ground expression assuming that it might be asserted back. If this ground expression will never be asserted to the system, then the system is wasting resources to cache useless branches.

6.3.3 The Overhead of Evaluating the Query for the First Time

When proving the query for the first time, JLOG is paying sufficient overhead since it caches the proof structure of the query rather than the end results. JLOG is not a good choice for the queries generates a large number of answers. The large number of answers for a query requires large JTMS network to be installed for the query in order to cache the proof structure. We need to study carefully the memory usage of JLOG and see how this issue can be resolved by controlling the memory management for the JTMS network.

6.4 Directions for Future Work

The aim of this section is to give directions for future work that could be done based on this thesis work.

6.4.1 Dirty Tag

As shown in the previous section, JLOG keeps the cached proof structure of the query up-to-date (when the database of facts/rules is changed) without paying any attention whether the query is re-evaluated or not. An alternative approach is that instead of keeping the proof structure up-to-date, whenever the state of database is changed, the system marks the query as **dirty**. When the query is re-evaluated next time, and it is marked as **dirty**, the system updates that proof structure for this query before returning the results. The dirty

tag approach requires extra book keeping in order to handle the situation. This approach is suitable for those systems where query re-evaluation is not often.

6.4.2 Deleting Proof Structure Branches

When a certain ground expression, a fact or rule, that is participated as an antecedent/-consequence in any justification is currently been retracted from the database, JLOG keep caching the justifications related to retracted ground expression assuming that it might be asserted back. An alternative approach is possible by implementing a new PROLOG predicate $retract\_r/1$. The behavior of $retract\_r/1$ is similar to the normal PROLOG predicate $retract/1$. When JLOG user retracts a fact or a rule using $retract\_r/1$, the system deletes all branches (justifications) of the proof structure where the retracted fact/rule is participating as an antecedent or as a consequence (fact only). The $retract\_r/1$ can be used when the user is assured that the retracted ground expression will never be asserted back.

6.4.3 Compacting the Size of The JTMS Network

When proving the query for the first time, JLOG is paying sufficient overhead since it caches the proof structure of the query rather than the end results. The main overhead is coming from the time it takes to build the JTMS network for the query results (justifications). Ideas to reduce the overhead of building the JTMS network are listed below.

1. The current version of JLOG coverts each successful branch of PROLOG proof into a justification in the JTMS network. It is possible that many justifications support the same consequence (redundant answers). This approach causes sufficient overhead especially when the number of successful branches in the proof is large. To reduce the size of the JTMS network, the system installs only one justification for the consequence. When this justification becomes inactive, JLOG has to request from the PROLOG inference engine to find another justification that might support the consequence of the current inactive justification. Using this approach, the number of

justification that should be installed, when the query is proved for the first time, is equivalent to the number of unique answers for the query. A typical example that can benefit from this approach is the reachability problem(see Section 5.1.1.2).

2. When constructing the JTMS network for the query proof, JLOG is using the textual representation of the PROLOG predicates (facts/rules) that are participating as antecedents and consequence of the justification. Another possible approach is to assign each PROLOG predicate an ID, those IDs are used to build the JTMS network.

6.4.4 Implementing JLOG by Modifying and Extending the Low Level Engine

We would like to incorporate the idea of incremental tabulation presented in this thesis into one of the existing PROLOG implementations (e.g. YAP-PROLOG, SWI-PROLOG). This can be done by modifying and extending the low level engine in those systems. This is the common approach used by most of systems to support tabled evaluation.

6.5 Conclusion

The research opens the door for logic based systems that depend on dynamic facts and rules to benefit in their performance from the idea of incremental evaluation of tabled PROLOG programs. More precisely JLOG favors the dynamic rules based logic systems. Below we summarize our findings about how effective and efficient JLOG is in handling the evaluation of PROLOG queries under the non-monotonic systems:

1. Evaluating the query for the first time

   The time it takes to execute a query for the first time on JLOG depends on the nature of the query. A query that generates reasonable number of answers is the ideal one for JLOG. A query that generates large number of answers suffers from overhead due to the time spent to cache the proof structure.

2. Re-evaluating the query

   When the query is re-evaluated, JLOG collects the valid answers from the query's cached proof structure and returns them. The speed query re-evaluation in JLOG is almost similar to other PROLOG implementations that support tabulation.

3. Evaluating a sub-query

   If a subquery, related to a previously proven query, is evaluated for the first time, JLOG filters the answers of the parent query to return the answers of the sub-query. The filtration is done from the parent query's JTMS network without referring to PROLOG inference engine.

4. Maintaing Soundness of the cached query answers

   JLOG is able to maintain the soundness of the cached proof structure through propagating the Inness/Outness of PROLOG facts and rules in the Jtms network associated with the query. This task does not require any work to be done on the PROLOG inference engine side.

5. Maintaing completeness of the cached query answers.

   Whenever a new fact/rule is asserted into the database, the system checks to see if the new fact/rule can contribute to a previously proven query. If yes, then the system resumes the query to update its cached proof structure with respect to the new data. A special form of the query is resumed rather than evaluating the general form the query. This work requires the help of PROLOG inference engine in order to complete the cached proof structure.

Bibliography

[Avoine et al., 2008] Avoine, G., Junod, P., & Oechslin, P. (2008). Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4).

[Boella & van der Torre, 2005] Boella, G. & van der Torre, L. W. N. (2005). A non-monotonic logic for specifying and querying preferences. In L. P. Kaelbling & A. Saffiotti (Eds.), *IJCAI* (pp. 1549–1550).: Professional Book Center.

[Brass, 1995] Brass, S. (1995). Magic sets vs. sld-resolution. In J. Eder & L. A. Kalinichenko (Eds.), *ADBIS*, Workshops in Computing (pp. 185–203).: Springer.

[Bratko, 2000] Bratko, I. (2000). *Prolog Programming for Artificial Intelligence*. Harlow, England: Pearson Addison-Wesley, 3 edition.

[Brewka et al., 1991] Brewka, G., Makinson, D., & Schlechta, K. (1991). Jtms and logic programming. In *LPNMR* (pp. 199–210).

[Calejo, 2004] Calejo, M. (2004). Interprolog: Towards a declarative embedding of logic programming in java. In J. J. Alferes & J. A. Leite (Eds.), *JELIA*, volume 3229 of *Lecture Notes in Computer Science* (pp. 714–717).: Springer.

[Carlsson & Mildner, 2010] Carlsson, M. & Mildner, P. (2010). Sicstus prolog – the first 25 years. *CoRR*, abs/1011.5640.

[Chen & Warren, 1996] Chen, W. & Warren, D. S. (1996). Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1), 20–74.

[Clocksin & Mellish, 1984] Clocksin, W. F. & Mellish, C. S. (1984). *Programming in Prolog (2nd ed.)*. New York, NY, USA: Springer-Verlag New York, Inc.

[Costa et al., 2012] Costa, V. S., Rocha, R., & Damas, L. (2012). The yap prolog system. *TPLP*, 12(1-2), 5–34.

[Covington, 1994] Covington, M. A. (1994). *Natural Language Processing for Prolog Programmers*. Englewood Cliffs, NJ: Prentice-Hall.

[Diaz et al., 2012] Diaz, D., Abreu, S., & Codognet, P. (2012). On the implementation of gnu prolog. *TPLP*, 12(1-2), 253–282.

[Doyle, 1979] Doyle, J. (1979). A truth maintenance system. *Artif. Intell.*, 12(3), 231–272.

[Dung, 1996] Dung, T. Q. (1996). A revision of dependency-directed backtracking for jtms. In G. Görz & S. Hölldobler (Eds.), *KI*, volume 1137 of *Lecture Notes in Computer Science* (pp. 57–60).: Springer.

[Fan & Dietrich, 1992] Fan, C. & Dietrich, S. W. (1992). Extension table built-ins for prolog. *Softw. Pract. Exper.*, 22(7), 573–597.

[Fickas, 1985] Fickas, S. (1985). Design issues in a rule-based system. *SIGPLAN Not.*, 20(7), 208–215.

[Forbus & de Kleer, 1993] Forbus, K. D. & de Kleer, J. (1993). *Building problem solvers*. Cambridge, MA, USA: MIT Press.

[Gallier, 1985] Gallier, J. H. (1985). *Logic for computer science: foundations of automatic theorem proving*. New York, NY, USA: Harper & Row Publishers, Inc.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

[Grabova et al., 2010] Grabova, O., Darmont, J., Chauchat, J.-H., & Zolotaryova, I. (2010). Business intelligence for small and middle-sized enterprises. *SIGMOD Record*, 39(2), 39–50.

[Hermenegildo et al., 2011] Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., & Puebla, G. (2011). An overview of ciao and its design philosophy. *CoRR*, abs/1102.5497.

[Lee et al., 2001] Lee, K. Y., Son, J. H., & Kim, M. H. (2001). Efficient incremental view maintenance in data warehouses. In *Proceedings of the tenth international conference on Information and knowledge management*, CIKM '01 (pp. 349–356). New York, NY, USA: ACM.

[Lloyd, 1987] Lloyd, J. (1987). *Foundations of Logic Programming (2nd Extended Edition)*. Springer-Verlag.

[Lloyd, 1994] Lloyd, J. W. (1994). Practical advantages of declarative programming. In *Joint Conference on Declarative Programming*.

[McDermott, 1982] McDermott, D. (1982). Nonmonotonic logic ii: Nonmonotonic modal theories. *J. ACM*, 29(1), 33–57.

[Mohania et al., 1997] Mohania, M., Konomi, S., & Kambayashi, Y. (1997). Incremental maintenance of materialized views. In A. Hameurlain & A. Tjoa (Eds.), *Database and Expert Systems Applications*, volume 1308 of *Lecture Notes in Computer Science* (pp. 551–560). Springer Berlin / Heidelberg.

[Nilsson & Maluszynski, 1995] Nilsson, U. & Maluszynski, J. (1995). *Logic, Programming, and PROLOG*. New York, NY, USA: John Wiley & Sons, Inc., 2nd edition.

[O'Keefe, 1990] O'Keefe, R. A. (1990). *The Craft of Prolog*. MIT Press.

[Perea, 2010] Perea, A. (2010). Backward induction versus forward induction reasoning. *Games*, 1(3), 168–188.

[Rao et al., 1997] Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., & Freire, J. (1997). XSB: A System for Effciently Computing WFS. In Dix (Ed.), *Logic Programming and Non-monotonic Reasoning*, volume 1265 (pp. 431–441).: Springer.

[Rocha et al., 2007a] Rocha, R., Silva, C., & Lopes, R. (2007a). Implementation of Suspension-Based Tabling in Prolog using External Primitives. In J. Neves, M. Santos, & J. Machado (Eds.), *Local Proceedings of the 13th Portuguese Conference on Artificial Intelligence, EPIA'2007* (pp. 11–22). Guimarães, Portugal.

[Rocha et al., 2007b] Rocha, R., Silva, C., & Lopes, R. (2007b). On applying program transformation to implement suspension-based tabling in prolog. In *ICLP* (pp. 444–445).

[Rocha et al., 2000] Rocha, R., Silva, F., Fern, R. R., & Costa, V. S. (2000). Yaptab: A tabling engine designed to support parallelism.

[Sagonas et al., 1993a] Sagonas, K., Swift, T., & Warren, D. S. (1993a). Xsb: An overview of its use and implementation. *SUNY Stony Brook*, (pp. 11794–4400).

[Sagonas et al., 1994] Sagonas, K., Swift, T., & Warren, D. S. (1994). Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2), 442–453.

[Sagonas et al., 1993b] Sagonas, K. F., Swift, T., & Warren, D. S. (1993b). The xsb programming system. In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS* (pp. 164).

[Saha, 2006] Saha, D. (2006). *Incremental evaluation of tabled logic programs*. PhD thesis, Stony Brook, NY, USA. AAI3258884.

[Saha & Ramakrishnan, 2006] Saha, D. & Ramakrishnan, C. (2006). Incremental evaluation of tabled prolog: Beyond pure logic programs. In P. Van Hentenryck (Ed.), *Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science* (pp. 215–229). Springer Berlin / Heidelberg.

[Saha & Ramakrishnan, 2003] Saha, D. & Ramakrishnan, C. R. (2003). Incremental evaluation of tabled logic programs. In *ICLP* (pp. 392–406).

[Saha & Ramakrishnan, 2005] Saha, D. & Ramakrishnan, C. R. (2005). Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05 (pp. 117–128). New York, NY, USA: ACM.

[Shapiro, 1998] Shapiro, S. C. (1998). *Belief Revision and Truth Maintenance Systems: An Overview and a Proposal*. Technical report.

[Smullyan, 1995] Smullyan, R. (1995). *First-order logic*. Dover Publications.

[Somogyi & Sagonas, 2006] Somogyi, Z. & Sagonas, K. F. (2006). Tabling in mercury: Design and implementation. In P. V. Hentenryck (Ed.), *PADL*, volume 3819 of *Lecture Notes in Computer Science* (pp. 150–167).: Springer.

[Swift, 2000] Swift, T. (2000). Principles, practice, and applications of tabled logic programming. *SIGSOFT Softw. Eng. Notes*, 25(1), 87–88.

[Swift & Warren, 1993] Swift, T. & Warren, D. S. (1993). A Summary of XSB Performance.

[Swift & Warren, 2012] Swift, T. & Warren, D. S. (2012). Xsb: Extending prolog with tabled logic programming. *TPLP*, 12(1-2), 157–187.

[Tamaki & Sato, 1986] Tamaki, H. & Sato, T. (1986). Old resolution with tabulation. In *Proceedings on Third international conference on logic programming* (pp. 84–98). New York, NY, USA: Springer-Verlag New York, Inc.

[Warren, 1983] Warren, D. H. D. (1983). *An Abstract Prolog Instruction Set*. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

[Warren, 1992] Warren, D. S. (1992). Memoing for logic programs. *Commun. ACM*, 35(3), 93–111.

[Wielemaker & Costa, 2010] Wielemaker, J. & Costa, V. S. (2010). Portability of prolog programs: theory and case-studies. *CoRR*, abs/1009.3796.

[Wielemaker et al., 2012] Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2), 67–96.

[Zhou, 2012] Zhou, N.-f. (2012). The language features and architecture of b-prolog. *Theory Pract. Log. Program.*, 12(1-2), 189–218.

[Zhou et al., 1996] Zhou, N.-F., Nagasawa, I., Umeda, M., Katamine, K., & Hirota, T. (1996). B-prolog: A high performance prolog compiler. In T. Tanaka, S. Ohsuga, & M. Ali (Eds.), *IEA/AIE* (pp. 790).: Gordon and Breach Science Publishers.

Appendix A

JLOG Class

```
package com.taher.jlog;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Scanner;
import com.taher.jlog.jtms.*;
public class Jlog {
  private static Jtms myJtms;
  private static long startTime;
  private static long endTime;
  public static PrintWriter out;

public static void main(String[] args) {
try{
// TODO Auto-generated method stub
    System.out.println("Welcome to jlog version 2.0");
myJtms = new Jtms("XSB");
Scanner f = new Scanner(System.in);
System.out.println("Enter folder name:");
String folderName = f.nextLine();
System.out.println("Enter consult file name:");
String fileName = f.nextLine();
File fName=new File(System.getProperty("user.dir")+
"/Tests"+"/"+folderName+"/"+fileName+".pl" );
FileWriter outFile = new FileWriter(System.getProperty("user.dir")+
"/Tests"+"/"+folderName+"/Results/"+fileName+".txt");
    PrintWriter out = new PrintWriter(outFile);
    Jlog.out=out;
System.out.println(Jtms.engine.getPrologVersion());
if (Jtms.engine.getPrologVersion().startsWith("XSB")) {
System.out.println("Import append/3");
Jtms.engine.deterministicGoal("import append/3 from basics");
}
```

```
long totalTime = 0;
long start = System.currentTimeMillis();
    if (!myJtms.consult(fName)) {
      System.exit(0);
    }
    long end = System.currentTimeMillis();
    totalTime = end - start;
    String message = "Consult time: "+totalTime+" ms.\n";
    System.out.println(message);
    fileName = f.nextLine();
    fName=new File(System.getProperty("user.dir")+
"/Tests"+"/"+folderName+"/"+fileName+".txt" );
FileInputStream fstream = new FileInputStream(fName.toString());
    DataInputStream in = new DataInputStream(fstream);
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String x;
   while ((x = br.readLine()) != null) {
   if (!x.startsWith("extnCommand") )
    x=x.replaceAll(" ","");
   if (x.compareTo("halt.")==0){
   out.println("Good Day....");
   out.close();
   break;
   }
     if (x.startsWith("assert") ) {
     x=x.replace(".", "");
      System.out.println(x);
      boolean t1 = Jtms.engine.deterministicGoal(x);
      if (t1) {
     x=x.replace("assert(", "");
     x=x+".";
     x=x.replace(").", "");
     myJtms.jAssert(x);
      } else
     System.out.println(x+" failed!");
      continue;
     }
     if (x.startsWith("retract") ) {
      x=x.replace(".", "");
      boolean t1 = Jtms.engine.deterministicGoal(x);
      if (t1) {
     x=x.replace("retract(", "");
     x=x+".";
     x=x.replace(").", "");
     myJtms.jRetract(x);
      } else
     System.out.println(x+" failed!");
      continue;
     }
     if (x.startsWith("extnCommand") ) {
          out.println("\njextnCommand:"+x);
      x=x.replace(".", "");
      x=x.replace("extnCommand(", "");
      x=x+".";
      x=x.replace(").", "");
      String msg =  myJtms.executeExternalPrologCommand(x);
```

```
      System.out.println(msg);
      out.write(msg);
      continue;
     }
     if (x.startsWith("extn") ) {
          out.println("\njextn:"+x);
      x=x.replace(".", "");
      x=x.replace("extn(", "");
      x=x+".";
      x=x.replace(").", "");
      String msg =  myJtms.executeExternalPrologQuery(x);
      System.out.println(msg);
      out.write(msg);
      continue;
     }
     if (x.startsWith("stat") ) {
      out.println("");
      out.println("external Nodes = "+ TmsNode.Ecount());
      out.println("internal Nodes = "+ TmsNode.Icount());
      out.println("Rule     Nodes = "+ TmsNode.Rcount());
      out.println("Total    Nodes = "
  + (TmsNode.Rcount()+TmsNode.Icount()+TmsNode.Ecount()));
      out.println(myJtms.jtmsStat());
      continue;
     }
     if (x.startsWith("printJust") ) {
      out.println(myJtms.printJustifications());
      continue;
     }
     if (x.startsWith("printJtms") ) {
      out.println(myJtms.toString());
      continue;
     }
     if (x.startsWith("startTimer") ) {
      startTime = System.currentTimeMillis();
      out.println("\n---------------------------------- ");
      out.println("Start Timer..... ");
      out.println("---------------------------------- ");
      continue;
     }
     if (x.startsWith("stopTimer") ) {
      endTime = System.currentTimeMillis();
      out.println("\n---------------------------------- ");
      out.println("Total time= "+ (endTime - startTime));
      out.println("---------------------------------- ");
      continue;
     }
     String msg= myJtms.query(x, false);
     out.write(msg);
    }
}
 catch (Exception e){
   System.err.println("Error: " + e.getMessage());
   System.out.println("\n*****************bye***************\n");
     }
   System.out.println("\n*****************bye***************\n");
```

```
}
}
```

JTMS Class

```
package com.taher.jlog.jtms;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.Vector;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import com.declarativa.interprolog.*;
import com.taher.jlog.jtms.tmsNodeLabel;
public class Jtms {
    private Tms myTms;
    public static XSBSubprocessEngine engine;
    //public static SWISubprocessEngine engine;
    //public static YAPSubprocessEngine engine;
    private Hashtable<String, justification> myJust;
    private static int  x= 0;
    public Jtms(String X) {
    engine = new XSBSubprocessEngine
("/Users/ch_taher/XSB/config/i386-apple-darwin11.4.0/bin/xsb");
    System.out.println("PROLOG engine is created...");
    myTms = new Tms();
    System.out.println("TMS="+myTms.toString());
    myJust = new Hashtable<String, justification>();
    System.out.println("Justifications="+myJust.toString());
}
    public void assertRule (String rule) {
      if (!rule.startsWith("("))
rule = "(" + rule+ ")";
     System.out.println ("Assert Rule: "+rule);
     TmsNode ruleNode = myTms.findNode(rule);
   if (ruleNode != null ) {
   ruleNode.setLabel(tmsNodeLabel.IN, this);
   } else {
   Tms.setRuleFlag(true);
   boolean flag=false;
   TermModel ruleTerm =
                              (TermModel)engine.deterministicGoal("buildTermModel
       ("+rule+", Model)","[Model]")[0];
   TermModel head = (TermModel) ruleTerm.getChild(0);
```

```
            String consequence = head.toString();
   TermModel body;
   body = (TermModel) ruleTerm.getChild(1);
   String goal = "nonDeterministicGoal(JLOG,("+ body.toString();
   String inList = "[";
   String outList = "[";
   do  {
   TermModel current;
   if ((body.getFunctorArity().startsWith(",")) ) {
   current = (TermModel) body.getChild(0);
   body = (TermModel) body.getChild(1);
   } else {
   current = body;
   flag = true;
            }
   Pattern p = Pattern.compile("([a-z])");
   Matcher m = p.matcher(current.getFunctorArity());
       if (!current.isVar() && m.find()) {
   if (current.toString().contains("not")) {
   System.out.println("Negitive Goal!");
   current = (TermModel) current.getChild(0);
            if (outList.length() >1 )
            outList += ",";
            outList +=current.toString();
   } else {
   if (inList.length() >1 )
               inList += ",";
             inList +=current.toString();
   }
   this.RelatedQueries(current, head);
             }
   } while (flag != true);
   inList += "],";
   outList += "],";
   String jLog =", append(['"+rule.toString()+"']
,["+inList+outList+consequence+"],JLOG)";
   goal += jLog +"),ListModel1)";
   ruleNode= myTms.addNode(this, ruleTerm, 'R',
tmsNodeLabel.IN, rule, goal, head);
      TmsNode ruleQuery = myTms.findNode(ruleNode.getGT());
   if (ruleQuery != null) {
     if (ruleQuery.getLabel() == tmsNodeLabel.F) {
   System.out.println("goal:"+ goal);
   this.executePrologRule(head, ruleNode);
     }
   }
   Tms.setRuleFlag(false);
 }
    }


    private void RelatedQueries(TermModel current, TermModel head) {
TermModel x = (TermModel)current.clone();
for (int i=0; i <x.getChildCount(); i ++) {
            x.setChild(i, new TermModel("JLOG"+i));
      }
```

```
TmsNode xT = myTms.addNode(this, x, 'I', tmsNodeLabel.P);
TermModel y = (TermModel)head.clone();
for (int i=0; i <y.getChildCount(); i ++) {
           y.setChild(i, new TermModel("JLOG"+i));
      }
TmsNode yT = myTms.addNode(this, y, 'I', tmsNodeLabel.P);
xT.addToRelatedQueries(yT);
}


public boolean consult(File  fName) {
       try {
       engine.consultAbsolute(fName);
       FileInputStream fstream = new FileInputStream(fName.toString());
  DataInputStream in = new DataInputStream(fstream);
  BufferedReader br = new BufferedReader(new InputStreamReader(in));
  String strLine;
  while ((strLine = br.readLine()) != null)   {
             System.out.println("Line:"+strLine);
    strLine = strLine.replace(".", " ");
         strLine = strLine.replaceAll(" ", "");
         if (strLine.isEmpty() )
           continue;
         if ( strLine.startsWith("%") )
           continue;
         if ( strLine.contains("nonDeterministicGoal") )
           continue;
         if ( strLine.startsWith(":-") )
           continue;
     if ( strLine.contains(":-") ) {
              this.assertRule(strLine);
              continue;
     }
     break;
       }
  br.close();
  in.close();
       } catch (Exception e){//Catch exception if any
        System.err.println("Error: " + e.getMessage());
        return false;
  }
       return true;
    }


    public void jRetract(String x) {
    TmsNode tmsNode = myTms.findNode(x);
if (tmsNode == null) {
  TermModel t1 = (TermModel)engine.deterministicGoal("buildTermModel
("+x+", Model)","[Model]")[0];
  tmsNode = myTms.addNode(this, t1, 'E', tmsNodeLabel.OUT);
}else {
  tmsNode.setLabel(tmsNodeLabel.OUT, this);
}
}
```

```
public void jAssert (String x) {
System.out.println("jAssert"+x);
if (x.contains(":-")) {
this.assertRule(x);
} else {
TermModel t1 =
(TermModel)engine.deterministicGoal
("buildTermModel("+x+", Model)","[Model]")[0];
TmsNode newNode = myTms.findNode(t1);
if (newNode == null) {
   newNode = myTms.addNode(this, t1, 'E', tmsNodeLabel.IN);
   newNode.propagateNewAtom(this);
   }else {
   newNode.setLabel(tmsNodeLabel.IN, this);
}
}
}

private void installJustifications (TermModel justification) {
    if (justification.isList()) {
TermModel []list = justification.flatList();
TermModel rule=list[0];
TermModel []inList=list[1].flatList();
TermModel []outList=list[2].flatList();
TermModel consequence=list[3];
if (!myJust.containsKey(justification.toString())) {
myJust.put(justification.toString(), new
justification(this,
justification, rule,inList,outList,
consequence));
}
    } else {
        myTms.addNode(this, justification, 'E', tmsNodeLabel.IN);
    }

}

public String executePrologRule(TermModel query, TmsNode rule) {
String msg="";
String body = rule.getBody();
TermModel head = (TermModel)rule.getNode().getChild(0);
for (int i=0; i<query.getChildCount();i++ ) {
      TermModel child = (TermModel)query.getChild(i);
      TermModel hChild = (TermModel)head.getChild(i);
      if ((child.isAtom() || child.isInteger()  )&& !
(child.toString().contains("JLOG")) &&
hChild.isVar()) {
     body=body.replaceAll(hChild.toString(),
child.toString());
      }
     }
     System.out.println("execute Prolog Query:"+body);
     msg+=this.executePrologQuery(body);
return msg;
}
```

```
public String executeExternalPrologCommand( final String goal) {
String msg="Prolog Command:"+ goal+"\n";
try {
boolean flag = engine.deterministicGoal(goal);
msg= Jtms.x++ +": command executed:"+ flag;
} catch (Exception e){//Catch exception if any
      System.out.println("Exception: "+e);
      return null;
}
return msg;
}

public String executeExternalPrologQuery( final String goal) {
String goal1 = "findall("+goal+","+goal+",L)";
String goal2 = "nonDeterministicGoal("+goal+"," +
goal +",ListModel1)";
String msg="Prolog Query:"+ goal1+"\n"+goal2+"\n";
long totalTime, end, start;
try {
start = System.currentTimeMillis();
boolean flag = engine.deterministicGoal(goal1);
end = System.currentTimeMillis();
   totalTime = end - start;
    msg+= flag+": Prolog Query excecution time: "+totalTime+" ms.\n";
    start = System.currentTimeMillis();
    TermModel solutionVars = (TermModel)
(engine.deterministicGoal(goal2,
"[ListModel1]")[0]);
end = System.currentTimeMillis();
   totalTime = end - start;
    if (solutionVars != null && solutionVars.getChildCount() > 0 ) {
  msg+="Number of sols: "+ solutionVars.getChildCount()+"\n";
  for (int i=0; i< solutionVars.getChildCount(); i++) {
  System.out.println(solutionVars.getChild(i));
  }
}
    totalTime = end - start;
    msg+= "Prolog Query excecution time with answers: "+totalTime+" ms.\n";
  } catch (Exception e){//Catch exception if any
      System.out.println("Exception: "+e);
      return null;
 }

return msg;
}

public String executePrologQuery(final String goal) {
String msg= "\n**********\nProlog Query:"+ goal+"\n";
long totalTime, end, start = System.currentTimeMillis();
TermModel solutionVars= null;
try {
   String tempGoal = goal.replaceFirst
("nonDeterministicGoal", "findall");
   start = System.currentTimeMillis();
   boolean flag =  engine.deterministicGoal(tempGoal);
   end = System.currentTimeMillis();
```

```
      totalTime = end - start;
       msg+=flag+": Prolog Query excecution time: "+totalTime+" ms.\n";
       System.out.println(msg);
       start = System.currentTimeMillis();
       System.out.println("start building termodel");
   solutionVars = (TermModel)
(engine.deterministicGoal(goal,
"[ListModel1]")[0]);
   end = System.currentTimeMillis();
      totalTime = end - start;
       msg+="Prolog Query excecution time: "+totalTime+" ms.\n";
       System.out.println("end build building term model");
       start = System.currentTimeMillis();
   if (solutionVars != null && solutionVars.getChildCount() > 0 ) {
  msg+="Number of sols: "+
solutionVars.getChildCount()+"\n";
  System.out.println("Number of sols: "
+ solutionVars.getChildCount());
  if (solutionVars.toString() != "[]")
   for (int i=0; i < solutionVars.getChildCount(); i++ ){
this.installJustifications
((TermModel)solutionVars.getChild(i));
   }
   }
  } catch (Exception e){//Catch exception if any
      System.out.println("Exception: "+e);
      return null;
 }
end = System.currentTimeMillis();
   totalTime = end - start;
        msg+="Jtms build time: "+totalTime+" ms.\n";
        boolean t1 = Jtms.engine.deterministicGoal("abolish_all_tables");
        msg+= "abolish_all_tables->>"+t1+"\n";
     return msg;
}

public Tms getTms() {
  return myTms;
}

public String proveQueryFirstTime(TmsNode tn, String x) {
String msg="";
msg += "Prove Query for first time!\n";
TmsNode gt = myTms.findNode(tn.getGrandParent());
Set<TmsNode> rules= gt.getRules();
  if (rules== null) {
    String goal = "nonDeterministicGoal("+x+"," +x +",ListModel1)";
     msg+=this.executePrologQuery(goal);
   } else {
   Iterator<TmsNode> r = rules.iterator();
 while (r.hasNext()) {
   TmsNode m = (TmsNode) r.next();
   if (m.getLabel()== tmsNodeLabel.IN)
    msg+=this.executePrologRule(tn.getNode(), m);
 }
  }
```

```
return msg;
}

public boolean isAtom(TermModel x) {
  for (int i=0; i< x.getChildCount(); i++) {
TermModel y = (TermModel) x.getChild(i);
if (y.isVar() || y.toString().contains("JLOG"))
 return false;
  }
  return true;
}

public String atomGoal (TermModel query) {
String msg ="";
TmsNode queryNode = myTms.findNode(query);
   if (queryNode == null) {
      queryNode = myTms.addNode(this,query,'E', tmsNodeLabel.IN);
     boolean answer = Jtms.engine.deterministicGoal(queryNode.getGT());
  if (!answer)
 queryNode.setLabel(tmsNodeLabel.OUT, this);
   }
    myTms.printSols(queryNode, null, this);
   return msg;
}

public String query( String x, boolean force) {
Vector<TmsNode> prints =null;
String msg="Query: "+ x+"\n";
x = x.replace(".", " ").replaceAll(" ", "");
   long totalTime, end, start = System.currentTimeMillis();
   TermModel query = (TermModel)engine.
deterministicGoal("buildTermModel("+x+",
Model)","[Model]")[0];
   if (this.isAtom(query)) {
     msg+= this.atomGoal(query);
     return msg;
}
   TmsNode queryNode = myTms.addNode(this, query, 'I', tmsNodeLabel.P);
   if ((queryNode.getLabel() == tmsNodeLabel.F ||
queryNode.getType() =='E' ) && !force ) {
   msg += " Query already proved!\n";
   System.out.println("Query already proved!");
   prints= myTms.printSols(queryNode, null, this);
   } else {
   TmsNode parentNode = myTms.addNode(this,
queryNode.getGrandParent(), 'I', tmsNodeLabel.P);
   TmsNode childNode = parentNode.findChildQuery(queryNode);
   if (!force && childNode != null   )  {
  msg += " Parent Query already proved!\n";
     System.out.println("Parent Query already proved!");
     prints= myTms.printSols(queryNode, childNode,this);
   } else {
       System.out.println("Prove Query for first time or by force!");
       if (parentNode != queryNode)
        parentNode.addToChildQueries(queryNode);
      msg += " Prove Query for first time or by force!\n";
```

```
   msg+=this.proveQueryFirstTime(queryNode, x);
   end = System.currentTimeMillis();
   totalTime = (end - start);
        msg += " Total Query time: "+totalTime+" ms.\n";
        start = System.currentTimeMillis();
        prints= myTms.printSols(queryNode, null,this);
   }
   }
   end = System.currentTimeMillis();
   totalTime = (end - start);
        msg += "JTMS get results time: "+totalTime+" ms.\n";
        start = System.currentTimeMillis();
        if (prints != null) {
            msg += "total query answers: "+prints.size()+"\n";
            int count =0;
    Iterator<TmsNode> it = prints.iterator();
        while (it.hasNext()) {
         TmsNode element = it.next();
    System.out.println(++count +": "
+ element.getNode().toString());
        }
        }
        end = System.currentTimeMillis();
        totalTime = (end - start);
        msg += "JTMS show results time: "+totalTime+" ms.\n";
        msg += "_____________________________________\n";
        return msg;

}

public String toString() {
return myTms.toString();
}

public String jtmsStat() {
String msg = "";
msg += myTms.tmsStat();
msg += "Justifications: "+ myJust.size();
return msg;
}

public String printJustifications() {
String msg ="";
Iterator<Entry<String, justification>>  it;
Map.Entry<String, justification>            entry;
it = myJust.entrySet().iterator();
while (it.hasNext()) {
    entry = it.next();
    msg+=  entry.getKey().toString()+"\n" ;
}
return msg;
}
}
```

TMS Class

```
package com.taher.jlog.jtms;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;
import java.util.Vector;
import com.declarativa.interprolog.*;
import java.util.Deque;
public class  Tms{
 private Hashtable<String, TmsNode> hashTable;
 private Hashtable<TmsNode, Set<TmsNode>> adj_hash;
 private static Boolean ruleFlag = false;

public Tms() {
  // TODO Auto-generated constructor stub
  hashTable = new Hashtable<String, TmsNode>();
  adj_hash = new Hashtable<TmsNode,Set<TmsNode>>();
}

public interface DequeAdder {
    void add(TmsNode vertex, Deque<TmsNode> deque);
}

public Vector<TmsNode> printSols(TmsNode query, TmsNode parent,
 Jtms myJtms) {
Vector<TmsNode> vc=new Vector<TmsNode>();
if (query.getType() == 'E')  {
         if (query.getLabel() == tmsNodeLabel.IN)
          System.out.println("true");
         else
          System.out.println("false");
} else {
if (query.getLabel()== tmsNodeLabel.P)
query.setLabel(tmsNodeLabel.F, myJtms);
if ((parent!= null) && (query.getGT() != parent.getGT())) {
Set<TmsNode> prints = parent.getGroundItems();
if (prints != null) {
Iterator<TmsNode> it = prints.iterator();
        while (it.hasNext()) {
       TmsNode element = it.next();
       if (element.getLabel()== tmsNodeLabel.IN) {

if(element.getNode().unifies
(query.getNode())) {
      vc.add(element);
      query.addToGroundItems(element);
     }
       }
    }
        }
```

```
} else {
Set<TmsNode> prints= query.getGroundItems();
if (prints != null) {
   Iterator<TmsNode> it = prints.iterator();
       while (it.hasNext()) {
      TmsNode element = it.next();
      if (element.getLabel()== tmsNodeLabel.IN) {
     vc.add(element);
       }
    }
        }
}
}
return vc;
    }

public TmsNode findNode (String x ) {
if (this.hashTable.containsKey(x)) {
   return this.hashTable.get(x);
}
else {
   return null;
}
}

public TmsNode findNode (TermModel node ) {
String x = TmsNode.getGT(node);
if (this.hashTable.containsKey(x)) {
  return this.hashTable.get(x);
}
else {
  return null;
}
}

private void addVertex(TmsNode t, String key ) {
if ( key == null )
this.hashTable.put(t.getGT(), t);
else
this.hashTable.put(key, t);
this.adj_hash.put(t, new HashSet<TmsNode>());
}

private void addEdge(TmsNode v1, TmsNode v2) {
        v1.addToGroundItems(v2); //adj_hash.get(v1).add(v2);

    }

    private TmsNode addNodeGenParent(Jtms myJtms, TermModel t, TmsNode n) {
          TermModel x = (TermModel)t.clone();
 for (int i=0; i <x.getChildCount(); i ++) {
            x.setChild(i, new TermModel("JLOG"+i));
        }
 TmsNode y = this.findNode(x);
        if (y == null ) {
           y= this.addNode(myJtms, x, 'I', tmsNodeLabel.P);
```

```
        }
        if (n.getType() =='R')
           y.addRule(n);
        else
          y.addChildQuery(n);
        return y;
    }

    private void addRuleToTmsNetwrok(Jtms myJtms, TmsNode R, TermModel head) {
          TmsNode g = this.addNodeGenParent(myJtms, head,R);
          R.setGT(g.getGT());
    }

    TmsNode addNode(Jtms myJtms, TermModel node, char nodeType, tmsNodeLabel l,
     String rule, String body, TermModel head) {
TmsNode t1=new TmsNode(node,l, nodeType);
     this.addVertex(t1, rule);
     this.addRuleToTmsNetwrok(myJtms,t1, head);
     t1.setBody(body);
     return t1;
}

    TmsNode addNode (Jtms myJtms, TermModel node, char nodeType, tmsNodeLabel l) {
TmsNode t1=null;
t1 = this.findNode(node);
if (t1 == null ) {
    t1 = new TmsNode(node,l, nodeType);
    this.addVertex(t1, null);
    switch (nodeType) {
    case 'E':
             TmsNode t2 = this.addNode(myJtms, t1.getGrandParent(),
'I', tmsNodeLabel.P);
             this.addEdge(t2, t1);
     Set<TmsNode> chQ = t2.getChildQueries();
             if (chQ != null) {
                  Iterator<TmsNode> r = chQ.iterator();
     while (r.hasNext()) {
       TmsNode m = (TmsNode) r.next();
       if (t1.getNode().unifies(m.getNode())) {
      m.addToGroundItems(t1);
       }
      }
         }
             if (!ruleFlag) {
               t1.propagateNewAtom(myJtms);
             }
         break;

    case 'I':
             t1.setLabel(tmsNodeLabel.P, myJtms);
             break;
  }
} else {
  if (t1.getType() == 'E')
            t1.setLabel(l, myJtms);
}
```

```
return t1;
}

public String toString() {
        StringBuffer buffer = new StringBuffer();
        int count =1;
        Enumeration<TmsNode> iterator = hashTable.elements();
        while(iterator.hasMoreElements()) {
             TmsNode temp = (TmsNode)iterator.nextElement();
             buffer.append(count++ +": ");
             buffer.append(temp.toString());
             Set<TmsNode> adjList = adj_hash.get(temp);
            if ( adjList == null )
               continue;
            Iterator<TmsNode> listIterator = adjList.iterator();
             while ( listIterator.hasNext() ) {
               TmsNode adjVertex = (TmsNode) listIterator.next();
               buffer.append(" -> " + adjVertex.toString());
             }
       }
        return buffer.toString();
    }

public String tmsStat() {
String msg = "";
msg += "HashTable: "+ hashTable.size()+"\n";
return msg;
}

static public void setRuleFlag (boolean x) {
ruleFlag = x;
}
}
```

TMS Node Class

```
package com.taher.jlog.jtms;
import java.util.HashSet;
import java.util.Set;
import java.util.Iterator;
import com.declarativa.interprolog.*;
public class TmsNode {
private static int Icount=0;
private static int Ecount=0;
    private static int Rcount=0;
    private String  genTemplate;
private tmsNodeLabel label;
private char type;
private TermModel node;
private Set<TmsNode> groundItems=null;
private Set<TmsNode> childQueries=null;
private Set<justification> support= null;
```

```
private Set<justification> inList=null;
    private Set<justification> outList=null;
    private Set<TmsNode> relatedQueries=null;
    private Set<TmsNode> rules = null;
    private String body = null;

    public TmsNode( TermModel s,  tmsNodeLabel l, char t) {
this.node = s;
    this.genTemplate= getGT(s);
    this.label =l;
    this.type = t;
     switch (t) {
      case  'E' : Ecount++;
      case  'R' : Rcount++;
      case  'I' : Icount++;
     }
}

public static String getGT(TermModel s) {
String x = s.getTemplate();
int count =0;
for (int i=0; i < s.getChildCount(); i++ ) {
  TermModel temp = (TermModel) s.getChild(i);
  String y = temp.toString();
  if (temp.isVar() || y.startsWith("'JLOG")) {
  x=x.replaceFirst("_", ("JLOG"+count++));
  } else {
  x=x.replaceFirst("_", temp.toString());
  }
}
return x;
}

public String getGT() {
return this.genTemplate;
}

public TermModel getNode() {
return this.node;
}

public void setGT(String s) {
this.genTemplate=s;
}

public void addRule (TmsNode r) {
if (this.rules == null) {
 this.rules = new HashSet<TmsNode>();
}
    this.rules.add(r);
}

public void addChildQuery (TmsNode c) {
if (this.relatedQueries == null) {
 this.relatedQueries = new HashSet<TmsNode>();
}
```

```
    this.relatedQueries.add(c);
}

public TermModel getGrandParent() {
TermModel genT = (TermModel) this.node.clone();
for (int i=0; i < genT.getChildCount();i++) {
TermModel temp = new TermModel("JLOG"+i);
genT.setChild(i, temp);
}
return genT;
}

public char getType() {
return this.type;
}

public String getBody() {
return this.body;
}

public void setBody(String b) {
this.body=b;
}

public tmsNodeLabel getLabel() {
return this.label;
}

public void setLabel(tmsNodeLabel x, Jtms myJtms) {
  if (this.label != x) {
boolean active = false;
if (x == tmsNodeLabel.OUT && this.type == 'E' )
  active = this.findAnotherSupport();
if (!active)  {
  this.label = x;
  if (this.type == 'E' || this.type == 'R') {
this.propagateInessOutness(myJtms);
  }
}
  }
}

private void propagateInessOutness(Jtms myJtms) {
if (this.inList == null && this.outList == null) {
  this.propagateNewAtom(myJtms);
}
if (this.inList != null) {
Iterator<justification> in = this.inList.iterator();
while (in.hasNext()) {
justification m = (justification) in.next();
if (this.label == tmsNodeLabel.IN)
  m.setActive(true, myJtms);
else
  m.setActive(false, myJtms);
}
}
```

```
if (this.outList != null) {
Iterator<justification> out = this.outList.iterator();
while (out.hasNext()) {
justification m = (justification) out.next();
if (this.label == tmsNodeLabel.IN)
  m.setActive(false,myJtms);
else
  m.setActive(true,myJtms);
}
}
}

private boolean findAnotherSupport() {
if (this.support != null) {
Iterator<justification> s = this.support.iterator();
while (s.hasNext()) {
  justification m = (justification) s.next();
  if (m.getActive() == true)
return true;
}
}
return false;
}

public Set<TmsNode> getRules () {
return this.rules;
}

private void executePartialQuery(Jtms myJtms, TmsNode rule) {
String newFact = this.getNode().toString();
String factName = newFact.substring(0,newFact.indexOf('('));
TermModel rulePart = (TermModel)rule.getNode().getChild(1);
TermModel factPart = this.getNode();
if (rulePart.toString().contains(factName)) {
Set<TermModel> ruleParts = new HashSet<TermModel>();
while (rulePart.toString().contains(factName)) {
if ((rulePart.getFunctorArity().contains(factName)) ) {
ruleParts.add(rulePart);
break;
}
if (rulePart.getChild(0).toString().contains(factName)) {
ruleParts.add((TermModel)rulePart.getChild(0));
}
if (rulePart.getChildCount() > 1)
   rulePart = (TermModel)rulePart.getChild(1);
else
   break;
}
Iterator<TermModel> r = ruleParts.iterator();
while (r.hasNext()) {
  rulePart= r.next();
  String body = rule.getBody();
  for (int i=0; i< factPart.getChildCount();i++ ) {
      TermModel child = (TermModel)factPart.getChild(i);
      TermModel hChild = (TermModel)rulePart.getChild(i);
      if ((child.isAtom()|| child.isInteger()) &&
```

```
!(child.toString().contains("JLOG"))
&& hChild.isVar()) {
     body=body.replaceAll(hChild.toString(),child.toString());
      }
  }
  myJtms.executePrologQuery(body);
}
}
}

private void executeRelatedQueries(Jtms myJtms, Set<TmsNode> set  ) {
if (set != null) {
Iterator<TmsNode> rules = set.iterator();
while (rules.hasNext()) {
TmsNode rule = rules.next();
if (rule.getLabel() == tmsNodeLabel.IN ) {
this.executePartialQuery(myJtms, rule);
}
}
}else
System.out.println("set is empty");
}

public void propagateNewRule(Jtms myJtms) {
TmsNode ruleQuery = myJtms.getTms().findNode(this.getGT());
   if (ruleQuery != null) {
     if (ruleQuery.getLabel() == tmsNodeLabel.F) {
   TermModel head = (TermModel) this.getNode().getChild(0);
        myJtms.executePrologRule(head, this);
     }
   }
}

public void propagateNewAtom(Jtms myJtms) {
TmsNode parentQuery =  myJtms.getTms().findNode(this.getGrandParent());
if ((parentQuery != null) && (parentQuery.relatedQueries != null))  {
Iterator<TmsNode> it = parentQuery.relatedQueries.iterator();
while (it.hasNext()) {
TmsNode element = it.next();
if ((element.getLabel() == tmsNodeLabel.F) )
{
  this.executeRelatedQueries
(myJtms, element.getRules());
}
}
}
}

public String toString() {
String x = this.node.toString();
    x+=",  Type="+this.type;
    x+=",  Label="+this.label;
    if (this.type == 'R')
      x+="\nBody= "+this.body;
    x+= "\n";
    if (rules != null) {
```

```
    Iterator<TmsNode> r = rules.iterator();
while (r.hasNext()) {
  TmsNode m = (TmsNode) r.next();
  x+="Rules :"+ m.toString(true);
}
    }
    if (this.relatedQueries != null) {
    Iterator<TmsNode> r = this.relatedQueries.iterator();
while (r.hasNext()) {
  TmsNode m = (TmsNode) r.next();
  x+="rQueries :"+ m.toString(true);
}
    }
    if (this.childQueries != null) {
    Iterator<TmsNode> r = this.childQueries.iterator();
while (r.hasNext()) {
  TmsNode m = (TmsNode) r.next();
  x+="cQueries :"+ m.toString(true);
}
    }
    return x;
}

public String toString(boolean flag) {
String x = this.node.toString();
x+=",  Label="+this.label;
x+= "\n";
return x;
}

public void addToInList(justification t) {
if ( this.inList == null ) {
    this.inList = new HashSet<justification>();
    }
this.inList.add(t);
}

public void addToOutList(justification t) {
if ( this.outList == null ) {
    this.outList = new HashSet<justification>();
    }
this.outList.add(t);
}

public void addToSupport(justification t) {
if ( this.support == null ) {
    this.support= new HashSet<justification>();
    }
this.support.add(t);
}

public void addToRelatedQueries(TmsNode t) {
if ( this.relatedQueries == null ) {
    this.relatedQueries= new HashSet<TmsNode>();
    }
this.relatedQueries.add(t);
```

```
}

public void addToRules(TmsNode t) {
if ( this.rules == null ) {
    this.rules= new HashSet<TmsNode>();
    }
this.rules.add(t);
}

public void addToGroundItems(TmsNode t) {
if ( this.groundItems == null ) {
    this.groundItems= new HashSet<TmsNode>();
    }
this.groundItems.add(t);
}

public void addToChildQueries(TmsNode t) {
if ( this.childQueries == null ) {
    this.childQueries= new HashSet<TmsNode>();
    }
this.childQueries.add(t);
}

public Set<TmsNode> getGroundItems () {
  return this.groundItems;
}

public Set<TmsNode> getChildQueries () {
  return this.childQueries;
}

public static int Icount () {
 return Icount;
}

public static int Rcount () {
 return Rcount;
}

public static int Ecount () {
 return Ecount;
    }

public TmsNode findChildQuery(TmsNode query) {
if (this.childQueries != null) {
    Iterator<TmsNode> r = this.childQueries.iterator();
while (r.hasNext()) {
  TmsNode m = (TmsNode) r.next();
  if (query == m)
return m;
  if  (m.getNode().unifies(query.getNode()) )
return m;
}
    } else {
      if (this.getLabel()== tmsNodeLabel.F) {
    return this;
```

```
      }
    }
return null;
}
}
```

Justification Class

```
package com.taher.jlog.jtms;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import com.declarativa.interprolog.TermModel;;

public class justification {
private boolean active;
private TmsNode rule;
private Set<TmsNode> inList=null;
private Set<TmsNode> outList=null;
private TmsNode consequence;

private void addToList (Jtms myJtms,  TermModel []list,  tmsNodeLabel l,
Set<TmsNode> myList) {
myList = new HashSet<TmsNode>();
    for (int j =0; j< list.length; j++) {
       TermModel in = list[j];
       TmsNode t1 = myJtms.getTms().addNode(myJtms, in, 'E', l);
       myList.add(t1);
       if (l == tmsNodeLabel.IN) {

         t1.addToInList(this);
         this.inList= myList;
       } else {

         t1.addToOutList(this);
         this.outList= myList;
       }
       TmsNode childQuery = myJtms.getTms().findNode(t1.getGrandParent());
       TmsNode parentQuery =
  myJtms.getTms().findNode
(this.consequence.getGrandParent());
       if (parentQuery != null)
         if (childQuery != null)
           childQuery.addChildQuery(parentQuery);
       }
       this.active = true;
       this.consequence.addToSupport(this);
}

public justification (Jtms myJtms,  TermModel just,  TermModel Rule,
TermModel []inList, TermModel []outList,
TermModel consequence) {
```

```
this.rule =  myJtms.getTms().findNode(Rule);
    this.consequence =  myJtms.getTms().addNode(myJtms,
consequence, 'E', tmsNodeLabel.IN);
    if ( inList.length > 0 ) {
       this.addToList (myJtms,inList,  tmsNodeLabel.IN,  this.inList);
    }
    if ( outList.length > 0 ) {
      this.addToList (myJtms, outList,tmsNodeLabel.OUT, this.outList);
    }
    rule.addToInList(this);

}

public String toString (){
String x="";
if (this.rule != null)
x+=  this.rule.toString();
x+= this.consequence.toString();
    return x;
}

public boolean getActive () {
return this.active;
}

private boolean checkJustification () {
if (this.rule.getLabel()== tmsNodeLabel.OUT)
  return false;
if (this.inList != null) {
Iterator<TmsNode> in = this.inList.iterator();
while (in.hasNext()) {
TmsNode m = (TmsNode) in.next();
if (m.getLabel()==tmsNodeLabel.OUT)
 return false;
}
}
if (this.outList != null) {
Iterator<TmsNode> in = this.outList.iterator();
while (in.hasNext()) {
TmsNode m = (TmsNode) in.next();
if (m.getLabel()==tmsNodeLabel.IN)
 return false;
}
}
return true;
}

public  void setActive (boolean a, Jtms myJtms) {
if (this.active != a) {
if (a == false) {
  this.active= a;
  this.consequence.setLabel(tmsNodeLabel.OUT, myJtms);
} else {
  if (this.checkJustification()) {
  this.active= a;
  this.consequence.setLabel(tmsNodeLabel.IN, myJtms);
```

```
  }
}
}
}
}
```

Appendix B

List of Scheduled Classes in the Academic Year 2002/2003

```
schedule(0201,000003,1002).
schedule(0201,000003,1003).
schedule(0201,000003,1004).
schedule(0201,000003,1005).
schedule(0201,000003,1006).
schedule(0201,000003,1007).
schedule(0201,000003,1008).
schedule(0201,000003,1009).
schedule(0201,000003,1010).
schedule(0201,000022,1011).
schedule(0201,000022,1012).
schedule(0201,000022,1013).
schedule(0201,000022,1014).
schedule(0201,000022,1015).
schedule(0201,000022,1016).
schedule(0201,000044,1017).
schedule(0201,000044,1018).
schedule(0201,000044,1019).
schedule(0201,000044,1020).
schedule(0201,000044,1021).
schedule(0201,000044,1022).
schedule(0201,000044,1023).
schedule(0201,000044,1024).
schedule(0201,000250,1025).
schedule(0201,000250,1026).
schedule(0201,000250,1027).
schedule(0201,000250,1028).
schedule(0201,000250,1029).
schedule(0201,000251,1030).
schedule(0201,000251,1031).
schedule(0201,000251,1032).
schedule(0201,000251,1033).
schedule(0201,000253,1034).
schedule(0201,000253,1035).
schedule(0201,000253,1036).
schedule(0201,000253,1037).
schedule(0201,000253,1038).
```

```
schedule(0201,000253,1039).
schedule(0201,000253,1040).
schedule(0201,000253,1041).
schedule(0201,000253,1042).
schedule(0201,000253,1043).
schedule(0201,000254,1044).
schedule(0201,000254,1045).
schedule(0201,000254,1046).
schedule(0201,000254,1047).
schedule(0201,000254,1048).
schedule(0201,000254,1049).
schedule(0201,000254,1050).
schedule(0201,000254,1051).
schedule(0201,000264,1052).
schedule(0201,000264,1053).
schedule(0201,000264,1054).
schedule(0201,000264,1055).
schedule(0201,000264,1056).
schedule(0201,000294,1057).
schedule(0201,000294,1058).
schedule(0201,000294,1059).
schedule(0201,000294,1060).
schedule(0201,000294,1061).
schedule(0201,000294,1062).
schedule(0201,000001,1063).
schedule(0201,000001,1064).
schedule(0201,000001,1065).
schedule(0201,000001,1067).
schedule(0201,000001,1068).
schedule(0201,000001,1069).
schedule(0201,000001,1070).
schedule(0201,000001,1071).
schedule(0201,000002,1077).
schedule(0201,000002,1079).
schedule(0201,000002,1081).
schedule(0201,000002,1082).
schedule(0201,000002,1086).
schedule(0202,000003,1002).
schedule(0202,000003,1003).
schedule(0202,000003,1004).
schedule(0202,000003,1005).
schedule(0202,000003,1006).
schedule(0202,000003,1007).
schedule(0202,000006,1008).
schedule(0202,000006,1009).
schedule(0202,000006,1010).
schedule(0202,000011,1011).
schedule(0202,000011,1012).
schedule(0202,000011,1013).
schedule(0202,000011,1014).
schedule(0202,000016,1015).
schedule(0202,000016,1016).
schedule(0202,000016,1017).
schedule(0202,000044,1018).
schedule(0202,000044,1019).
schedule(0202,000044,1020).
```

```
schedule(0202,000044,1021).
schedule(0202,000044,1022).
schedule(0202,000044,1023).
schedule(0202,000068,1024).
schedule(0202,000068,1025).
schedule(0202,000136,1026).
schedule(0202,000136,1027).
schedule(0202,000250,1028).
schedule(0202,000250,1029).
schedule(0202,000250,1030).
schedule(0202,000251,1031).
schedule(0202,000251,1032).
schedule(0202,000252,1033).
schedule(0202,000252,1034).
schedule(0202,000252,1035).
schedule(0202,000252,1036).
schedule(0202,000252,1037).
schedule(0202,000252,1038).
schedule(0202,000253,1039).
schedule(0202,000253,1040).
schedule(0202,000253,1041).
schedule(0202,000253,1042).
schedule(0202,000253,1043).
schedule(0202,000253,1044).
schedule(0202,000253,1045).
schedule(0202,000254,1046).
schedule(0202,000254,1047).
schedule(0202,000254,1048).
schedule(0202,000254,1049).
schedule(0202,000254,1050).
schedule(0202,000254,1051).
schedule(0202,000254,1052).
schedule(0202,000254,1053).
schedule(0202,000255,1054).
schedule(0202,000255,1055).
schedule(0202,000256,1056).
schedule(0202,000256,1057).
schedule(0202,000257,1058).
schedule(0202,000257,1059).
schedule(0202,000257,1060).
schedule(0202,000257,1061).
schedule(0202,000257,1062).
schedule(0202,000257,1063).
schedule(0202,000264,1064).
schedule(0202,000264,1065).
schedule(0202,000264,1066).
schedule(0202,000264,1067).
schedule(0202,000264,1068).
schedule(0202,000264,1069).
schedule(0202,000294,1070).
schedule(0202,000294,1071).
schedule(0202,000294,1072).
schedule(0202,000294,1073).
schedule(0202,000294,1074).
schedule(0202,000294,1075).
schedule(0202,000294,1076).
```

```
schedule(0202,000001,1077).
schedule(0202,000001,1078).
schedule(0202,000001,1079).
schedule(0202,000001,1080).
schedule(0202,000001,1081).
schedule(0202,000002,1082).
schedule(0202,000002,1083).
schedule(0202,000002,1084).
schedule(0202,000002,1085).
schedule(0202,000002,1086).
schedule(0202,000002,1087).
schedule(0202,000002,1088).
schedule(0202,000268,1089).
schedule(0202,000268,1090).
schedule(0202,000268,1091).
schedule(0202,000268,1092).
schedule(0202,000268,1093).
schedule(0203,000003,1002).
schedule(0203,000003,1003).
schedule(0203,000006,1004).
schedule(0203,000044,1005).
schedule(0203,000044,1006).
schedule(0203,000068,1007).
schedule(0203,000136,1008).
schedule(0203,000136,1009).
schedule(0203,000252,1010).
schedule(0203,000252,1011).
schedule(0203,000254,1012).
schedule(0203,000254,1013).
schedule(0203,000254,1014).
schedule(0203,000255,1015).
schedule(0203,000255,1016).
schedule(0203,000264,1017).
schedule(0203,000264,1018).
schedule(0203,000264,1019).
schedule(0203,000265,1020).
schedule(0203,000265,1021).
schedule(0203,000002,1022).
schedule(0203,000002,1023).
schedule(0203,000002,1024).
```

Appendix C

A Sample Enrollment Data for the Academic Year 2002/2003

```
reg(0201,000003,0000095,1002).
reg(0201,000003,0000116,1002).
reg(0201,000003,0000123,1002).
reg(0201,000003,0000127,1002).
reg(0201,000003,0000162,1002).
reg(0201,000003,0000177,1002).
reg(0201,000003,0000192,1002).
reg(0201,000003,0000196,1002).
reg(0201,000003,0000249,1002).
reg(0201,000003,0000357,1002).
reg(0201,000003,0000371,1002).
reg(0201,000003,0000375,1002).
reg(0201,000003,0000407,1002).
reg(0201,000003,0000418,1002).
reg(0201,000003,0000429,1002).
reg(0201,000003,0000489,1002).
reg(0201,000003,0000556,1002).
reg(0201,000003,0000059,1003).
reg(0201,000003,0000082,1003).
reg(0201,000003,0000112,1003).
reg(0201,000003,0000131,1003).
reg(0201,000003,0000163,1003).
reg(0201,000003,0000164,1003).
reg(0201,000003,0000187,1003).
reg(0201,000003,0000234,1003).
reg(0201,000003,0000251,1003).
reg(0201,000003,0000303,1003).
reg(0201,000003,0000309,1003).
reg(0201,000003,0000313,1003).
reg(0201,000003,0000314,1003).
reg(0201,000003,0000355,1003).
reg(0201,000003,0000402,1003).
reg(0201,000003,0000425,1003).
reg(0201,000003,0000426,1003).
reg(0201,000003,0000433,1003).
reg(0201,000003,0000037,1004).
reg(0201,000003,0000084,1004).
reg(0201,000003,0000086,1004).
```

```
reg(0201,000003,0000111,1004).
reg(0201,000003,0000119,1004).
reg(0201,000003,0000126,1004).
reg(0201,000003,0000157,1004).
reg(0201,000003,0000158,1004).
reg(0201,000003,0000226,1004).
reg(0201,000003,0000456,1004).
reg(0201,000003,0000466,1004).
reg(0201,000003,0000495,1004).
reg(0201,000003,0000500,1004).
reg(0201,000003,0000528,1004).
reg(0201,000003,0000557,1004).
reg(0201,000003,0000061,1005).
reg(0201,000003,0000075,1005).
reg(0201,000003,0000080,1005).
reg(0201,000003,0000132,1005).
reg(0201,000003,0000166,1005).
reg(0201,000003,0000195,1005).
reg(0201,000003,0000197,1005).
reg(0201,000003,0000310,1005).
reg(0201,000003,0000323,1005).
reg(0201,000003,0000344,1005).
reg(0201,000003,0000348,1005).
reg(0201,000003,0000368,1005).
reg(0201,000003,0000376,1005).
reg(0201,000003,0000390,1005).
reg(0201,000003,0000394,1005).
reg(0201,000003,0000457,1005).
reg(0201,000003,0000537,1005).
reg(0201,000003,0000555,1005).
reg(0201,000003,0000326,1006).
reg(0201,000003,0000328,1006).
reg(0201,000003,0000350,1006).
reg(0201,000003,0000377,1006).
reg(0201,000003,0000387,1006).
reg(0201,000003,0000393,1006).
reg(0201,000003,0000474,1006).
reg(0201,000003,0000475,1006).
reg(0201,000003,0000487,1006).
reg(0201,000003,0000498,1006).
reg(0201,000003,0000507,1006).
reg(0201,000003,0000521,1006).
reg(0201,000003,0000527,1006).
reg(0201,000003,0000530,1006).
reg(0201,000003,0000003,1007).
reg(0201,000003,0000006,1007).
reg(0201,000003,0000014,1007).
reg(0201,000003,0000033,1007).
reg(0201,000003,0000047,1007).
reg(0201,000003,0000051,1007).
reg(0201,000003,0000053,1007).
reg(0201,000003,0000057,1007).
reg(0201,000003,0000067,1007).
reg(0201,000003,0000098,1007).
reg(0201,000003,0000105,1007).
reg(0201,000003,0000148,1007).
```

```
reg(0201,000003,0000175,1007).
reg(0201,000003,0000213,1007).
reg(0201,000003,0000214,1007).
reg(0201,000003,0000221,1007).
reg(0201,000003,0000236,1007).
reg(0201,000003,0000237,1007).
reg(0201,000003,0000287,1007).
reg(0201,000003,0000311,1007).
reg(0201,000003,0000346,1007).
reg(0201,000003,0000423,1007).
reg(0201,000003,0000438,1007).
reg(0201,000003,0000015,1008).
reg(0201,000003,0000017,1008).
reg(0201,000003,0000019,1008).
reg(0201,000003,0000021,1008).
reg(0201,000003,0000022,1008).
reg(0201,000003,0000065,1008).
reg(0201,000003,0000091,1008).
reg(0201,000003,0000101,1008).
reg(0201,000003,0000103,1008).
reg(0201,000003,0000110,1008).
reg(0201,000003,0000146,1008).
reg(0201,000003,0000181,1008).
reg(0201,000003,0000183,1008).
reg(0201,000003,0000191,1008).
reg(0201,000003,0000292,1008).
reg(0201,000003,0000301,1008).
reg(0201,000003,0000318,1008).
reg(0201,000003,0000481,1008).
reg(0201,000003,0000486,1008).
reg(0201,000003,0000502,1008).
reg(0201,000003,0000511,1008).
reg(0201,000003,0000512,1008).
reg(0201,000003,0000514,1008).
reg(0201,000003,0000519,1008).
reg(0201,000003,0000536,1008).
reg(0201,000003,0000538,1008).
reg(0201,000003,0000539,1008).
reg(0201,000003,0000540,1008).
reg(0201,000003,0000010,1009).
reg(0201,000003,0000026,1009).
reg(0201,000003,0000029,1009).
reg(0201,000003,0000032,1009).
reg(0201,000003,0000048,1009).
reg(0201,000003,0000050,1009).
reg(0201,000003,0000184,1009).
reg(0201,000003,0000265,1009).
reg(0201,000003,0000268,1009).
reg(0201,000003,0000279,1009).
reg(0201,000003,0000284,1009).
reg(0201,000003,0000315,1009).
reg(0201,000003,0000319,1009).
reg(0201,000003,0000342,1009).
reg(0201,000003,0000388,1009).
reg(0201,000003,0000431,1009).
reg(0201,000003,0000477,1009).
```

```
reg(0201,000003,0000543,1009).
reg(0201,000003,0000549,1009).
reg(0201,000003,0000052,1010).
reg(0201,000003,0000261,1010).
reg(0201,000003,0000276,1010).
reg(0201,000003,0000288,1010).
reg(0201,000003,0000295,1010).
reg(0201,000003,0000336,1010).
reg(0201,000003,0000364,1010).
reg(0201,000003,0000370,1010).
reg(0201,000003,0000398,1010).
reg(0201,000003,0000400,1010).
reg(0201,000003,0000406,1010).
reg(0201,000003,0000416,1010).
reg(0201,000003,0000467,1010).
reg(0201,000022,0000063,1011).
reg(0201,000022,0000084,1011).
reg(0201,000022,0000126,1011).
reg(0201,000022,0000164,1011).
reg(0201,000022,0000208,1011).
reg(0201,000022,0000226,1011).
reg(0201,000022,0000330,1011).
reg(0201,000022,0000352,1011).
reg(0201,000022,0000369,1011).
reg(0201,000022,0000433,1011).
reg(0201,000022,0000441,1011).
reg(0201,000022,0000449,1011).
reg(0201,000022,0000501,1011).
reg(0201,000022,0000518,1011).
reg(0201,000022,0000529,1011).
reg(0201,000022,0000531,1011).
reg(0201,000022,0000545,1011).
reg(0201,000022,0000556,1011).
reg(0201,000022,0000064,1012).
reg(0201,000022,0000112,1012).
reg(0201,000022,0000123,1012).
reg(0201,000022,0000127,1012).
reg(0201,000022,0000162,1012).
reg(0201,000022,0000177,1012).
reg(0201,000022,0000192,1012).
reg(0201,000022,0000196,1012).
reg(0201,000022,0000249,1012).
reg(0201,000022,0000357,1012).
reg(0201,000022,0000375,1012).
reg(0201,000022,0000402,1012).
reg(0201,000022,0000418,1012).
reg(0201,000022,0000426,1012).
reg(0201,000022,0000429,1012).
reg(0201,000022,0000001,1013).
reg(0201,000022,0000020,1013).
reg(0201,000022,0000025,1013).
reg(0201,000022,0000046,1013).
reg(0201,000022,0000066,1013).
reg(0201,000022,0000140,1013).
reg(0201,000022,0000224,1013).
reg(0201,000022,0000262,1013).
```

```
reg(0201,000022,0000266,1013).
reg(0201,000022,0000275,1013).
reg(0201,000022,0000281,1013).
reg(0201,000022,0000304,1013).
reg(0201,000022,0000338,1013).
reg(0201,000022,0000340,1013).
reg(0201,000022,0000041,1014).
reg(0201,000022,0000052,1014).
reg(0201,000022,0000285,1014).
reg(0201,000022,0000291,1014).
reg(0201,000022,0000372,1014).
reg(0201,000022,0000404,1014).
reg(0201,000022,0000408,1014).
reg(0201,000022,0000494,1014).
reg(0201,000022,0000520,1014).
reg(0201,000022,0000021,1015).
reg(0201,000022,0000065,1015).
reg(0201,000022,0000103,1015).
reg(0201,000022,0000110,1015).
reg(0201,000022,0000146,1015).
reg(0201,000022,0000183,1015).
reg(0201,000022,0000191,1015).
reg(0201,000022,0000261,1015).
reg(0201,000022,0000276,1015).
reg(0201,000022,0000288,1015).
reg(0201,000022,0000336,1015).
reg(0201,000022,0000364,1015).
reg(0201,000022,0000370,1015).
reg(0201,000022,0000400,1015).
reg(0201,000022,0000406,1015).
reg(0201,000022,0000502,1015).
reg(0201,000022,0000512,1015).
reg(0201,000022,0000538,1015).
reg(0201,000022,0000539,1015).
reg(0201,000022,0000540,1015).
reg(0201,000022,0000029,1016).
reg(0201,000022,0000050,1016).
reg(0201,000022,0000145,1016).
reg(0201,000022,0000184,1016).
reg(0201,000022,0000337,1016).
reg(0201,000022,0000470,1016).
reg(0201,000022,0000473,1016).
reg(0201,000022,0000479,1016).
```

Appendix D

A Smaple List of Grades for the Academic Year 2002/2003

```
grade(0201,000003,0000181,'B-').
grade(0201,000003,0000183,'B' ).
grade(0201,000003,0000486,'Y' ).
grade(0201,000003,0000301,'A-').
grade(0201,000003,0000481,'D' ).
grade(0201,000003,0000015,'C+').
grade(0201,000003,0000017,'Y' ).
grade(0201,000003,0000019,'Y' ).
grade(0201,000003,0000021,'A' ).
grade(0201,000003,0000022,'C+').
grade(0201,000003,0000146,'A' ).
grade(0201,000003,0000101,'A-').
grade(0201,000003,0000103,'F' ).
grade(0201,000003,0000110,'C+').
grade(0201,000003,0000540,'B' ).
grade(0201,000003,0000536,'A' ).
grade(0201,000003,0000538,'C+').
grade(0201,000003,0000539,'C' ).
grade(0201,000003,0000318,'B-').
grade(0201,000003,0000292,'F' ).
grade(0201,000003,0000065,'B-').
grade(0201,000003,0000191,'A-').
grade(0201,000003,0000091,'B' ).
grade(0201,000003,0000512,'B' ).
grade(0201,000003,0000514,'B+').
grade(0201,000003,0000519,'C+').
grade(0201,000003,0000502,'C+').
grade(0201,000003,0000511,'F' ).
grade(0201,000022,0000512,'B' ).
grade(0201,000022,0000370,'A-').
grade(0201,000022,0000364,'C+').
grade(0201,000022,0000502,'B' ).
grade(0201,000022,0000065,'A-').
grade(0201,000022,0000191,'B-').
grade(0201,000022,0000288,'A-').
grade(0201,000022,0000406,'B' ).
grade(0201,000022,0000400,'B' ).
grade(0201,000022,0000539,'A-').
```

```
grade(0201,000022,0000538,'C+').
grade(0201,000022,0000540,'B' ).
grade(0201,000022,0000110,'C' ).
grade(0201,000022,0000103,'C' ).
grade(0201,000022,0000276,'D' ).
grade(0201,000022,0000336,'B' ).
grade(0201,000022,0000021,'C-').
grade(0201,000022,0000146,'B-').
grade(0201,000022,0000261,'C+').
grade(0201,000022,0000183,'B-').
grade(0201,000253,0000467,'F' ).
grade(0201,000253,0000292,'C' ).
grade(0201,000253,0000293,'C-').
grade(0201,000253,0000071,'C+').
grade(0201,000253,0000363,'B-').
grade(0201,000253,0000358,'B' ).
grade(0201,000253,0000519,'C' ).
grade(0201,000253,0000511,'C' ).
grade(0201,000253,0000502,'B-').
grade(0201,000253,0000225,'B' ).
grade(0201,000253,0000217,'B' ).
grade(0201,000253,0000219,'D' ).
grade(0201,000253,0000220,'D' ).
grade(0201,000253,0000295,'A-').
grade(0201,000253,0000181,'C+').
grade(0201,000253,0000168,'Y' ).
grade(0201,000253,0000301,'B-').
grade(0201,000253,0000308,'D' ).
grade(0201,000253,0000478,'C' ).
grade(0201,000253,0000267,'C-').
grade(0201,000253,0000264,'D+').
grade(0201,000253,0000254,'C' ).
grade(0201,000253,0000255,'B' ).
grade(0201,000253,0000256,'C+').
grade(0201,000253,0000009,'B' ).
grade(0201,000253,0000017,'Y' ).
grade(0201,000253,0000015,'C' ).
grade(0201,000253,0000141,'C+').
grade(0201,000253,0000280,'C+').
grade(0201,000253,0000318,'B-').
grade(0201,000253,0000550,'C-').
grade(0201,000253,0000536,'D+').
grade(0201,000253,0000339,'B-').
grade(0201,000253,0000351,'D+').
grade(0201,000253,0000129,'C+').
grade(0201,000254,0000416,'A-').
grade(0201,000254,0000400,'D' ).
grade(0201,000254,0000403,'Y' ).
grade(0201,000254,0000406,'F' ).
grade(0201,000254,0000288,'A' ).
grade(0201,000254,0000364,'C+').
grade(0201,000254,0000370,'C' ).
grade(0201,000254,0000181,'C' ).
grade(0201,000254,0000261,'F' ).
grade(0201,000254,0000270,'A-').
grade(0201,000254,0000318,'C' ).
```

```
grade(0201,000294,0000407,'A-').
grade(0201,000294,0000192,'A' ).
grade(0201,000294,0000357,'A-').
grade(0201,000294,0000086,'C+').
grade(0201,000294,0000371,'A-').
grade(0201,000294,0000429,'A' ).
grade(0201,000294,0000162,'B-').
grade(0201,000294,0000375,'B' ).
grade(0201,000294,0000127,'B' ).
grade(0201,000294,0000123,'C+').
grade(0201,000294,0000095,'B+').
grade(0201,000294,0000116,'A-').
grade(0201,000001,0000442,'P' ).
grade(0201,000001,0000443,'NP').
grade(0201,000001,0000465,'P' ).
grade(0201,000001,0000505,'NP').
grade(0201,000001,0000227,'P' ).
grade(0201,000001,0000230,'P' ).
grade(0201,000001,0000034,'Y' ).
grade(0201,000001,0000496,'P' ).
grade(0201,000001,0000480,'P' ).
grade(0201,000001,0000161,'NP').
grade(0201,000001,0000380,'P' ).
grade(0201,000001,0000238,'P' ).
grade(0201,000001,0000130,'Y' ).
grade(0201,000001,0000384,'P' ).
grade(0201,000001,0000124,'NP').
grade(0201,000001,0000347,'P' ).
grade(0201,000001,0000349,'P' ).
grade(0201,000001,0000325,'P' ).
grade(0201,000001,0000113,'P' ).
grade(0201,000001,0000108,'P' ).
grade(0201,000001,0000092,'Y' ).
grade(0201,000001,0000542,'NP').
grade(0201,000001,0000253,'P' ).
grade(0201,000001,0000143,'Y' ).
grade(0201,000001,0000145,'P' ).
grade(0201,000001,0000149,'P' ).
grade(0201,000001,0000008,'P' ).
grade(0201,000001,0000483,'P' ).
grade(0201,000001,0000479,'P' ).
grade(0201,000001,0000305,'P' ).
grade(0201,000001,0000005,'NP').
grade(0201,000001,0000044,'P' ).
grade(0201,000001,0000172,'P' ).
grade(0201,000001,0000296,'NP').
grade(0201,000001,0000524,'NP').
grade(0201,000001,0000223,'P' ).
grade(0201,000001,0000211,'P' ).
grade(0201,000001,0000210,'P' ).
grade(0201,000001,0000470,'P' ).
grade(0201,000001,0000473,'P' ).
grade(0201,000001,0000448,'NP').
grade(0201,000001,0000286,'NP').
grade(0201,000250,0000084,'C' ).
grade(0201,000250,0000507,'A-').
```

```
grade(0201,000250,0000456,'F' ).
grade(0201,000250,0000474,'C-').
grade(0201,000250,0000475,'C-').
grade(0201,000250,0000226,'B-').
grade(0201,000250,0000530,'D' ).
grade(0201,000250,0000528,'A-').
grade(0201,000250,0000119,'F' ).
grade(0201,000250,0000126,'C' ).
grade(0201,000250,0000350,'Y' ).
grade(0201,000250,0000377,'C+').
grade(0201,000250,0000393,'A-').
grade(0201,000250,0000157,'B-').
grade(0201,000250,0000487,'F' ).
grade(0201,000250,0000498,'D' ).
grade(0201,000250,0000500,'B' ).
grade(0201,000250,0000037,'C' ).
grade(0201,000251,0000281,'A-').
grade(0201,000251,0000224,'C-').
grade(0201,000251,0000340,'C-').
grade(0201,000251,0000342,'B' ).
grade(0201,000251,0000275,'B' ).
grade(0201,000251,0000046,'F' ).
grade(0201,000251,0000304,'B' ).
grade(0201,000251,0000262,'B' ).
grade(0201,000251,0000020,'F' ).
grade(0201,000022,0000352,'D+').
grade(0201,000022,0000164,'A' ).
grade(0201,000022,0000433,'A' ).
grade(0201,000022,0000441,'B+').
grade(0201,000022,0000126,'B' ).
grade(0201,000022,0000531,'D' ).
grade(0201,000022,0000330,'B+').
grade(0201,000022,0000529,'B' ).
grade(0201,000022,0000545,'D' ).
grade(0201,000022,0000449,'D+').
grade(0201,000022,0000556,'C' ).
grade(0201,000022,0000501,'C' ).
grade(0201,000022,0000518,'C' ).
grade(0201,000022,0000369,'D+').
grade(0201,000022,0000208,'D+').
grade(0201,000022,0000063,'D+').
grade(0201,000022,0000226,'A' ).
grade(0201,000022,0000084,'B-').
grade(0201,000022,0000402,'B-').
grade(0201,000022,0000418,'C+').
grade(0201,000022,0000064,'D' ).
grade(0201,000022,0000192,'A-').
grade(0201,000022,0000196,'B' ).
grade(0201,000022,0000357,'C+').
grade(0201,000022,0000112,'B+').
grade(0201,000022,0000249,'C' ).
grade(0201,000022,0000375,'B-').
grade(0201,000022,0000123,'B-').
grade(0201,000022,0000127,'C-').
grade(0201,000022,0000429,'C+').
grade(0201,000022,0000426,'B+').
```

```
grade(0201,000022,0000162,'A-').
grade(0201,000022,0000177,'C-').
grade(0201,000250,0000431,'C-').
grade(0201,000250,0000268,'C-').
grade(0201,000250,0000265,'D+').
grade(0201,000250,0000319,'Y' ).
grade(0201,000250,0000001,'D+').
grade(0201,000250,0000010,'B-').
grade(0201,000250,0000477,'C+').
grade(0201,000250,0000032,'C-').
grade(0201,000250,0000026,'A-').
grade(0201,000250,0000279,'B' ).
grade(0201,000250,0000284,'B+').
grade(0201,000250,0000053,'D+').
grade(0201,000251,0000196,'C-').
grade(0201,000251,0000082,'B+').
grade(0201,000251,0000355,'C-').
grade(0201,000251,0000303,'C+').
grade(0201,000251,0000309,'C-').
grade(0201,000251,0000313,'A' ).
grade(0201,000251,0000164,'A' ).
grade(0201,000251,0000163,'D-').
grade(0201,000251,0000537,'A' ).
grade(0201,000251,0000425,'EX').
grade(0201,000251,0000234,'C+').
grade(0201,000251,0000131,'A-').
grade(0201,000254,0000132,'C-').
grade(0201,000254,0000166,'D-').
grade(0201,000254,0000500,'B' ).
grade(0201,000254,0000061,'D-').
grade(0201,000254,0000513,'B+').
grade(0201,000254,0000197,'Y' ).
grade(0201,000254,0000111,'Y' ).
grade(0201,000254,0000457,'Y' ).
grade(0201,000254,0000466,'C-').
grade(0201,000254,0000119,'D' ).
grade(0201,000254,0000075,'D-').
grade(0201,000254,0000080,'A-').
grade(0201,000254,0000086,'D-').
grade(0201,000254,0000394,'B' ).
grade(0201,000254,0000474,'C' ).
grade(0201,000254,0000368,'A-').
grade(0201,000254,0000507,'A-').
grade(0201,000254,0000487,'F' ).
```