TOWARDS NEXT GENERATION LOGIC PROGRAMMING SYSTEMS

by

Ajay Bansal

For my wife, Vidya,

my parents,

my brother Amit, and

my sister Dolly

TOWARDS NEXT GENERATION LOGIC PROGRAMMING SYSTEMS

by

AJAY BANSAL, B.TECH., M.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2007

ACKNOWLEDGEMENTS

First and foremost, I thank my advisor Dr. Gopal Gupta for his guidance throughout my doctoral experience. It was his encouragement and support that made this dissertation possible. He was always available for long discussions and maintained an informal and friendly relationship. I have learned from him the art of teaching, research and writing. I would always be grateful to him for his support in both academic and non-academic matters. He is a true friend, philosopher and guide for his students.

I express my gratitude to Dr. Farokh Bastani, Dr. Kevin Hamlen, and Dr. Neeraj Mittal for agreeing to be on my dissertation committee and providing their valuable feedback on my work. I thank Dr. Vitor Santos Costa and Dr. Ricardo Rocha for their help with implementation of co-LP and ASP.

A special thanks is due for all my colleagues and friends who made my graduate student years such a memorable experience. I thank my colleagues Richard Min, Luke Simon, and Ajay Mallya for being my co-researchers. I thank Shashidhar Gandham, a graduate student and friend for all the stimulating discussions, support and suggestions. He was always there to review my papers and provide valuable feedback. I thank my friends Rupesh, Arnab, Raj, and Spandana for being around at all times. A special thanks to all the members of my Toastmasters club who helped me improve my leadership, communication and presentation skills by constantly providing feedback and evaluations at our club meetings.

I thank my parents, my sister Dolly and my brother Amit for their constant support and encouragement without which none of my academic goals would have been a reality. I thank my other family members for their love and support.

Last but not the least, I thank my wife and co-researcher Vidya, for her love, support, and constant encouragement. I thank her for always believing in me and motivating me to pursue my dreams. It is her love that keeps me going.

October 2007

TOWARDS NEXT GENERATION LOGIC PROGRAMMING SYSTEMS

Publication No. ___________________

Ajay Bansal, Ph.D.
The University of Texas at Dallas, 2007

Supervising Professor: Gopal Gupta

Various powerful extensions of Logic Programming (LP) have been proposed in the last few decades. Some of the highly successful extensions include *coinductive logic programming*, *tabled logic programming*, *constraint logic programming* and *answer set programming*. Although all of these extensions have been developed by extending standard logic programming (Prolog) systems, each one has been developed in isolation. While each extension has resulted in very powerful applications in reasoning, considerably more powerful applications will become possible if all these extensions are combined into a *single system*. Realizing multiple or all extensions in a single framework is quite challenging because the techniques devised so far for implementing these extensions are quite complex.

In this dissertation, we develop simple, elegant and easy techniques for implementing coinductive logic programming and answer set programming atop an existing prolog engine. We present our implementations of these extensions of LP atop YAP prolog. With these implementations in place, we have realized an integrated framework that combines *coinductive logic programming*, *tabled logic programming*, *constraint logic programming* and *answer set programming* into a single logic programming system.

We present the practice of *co-logic programming* (co-LP for brevity), a paradigm that combines both inductive and coinductive logic programming. Co-LP is a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent LP, model checking, bisimilarity proofs, etc.

All approaches to executing answer set programs have been primarily bottom-up (non goal-directed), which prevents any possibility of its merging with other extensions as their implementations are primarily top-down (goal-directed). We present a goal-directed top-down execution method for executing answer-set programs by extending co-LP.

The integrated system presented in this dissertation is capable of handling constraints, tabling, coinduction and non-monotonic reasoning, all at the same time. Applications to model checking and knowledge representation, presented in this dissertation, justify the power and elegance of combining these powerful extensions in a single system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Logic programming (LP), with Prolog as the most representative logic programming language, has emerged as a powerful paradigm for intelligent reasoning and deductive problem solving. In the last two decades several powerful and highly successful extensions of logic programming have been proposed and researched. These include *coinductive logic programming (coLP)* [72, 71] *constraint logic programming (CLP)* [36, 81], *tabled logic programming* [13], *inductive logic programming* [52], *concurrent/parallel logic programming* [33], Deterministic Coroutining [18], and adaptations for non-monotonic reasoning such as *answer set programming* [5]. These extensions have led to very powerful applications in reasoning; for example, CLP has been used in industry for intelligently and efficiently solving large scale scheduling and resource allocation problems, tabled logic programming has been used for intelligently solving large verification problems as well as non-monotonic reasoning problems, inductive logic programming has been used for new drug discoveries [21], and answer set programming has been used for practical planning problems [57], etc.

1.1 Next generation logic programming systems

All of these powerful extensions of logic programming have been developed by extending standard logic programming (Prolog) systems, and each one has been developed in isolation. While each extension has resulted in very powerful applications in reasoning, considerably more powerful applications will become possible if all these extensions are combined into a *single system*. This power will come about due to the search space being dramatically pruned, reducing the time taken to find a solution given a problem coded as a logic program.

Realizing multiple or all extensions in a single framework has been rendered difficult by the enormous complexity of implementing reasoning systems in general and logic programming systems in particular. Implementing standard Prolog systems is itself quite complex due to the built-in unification and backtracking that have to be supported. Extending a Prolog system to incorporate advanced features such as coinduction, constraints, tabling, or parallelism further presents formidable challenges. As a result, even though having multiple extensions in a single system will result in considerably more efficient deductive procedures and more powerful applications, researchers have not ventured beyond attempting to combine atmost two of the features. In our opinion, the main barrier that prevents researchers from building a single system combining all the features is the complexity that is involved in implementing each one of them individually. This complexity poses difficulty in two ways when the task of combining all systems is considered. First, it takes a long time to implement each feature as an extension of the Prolog system. Second, combining them becomes extremely hard, resulting in an enormously complex and buggy system, especially given that a WAM-based implementation [83, 39] of standard Prolog itself is quite complex.

1.2 Dissertation Objectives

The goal of this research is to develop the next generation logic programming system that combines various advancements that have recently been made in the area of logic programming. Combining these features all into one will result in considerably more powerful deductive systems. The advancements that we seek to combine in a single system are coinductive logic programming, constraint logic programming, tabled logic programming, and answer set programming. Each of these advancements (except ASP) have been realized individually as extensions to a Prolog engine; however, their integration into a single logic programming system has never been attempted. Having all these advanced features available in a single system will allow for more advanced applications to be developed more elegantly and efficiently. It should be noted that each of the individual advancements has already resulted in new reasoning applications being developed more easily and efficiently.

Our insight is that if we wish to realize all these features (co-LP, CLP, tabled LP, ASP) in a single system, the techniques used for implementing them have to be extremely simple. The simplicity of the implementation techniques will not only make it possible for other implementors to incorporate these features into their systems, it will become possible to elegantly combine them into a single system. With this in mind, we have been working towards developing very simple techniques for implementing various advanced features that can be realized in a standard WAM-based logic programming engine in a few man months of work. The simplicity of these techniques also allows for their integration in a single system.

The preliminary design of these implementation techniques for many of the features have, in fact, already been done by us [71, 31]. Our ultimate goal is to refine these implementation techniques, and to implement them all in a single integrated system together on top of an existing logic programming engine. This single integrated system will support coinduction, constraints, tabling, answer set programming in a seamless fashion.

The main contributions of this work are as follows:

(i) Design of “Coinductive Logic Programming” and “Co-Logic Programming” languages.

(ii) Design a simple & efficient technique to implement these languages atop an existing Prolog engine.

(iii) Design a Top-Down algorithm for goal-directed execution of propositional Answer-Set Programs.

(iv) Design a simple & efficient technique to implement the Top-Down propositional ASP atop an existing Prolog engine.

(v) Applications justifying the combinations of co-LP, CLP, ASP and Tabled LP.

1.3 Dissertation Outline

The remainder of this document is organized as follows. Chapter 2 presents the background concepts required for the research presented in this dissertation. It also presents the literature survey done for this research. It describes various extensions of logic programming and their current implementations, which we intend to combine in this research. This chapter also presents the concept of *Coinduction*, which has been used in universal algebra, category theory, etc., and which forms the basis of *Co-Logic Programming* presented in Chapter 3. The motivation behind *Co-Logic Programming*, the language, its implementation details and a few applications are also presented in Chapter 3. A goal-directed top-down execution method for execution of Answer-Set programs (ASP) is presented in Chapter 4. This chapter also presents the details of the implementation of our goal-directed execution engine for ASP atop an existing YAP prolog engine. Chapter 5 presents applications justifying the combination of various extensions of logic programming. An elegant framework to verify both safety and liveness properties in the field of model checking is presented as an application of tabled logic programming, constraint logic programming and coinductive logic programming combined. Other applications from fields such as AI and knowledge representation, justifying the combination of Answer-Set programming and Constraint Logic programming, are also presented. Finally, Chapter 7 presents conclusions and some possible future directions for extending the research presented in this dissertation.

CHAPTER 2
BACKGROUND

This chapter presents the requisite background concepts for realizing a next generation logic programming system. The mathematical concepts of induction and coinduction are presented in Section 2.3. Coinduction forms the basis for realizing Co-Logic programming, a powerful extension to logic programming, presented in the next chapter. A brief overview of other extensions to logic programming that we intend to combine–i.e., Constraint Logic programming, Tabled logic programming and Answer-set programming–are presented next. Finally, we conclude the chapter with a discussion on incorporating these extensions into a single logic programming system.

2.1 Logic Programming

Logic Programming (LP) languages belong to the class of languages called *declarative languages*, which are high-level languages compared to traditional imperative languages. Declarative languages allow the programmer to specify what the problem is rather than having to specify the details on how to solve the problem. These languages have a strong mathematical basis which makes the programming task easier. The programmer can specify the problem at a very high level, without getting into the step-by-step details for solving it.

Logic Programming languages are based on predicate calculus. The Logic Programming [45] paradigm uses Horn Clause Logic which is a subset of First Order Logic. Given a theory (or a program) and a query, Logic Programming uses the theory to search for different ways to satisfy the query. Logic Programming has a number of features that are unique to it. For example, variables are instantiated only once and variables are untyped until they are instantiated. Variables are instantiated via unification, which is the process of finding the most general, common instance between two objects. When unification fails, the execution backtracks and tries to find another solution for the query. Prolog is a popular logic programming language with a simple declarative semantics, simple procedural semantics, high expressive power, and inherent non-determinism. Logic programs written in Prolog can be seen as executable specifications. The expressive power of Prolog allows for designing complex and efficient algorithms. These features lead to compact and elegant code that can be easily understood, implemented, and transformed.

A Logic Program consists of a collection of *Horn clauses* which are called *rules* in Prolog and are of the form

A :- B_1, ..., B_n.

A rule is made up of the following:

(i) *head*: The head of the rule above is A.

(ii) *body*: The body of the rule above is B_1, ..., B_n.

A rule without a body is called a *fact* which is simply written as follows:

A.

A rule without a head is called a *query* and is written as follows:

:- B_1, ..., B_n.

Prolog programs have *predicates,* which are applied to *terms* or arguments and are of the form:

p(t_1, ..., t_n).

where p is the predicate name and t_1, ..., t_n are terms used as arguments.

A *term* is recursively defined as follows:

(i) A constant is a term.

(ii) A variable is a term.

(iii) If f is an n-place function symbol and $t_1, t_2, ..., t_n$ are terms, then the function $f(t_1, t_2, \ldots, t_n)$ is a term.

(iv) All terms generated by applying these rules are terms.

The execution of a query Q against a logic program P, leads to consecutive assignments of terms to the variables of Q until a substitution θ satisfied by P is found. A substitution is a function that given a variable of Q, returns a term assignment. The solutions are reported by listing each variable X in Q along with the corresponding assignment $\theta(X)$.

Prolog was invented by Colmerauer and stands for *PROgramation en LOGic*. The pioneering work on resolution by Robinson described the general inference rule *Resolution with Unification*. Following this work, Colmerauer and Kowalski found the procedural semantics of Horn clauses that led to Prolog. In 1977, the first compiler [83] for Prolog was developed, which attracted more programmers to this language. Warren proposed a new abstract machine for executing compiled Prolog code that is now known as Warren Abstract Machine, or simply WAM.

2.2 Warren Abstract Machine

Most logic programming languages rely on WAM's technology, which is a stack-based architecture with simple data structures and a low-level instruction set. The contents of WAM's data areas, data structures, and registers are used to obtain the state of the computation. Figure 2.1 shows in detail the organization of WAM's memory, frames, and registers. WAM defines the following execution stacks.

- *PDL* is a push down list used by the unification process.
- *Trail* is an array of addresses. It stores the addresses of the stack or heap variables, which are reset upon backtracking. The register *TR* points to the top of this array.

- *Stack*, also known as the *local stack*, stores the *environment* and *choice point* frames. The environment tracks the flow control of the program. It is pushed onto the stack whenever a clause containing several body subgoals is picked for execution and is popped off before the last body subgoal gets executed. The choice points store the untried alternatives. A choice point contains all the necessary data to restore the state of the computation back to when the clause was entered. It also has a pointer to the next clause that can be tried, in case the current one fails.
- *Heap*, also referred to as *global stack*, is an array of data cells used to store variables and compound terms that cannot be stored on the stack. The *H* register points to the top of this stack.
- *Code Area* contains the WAM instructions comprising the compiled form of loaded programs.

WAM instructions can be categorized into four main groups:

- *Choice point instructions* manipulate the choice points. They allow the allocation or removal of choice points and recover the state of a computation through the data stored in choice points.
- *Control instructions* allocate or remove environments and manage the call and return sequence of sub-goals.
- *Unification instructions* implement specialized versions of the unification algorithm according to the position and type of arguments. These specialized algorithms can be further categorized as: treating the first occurrence of variables in a clause, non-first occurrences, constants in clauses, lists, and compound terms.
- *Indexing instructions* help in accelerating the process of determining which clauses unify with a given subgoal call. Based on the first argument of the call, these instructions jump to the specialized code that can directly index the unifying clauses.

The simplicity of WAM hides several involved implementation issues.

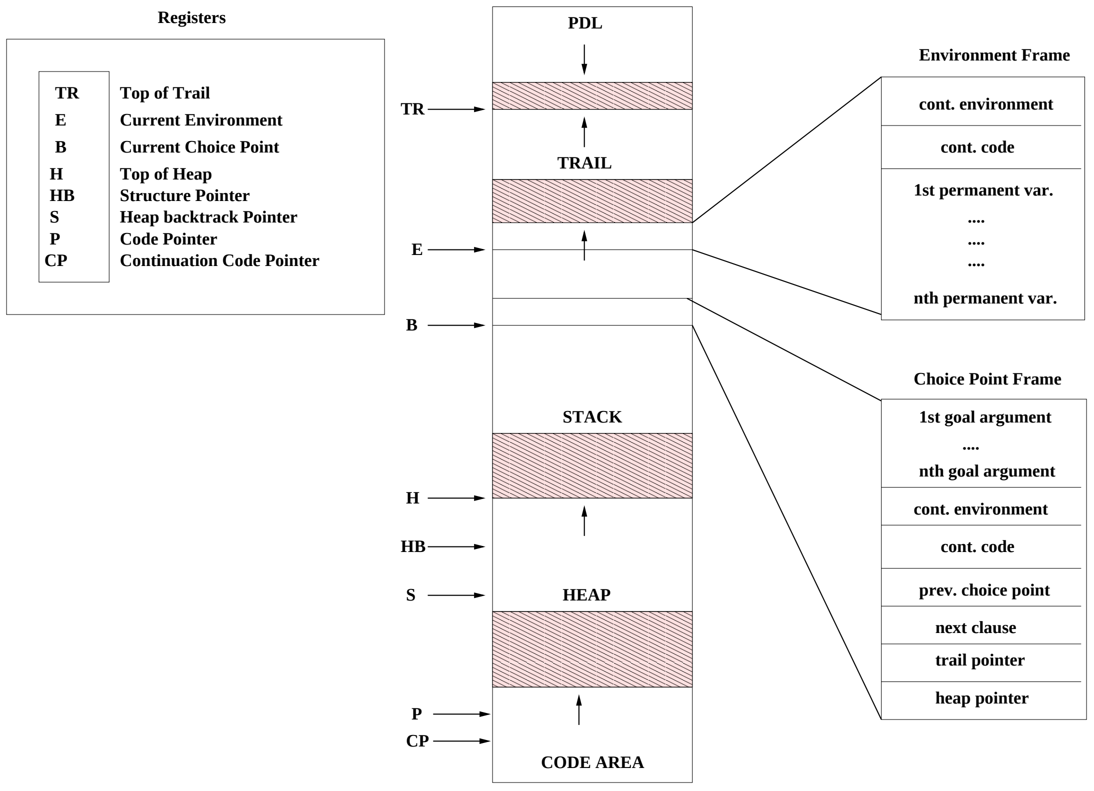

Figure 2.1. WAM Organization

2.3 Induction and Coinduction

Induction corresponds to well-founded structures that start from a basis that serves as a foundation for building more complex structures. For example, natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality [29]. Thus, the inductive definition of a list of natural numbers is as follows:

(i) `[]` (empty list) is a list (initiality).
(ii) `[H|T]` is a list if `T` is a list and `H` is some number (iteration).
(iii) the set of lists is the minimal set satisfying (i) & (ii) (minimality).

Minimality implies that infinite-length lists of numbers are not members of the inductively defined set of lists of numbers. Inductive definitions correspond to least fixed point interpretations of recursive definitions.

Coinduction is the dual of induction. Coinduction eliminates the initiality condition and replaces the minimality condition with maximality. Thus, the coinductive definition of a list of numbers is:

(i) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration).
(ii) the set of lists is the maximal set satisfying (i) (maximality).

There is no base case in coinductive definitions, and while this may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point interpretation of recursive definitions. (Recursive definitions for which gfp interpretation is intended are termed corecursive definitions.) Thus, the set of lists under coinduction is the set of all infinite lists of numbers (no finite lists are contained in this set). Note, however, that if we have a recursive definition with a base case, then under coinductive interpretation, the set defined will contain both finite and infinite-sized elements, since in this case the gfp will also contain the lfp. In the context of logic programming, in the presence of coinduction proofs may be of infinite length. A coinductive proof is essentially an infinite-length proof.

2.4 Constraint Logic Programming

Constraint logic programming [81, 36, 11, 76] is a generalization of logic programming in which logical variables can range over a given domain (such as real numbers, booleans, rationals, finite sets of integers, etc.) instead of the set of terms induced by the constants and function symbols present in the program (the Herbrand Universe). For variables and values in these domains, unification is replaced by constraint solving. Subgoals that consist entirely of variables and values ranging over these domains are called constraints, and can be solved only by invoking a special constraint solver that corresponds to this domain.

Constraint logic programming (CLP) systems are realized by extending standard logic programming systems with constraints and a constraint solver. Most modern logic programming systems have constraint solving capabilities built in. In such systems, subgoals that do not qualify as constraints are solved using the standard logic programming mechanism of unification. Constraints that cannot be solved immediately when they are encountered, are suspended. During execution, the constraint solver is invoked every time a variable receives a value to solve the constraints that have accumulated so far.

CLP has been immensely successful in the commercial world and, in fact, in its general form called *constraint programming*, it has been incorporated in other languages as well such as C++ and Java. Many successful commercial companies have been built around the CLP technology (e.g., ILOG, Cosytec). In many instances, CLP has been used to solve difficult operations research problems with very small programming effort and considerable efficiency.

While logic programming can be thought of as following the *generate and test* paradigm, CLP can be thought of as following the *test and generate* paradigm. Thus, while in logic programming programmers have to worry about the order in which they should place the subgoals in a rule, in CLP they can simply put the *test* constraints first before the *generate* goals. The suspension of constraints that are not sufficiently instantiated allows execution of subgoals to take place in the “data-flow” order, reducing the search space by many orders of magnitude.

Constraint programming systems break the rigid left to right execution order of subgoals mandated by Prolog, though only for subgoals that qualify as constraints. Ordinary subgoals are still executed in left-to-right order, and the rules are still tried in a top-to-bottom order.

Most implementations of CLP have been obtained by extending the Warren Abstract Machine (WAM) based implementations of Prolog. Two prominent approaches have been to use an approach based on *attributed variables* [34] and compiling all constraints to the primitive constraint “`X in R`” [16]. We give only a brief outline of the approaches below.

In the attributed variables approach, logical variables have attributes in which additional information about the variable can be held. Thus, it is possible to attach a term to one variable. When a variable with an attached term has to be unified, a user-defined predicate is called. Thus, constraints suspended on a variable can be stored in the variable’s attribute. When an attempt to unify this variable is made, the constraint solver is invoked to see if any of the constraints suspended on this variable can be solved if unification produces a binding for the variable. If the WAM engine has support for attributes built in, then constraints can be cleanly incorporated without extensive modification to the design of the WAM. The attributed variable approach has been used in the SICStus CLP system [11].

In the second approach, all types of constraints are translated into a primitive constraint of one single type: *X in r*, where *r* is a range of constants (e.g. 1..20) or a range of *indexicals*. Loosely speaking, an indexical is a set derived from the domain of the variable *X* [16]. The indexical-based approach can be easily incorporated into a WAM-based implementation due to its simplicity. This incorporation can be done in a way that most of the data-structures and control-structures of the WAM are untouched. Thus, the representation of choice-points, environments, and terms without constraints remains unchanged.

2.5 Tabled Logic Programming

Traditional logic programming systems (e.g., Prolog) use SLD resolution [75] with the following *computation strategy* [75]: subgoals of a resolvent are tried from left to right and clauses that match a subgoal are tried in the textual order they appear in the program.

It is well known that Prolog's computation strategy (SLD resolution [75]) may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics (e.g., datalog-like programs with left recursive rules). In fact, this is true of any "static" computation strategy that is adopted. To get around this problem, researchers have suggested computation strategies that are *dynamic* in nature coupled with recording solutions in a *memo table*. In a dynamic computation strategy, the decision regarding which clause to use next for resolution is taken based on runtime properties, e.g., the nature and type of goals in the current resolvent. OLDT [78] is one such computation strategy. OLDT is adopted in the XSB system [1, 13], the most mature tabled LP system. In OLDT resolution, solutions to certain subgoals are recorded in a *memo table* (heretofore referred to simply as a *table*; such a subgoal table is referred to as a *tabled call*). In OLDT resolution, solutions to tabled calls are recorded in the table (termed *tabled solutions*). A call to a subgoal that is identical to a previous call is called a *variant call* and may possibly lead to non-termination if SLD resolution is used. When a variant call is encountered while computing a tabled call, the OLDT resolution strategy will not expand it as SLD resolution will; rather the solutions to the variant call will only be obtained by matching it with tabled solutions. If any solutions are found in the table, they are *consumed* one by one just as a list of fact clauses by the variant call, each producing a solution for the variant call. After consuming, the computation of the variant subgoal is suspended until some new solutions appear in the table. This consumption and suspension continues until we can detect that all the solutions for the tabled call have been generated and a *fixpoint* reached. Tabled logic programming systems have been put to many innovative uses. These include:

(i) *Non-monotonic reasoning:* Tabled LP systems can support more powerful forms of negations [12].

(ii) *Efficient implementation of deductive databases:* Tabled systems can be considerably faster than traditional deductive databases [1].

(iii) *Program Analysis:* Tabled LP systems provides a generic framework for developing abstract interpreters [64, 15].

(iv) *Model checking:* Model checkers based on tabled LP systems are comparable in speed to state-of-the-art model checkers but are considerably easier to program [65].

It should be noted that tabled logic programming is a combination of two techniques: (i) memoing, which makes execution more efficient by avoiding redundant computations, and (ii) exploring the matching clauses in a dynamic order (with variant checking). In fact, the incompleteness of SLD resolution mentioned above can be cured with (ii) alone; memoing merely makes the process more efficient. Thus, tabled logic programming can be thought of as a mechanism for finding the "best" order in which clauses (rules) will be tried so as to find a solution as quickly as possible. As is evident from the various applications of tabled LP mentioned above, trying the clauses in this optimal order has tremendous benefits.

The ability to table calls and solutions results in a more complete logic programming system. Thus, tabling should be an indispensable part of any Prolog system. However, this has not happened, mainly, we believe, due to the techniques that have traditionally been used to incorporate tabling in existing LP systems.

There are very few implemented tabled logic programming systems. The most well known is the XSB system, based on OLDT resolutions [1, 13]. In XSB, OLDT resolution has been implemented by a combination of computation suspension via *stack freezing* and maintaining a forest of SLD trees [1, 13]. In the XSB system [1] that is based on OLDT, certain predicates are declared to be *tabled*. Every time a tabled call is encountered, the current computation is frozen and a new SLD computation is begun. This SLD computation *produces* answers that will be *consumed* by another SLD tree waiting for answers. Forest of SLD trees, stack freezing, suspensions and resumptions, all make implementing OLDT in this way quite complex. Also, the freezing of stacks results in space overheads. Several man-years have been invested in the design and development of the XSB system [13, 68, 84] due to this complexity. This investment in effort has indeed yielded results, turning XSB into the extremely efficient and most widely used tabled LP system in use today. Variants of OLDT, such as SLDT, are simpler to implement, incur less space overhead, and have been recently implemented in the B-Prolog system [86]. However, in SLDT, ensuring that all tabled solutions have been found is very expensive in time because of the execution strategy used.

2.6 Answer Set Programming

Answer Set Programming (ASP) arises from the research on using logic programming as a language for declaratively programming systems that exhibit non-monotonic behavior [5]. ASP programs are expressed as collections of propositional rules of the form

$$L_0 \vee \cdots \vee L_k \leftarrow L_{k+1} \wedge \cdots \wedge L_m \wedge \ not\ L_{m+1} \wedge \cdots \wedge \ not\ L_n$$

where L_i are literals in the sense of classical logic. The semantics of a program is expressed in terms of *answer sets* of the program [23], where answer sets correspond to minimal supported models of the underlying logic theory. ASP rules can be naturally interpreted as constraints on the admissible answer sets of the programs, while the answer sets themselves can be viewed as set-based representations of the solutions to the problem encoded by the program.

ASP offers a number of advantages over Prolog as well as existing non-monotonic logics:

(i) ASP offers a purely declarative language free of dependencies from operational issues (e.g., ordering of rules/literals).

(ii) relative to Prolog, it avoids issues like floundering and non-terminating computations.

(iii) ASP has been shown to be as expressive as many other non-monotonic logics, yet it provides a much simpler syntax coupled with well-developed and efficient implementations [56, 54].

(iv) ASP is more expressive than propositional and first-order logic, allowing the programmer to elegantly encode causality, transitive closure and aggregation.

(v) there is a large body of support structure built for ASP (considerably larger than any other knowledge representation language) including theoretical and practical building blocks, laying the foundations for systematic development of programs.

ASP has been successfully employed in the construction of large reasoning systems, e.g., in the areas of diagnosis [57], web services [48], and bioinformatics [74].

The execution of an ASP program is aimed at determining the minimal stable models of the program [24]. Existing inference engines developed for ASP [56, 54, 43, 4] rely on variations of the Davis-Putnam procedure for the construction of the stable models. They construct the stable model by a series of *pick* (choose a truth value for an atom not in the model) and *expand* steps (propagate the effect of this choice). The existing inference engines implement this basic structure along with a variety of optimization techniques (e.g., clause learning, lookahead, backjumping).

Although all existing ASP inference engines have been developed from scratch as stand-alone systems, in this research we propose to develop an efficient and extensible implementation using the logic programming technology presented in the previous sections. The motivations behind our approach arise from the following observations:

- Theoretical results show that all stable models of an ASP program include the (unique) well-founded model of the program, and recent research results [66] have shown how tabling can be employed to efficiently compute the well-founded semantics of a logic program. Thus, the tabling engine can be employed as a pre-processor to generate the fixed core shared by all stable models. The ability to combine tabling with parallelism affords the possibility of generating this core very efficiently.

- The basic structure of the computation of distinct stable models clearly resembles the familiar structure of a constraint propagation computation (where the labeling step corresponds to a pick and the propagation corresponds to the expansion of the model). Indeed, researchers have recognized the inherent link between ASP computations and constraint resolution [10]. We propose to expand on this research by developing an effective mapping of the *residual program*—i.e., the ASP program simplified w.r.t. its well-founded semantics—to constraints over finite domains. Modern constraint solvers over finite domains effectively prune the search space through the use of techniques analogous to those used in state-of-the-art ASP engines, thus guaranteeing comparable levels of performance. Furthermore, the capability of explicitly programming the search strategy using the control offered by logic programming allows the flexibility of modifying the process of searching for stable models, in the same style and with the same advantages as programming search strategies for constraint programming [14].
- Leveraging logic programming to describe the basic control in the generation of the stable models of an ASP program opens the door for investigating the issues of automated parallelization of ASP computations. The non-determinism in the *pick* step directly maps to or-parallelism, while model expansion maps to dependent and-parallelism [61]. This considerably reduces the execution time of complex applications (e.g., complex planning problems).
- The encoding of ASP as constraint problems offers the possibility of enriching the language of ASP with new structures derived from the existing constraint domains—

e.g., numerical constraints—which can lead to a further enhancement of the declarative capabilities of the language [62].

2.7 Conclusions

To realize all these features (co-LP, CLP, tabled LP, ASP) in a single system, the techniques used for implementing them have to be extremely simple. The simplicity of the implementation techniques will not only make it possible for other implementors to incorporate these features into their systems, it will become possible to elegantly combine them into a single system. With this in mind we have been working towards developing simple techniques for implementing various advanced features that can be realized in a standard WAM-based logic programming engine in a few man-months of work. The simplicity of these techniques also allows for their integration in a single system.

CHAPTER 3
CO-LOGIC PROGRAMMING

Traditional logic programming with its minimal Herbrand model semantics is useful for declaratively defining finite data structures and properties, while *coinductive logic programming* allows for logic programming with infinite data structures and properties. Note that coinductive LP is not at all related to *inductive* LP, the common term used to refer to LP systems for learning rules [51]. In fact, throughout this dissertation we use the term inductive LP to refer to traditional SLD (or OLDT) resolution-based LP. In this chapter we present the practice of *co-logic programming* (co-LP for brevity), a paradigm that combines both inductive and coinductive logic programming. This combination is not straightforward as in a naive combination, coinductive predicates and inductive predicates may mutually call each other resulting in cycles whose semantics is hard to define. Co-LP allows predicates to be annotated as coinductive; by default, unannotated predicates are assumed to be inductive. Coinductive predicates can call inductive predicates and *vice versa*, the only exception being that no cycles are allowed through alternating calls to inductive and coinductive predicates. Co-LP is a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. The declarative semantics for co-LP is defined, and a corresponding top-down, goal-directed operational semantics is provided in terms of alternating SLD and *co-SLD* semantics [71, 70]. This operational semantics is implemented by extending the WAM. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent logic programming, model checking, bisimilarity proofs, Answer Set Programming (ASP), etc.; some of these applications are discussed in this chapter.

Table 3.1. Infinite Binary Streams

```
bit(0).
bit(1).
bitstream([H|T]) :- bit(H), bitstream(T).
| ?- X = [0, 1, 1, 0 | X], bitstream(X).
```

3.1 Introduction

The traditional semantics for logic programming (LP) is inadequate for various programming practices such as programming with infinite data structures and *corecursion* [9]. Not only are such programs theoretically interesting, their practical applications include improved modularization of programs as seen in lazy functional programming languages [38], rational terms, and model checking. For example, we would like programs such as the program in Table 3.1, which describes infinite binary streams, to be semantically meaningful, i.e. not semantically null.

We would like the above query to have a finite derivation and return a positive answer; however, aside from the *bit* predicate, the least fixed-point (lfp) semantics of the above program is null, and its evaluation using SLD resolution lacks a finite derivation. The problems are two-fold. The Herbrand universe does not allow for infinite terms such as `X` and the least Herbrand model does not allow for infinite proofs, such as the proof of `bitstream(X)`; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [9]. Therefore, we must not exclude these concepts from logic programming, just because they break with tradition. The traditional declarative semantics of LP must be extended in order to reason about infinite and cyclic structures and properties. This has indeed been done and the paradigm of coinductive logic programming defined and its declarative and operational semantics given [72]. In the coinductive LP paradigm the declarative semantics of the predicate `bitstream/1` above is given in terms of *infinitary Herbrand universe, infinitary Herbrand base, and maximal models (computed using greatest fixed-points)*. The operational semantics is given in terms of the *coinductive hypothesis rule* which states that during execution, if the current resolvent R contains a call C' that unifies with a call C encountered earlier, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension, a clause such as `p([1|T]) :- p(T)` and the query `p(Y)` will produce an infinite answer `Y = [1|Y]`.

Applications of purely coinductive logic programming to fields such as model checking, concurrent logic programming, real-time systems, etc., can also be found in [72]. There are problems for which coinductive LP is better suited than traditional inductive LP. Conversely, there are problems for which inductive LP is better suited than coinductive LP. But there are even more problems where both coinductive and inductive logic programming paradigms are simultaneously useful. In this chapter we present the combination of coinductive and inductive LP. We christen the new paradigm *co-logic programming* (co-LP for brevity). *Thus, co-LP subsumes both coinductive LP and inductive LP.* A combination of inductive and coinductive LP is not straightforward as cyclical nesting of inductive and coinductive definitions results in programs to which proper semantics cannot be given. *Co-logic programming* combines traditional and coinductive logic programming by allowing predicates to be optionally annotated as being coinductive; by default, unannotated predicates are interpreted as inductive. In our formulation of co-LP, coinductive predicates can call inductive predicates and *vice versa*, with the only exception being that no cycles are allowed through alternating calls to inductive and coinductive predicates. This results in a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. The declarative semantics for co-LP is defined, and a corresponding top-down, goal-directed operational semantics is provided in terms of alternating SLD and co-SLD semantics [71]. This operational semantics is implemented atop YAP prolog [17]. Applications of co-LP are also discussed. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent logic programming, model checking, bisimilarity proofs, Answer Set Programming (ASP), etc. Finally, an outline of a high-level implementation realized atop YAP Prolog's engine and insights for a more efficient low level (WAM level) implementation are also described.

3.2 Language Overview

Coinductive logic programming augments traditional logic programming with coinduction. It provides logic programming the capability to reason about infinite data structures and properties. The syntax and declarative semantics of Coinductive Logic programming is presented in [72]. The operational semantics of coinductive logic programming is given in terms of the *coinductive hypothesis rule* which states that during execution, if the current resolvent R contains a call C' that unifies with a call C encountered earlier, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension, a clause such as

```
p([1|T]) :- p(T)
```

and the query `?- p(Y)` will produce an infinite answer `Y = [1|Y]`.

Thus, given a call during execution of a logic program, where, earlier, the candidate clauses were tried one by one via backtracking, under coinductive logic programming the trying of candidate clauses is extended with yet more alternatives: applying the coinductive hypothesis rule to check if the current call will unify with any of the earlier calls. The coinductive hypothesis rule will work for only those infinite proofs that are *regular* in nature, i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved infinite number of times (such as a circular linked list). More general implementations of coinduction are possible [70].

Even with regular proofs, there are many applications of coinductive logic programming, some of which are discussed next. These include model checking, concurrent logic programming, real-time systems, non-monotonic reasoning, etc. Note that to implement coinductive LP, one needs to remember in a memo-table (memoize) all the calls made to coinductive predicates.

Finally, note that one has to be careful when using both inductive and coinductive predicates together, since careless use can result in interleaving of least fixed point and greatest fixed point computations. Such programs cannot be given meaning easily. Consider the following program where the predicate `p` is coinductive and `q` is inductive.

```
p :- q.
q :- p.
```

For computing the result of goal `?- q.`, we will use lfp semantics, which will produce null, implying that `q` should fail. Given the goal `?- p.` now, it should also fail, since `p` calls `q`. However, if we use gfp semantics (and the coinductive hypothesis computation rule), the goal `p` should succeed, which, in turn, implies that `q` should succeed. Thus, naively mixing coinduction and induction leads to contradictions. This contradiction is resolved by disallowing such cyclical nesting of inductive and coinductive predicates, i.e., *stratifying* inductive and coinductive predicates in a program. An inductive predicate in a given strata cannot call a coinductive predicate in a higher strata and vice versa [71, 70].

3.3 Examples

Next, we illustrate coinductive Logic Programming via more examples [72]. In addition to allowing infinite terms, the operational semantics of co-LP allows for an execution to succeed when it encounters a subgoal that unifies with an ancestor subgoal (coinductive hypothesis rule). Note that while this is somewhat similar to tabled logic programming in that called atoms are recorded so as to avoid unnecessary redundant computation, the difference is that co-LP's memo'ed atoms represent a (dynamically generated) coinductive hypothesis, while tabled logic programming's table represents a list of results for each called goal in the traditional inductive semantics. Hence the memo'ed atoms in co-LP correspond to a *dynamically generated* coinductive hypothesis.

Infinite Streams: The following example involves a combination of an inductive predicate and a coinductive predicate. By default, predicates are inductive, unless indicated otherwise. Consider the execution of the following program, which defines a predicate that recognizes infinite streams of natural numbers. Note that only the `stream/1` predicate is coinductive, while the `number/1` predicate is inductive.

```
:- coinductive stream/1.
stream([ H | T ]) :- number(H), stream(T).
number(0).
number(s(N)) :- number(N).
| ?- stream([ 0, s(0), s(s(0)) | T ]).
```

The following is an execution trace, for the above query, of the memoization of calls by the operational semantics. Note that calls of `number/1` are not memo'ed because `number/1` is inductive.

```
MEMO: stream([ 0, s(0), s(s(0)) | T ])
MEMO: stream([ s(0), s(s(0)) | T ])
MEMO: stream([ s(s(0)) | T ])
```

The next goal call is `stream(T)`, which unifies with the first memo'ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with the infinite solution:

```
T = [ 0, s(0), s(s(0)) | T ]
```

The user could force a failure here, which would cause the goal to be unified with the next two matching memo'ed ancestor producing `T = [s(0),s(s(0))|T]` and `T = [s(s(0))|T]` respectively. If no remaining memo'ed elements exist, the goal is memo'ed, and expanded using the coinductively defined clauses, and the process repeats—generating additional results, and effectively enumerating the set of (rational) infinite lists of natural numbers that begin with the prefix `[0,s(0),s(s(0))]`.

The goal `stream(T)` is true whenever `T` is some infinite list of natural numbers. If `number/1` was also coinductive, then `stream(T)` would be true whenever `T` is a list containing either natural numbers or ω, i.e., infinity, which is represented as an infinite application of successor `s(s(s(...)))`. Such a term has a finite representation as `X = s(X)`.

Note that excluding the occurs check is necessary as such structures have a greatest fixed-point interpretation and are in the co-Herbrand Universe. This is in fact one of the benefits of coinductive LP. Unification without occurs check is typically more efficient than unification with occurs check, and now it is even possible to define non-trivial predicates on the infinite terms that result from such unification, which are not definable in LP with rational trees. Traditional logic programming's least Herbrand model semantics requires SLD resolution to unify with occurs check (or lack soundness), which adversely affects performance in the common case. Coinductive LP, on the other hand, has a declarative semantics that allows unification without doing occurs check, and it also allows for non-trivial predicates to be defined on infinite terms resulting from such unification.

List Membership: This example illustrates that some predicates are naturally defined inductively, while other predicates are naturally defined coinductively. The `member/2` predicate is an example of an inherently inductive predicate.

```
member(H, [ H | _ ]).

member(H, [ _ | T ]) :- member(H, T).
```

If this predicate was declared to be coinductive, then `member(X, L)` is true whenever `X` is in `L` or whenever `L` is an infinite list, even if `X` is not in `L`! The definition above, whether declared coinductive or not, states that the desired element is the last element of some prefix of the list, as the following equivalent reformulation of `member/2`, called `membera/2` demonstrates, where `drop/3` drops a prefix ending in the desired element and returns the resulting suffix.

```
    membera(X, L) :- drop(X, L, _).
    drop(H, [ H | T ], T).
    drop(H, [ _ | T ], T1) :- drop(H, T, T1).
```

When the predicate is inductive, this prefix must be finite, but when the predicate is declared coinductive, the prefix may be infinite. Since an infinite list has no last element, it is trivially true that the last element unifies with any other term. This explains why the above definition, when declared to be coinductive, is always true for infinite lists regardless of the presence of the desired element.

A mixture of inductive and coinductive predicates can be used to define a variation of `member/2`, called `comember/2`, which is true if and only if the desired element occurs an infinite number of times in the list. Hence it is false when the element does not occur in the list or when the element only occurs a finite number of times in the list. On the other hand, if `comember/2` was declared inductive, then it would always be false. Hence, coinduction is a necessary extension.

```
 :- coinductive comember/2.

 comember(X, L) :- drop(X, L, L1), comember(X, L1).

 ?- X = [ 1, 2, 3 | X ], comember(2, X).
```

```
        Answer: yes.

?- X = [ 1, 2, 3, 1, 2, 3 ], comember(2, X).
        Answer: no.

?- X = [ 1, 2, 3 | X ], comember(Y, X).
        Answer: Y = 1;
                Y = 2;
                Y = 3;
```

Note that `drop/3` will have to be evaluated using OLDT tabling for it not to go into an infinite loop for inputs such as `X = [1,2,3|X]` (if `X` is absent from the list `L`, the lfp of `drop(X,L)` is null).

List Append: Let us now consider the definition of standard `append` predicate.

```
append([], X, X).
append([H|T], Y, [H|Z]) :- append(T, Y, Z).
```

Not only can the above definition append two finite input lists, as well as split a finite list into two lists in the reverse direction, it can also append infinite lists under coinductive execution. It can even split an infinite list into two lists that, when appended, equal the original infinite list. For example:

```
| ?- Y = [4, 5, 6, | Y], append([1, 2, 3], Y, Z).

        Answer:  Z = [1, 2, 3 | Y], Y = [4, 5, 6, | Y]
```

More generally, the coinductive append has interesting algebraic properties. When the first argument is infinite, it doesn't matter what the value of the second argument is, as the third argument is always equal to the first. However, when the second argument is infinite, the value of the third argument still depends on the value of the first. This is illustrated below:

```
| ?- X = [1, 2, 3 | X], Y = [3, 4 | Y], append(X, Y, Z).

        Answer:  Z = [1, 2, 3 | Z], X = [1,2,3|X], Y = [3,4|Y]
```

The coinductive append can also be used to split infinite lists as in:

```
| ?- Z = [1, 2 | Z], append(X, Y, Z).
        Answers: X = [], Y = [1, 2 | Z], Z = [1, 2 | Z];
                 X = [1], Y = [2 | Z], Z = [1, 2 | Z];
                 X = [1, 2], Y = Z, Z = [1, 2 | Z];
                 X = [1, 2 | X], Y = _, Z = [1, 2 | Z];
                 X = [1, 2, 1], Y = [2 | Z], Z = [1, 2 | Z];
                 X = [1, 2, 1, 2], Y = Z, Z = [1, 2 | Z];
                 X = [1, 2, 1, 2 | X], Y = _, Z = [1, 2 | Z];

                 ......
```

Note that application of the coinductive hypothesis rule will produce solutions in which `X` gets bound to an infinite list (fourth and seventh solutions above). Coinductive LP further extends LP with rational trees, in that terms need not be rational, as demonstrated by the term `[0,1,...]` mentioned above, which has infinitely many distinct subterms.

Sieve of Eratosthenes: Coinductive LP also allows for lazy evaluation to be elegantly incorporated into Prolog. Lazy evaluation allows for manipulation of, and reasoning about, cyclic and infinite data structures and properties. Lazy evaluation can be put to fruitful use in situations where only a finite part of the infinite term is of interest. In the presence of coinductive LP, if the infinite terms involved are rational, then given the goal `p(X), q(X)` with coinductive predicates `p/1` and `q/1`, then `p(X)` can coinductively succeed and terminate, and then pass the resulting `X` to `q(X)`. If `X` is bound to an infinite irrational term during the computation, then `p` and `q` must be executed in a coroutined manner to

Table 3.2. Sieve of Eratosthenes

```
:- coinductive sieve/2, filter/3, comember/2.

primes(X) :-  generate_infinite_list(I),
              sieve(I,L), comember(X,L).

sieve([H|T], [H|R]) :- filter(H,T,F), sieve(F,R).

filter(H,[],[]).
filter(H,[K|T],[K|T1]) :-
               R is K mod H, R > 0, filter(H,T,T1).
filter(H,[K|T],T1) :- 0 is K mod H, filter(H,T,T1).
```

produce answers. That is, one of the goals must be declared the producer of `X` and the other the consumer of `X`, and the consumer goal must not be allowed to bind `X`. Consider the (coinductive) lazy logic program for the sieve of Eratosthenes shown in Table 3.2.

In this program `filter/3` removes all multiples of the first element in the list, and then passes the filtered list recursively to `sieve/2`. If the call *generate_infinite_list(I)* binds `I` to an inductive or rational list (e.g., `X = [2, ..., 20]` or `X = [2, .., 20 | X]`), then filter can be *completely* processed in each call to `sieve/2`. However, in contrast, if `I` is bound to an irrational infinite list as in:

```
:- coinductive int/2.
int(X, [X|Y]) :- X1 is X+1, int(X1, Y).
generate_infinite_list(I) :- int(2,I).
```

then in `primes/1` predicate, the calls `generate_infinite_list/1`, `comember/2`, and `sieve/2` should be co-routined, and likewise, in the `sieve/2` predicate, the calls `filter/3` and the recursive call `sieve/2` must be coroutined.

3.4 Implementation of Co-Logic Programming

In the coinductive LP paradigm the declarative semantics of a co-inductive predicate is given in terms of *infinitary Herbrand universe, infinitary Herbrand base, and maximal models (computed using greatest fixed-points)*. The operational semantics is given in terms of the *coinductive hypothesis rule* which states that during execution, if the current resolvent R contains a call C' that unifies with a call C encountered earlier, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension a clause such as `p([1|T]) :- p(T)` and the query `p(Y)` will produce an infinite answer `Y = [1|Y]`.

Note that to implement coinductive LP, one needs to remember in a memo-table (memoize) all the calls made to coinductive predicates.

The operational semantics allows for a co-inductive recursive call to terminate (coinductively succeed) if it *unifies* with an ancestor call. The following example illustrates the execution of a CoLP program based on this operational semantics. The stream predicate below recognizes infinite streams of natural numbers. Note that only the `stream/1` predicate is coinductive, while the `number/1` predicate is inductive.

```
:- coinductive stream/1.
stream( [ H | T ] ) :- number( H ), stream( T ).
number( 0 ).
number( s(N) ) :- number( N ).


| ?- stream( [ 0, s(0), s(s(0)) | T ] ).
```

The following is an execution trace, for the above query, of the memoization of calls by the operational semantics. Note that calls of `number/1` are not memo'ed because `number/1` is inductive.

```
MEMO: stream( [ 0, s(0), s(s(0)) | T ])
MEMO: stream( [ s(0), s(s(0)) | T ] )
MEMO: stream( [ s(s(0)) | T ] )
```

The next goal call is `stream(T)`, which unifies with the first memo'ed ancestor, and therefore immediately succeeds. Hence the original query succeeds with the infinite solution:

```
T = [ 0, s(0), s(s(0)) | T ]
```

The user could force a failure here, which would cause the goal to be unified with the next two matching memo'ed ancestor producing `T = [s(0),s(s(0))|T]` & `T = [s(s(0))|T]` respectively. If no remaining memo'ed elements exist, the goal is memo'ed, and expanded using the coinductively defined clauses, and the process repeats—generating additional results, and effectively enumerating the set of (rational) infinite lists of natural numbers that begin with the prefix `[0,s(0),s(s(0))]`.

The goal `stream(T)` is true whenever `T` is some infinite list of natural numbers. If `number/1` was also coinductive, then `stream(T)` would be true whenever `T` is a list containing either natural numbers or ω, i.e., infinity, which is represented as an infinite application of successor `s(s(s(...)))`. Such a term has a finite representation as `X = s(X)`.

Table 3.3 presents our code for a high-level implementation of co-LP atop YAP prolog engine. The simplicity of the implementation technique makes it possible to integrate this implementation with any existing prolog implementation. The efficiency of executing co-LP programs can be improved greatly by realizing the implementation technique, mentioned above, at the WAM level. The following are the ideas we have to realize a more efficient low-level (WAM level) implementation of co-LP:

1. The inductive predicates are realized by the syntax and semantics of normal Prolog

Table 3.3. Implementation of co-LP atop YAP Prolog

```
:- write_depth(10,10).
:- dynamic coinductive/4.
:- dynamic inductive/4.
:- nb_setval(global_hypo, []).

%------------------------------------------------------
coinductive(F/N) :-
  functor(S,F,N), atom_concat(coinductive_,F,NF),
  functor(NS,NF,N), match_args(N,S,NS),
  atom_concat(stack_,F,SFn), atomic_concat(SFn,N,SF),
  nb_setval(SF, []),
  assert((S :- b_getval(SF,L),
               (in_stack(S,L), co_success;
                \+ in_stack(S,L), b_setval(SF,[S|L]),
                  b_getval(global_hypo, GPSL),
                  b_setval(global_hypo,[S|GPSL]), !, NS);
                NS))),
  assert(coinductive(S,F,N,NS)).

inductive(F/N) :-
  functor(S,F,N), atom_concat(inductive_,F,NF),
  functor(NS,NF,N), match_args(N,S,NS),
  atom_concat(stack_,F,SFn), atomic_concat(SFn, N, SF),
  nb_setval(SF, []),
  assert((S :- b_getval(SF,L),
               not_in_stack(S,L), b_setval(SF,[S|L]), NS)),
  assert(inductive(S,F,N,NS)).

match_args(0,_,_) :- !.
match_args(I,S1,S2) :- arg(I,S1,A), arg(I,S2,A),
                       I1 is I-1, match_args(I1,S1,S2).
co_success.

%------------------------------------------------------
term_expansion((H:-B),(NH:-B) ) :-  coinductive(H,_F,_N,NH), !.
term_expansion(H,NH) :- coinductive(H,_F,_N,NH), !.

term_expansion((H:-B),(NH:-B) ) :- inductive(H,_F,_N,NH), !.
term_expansion(H,NH) :- inductive(H,_F,_N,NH), !.

%------------------------------------------------------
in_stack(G,[G|_]).
in_stack(G,[_|T]) :- in_stack(G,T).

not_in_stack(_G,[]).
not_in_stack(G,[H|T]) :- G \== H, not_in_stack(G,T).
```

predicates. A co-inductive predicate is declared via a directive of the form

`:- coinductive p/n.`

where `p` is the predicate name and `n` its arity. All other predicates are treated as inductive by default.

2. A choice point is stored for every call of a co-inductive predicate.
3. A data-structure that maintains a list of pointers to the corresponding choice points for every co-inductive predicate is maintained.
4. Instead of “Call” which is used for inductive predicates, a new WAM instruction “CoCall” is used to call co-inductive predicates. A “CoCall” instruction first tries to unify the current call to the co-inductive predicate with the previous (ancestor) calls of the same predicate. It does so by retrieving the list of pointers to the choice points for the predicate from the above mentioned data-structure and then unifying the current call to each of the previous call (by unifying the arguments of the current call to those of the previous call stored in the choice point). If the unification is successful with any of the previous call, then it succeeds, else it stores the address of the current choice point in the corresponding list in the data-structure, and then behaves like the regular ’Call’ instruction.

Note that, even with the above mentioned implementation, a co-LP program (having both inductive and co-inductive predicates) works only if the program is *stratified* w.r.t. inductive and coinductive predicates. That is, an inductive predicate in a given strata cannot call a coinductive predicate in a higher strata and vice versa.

We expect that this implementation of co-LP will work with any tabled logic programming implementation as well, because co-inductive predicates are never tabled. So they work as non-tabled predicates and thus have no effect on the completion algorithm of the tabled prolog implementation.

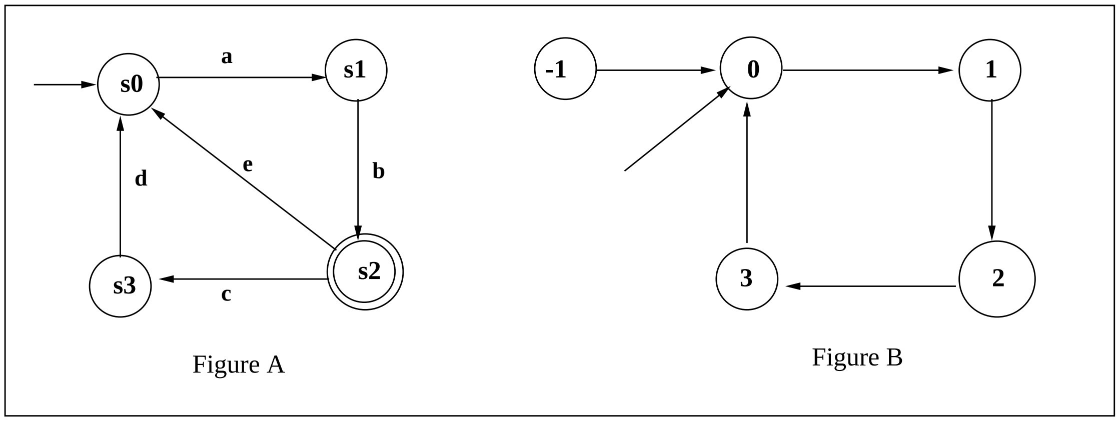

Figure 3.1. Example of a regular and an ω-automaton

3.5 Application to Model Checking and Verification

Model checking is a popular technique used for verifying hardware and software systems. It works by constructing a model of the system in terms of a finite state Kripke structure and then determining if the model satisfies various properties specified as temporal logic formulae. The verification is performed by means of systematically searching the state space of the Kripke structure for a counter-example that falsifies the given property. The vast majority of properties that are to be verified can be classified into *safety* properties and *liveness* properties. Intuitively, safety properties are those which assert that 'nothing bad will happen' while liveness properties are those that assert that 'something good will eventually happen.'

An important application of coinductive LP is in directly representing and verifying properties of Kripke structures and ω-automata (automata that accept infinite strings). Just as automata that accept finite strings can be directly programmed using standard LP, automata that accept infinite strings can be directly represented using coinductive LP (one merely has to drop the base case). Consider the automata (over finite strings) shown in Figure 3.1(A) which is represented by the logic program shown in Table 3.4. A call to `?- automata(X, s0)` in a standard LP system will generate all finite strings accepted

Table 3.4. Encoding of a Regular Automaton

```
automata([X|T], St) :-
    trans(St, X, NewSt), automata(T, NewSt).
automata([], St) :- final(St).

trans(s0, a, s1).
trans(s1, b, s2).
trans(s2, c, s3).
trans(s3, d, s0).
trans(s2, e, s0).

final(s2).
```

by this automata. Now suppose we want to turn this automata into an ω-automata, i.e., it accepts infinite strings (an infinite string is accepted if states designated as final state are traversed infinite number of times), then the (coinductive) logic program that simulates this automata can be obtained by simply dropping the base case (for the moment, we'll ignore the requirement that final-designated states occur infinitely often; this can be easily checked by `comember/2`).

```
automata([X|T],St) :- trans(St,X,NewSt), automata(T,NewSt).
```

Under coinductive semantics, posing the query `| ?- automata(X, s0).` will yield the solutions:

```
X = [a, b, c, d | X];
X = [a, b, e | X];
```

This feature of coinductive LP can be leveraged to directly and elegantly verify liveness properties in model checking, multi-valued model checking, for modeling and verifying properties of timed ω-automata, checking for bisimilarity, etc.

3.5.1 Verifying Liveness Properties

It is well known that safety properties can be verified by reachability analysis, i.e, if a counter-example to the property exists, it can be finitely determined by enumerating all the reachable states of the Kripke structure. Verification of safety properties amounts to computing least fixed-points and thus is elegantly handled by standard LP systems extended with tabling [63]. Verification of liveness properties under such tabled LP systems is however problematic. This is because counterexamples to liveness properties take the form of infinite traces, which are semantically expressed as greatest fixed-points. These infinite traces have a ‘lasso’-like representation [69] that consists of a loop. Since traditional tabled LP systems are designed to not get trapped in loops by explicitly recognizing them and ignoring them, it is not possible to search for such counterexamples directly using standard tabled LP methods. Tabled LP systems [63] work around this problem by transforming the temporal formula denoting the property into a semantically equivalent least fixed-point formula, which can then be executed as a tabled logic program. This transformation is quite complex as it uses a sequence of nested negations.

In contrast, coinductive LP can be directly used to verify liveness properties. Coinductive LP can directly compute counterexamples using greatest fixed-point temporal formulae without requiring any transformation. Intuitively, a state is not live if it can be reached via an infinite loop (cycle). Liveness counterexamples can be found by (coinductively) enumerating all possible states that can be reached via infinite loops and then by determining if any of these states constitutes a valid counterexample. Consider the example of a modulo-4 counter, adapted from [69] (See Figure 3.1(B)). The counter has initial state s_0; all other states are reachable from the state s_{-1}, but this state itself is not reachable from other states, since the counter wraps around from state s_3 to s_0. For correct operation of the counter, we must verify that along every path the state s_{-1} is not reached, i.e., there is at least one infinite trace of the system along which s_{-1} never occurs. This property is naturally specified as a greatest fixed-point formula and can be verified coinductively. A

Table 3.5. Encoding of a Modulo-4 counter

```
:- coinductive s0/2, s1/2, s2/2, s3/2, sm1/2.

sm1(N,[sm1|T]) :-
                N1 is N+1 mod 4, s0(N1,T), N1>=0.
s0(N,[s0|T]) :- N1 is N+1 mod 4, s1(N1,T), N1>=0.
s1(N,[s1|T]) :- N1 is N+1 mod 4, s2(N1,T), N1>=0.
s2(N,[s2|T]) :- N1 is N+1 mod 4, s3(N1,T), N1>=0.
s3(N,[s3|T]) :- N1 is N+1 mod 4, s0(N1,T), N1>=0.
```

simple coinductive logic program S_P to solve the problem is shown in Table 3.5. We compose the counter program with the negation of the property, i.e., `N1 >= 0`. Note that `sm1` represents the state corresponding to `-1`.

The counter is coded as a cyclic program that loops back to state s_0 via states s_1, s_2 and s_3. State s_{-1} represents a state where the counter has the value -1. The property P to be verified is whether the state s_{-1} is live. The query `:- sm1(-1,X), comember(sm1,X)` where the `comember` predicate coinductively checks that `sm1` occurs in `X` infinitely often, will fail implying inclusion of the property in the model, i.e., the absence of a counterexample to the property. The benefit of our approach is that we do not have to transform the model into a form amenable to safety checking. This transformation is expensive in general and can reportedly increase the time and memory requirements by 6-folds [69].

This direct approach to verifying liveness properties also applies to *multi-valued model checking* of the μ-calculus [46]. Multi-valued model checking is used to model systems, whose specification has varying degrees of inconsistency or incompleteness. Earlier effort [46] verified liveness properties by computing the *gfp* which was found using nega-

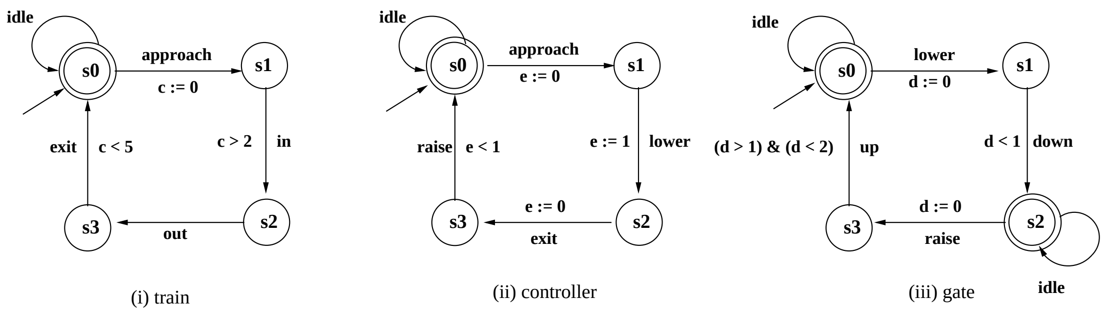

Figure 3.2. Train-Gate-Controller Timed Automata

tion based transformation described earlier. With coinduction, the *gfp* can be computed directly as in standard model checking as described above. Coinductive LP can also be used to check for *bisimilarity*. Bisimilarity is reduced to coinductively checking if two ω-automata accept the same set of rational infinite strings.

3.5.2 Verifying Properties of Timed Automata

Timed automata are simple extensions of ω-automata with stopwatches [3], and are easily modeled as coinductive logic programs with CLP(R) [32]. Timed automata can be modeled with coinductive logic programs together with constraints over reals for modeling clock constraints. The coinductive logic program with CLP(R) constraints for modeling the classic train-gate-controller problem [70] is shown below. This program runs on our implementation of co-LP atop YAP prolog[17] extended with CLP(R). The system can be queried to enumerate all the infinite strings that will be accepted by the automata and that meet the time constraints. Safety and liveness properties can be checked by negating those properties, and checking that they fail for each string accepted by the automata with the help of `comember/2` predicate.

The code for the timed automata represented in Fig 3.2 is given in Table 3.6. The predicate `driver/4`, that composes the three automata, is coinductive, as it executes forever.

Table 3.6. Implementation of Train-Gate-Controller Automata using co-LP

```
:- use_module(library(clpr)).
:- coinductive(driver/4).

init_Gvars :- nb_setval(wallClock, 0), nb_setval(trainClock, 0),
              nb_setval(gateClock, 0), nb_setval(contrClock, 0),
              nb_setval(result, []).
:- init_Gvars.

%---------------------------------------------------------------
train(s0,approach,s1, T1,_T2,T3) :- {T3 = T1}.
train(s1,in,      s2, T1, T2,T3) :- {T1 - T2 > 2, T3 = T2}.
train(s2,out,     s3,_T1, T2,T2).
train(s3,exit,    s0, T1, T2,T3) :- {T3 = T2, T1 - T2 < 5}.
train(X,lower,X,_T1,T2,T2).        train(X,down, X,_T1,T2,T2).
train(X,raise,X,_T1,T2,T2).        train(X,up,   X,_T1,T2,T2).

gate(s0,lower,s1,T1,_T2,T3) :- {T3 = T1}.
gate(s1,down, s2,T1, T2,T3) :- {T3 = T2, T1 - T2 < 1}.
gate(s2,raise,s3,T1,_T2,T3) :- {T3 = T1}.
gate(s3,up,   s0,T1, T2,T3) :- {T3 = T2, T1-T2 > 1, T1-T2 < 2}.
gate(X,approach,X,_T1,T2,T2).      gate(X,in,   X,_T1,T2,T2).
gate(X,out,     X,_T1,T2,T2).      gate(X,exit, X,_T1,T2,T2).

contr(s0,approach,s1,T1,_T2,T3) :-  {T3 = T1}.
contr(s1,lower,   s2,T1, T2,T3) :- {T3 = T2, T1 - T2 = 1}.
contr(s2,exit,    s3,T1,_T2,T1).
contr(s3,raise,   s0,T1, T2,T2) :- {T1-T2 < 1}.
contr(X,in,  X,_T1,T2,T2).         contr(X,out, X,_T1,T2,T2).
contr(X,up,  X,_T1,T2,T2).         contr(X,down,X,_T1,T2,T2).

driver(S0,S1,S2,[X|Rest]) :-
        b_getval(wallClock, T),  b_getval(trainClock, T0),
        b_getval(gateClock, T1), b_getval(contrClock, T2),
        train(S0,X,S00,T,T0,T00),
        gate( S1,X,S10,T,T1,T10),
        contr(S2,X,S20,T,T2,T20),
        {TA > T},
        b_setval(wallClock, TA),  b_setval(trainClock, T00),
        b_setval(gateClock, T10), b_setval(contrClock, T20),
        driver(S00,S10,S20,Rest),
        b_getval(result,R), b_setval(result,[f(X,T)|R]).

append([],L,L).
append([X|A],B,[X|C]) :- append(A,B,C).

getResult(R) :- b_getval(result,R).
```

Table 3.7. Infinite Lists accepted by Train-Gate-Controller Automata

```
R = [(approach,A),(lower,B),(down,C),(in,D),(out,E),
                        (exit,F),(raise,G),(up,H)|R],
X = [approach,lower,down,in,out,exit,raise,up | X] ? ;

R= [(approach,A),(lower,B),(down,C),(in,D),(out,E),
          (exit,F),(raise,G),(approach,H),(up,I)|R],
X = [approach,lower,down,in,out,exit,raise,approach,up|X] ? ;

no
```

Given the query:

```
| ?- driver(s0, s0, s0, X), getResult(R).
```

Table 3.7 shows the infinite lists we obtain, as answers (`A, B, C, ..`, etc. are the time on the wall clock when the corresponding event occurs).

Table 3.8 gives the code for various safety and liveness properties of the Train-Gate-Controller automata. A brief description of each property is given next.

The `downbeforein` predicate can be used to check the safety property that the signal `down` occurs before `in` by checking that the infinite list `Y = [down, in | Y]` is coinductively contained in the infinite string `X` above. This ensure that the system satisfies the safety property, namely, that the gate is down before the train is in the gate area.

A similar approach is used to verify the liveness property, namely that the gate will eventually go up [32], by finding the maximum difference between the times the gate goes down and later comes up. The `downmorethan10` predicate encodes the check that the gate can never be down for more than 10 units of time. This encodes the liveness property that the gate will eventually go up.

Table 3.8. Safety and Liveness Properties for Train-Gate-Controller Automata

```
%Property 1: check that the gate will always be "down"
%when the train is "in" subject to the timing constra-
%ints. A call to "downbeforein" will produce the answer
%"no" since the property is negated in the query.

downbeforein :- driver(s0,s0,s0,_), b_getval(result,R),
                append(A, [f(down,_)|_], R),
                append(_, [f(in,_)|_], A).

%-------------------------------------------------------
%Property 2: check that the gate can never be down more
%than 10 units of time.  A call to downmorethan10 will
%produce the answer "no" since the property is negated
%in the query.

downmorethan10 :-
    driver(s0,s0,s0,_),
    b_getval(result,R), append(A, [f(up,T2) |_], R),
    append(_, [f(down,T1)|_], A), {10 < T2 - T1}.

%--------------------------------------------------------
%Property 3: Find the upper and lower bounds on the time
%the gate can be down (assuming there is only one train).
%Variables M and N denote the Max time and the Min time.
%More precisely this query will find the upper and lower
%bounds on the times at which two approach signals can
%occur with no other intervening approach signal.
%Calling "findrange(M,N)" on sicstus will produce:
%          N < 7,  M > 1, M > N
%This means that the largest value that difference of
%T2 and T1 can have is 7, while the smallest value it
%can have is larger than 1.

findrange(M,N) :-
    driver(s0,s0,s0,_), b_getval(result,R),
    append(A, [f(up,T2) |_], R),
    append(_, [f(down,T1)|_], A),
    {N < T2 - T1,  T2 - T1 < M, M > 0, N > 0}.
```

From the answers to the above mentioned queries of the driver predicate, it is easy to see that another train can approach before the gate goes up. This behavior, however, is entirely consistent as the gate goes up and comes down again, by the time the second train arrives in the gate area (see Figure 3.2). We can find out that the minimum time that must elapse between two trains for the system to be safe by finding the minimum value of the time that elapses between two approach signals with the above constraints. Predicate `findrange(M,N)` computes the answer to be 7 units of time.

3.5.3 Verification of Nested Finite and Infinite Automata

Next, we present the application of co-logic programming to verification of a nested infinite (coinductive) and finite (inductive) automata. It is well known that reachability-based (inductive) techniques are not suitable for verifying liveness properties [60]. Further, it is also well known that, in general, verification of liveness properties can be reduced to verification of termination under the assumption of fairness [82]. Fairness properties can be specified in terms of alternating fixed-point temporal logic formulas [44]. Earlier we showed that co-LP allows one to verify a class of liveness properties in the absence of fairness constraints. Co-LP further permits us to verify a more general class of all liveness properties that can only be verified in the presence of fairness constraints.

A co-LP based approach shows that if a model satisfies the fairness constraint, then it also satisfies the liveness property. This is achieved by composing a program, P_M, which encodes the model with a program, P_F, which in turn encodes the fairness constraint and a program, P_{NP}, which encodes the negation of the liveness property, to obtain a composite program, P_μ. We then compute the stratified alternating fixed-point of the logic program P_μ and check for the presence of the initial state of the model in it. If the stratified alternating fixed-point contains the initial state, then that implies the presence of a valid counterexample that violates the given liveness property. On the other hand, if it is empty, then that

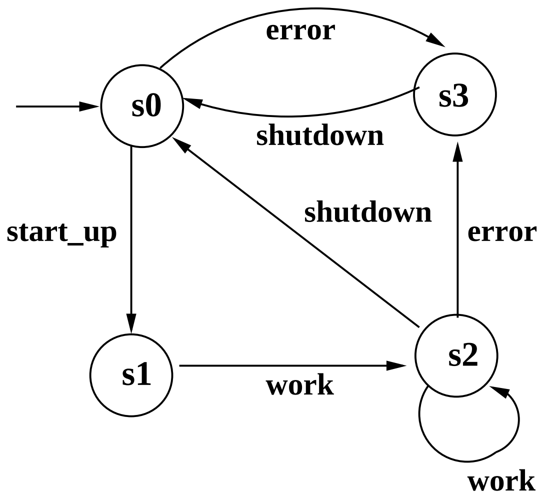

Figure 3.3. Nested Finite and Infinite Automata

implies that no counterexample can be constructed, which in turn implies that the model satisfies the given liveness property.

Consider the model shown in Figure 3.3, consisting of four states. The system starts off in state `s0`, enters state `s1` on *start_up*, and then does some work to transition to state `s2`. It performs a finite amount of work in state `s2` and then on *shutdown*, it transitions back to state `s0`, and repeats the entire loop again, an infinite number of times. The system might encounter an error in either state `s0` or `s2`, causing a transition to state `s3`; corrective action is taken, followed by a transition back to `s0` (this can also happen infinitely often). The system is modeled by the Prolog code shown in Table 3.9.

This simple example illustrates the power of co-logic programming (co-LP, for brevity, that contains both inductive and coinductive LP) when compared to purely inductive or purely coinductive LP. Note that the computation represented by the state machine in the example consists of two stratified loops, represented by recursive predicates. The outer loop (predicate `state/2`) is coinductive and represents an infinite computation (hence it is declared as coinductive as we are interested in its gfp). The inner loop (predicate `work/0`) is inductive and represents a bounded computation (we are interested in its lfp). The se-

Table 3.9. Encoding of a Nested Finite and Infinite Automata

```
:- coinductive state/2.
:- table state/1.
state(s0,[s0|T]) :- start_up, state(s1,T).
state(s1,[s1,is1|T]) :- work, state(s2,T).
state(s2,[s2|T]) :- shutdown, state(s0,T).
state(s2,[s2|T]) :- error,    state(s3,T).
state(s0,[s0|T]) :- error,    state(s3,T).
state(s3,[s3|T]) :- shutdown, state(s0,T).
start_up.
shutdown.
error.
work :- state(is1).
state(is1) :- state(is1).
state(is1).
```

mantics therefore evaluates `work/0` using SLD resolution and `state/2` using co-SLD resolution.

The property that we would like to verify is that if the system *start_up* is successful (i.e., if it transitions to state `s1`), then it always performs some (finite) amount of work in state `s2`. In order to do so, we require the fairness property: “if the transition *start_up* occurs infinitely often, then the transition *shutdown* also occurs infinitely often”. The stratified alternating fixed-point semantics ensures that this fairness constraint holds by computing the minimal model of the inductive program represented by the predicate `state/1` and then composing it with the coinductive program. The resulting program is then composed with the property, “the state `s1` is not present in any trace of the infinite computation,” which is the negation of the given liveness property. The negated property is represented by the predicate `absent/2`. Thus, given the program above, the user will pose the query:

```
| ?- state(s0,X), absent(s1,X).
```

where `absent/2` is a coinductive predicate that checks that the state `s1` is not present in the (infinite) list `X` infinitely often (it is the negated version of the coinductive comember predicate described earlier). The co-LP system will respond with a solution: `X = [s0, s3 | X]`, a counterexample which states that there is an infinite path not containing `s1`. One can see that this corresponds to the (infinite) behavior of the system if `error` is encountered.

3.6 Application to Non-Monotonic Reasoning

We next consider application of coinductive LP to non-monotonic reasoning, in particular its manifestation as *answer set programming* (ASP) [24, 47]. ASP has been proposed as an elegant way of introducing non-monotonic reasoning into logic programming [5]. ASP has been steadily gaining popularity since its inception due to its applications to planning, action-description, AI, etc.

We believe that if one were to add negation as failure to coinductive LP then one would obtain something resembling answer set programming. To support negation as failure in coinductive LP, we have to extend the coinductive hypothesis rule: given the goal `not(G)`, if we encounter `not(G′)` during the proof, and `G` and `G′` are unifiable, then `not(G)` succeeds. Since the coinductive hypothesis rule provides a method for goal-directed execution of coinductive logic programs, it can also be used for goal-directed execution of answer set programs. As discussed earlier, coinduction is a technique for specifying unfounded set; likewise, answer set programs are also recursive specifications (containing negation as failure) for computing unfounded sets. Obviously, coinductive LP and ASP must be related.

All approaches to implementing ASP are based on bottom up execution of finitely grounded programs. If a top-down execution scheme can be designed for ASP, then ASP can be extended to include predicates over general terms. We outline how this can be achieved using our top-down implementation of coinduction. In fact, a top-down interpreter for ASP (restricted to propositions at present) can be trivially realized on top of our implementation of coinductive LP. Work is in progress to implement a top-down interpreter for ASP with general predicates [49].

3.6.1 A Top-Down Algorithm for Computing Answer Sets

In top-down execution of answer set programs, given a (propositional) query goal Q, we are interested in finding out all the answer sets that contain Q, one by one, via backtracking. If Q is not in any answer set or if there are no answer sets at all, then the query should fail. In the former case, the query *not(Q)* should succeed and should enumerate all answer sets which do not contain Q, one by one, via backtracking.

The top down execution algorithm is quite simply realized with the help of coinduction. The traditional Gelfond-Lifschitz (GL) method [5] starts with a candidate answer set, computes a residual program via the GL-transformation, and then finds the lfp of the residual program. The candidate answer set is an answer set, if it equals the lfp of the residual program. Intuitively, in our top down execution algorithm, the propositions in the candidate answer set are regarded as hypotheses which are treated as facts during top-down coinductive execution. A call is said to be a *positive* call if it is in the scope of an even number of negations; similarly, a call is said to be a *negative* call if it is in the scope of an odd number of negations.

The top down algorithm works as follows: suppose the current call (say, p) is a positive call, then it will be placed in a *positive coinductive hypothesis set* (PCHS), a matching rule will be found and the Prolog-style expansion done. The new resolvent will be processed left to right, except that every positive call will be continued to be placed in the positive hypothesis set, while a negative call will be placed in the *negative coinductive hypothesis set* (NCHS). If a positive call p is encountered again, then if p is in PCHS, the call immediately succeeds; if it is in NCHS, then there is an inconsistency and backtracking takes place. If a negative call (say, *not(p)*) is encountered for the first time, *p* will be placed in the NCHS. If a negative proposition *not(p)* is encountered later, then if *p* is in NCHS, *not(p)* succeeds; if *p* is in PCHS, then there is an inconsistency and backtracking takes place. Once the execution is over with success, (part of) the *potential* answer set can be found in the PCHS. The set NCHS contains propositions that are *not* in the answer set.

Essentially, the algorithm explicitly keeps track of propositions that are in the answer set (PCHS) and those that are not in the answer set (NCHS). Any time a situation is encountered in which a proposition is both in the answer set and not in the answer set, an inconsistency is declared and backtracking ensues. In the top down algorithm, whenever a positive (resp. negative) predicate is called, it is stored in the positive (resp. negative) coinductive hypothesis set. This is equivalent to removing the positive (resp. negative) predicate from the body of all the rules, since a second call to predicate will trivially succeed by the principle of coinduction. Given a predicate *p* (resp. *not(p))* where *p* is in the positive (resp. negative) coinductive hypothesis set, if a call is made to *not(p)* (resp. *p*), then the call fails. This amounts to 'removing the rule' in GL transform.

The next chapter describes the above mentioned top-down algorithm in detail and also shows its equivalence with the GL method (using GL Transform) of executing answer-set programs. The top down method described above can also be extended for deducing answers sets of programs containing first order predicates [49].

3.7 Conclusions

In this chapter, we gave an introduction to co-logic programming. Practical applications of co-LP to verification and model checking and to non-monotonic reasoning were also discussed. Coinductive reasoning can also be included in first order theorem proving in a manner similar to coinductive LP [49].

CHAPTER 4
ANSWER-SET PROGRAMMING

Answer Set Programming (ASP) has been proposed as an elegant way of introducing non-monotonic reasoning into logic programming. ASP has been steadily gaining popularity since its inception due to its applications to planning, action-description, AI, etc. However, without exception, all of the approaches and current implementations for ASP have not been goal-directed, i.e., they process the entire knowledge base to compute the whole answer set. Such non-goal directed approaches to execution of ASP programs have inherent limitations, a major one being that they only work for finitely groundable programs. Recent discovery of Coinductive Logic Programming (CoLP) has been very promising in solving some of these inherent problems and limitations of current ASP implementation techniques. In this chapter we present a goal-directed, top-down execution mechanism to execute Answer Set Programs that is based on CoLP.

4.1 Introduction

Answer Set Programming (ASP) is an elegant way of developing non-monotonic reasoning applications. ASP has gained wide acceptance in the last 20 years, and considerable research has been done in developing the paradigm as well as its implementations and applications. ASP has been applied to important areas such as planning, scheduling, default reasoning, reasoning about actions [5], etc. Very efficient implementations of ASP have been developed such as DLV [54] and Smodels [55]. However, all implementations of ASP developed so far examine the entire knowledge-base, and compute the whole answer set: i.e., *they are not goal-directed* in the fashion of Prolog. Given an answer set program and a goal `Q`, a goal-directed execution will systematically enumerate—via SLD style call expansions and backtracking—all answer sets that contain the propositions/predicates in `Q`.

In this chapter, we report a goal-directed method for executing answer set programs. The method relies on the technique of *co-inductive logic programming*. Coinductive Logic Programming (CoLP) [71] is a recently developed technique that seems promising for many useful applications. CoLP can be regarded as providing an operational semantics for computing greatest fixpoints (gfp) of logic programs. It has been applied to a number of interesting applications [31] such as reasoning about unfounded sets [35], reasoning about behavioral properties of infinite programs and streams [9, 28], elegant modeling of liveness properties in model checking [46, 31], and incorporating lazy evaluation in LP. In this chapter, we show how CoLP allows us to develop an efficient, goal-directed execution model for A-Prolog, a realization of ASP.

A goal-directed execution method for answering queries for an answer set program has several advantages:

(i) during problem solving, given a theory (knowledge-base) most often we are interested in finding out whether a particular proposition/predicate is true in that theory; we are not usually interested in finding out *everything* that is true in that theory.

(ii) ASP need not be restricted to finitely groundable programs, i.e., programs that only contain propositions.

(iii) it provides an operational semantics to answer set programming—akin to SLD resolution—which is important for usability.

(iv) Current execution strategies rely on an exhaustive search which employs heuristics to reduce the search space; top-down execution can be potentially more efficient as it only explores a small, relevant part of the search space.

(v) If one part of the knowledge-base has inconsistencies, then queries can still be answered as long as the inconsistent part is not invoked.

(vi) While current execution strategies are not goal-directed, frequently, programs are written in a style that is goal directed; thus, if `Q` is the goal we are interested in, the constraint `:- not Q.` is added to ensure that every answer set reported contains `Q`.

(vii) Finally, ASP can be more naturally integrated with other advanced LP technologies such as constraints, tabling, etc.

Our goal-directed execution strategy for ASP is based on two insights:

1. Given rules of the form

   ```
   p :- not q.
   q :- not p.
   ```

   the query `p` can succeed via coinductive reasoning:

   `p` $\rightarrow$ `not q` $\rightarrow$ `not not p` $\rightarrow$ `p` $\rightarrow$ `success`.

   The query `q` can succeed likewise. In general the proposition `p` (resp. `q`) can be reached from the rule head `p` (resp. `q`) through an even number of nots.

2. Constraints of the form `p :- q, r, not p` essentially state that one of `q` or `r` must be false (i.e., `q` or `r` must not belong to the answer set). `p` must not also belong to the answer set (note that, however, if `p` belongs to the answer set through other means, then this rule is inapplicable).

With these observations, we can design a goal-directed top-down strategy. In the rest of the chapter, we develop such a goal-directed strategy, describe its implementation, as well as prove that it is equivalent to the method of Gelfond and Lifschitz. We restrict ourselves to only propositional (grounded) answer set programs, however, we do give an outline of how our method can be extended to answer set programs with full first order predicates. Finally, note that a top-down goal-directed execution strategy for answer set programs has been regarded as impossible by some researchers [6].

4.2 Coinduction and Logic Programming

Recently *coinduction* has been introduced as a technique for reasoning about unfounded sets [35], behavioral properties of programs [9, 28], and proving liveness properties in model checking [46]. Coinduction also serves as the foundation for lazy evaluation [30] and type inference [59] in functional programming as well as for interactive computing [29, 85].

Coinduction is the dual of induction. Induction corresponds to well-founded structures that start from a basis which serve as the foundation for building more complex structures. For example, natural numbers are inductively defined via the base element zero and the successor function. Inductive definitions have 3 components: initiality, iteration and minimality [29]. Thus, the inductive definition of list of numbers is as follows: (i) `[]` (empty list) is a list (initiality); (ii) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (iii) nothing else is a list (minimality). Minimality implies that infinite-length lists of numbers are not members of the inductively defined set of lists of numbers. Inductive definitions correspond to least fixed point interpretations of recursive definitions.

Coinduction eliminates the initiality condition and replaces the minimality condition with maximality. Thus, the coinductive definition of a list of numbers is: (i) `[H|T]` is as a list if `T` is a list and `H` is some number (iteration); and, (ii) the set of lists is the maximal set of such lists. There is no base case in coinductive definitions, and while this may appear circular, the definition is well formed since coinduction corresponds to the greatest fixed point interpretation of recursive definitions (recursive definitions for which gfp interpretation is intended are termed corecursive definitions). Thus, the set of lists under coinduction is the set of all infinite lists of numbers (no finite lists are contained in this set). Note, however, that if we have a recursive definition with a base case, then under coinductive interpretation, the set defined will contain both finite and infinite-sized elements, since in this case the gfp will also contain the lfp. In the context of logic programming, in the presence of coinduction, proofs may be of infinite length. A coinductive proof essentially is an infinite-length proof.

Coinduction has been incorporated in logic programming in a systematic way only recently [72, 70, 31], where an operational semantics—similar to SLD—is given for computing the greatest fixed point of a logic program. This operational semantics called co-SLD relies on a *coinductive hypothesis rule* and systematically computes elements of the gfp of a program via backtracking. The semantics is limited to only *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors.

Consider the list example above. The normal logic programming definition of a stream (list) of numbers is given as program P1 below:

```
stream([]).
stream([H|T]) :- number(H), stream(T).
```

Under SLD resolution, the query `?- stream(X)` will systematically produce all finite streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program P2:

```
stream([H|T]) :- number(H), stream(T).
```

In the program P2, the meaning of the query `?- stream(X)` is semantically null under standard logic programming. The problems are two-fold. The Herbrand universe does not allow for infinite terms such as `X` and the least Herbrand model does not allow for infinite proofs, such as the proof of `stream(X)` in program P2; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [9]. Coinductive LP extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties [72, 71, 70]. In the coinductive LP paradigm, the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary (or co-Herbrand) Herbrand base [45], and maximal models (computed using greatest fixed-points).*

Thus, under coinductive interpretation of P2, the query `?- stream(X)` produces all infinite sized stream as answers, e.g., `X = [1, 1, 1, ... ]`, `X = [1, 2, 1, 2, ... ]`, etc., thus, P2 is not semantically null (but proofs may be of infinite-length).

If we take a coinductive interpretation of program P1, then we get all finite and infinite stream as answers to the query `?- stream(X)`. Coinductive logic programming allows programmers to manipulate infinite structures. As a result, unification has to be necessarily extended and "occurs check" removed. Thus, unification equations such as `X = [1 | X]` are allowed in coinductive logic programming; in fact, such equations will be used to represent infinite (regular) structures in a finite manner.

Even with regular proofs, there are many applications of coinductive logic programming. These include model checking, concurrent logic programming, real-time systems, etc. In this chapter we show how CoLP can be used to realize a goal-directed execution strategy for evaluating answer set programs.

4.3 Answer Set Programming

Answer Set Programming (ASP) [47] (A-Prolog [24] or AnsProlog [5]) is a declarative logic programming paradigm which encapsulates non-monotonic or common sense reasoning. The rules in an ASP program have the following syntax:

p `:-` q_1`, ...,` q_m`, not` r_1`, ..., not` r_n`.`

where $m \geq 0$ and $n \geq 0$. Each p and q_i ($\forall i \leq m$) is a literal, each 'not r_j' ($\forall j \leq n$) is a naf-literal, and 'not' is a logical connective called 'negation as failure' or 'default negation'.

The semantics of an Answer Set program P is given in terms of the answer sets of the program *ground(P)* (which is obtained by grounding the variables in the program P).

Gelfond-Lifschitz Transform (GLT): Given a grounded Answer Set program P and a candidate answer set A, a residual program R is obtained by applying the following transformation rules: for all literals $L \in A$,

1. delete all rules in P which have naf-literal 'not L' in its body.
2. delete all naf-literals of the form 'not L' in the bodies of the remaining rules.

The least fixed-point (say, F) of the residual program R (obtained by applying GLT to grounded P) is computed. If F is equal to A (the candidate answer set for GLT), then A is an answer set for program P.

Answer set programs can also have rules of the following forms:

:- q_1, ..., q_m, not r_1, ..., not r_n.

p :- q_1, ..., q_m, not r_1, ..., not r_n, not p.

These are called constraint rules. These rules capture the non-monotonic aspect of Answer Set Programming. Consider a simple rule of the form `p :- q, not p.` It restricts `q` (and `p`) to not be in the answer set (unless `p` happens to be in the answer set via other rules). Note that, even though the ASP program can have other rules to establish that `q` is in the answer set, addition of this one rule to the program can force `q` to not be in the answer set, thus making ASP non-monotonic.

Consider another example of a simple ASP program.

```
p :- not q.

q :- not p.
```

This program exemplifies the cyclical reasoning that ASP encapsulates. It forces `p` and `q` to be mutually exclusive, i.e., either `p` or `q` is true. It is this cyclical reasoning in

ASP that became our main intuition behind the goal-directed top-down execution strategy for Answer Set Programs, which is described next.

4.4 Goal-directed execution of Answer Set Programs

ASP has been proposed as an elegant way of introducing non-monotonic reasoning into logic programming [5]. ASP has been steadily gaining popularity since its inception due to its applications to planning, action-description, AI, etc. But the non goal-directed/bottom-up approaches to execution of ASP programs are inefficient and have inherent limitations.

4.4.1 Motivation

Current approaches to execute ASP programs examine the entire knowledge-base and compute all possible answer sets. For each answer, the whole answer set is computed. One could argue that computing the whole answer set is an overkill, and frequently one is only interested in checking if a given goal is entailed by the knowledge-base. Current approaches rely on a “guess and check” approach, where propositional symbols in an answer set program are respectively assigned values true and false, and the program checked for consistency for that assignment (however, many implementations, e.g., Smodels, employ extremely smart heuristics and are quite efficient). This “guess and check” approach also dictates that the answer set programs be finitely groundable, i.e., all propositions be known, so that a truth table can be constructed. In contrast, one could desire a goal-directed approach that starts from the query and expands calls in the manner of SLD resolution until success is achieved, or failure takes place. Such a goal-directed strategy will not have to be restricted to propositional answer set programs. A goal-directed top-down approach also provides an *operational semantics* to ASP which, we believe, increases its usability. A goal-directed strategy will only use the “relevant” part of the knowledge-base, thus inconsistencies lurking in the knowledge-base will not scuttle every query. Thus, queries that do not depend on the inconsistent part of the knowledge-base can still terminate.

Note also that while solving practical problems, in most cases the programmer adds special rules to ensure that certain goals are always included in the answer set. These goals can be regarded as the query the user is interested in. Thus, if the user is interested in query `Q`, they will add the constraint `:- not Q.` to ensure that every answer set reported will contain `Q`. The implementation may or may not be equipped to take advantage of this fact.

A goal directed, top-down execution method is a more efficient way of finding answer sets as in this approach we do not generate unnecessary answers sets. We traverse only that part of the search tree where our intended answer set lies. As a goal-directed approach will rely on standard logic programming techniques, it will work well with constraints to give us Constraint ASP. For similar reasons, it can easily be parallelized to make it more efficient using off-the-shelf technology for parallelizing logic programming systems. It can also be incorporated easily into existing Prolog systems. Because of the inherent advantage of being goal directed, top-down approaches to execute programs have always fared better than non goal-directed/bottom-up approaches in terms of efficiency, execution of datalog programs being perhaps the best example.

Our approach to obtaining a goal-directed, top-down execution strategy for ASP relies on adding negation to CoLP. Addition of negation as failure to CoLP brings us one step closer to ASP. To support negation as failure in coinductive LP, we have to extend the coinductive hypothesis rule: given the goal `not(G)`, if we encounter `not(G′)` during the proof, and `G` and `G′` are unifiable, then `not(G)` succeeds (however, for soundness, note that `G` must be ground either to finite terms or to infinite but regular terms). Since the coinductive hypothesis rule provides a method for goal-directed execution of coinductive logic programs, it can also be used for goal-directed execution of answer set programs. As discussed earlier, coinduction is a technique for specifying unfounded sets; likewise, answer set programs are also recursive specifications (containing negation as failure) for computing unfounded sets. Thus, the two are closely related.

All approaches to implementing ASP are based on Gelfond-Lifschitz method (not goal-directed) of executing finitely grounded programs. If a goal-directed execution scheme can be designed for ASP, then ASP can be extended to include predicates over general terms. We outline how this can be achieved using our goal-directed implementation of coinduction. In fact, a goal-directed interpreter for ASP (restricted to propositions at present) can be trivially realized on top of our implementation of coinductive LP. Work is in progress to implement a goal-directed interpreter for ASP with general predicates [49].

4.4.2 A Goal-directed Method for Computing Answer Sets

In goal-directed execution of answer set programs, given a (propositional) query goal `Q`, we are interested in finding out all the answer sets that contain `Q`, one by one, via backtracking. If `Q` is not in any answer set or if there are no answer sets at all, then the query should fail. Otherwise, the query `not(Q)` should succeed and should enumerate all answer sets which do not contain `Q`, one by one, via backtracking. If a query succeeds, the answer set is obtained by finding all the propositions that were true along the branch that lead to success. Note, however, that complete answer set may not be computed, as only those parts of the answer sets will be computed that are needed to conclude that propositions in `Q` are in the answer set.

The goal-directed execution method is quite simply realized with the help of coinduction. Intuitively, the method works by separating constraint rules from non-constraint rules. Non-constraint rules are treated as a coinductive logic program, and used to execute the given query `Q`. This execution produces many answer sets via backtracking. For each potential answer set, the constraints rules are checked to see which ones should be reported as the final answer set.

The traditional Gelfond-Lifschitz (GL) method [5] starts with a candidate answer set, computes a residual program via the GL-transformation, and then finds the lfp of the residual program. The candidate answer set is an answer set if it equals the lfp of the residual program. Intuitively, in our goal-directed execution method, the propositions in the candidate answer set are regarded as hypotheses which are treated as facts during goal-directed coinductive execution. A call is said to be a *positive* call if it is in the scope of an even number of negations; similarly, a call is said to be a *negative* call if it is in the scope of an odd number of negations.

The goal-directed method works as follows: suppose the current call (say, `p`) is a positive call, then it will be placed in a *positive coinductive hypothesis set* (PCHS), a matching rule will be found and the Prolog-style expansion done. The new resolvent will be processed left to right, except that every positive call will be continued to be placed in the positive hypothesis set, while a negative call will be placed in the *negative coinductive hypothesis set* (NCHS). If a positive call `p` is encountered again, then if `p` is in PCHS, the call immediately succeeds; if it is in NCHS, then there is an inconsistency and backtracking takes place. If a negative call (say, `not(p)`) is encountered for the first time, `p` will be placed in the NCHS. If a negative proposition `not(p)` is encountered later, then if `p` is in NCHS, `not(p)` succeeds; if `p` is in PCHS, then there is an inconsistency and backtracking takes place. Once the execution is over with success, (part of) the *potential* answer set can be found in the PCHS. The set NCHS contains propositions that are *not* in the answer set.

Essentially, the method explicitly keeps track of propositions that are in the answer set (PCHS) and those that are not in the answer set (NCHS). Any time, a situation is encountered in which a proposition is both in the answer set and not in the answer set, an inconsistency is declared and backtracking ensues.

We still need one more step. ASP can specify the falsification of a goal via constraints. For example, the constraint `p :- q, not p.` restricts `q` (and `p`) to not be in the answer set (unless `p` happens to be in the answer set via other rules). For rules of the form

```
p :- B.
```

if `not(p)` is reachable via goals in the body `B`, we need to explicitly ensure that the potential answer set does not contain a proposition that is falsified by this rule.

Given an answer set program, a rule `p :- B.` is said to be constraint rule (C-rule) if `p` is reachable through calls in the body `B` through an odd number of negation as failure calls, otherwise it is said to be a non-constraint rule (NC-rule). Thus, given the ASP program:

```
p :- a, not q.                        .......(i)
q :- b, not r.                        .......(ii)
r :- c, not p.                        .......(iii)
q :- d, not p.                        .......(iv)
```

rules (i), (ii) and (iii) are C-rules, while (i) and (iv) are NC-rules. A rule can be both an NC-rule as well as a C-rule (such as rule (i)). NC-rules will be used to compute the potential answers sets, while C-rules will only be used to reject or accept potential answer sets. Rejection or acceptance of a potential answer set is accomplished as follows: For each C-rule of the form $\texttt{r}_i$ `:- B`, where `B` directly or indirectly leads to $\texttt{not(r}_i\texttt{)}$, we construct a new rule:

$$\texttt{chk\_r}_i\ \texttt{:- not(r}_i\texttt{), B.}$$

Next, we construct a new rule:

$$\texttt{nmr\_check :- not(chk\_r}_1\texttt{), not(chk\_r}_2\texttt{), ..., not(chk\_r}_i\texttt{), ...}$$

Now, the top level query, `?-Q`, is transformed into: `?- Q, nmr_check.` `Q` will be executed using only the NC-rules to generate potential answer sets, which will be subsequently either rejected or accepted by the call to `nmr_check`. If `nmr_check` succeeds, the potential answer set is an answer set. If `nmr_check` fails, the potential answer set is rejected and backtracking occurs. Note that during the execution of `nmr_check`, if a proposition `m` is encountered, then coinductive success/failure should be first used to solve it (i.e., check if `m` $\in$ `PCHS` or `m` $\in$ `NCHS`); it should be expanded using other rules only if it cannot succeed or fail coinductively.

For simplicity of illustration, we assume that for each NC-rule, we construct its negated version which will be expanded when a corresponding negative call is encountered (in the implementation, however, this is done implicitly). Thus, given an NC-rule for a proposition `p` of the form:

```
p :- B₁.
p :- B₂.
...
p :- Bᵢ.
...
```

its negated version will be:

```
not_p :- not(B₁), not(B₂), ..., not(Bᵢ), ...
```

If a call to `not(p)` is encountered, then this negated `not_p` rule will be used to expand it.

The goal-directed top-down method for computing the (partial) answer set of an ASP program can be summarized as follows.

1. Initialize PCHS and NCHS to empty (these are maintained as global variables in our implementation of top-down ASP atop our coinductive YAP [17] implementation). Declare every proposition in the ASP as a coinductive proposition.
2. Identify the set of C-rules and NC-rules in the program.
3. Assert `chk_r`$_i$ rule for every C-rule with `r`$_i$ as head and build the `nmr_check` rule; append the call `nmr_check` to the initial query.
4. For each NC-rule, construct its negated version.

5. For every positive call `p`: if `p` $\in$ PCHS, then `p` succeeds coinductively and the next call in the resolvent is executed, else if `p` $\in$ NCHS, then there is an inconsistency and backtracking ensues, else (`p` is not in PCHS or in NCHS) add `p` to PCHS and expand `p` using NC-rules that have `p` in the head (create a choice-point if there are multiple matching rules).
6. For every negative call of the form `not(p)`: if `p` $\in$ NCHS, then `not(p)` succeeds coinductively and the next call in the resolvent is executed, else if `p` $\in$ PCHS, then there is an inconsistency and backtracking ensues, else (`p` is not in PCHS or in NCHS) add `p` to NCHS and expand `not(p)` using the negated `not_p` rule for `p`.

Note, finally, that the answer sets reported may be partial, because an answer set may be a union of multiple independent answer subsets, and the other subsets may not be deducible from the query goal due to the nature of the rules. Next we present some examples to illustrate the above mentioned method.

4.4.3 Examples

Consider the following program:

```
p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).
```

After, step 1-4, we obtain the following program.

```
:- coinductive p/0, q/0, r/0.
```

```
p :- not(q).
q :- not(r).
r :- not(p).
q :- not(p).

chk_p :- not(p), not(q).
chk_q :- not(q), not(r).
chk_r :- not(r), not(p).

not_p :- q.
not_q :- r,p.
not_r :- p.

nmr_chk :- not(chk_p), not(chk_q), not(chk_r).
```

The ASP program above has {`q,r`} as the only answer set. Given the transformed program, the query: `?- q` will produce {`q`} as the answer set (with `p` known to be not in the answer set). The query `?- r` will produce the answer set {`q,r`} (with `p` known to be not in the answer set). It is easy to see why the first query `?- q` will not deduce `r` to be in the answer set: there is nothing in the NC-rules that relates `q` and `r`.

Now let's consider another example. It is believed that a "fully top-down" procedure for the execution of this program is impossible [6].

```
q :- not(r).
r :- not(q).
p :- not(p).
p :- not(r).
```

After, step 1-4, we obtain the following program.

```
:- coinductive p/0, q/0, r/0.

q :- not(r).
r :- not(q).
p :- not(p).
p :- not(r).

chk_p :- not(p).

not_p :- p,r.
not_q :- r.
not_r :- q.

nmr_chk :- not(chk_p).
```

The ASP program above has $\{$`p,q`$\}$ as the only answer set. Given the transformed program, the query: `?- p` will produce $\{$`p,q`$\}$ as the answer set (with `r` known to be not in the answer set). The query `?- q` will also produce the answer set $\{$`p,q`$\}$ (with `r` known to be not in the answer set). In this query, as NC rules do not relate `q` to `p`, `p` is not deduced as part of the candidate answer set. But the evaluation of the `nmr_chk` deduces `p` to be in the answer set.

4.5 Implementation

A prototype implementation of our Goal-directed ASP engine has been realized by extending our implementation of coinductive LP [72], which in turn was realized atop the YAP

Prolog system [17]. It is easier to describe our implementation of goal-directed ASP by describing our implementation of coinductive LP [72] first.

The general operational semantics of coinductive LP [72] allows for a coinductively recursive call to terminate (coinductively succeed) if it *unifies* with a call that has been seen earlier. Predicates have to be declared coinductive via the directive:

```
:- coinductive p/n.
```

where `p` is the predicate name and `n` its arity. When a coinductive call is encountered for the first time, it is recorded in a memo-table that we call coinductive hypothesis. The variables in the recorded call are interpreted w.r.t. the environment of the coinductive call (so effectively the closure of the call is saved). When the same call is encountered later, it is unified with the saved call in the table and made to succeed. Note that everything recorded in the memo-table for a specific coinductive predicate `p` will be deleted, when execution backtracks over the first call of `p`.

4.5.1 A Top-down ASP Engine

With the above mentioned implementation of coLP in place, it is reasonably straightforward to realize our goal-directed ASP engine by extending it. For our implementation of top-down ASP, we need two sets of coinductive hypotheses (PCHS & NCHS) as mentioned in Section 4.4.2. PCHS is used to memo all positive calls (say p) while NCHS is used to record negative calls (say ‘not p’). All the predicates in an ASP program are declared coinductive. For every predicate p in the ASP program, we add ‘not_p’ rules, as described in Section 4.4.2. Also for each predicate p, we assert two special predicates say S_p and nnot(S_p) in the program. S_p is called for every call to p. It first checks if p could coinductively succeed by checking for its presence in PCHS, otherwise it calls the actual rules for p in the program. A call to nnot(S_p) is made whenever ‘not p’ is called. It first checks if ‘not p’ could coinductively succeed by checking if p is present in NCHS, otherwise it calls ‘not_p’.

For efficient checking of coinductive success, we implement a pair of PCHS & NCHS for every predicate in the program.

Table 4.1 presents our code for a high-level implementation of our goal-directed ASP engine atop YAP prolog. The simplicity of the implementation technique makes it possible to integrate this implementation with any existing prolog implementation. It works seamlessly with YAP prolog's CLP engine and its tabling engine (YAPTAB). Thus, it provides a framework for elegant encodings of complex applications which require both constraints and ASP (e.g., Timed Planning) or Tabling and ASP (e.g., Planning and Verification). It can easily be extended to work with any other logic programming extensions. We are looking into realizing a more efficient, low-level implementation by extending YAP prolog's code to support our goal-directed ASP execution strategy. This low-level implementation would not require one to assert the special rules (to handle coinduction) or 'not_p' rule for every predicate p, thereby avoiding any increase in the size of the code. It would realize these rules implicitly at run-time.

4.5.2 Performance Evaluations

We have performed a preliminary assessment of the performance of our prototype goal-directed implementation of ASP on top of our YAP-based coinductive LP engine. This performance evaluation has been done using some of the propositional ASP program examples found in the literature. Our evaluation was run on a Linux system with a 2.80GHz Xeon processor and 2 GB of main memory. The results are quite promising and comparable with Smodels (running with Lparse v1.1.1 [77] and Smodels v2.32 [55]). Programs were grounded using Lparse and then transformed for top-down execution using the method outlined above. Most programs run in a few milliseconds on both the Smodels systems as well as our system. The programs used consisted of:

(i) a program to compute reachability.

(ii) a program to compute the winning move in a game.

Table 4.1. Implementation of goal-directed ASP atop YAP Prolog

```
:- write_depth(10,10).
:- dynamic coinductive/4.
:- nb_setval(positive_hypo, []).
:- nb_setval(negative_hypo, []).

%---------------------------------------------------------------
coinductive(F/N) :-
  atom_concat(coinductive_,F,NF), functor(S,F,N),
  functor(NS,NF,N), match_args(N,S,NS),

  atom_concat(stack_,F,SFn), atomic_concat(SFn,N,SF),
  atom_concat(SF,'_posHypo',SFpos), nb_setval(SFpos,[]),
  atom_concat(SF,'_negHypo',SFneg), nb_setval(SFneg,[]),

  assert((S :- b_getval(SFpos,PSL),
               (in_stack(S,PSL), !; b_getval(SFneg,NSL),
                in_stack(S,NSL), !, fail;
                b_setval(SFpos,[S|PSL]),
                b_getval(positive_hypo, GPSL),
                b_setval(positive_hypo,[S|GPSL]), NS))),
  assert((nnot(S) :- b_getval(SFneg,NSL),
               (in_stack(S,NSL), !; b_getval(SFpos,PSL),
                in_stack(S,PSL), !, fail;
                b_setval(SFneg,[S|NSL]),
                b_getval(negative_hypo, GNSL),
                b_setval(negative_hypo,[S|GNSL]),
                (clause(nt(S),_) -> nt(S); nt_succ)))),
  assert(coinductive(S,F,N,NS)).

%---------------------------------------------------------------
match_args(0,_,_) :- !.
match_args(I,S1,S2) :- arg(I,S1,A), arg(I,S2,A),
                       I1 is I-1, match_args(I1,S1,S2).

term_expansion((H:-B),(NH:-B) ) :- coinductive(H,_F,_N,NH), !.
term_expansion(H,NH) :- coinductive(H,_F,_N,NH), !.

in_stack(G,[G|_]) :- !.
in_stack(G,[_|T]) :- in_stack(G,T).

%---------------------------------------------------------------
ans :- write('Answer Set:'), nl, writeG_val(positive_hypo),
       writeG_val(negative_hypo), nl.

writeG_val(G_var) :- b_getval(G_var, G_val), write(G_var),
                     write(' ==> '), write(G_val), nl.
```

(iii) the Schur number [5] problem of various sizes (largest one being 5 boxes and 12 numbers).

Note that both for Smodels as well as our implementation, during performance evaluation, the query used consisted of a complete answer set. This is because our goal-directed implementation currently does not take advantage of any constraint propagation, while Smodels does. Given the extended query `Q, nmr_check`, one can regard `Q` as the generator, and `nmr_check` as the tester. In our system at present, `Q` is executed in a goal-directed manner first, followed by `nmr_chk`. This can be quite inefficient, as `Q` will produce potential answer sets that `nmr_chk` may reject. `Q` may produce a large number of candidate answer sets that `nmr_chk` may keep rejecting. For efficient performance, `nmr_chk` should be called first, followed by `Q`. However, in such a case, any call in `nmr_chk` that cannot be solved by coinduction, should be solved by call expansion only after `Q` has finished. Thus, if `nmr_chk` is executed first, then any call it makes that cannot coinductively succeed or fail immediately should be suspended. A suspended call should be resumed for coinductive success or failure as soon as a matching proposition appears in PCHS or NCHS. All suspended calls in `nmr_chk` should be executed via call expansion only after the execution of `Q` is over. Note that one can use this clean separation of an ASP program into constraint and non-constraint rules along with our formulation of `nmr_chk` to transform answer set programs written in Smodels syntax to CLP(FD) programs. A goal-directed implementation of answer set programs that gives priority to constraints in `nmr_chk` is in progress.

4.6 Correctness of the Goal-directed Method

Next, we show our intuition behind our goal-directed method to execute ASP programs. First, note that the above mentioned goal-directed method corresponds to computing the greatest fix point of the residual program rather than the least fix point. Second, we argue that the Gelfond-Lifschitz method for checking if a given set is an answer set [24] should compute the greatest fix point of the residual program instead of the least fixed point. A little thought will reflect that circular reasoning entailed by rules such as

```
p :- p.
```

is present in ASP through rules of the form:

```
p :- not(q).
q :- not(p).
```

If we extend the GL method, so that instead of computing the lfp, we compute the gfp of the residual program, then the GL transformation can be modified to remove positive goals as well. Given an answer set program, whose answer set is guessed to be A, the modified GL transform then becomes as follows:

1. Remove all those rules whose body contains `not(p)`, where $p \in A$.
2. Remove all those rules whose body contains p, where $p \notin A$.
3. From body of each rule, remove all goals of the form `not(p)`, where $p \notin A$.
4. From body of each rule, remove all positive goals p, where $p \in A$.

After application of this transform, the residual program is a set of facts. If this set of facts is the same as A, then A is an answer set. It is easy to see that our goal-directed method mimics the modified GL-transform (note, however, that our goal-directed method can be easily modified to work with the original GL method which computes the lfp of the residual program: we merely have to disallow making inference from positive coinductive loops of the form `p :- p`).

In the goal-directed method, whenever a positive (resp. negative) predicate is called, it is stored in the positive (resp. negative) coinductive hypothesis set. This is equivalent to removing the positive (resp. negative) predicate from the body of all the rules, since a second call to predicate will trivially succeed by the principle of coinduction. Given a predicate p (resp. `not(p)`) where p is in the positive (resp. negative) coinductive hypothesis set, if a call is made to `not(p)` (resp. `p`), the call fails. This amounts to 'removing the rule' in GL transform.

The top down method described above can also be extended for deducing answers sets of programs containing first order predicates [49]. With the help of the above mentioned intuition, we next outline a proof of correctness of our goal-directed method. We show our goal-directed execution method is correct by showing its equivalence with Gelfond-Lifschitz method of executing Answer-Set Programs.

Lemma 1

Let P be an Answer Set Program, and $\mathcal{A}$ is a known and correct Answer Set of P. For every proposition p, $p \in \mathcal{A}$ if and only

1. if there exists at least one rule in P of the form
   p :- q_1, ..., q_m, not r_1, ..., not r_n (where $m \geq 0$ and $n \geq 0$)
   s.t., $\forall i \leq m$, $q_i \in \mathcal{A}$ and $\forall j \leq n$, $r_j \notin \mathcal{A}$, and
2. there exists no rule of the form "q :- p, not q" (constraint rule, falsifying p), s.t., $q \notin \mathcal{A}$.

Proof. 1. Let us assume that such a rule does not exist for p in the program P. Let P' be the residual program that is obtained after applying Gelfond-Lifschitz Transform to P. It is easy to see that P' would not contain a rule for p, s.t., the least fixed point of P' would contain p. So $p \notin \mathcal{A}$. But that is a contradiction, as $p \in \mathcal{A}$. So, the above mentioned rule must exist for p in the program P.

2. If a rule C of the form "q :- p, not q" exists in P and $q \notin \mathcal{A}$, then C would prevent p to ever be true, thereby preventing it from being in any answer set.

Once we have Lemma 4.6, it is easy to show the equivalence between our goal-directed execution strategy and Gelfond-Lifschitz method of executing Answer-Set Programs. The next theorem establishes this fact.

Theorem 2

Let P be an Answer Set Program, and $\mathcal{A}$ is a known and correct Answer Set of P. Let q be a proposition in P. Let $PCHS$ be the positive coinductive hypothesis set obtained after executing the above mentioned goal-directed method with q as the query. Then $q \in PCHS$ if and only if $q \in \mathcal{A}$.

Proof. Let P be an Answer Set Program and q a goal proposition that we are interested in establishing with our goal-directed execution of P. Essentially, we want to find an answer set of P that contains q. Let's assume that such an answer set exists and is called A. Since $q \in A$, from Lemma 4.6, we can conclude that there exists at least one rule (call it $rule_q$) in P that is of the form

q :- q_1, ..., q_m, not r_1, ..., not r_n (where $m \geq 0$ and $n \geq 0$),

s.t., $\forall i \leq m$, $q_i \in A$ and $\forall j \leq n$, $r_j \notin A$. This is the rule in P that our goal-directed strategy will use to expand q (we do not need to know which rule in P is $rule_q$ beforehand; if no other rule can establish q, then backtracking will ensure that $rule_q$ is used to expand q). When $rule_q$ is called, q is added to *positive coinductive hypothesis set* (PCHS), and $\forall i \leq m$, each of q_i is called, thereby adding each of them to PCHS. As $\forall i \leq m$, $q_i \in A$ for q to belong to A, from Lemma 4.6, we conclude that for every q_i, there exists at least one rule of the above form. Our goal-directed strategy will use this rule to expand q_i. We also need to establish 'not r_j', $\forall j \leq n$. In process of establishing q, when 'not r_j' is called, r_j is added to *negative coinductive hypothesis set* (NCHS). From Lemma 4.6 we know that for r_j, no rule of the above form exist. So our top down strategy will never be able to establish r_j, $\forall j \leq n$, thereby establishing 'not r_j'. During call expansion one may encounter the same predicate again (for example, in process of establishing q, one may encounter q again by expanding q_i or 'not r_j'. This cycle is broken by succeeding coinductively, i.e., by checking for q in the PCHS. Similarly 'not r_j' can succeed coinductively by checking for r_j in NCHS.

If P contains constraint rules, as described in Section 4.4.2, then for each rule of the form `q :-` $s_1, \ldots, s_k$`, not q`, the call `nmr_chk` contains `not(chk_q)`, where `chk_q :- not(q),` $s_1, \ldots, s_k$`, not(q)`. If `q` $\in$ PCHS, then `chk_q` will fail and the call `not(chk_q)` will succeed in effect discarding the constraint rule for `q`. If `q` $\in$ NCHS then at least one s_i must fail for not(chk_q) to succeed essentially obtaining the effect of falsifying at least one s_i. If `q` is not in PCHS or NCHS, then our goal-directed method will try to expand `q` to determine whether it is true or false. If `q` fails, then it has the same effect as above when `q` $\in$ NCHS. Otherwise, `chk_q` will fail and it will have the same effect as above when `q` $\in$ PCHS. Thus, the way `nmr_chk` is formulated ensures that exactly those propositions that are falsified in the GL method are also falsified during goal-directed execution.

Thus, we can see that our goal-directed execution strategy will only succeed for a proposition if and only if that proposition belongs to an answer set computed using Gelfond-Lifschitz method.

4.7 Conclusions and Future Work

In this chapter we presented a Top-Down method for goal directed execution of Answer Set programs. This is realized by extending Coinductive Logic Programming to execute Answer Set Programs. A goal-directed procedure has many advantages, the chief one being that execution of answer set programs does not have to be restricted to only finitely groundable ones.

W.r.t. related work, a top-down, goal-directed execution strategy has been the aim of many researchers in the past. It has been some times cited as the “holy grail” of answer set programming. Many attempts have been made such as those by Lobo et al. [20]. However, it is widely believed that a goal-directed strategy is impossible to attain [6]. As such, there are not many efforts aimed at designing goal-directed strategies for ASP. Recently, Gebser and Schaub have developed a tableau based method which can be regarded as a step in this direction; however, the motivation for their work is completely different [22].

Our current and future work includes developing a goal directed execution strategy for general answer set programs with predicates over arbitrary terms [49]. The methodology is similar except that the definition of `nmr_chk` is more complex. Additionally, given a general constraint clause of the form:

$$p(\bar{t}) : -...., \texttt{not}\ p(\bar{s}), ...$$

where $\bar{t}$ and $\bar{s}$ are terms, this clause devolves into two clauses corresponding to the two cases: one in which $\bar{s}$ unifies with $\bar{t}$ with mgu θ, and the other in which $\bar{s}$ and $\bar{t}$ do not unify. The former case gives rise to a constraint rule, while the latter case gives rise to a non-constraint rule. The two clauses are as follows:

$$p(\bar{t}){:} -\bar{t} = \bar{s}, ..., \texttt{not}\ p(\bar{t}), ... \qquad \text{(C-rule)}$$

$$p(\bar{t}){:} -\bar{t} \neq \bar{s}, ..., \texttt{not}\ p(\bar{s}), ... \qquad \text{(NC-rule)}$$

where $=$ is the unification operator and $\bar{t} \neq \bar{s}$ indicates that $\bar{t}$ and $\bar{s}$ do not unify. Work is in progress to design and implement a goal-directed ASP system with full first order predicates on top of a coinductive logic programming engine. Our future work also includes developing goal-directed SAT solvers since any set of propositional formulas $\mathcal{F}$ can be transformed into an answer set program $\mathcal{A}$ such that every answer set of $\mathcal{A}$ is a model of $\mathcal{F}$ [50].

CHAPTER 5
APPLICATION (CO-LOGIC PROGRAMMING): MODEL CHECKING

Model checking is a popular technique used for verifying hardware and software systems. Most of the properties that can be verified using model checking can be classified into *safety* properties and *liveness* properties. Intuitively, safety properties are those which assert that ‘nothing bad will happen’ while liveness properties are those that assert that ‘something good will eventually happen’. Verification of safety properties amounts to computing least fixed-points [82, 44] and thus is elegantly handled by standard LP systems extended with tabling [63]. Co-LP on the other hand, as we have seen in Chapter 3, can be leveraged to directly and elegantly verify liveness properties, as verifying such properties amounts to computing greatest fixed-point [60].

It would be great to have a single framework for model checking in which one can encode both safety and liveness properties. This would require an integration of a co-LP engine with a Tabled LP engine. In this dissertation, we have presented a co-LP engine atop YAP prolog. YAP has a tabling engine (YAPTAB) and a CLP(R) engine integrated with it. So with our realization of co-LP atop YAP prolog, we have a single framework capable of handling constraints, tabling and coinduction. In this chapter we justify the power and need of our integrated Tabled LP + CLP + co-LP engine by presenting its application to elegantly solving the classical Dining Philosopher’s Problem. Various algorithms to solving this model checking application are elegantly encoded in our system. Verifying safety properties (free from deadlock) and liveness properties (free from starvation) of these algorithms can be elegantly encoded in a single framework.

5.1 Dining Philosopher's Problem

The Dining Philosopher's Problem has attracted and challenged both theoreticians and programmers ever since E. W. Dijkstra invented it in early 1970s and posed it as an exercise in concurrent programming. The problem can be summarized as follows: There are five philosophers who spend their lives just thinking and eating. While eating, they are not thinking, and while thinking, they are not eating. In the middle of the dining room there is a circular table around which the philosophers dine. The table has a big bowl of spaghetti in the center. There are five forks, each placed in between a pair of philosopher, and so each philosopher has one fork to his left and one fork to his right. Figure 5.1 depicts this table configuration. Each philosopher thinks for some time and when he gets hungry, he picks up the two forks (if they are free, i.e., not being used by the other philosophers) that are closest to him. If he can pick up both forks, he eats for a while. After he is done eating, he puts down the forks and starts to think. While eating, a philosopher can only use the fork on his immediate left or right. The philosophers are not allowed to communicate with each other. This creates a possibility of *deadlock* in which each philosopher holds a fork and waits (forever) for the other fork. There is a possibility of *Starvation* too, if one or more philosophers may never get a chance to acquire both forks and be able to eat. The problem is to design an algorithm to grant access of forks to the philosophers in a fair manner and show that the algorithm is free from deadlock and starvation.

A variety of different solutions to this problem have been proposed. The challenge comes in checking if the solution is free from deadlock and starvation. Next we present two solutions for this problem, one free from deadlock and the other free from both deadlock and starvation. We present the encodings of these solutions in our integrated framework and how we can elegantly check for the above mentioned properties in these solutions.

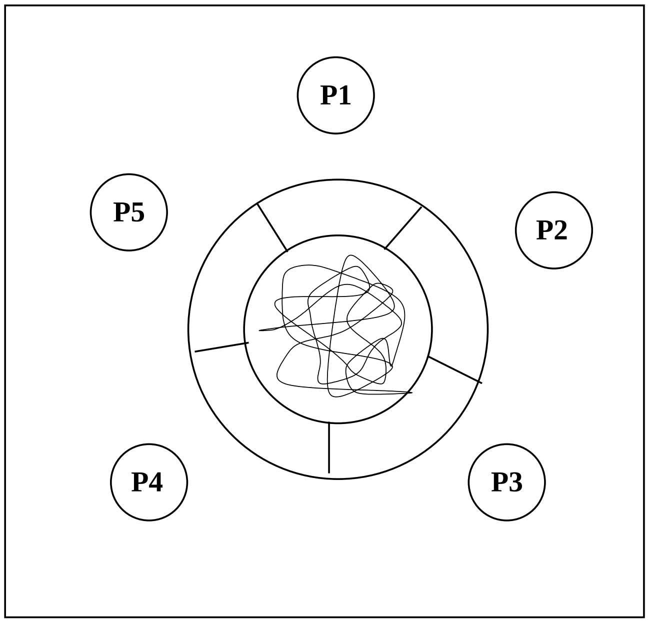

Figure 5.1. Dining Philosopher's Problem

5.1.1 A deadlock-free solution

First, we present a solution to the Dining Philosopher's Problem which is free from deadlock. Note that, the philosophers are in a thinking - picking up forks - eating - putting down forks cycle as shown in Figure 5.2.

One potential fork access mechanism is to force each philosopher to pick up his left fork before picking up his right fork. This is like assuming that each philosopher is a 'lefty' and so this would be his natural order of picking up the forks. Figure 5.3 presents this pictorially.

The problem with this solution is that it can easily lead the system to deadlock. Figure 5.4 depicts a deadlock situation. It represents a situation where each philosopher got access to their left forks simultaneously with others. Now each of them is waiting to get access to his right fork. That will never happen as none of the philosophers would put down their fork without eating. This produces a circular waiting. So the system is in deadlock. If this circular waiting can be broken, deadlocks will go away.

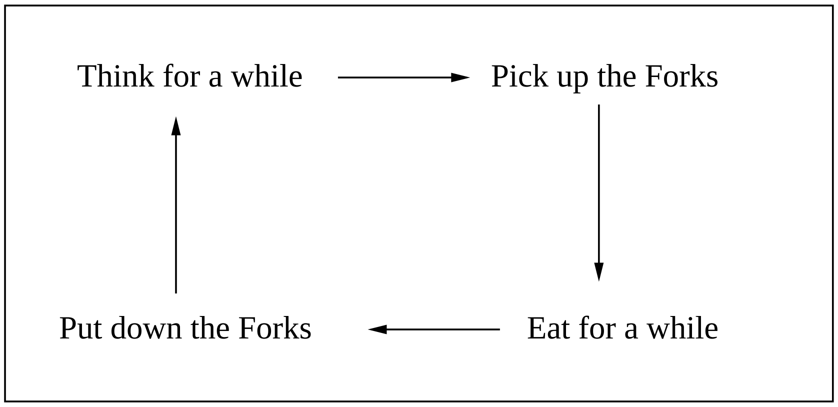

Figure 5.2. Dining Philosopher’s Problem Cycle

As it turns out, eliminating deadlock in the above mentioned solution is quite simple. If we choose one of the philosophers to be a righty instead of a lefty, then we never get into deadlock. The solution works deadlock free if we have at least one philosopher who is a lefty and at least one who is a righty.

Table 5.1 and 5.2 present the encoding of the above mentioned algorithm in our integrated framework. Verifying that this solution is deadlock free is done by calling the `reach/2` predicate to check if the system can reach the state where *all philosophers are waiting*, from the start state of *all philosophers thinking*. The `deadlock/0` predicate does precisely that. When called, the deadlock predicate fails, implying that this solution is deadlock free.

The solution presented in Table 5.1 and 5.2 is deadlock free but is prone to starvation. Deadlock is a safety property and starvation is a liveness property of this model and the above mentioned encoding shows how both the properties can be checked in our framework elegantly. Next, we present a solution to the Dining Philosopher’s Problem that is both deadlock and starvation free.

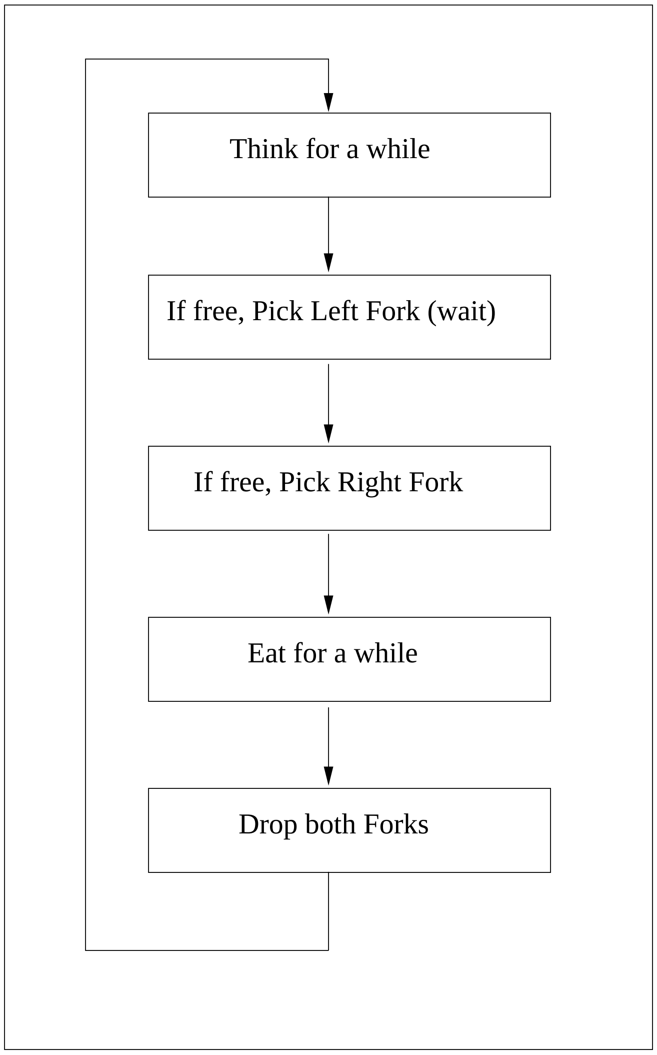

Figure 5.3. A potential fork access rule

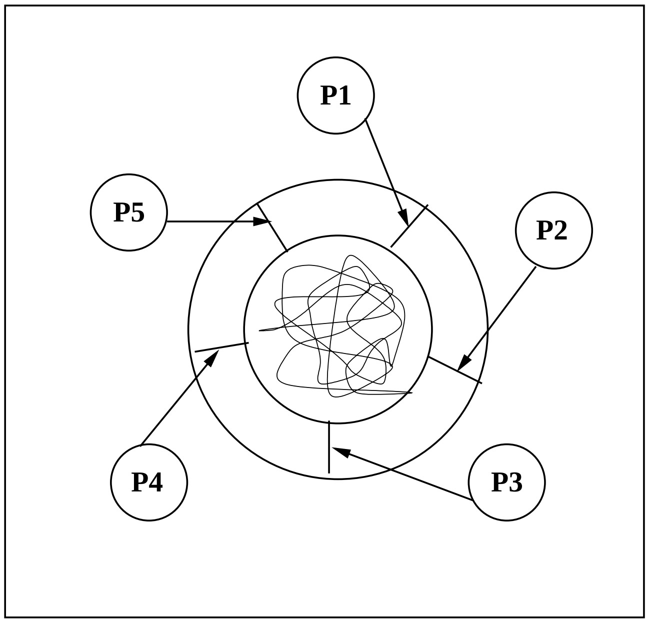

Figure 5.4. Deadlock

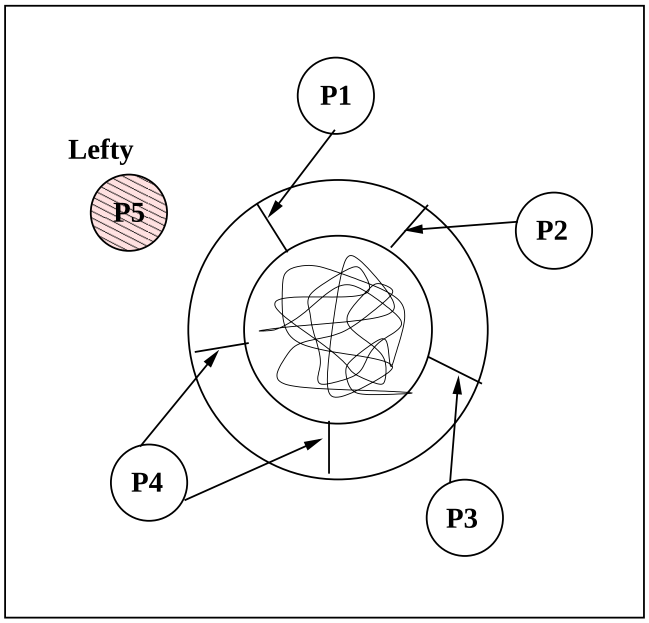

Figure 5.5. A deadlock-free solution

Table 5.1. Code for Dining Philosopher's problem (Deadlock-free)

```
:- [colp].
%-------------------------------------------------------

% 1 - think; 2 - wait; 3 - eat;

phil_1([1,P2,P3,P4,P5], [2,P2,P3,P4,P5]) :- (P5==1).
phil_1([2,P2,P3,P4,P5], [3,P2,P3,P4,P5]) :- (P2==1).
phil_1([3,P2,P3,P4,P5], [1,P2,P3,P4,P5]).
phil_1(St, St).

phil_2([P1,1,P3,P4,P5], [P1,2,P3,P4,P5]) :- (P1==1;P1==2).
phil_2([P1,2,P3,P4,P5], [P1,3,P3,P4,P5]) :- (P3==1).
phil_2([P1,3,P3,P4,P5], [P1,1,P3,P4,P5]).
phil_2(St, St).

phil_3([P1,P2,1,P4,P5], [P1,P2,2,P4,P5]) :- (P2==1;P2==2).
phil_3([P1,P2,2,P4,P5], [P1,P2,3,P4,P5]) :- (P4==1).
phil_3([P1,P2,3,P4,P5], [P1,P2,1,P4,P5]).
phil_3(St, St).

phil_4([P1,P2,P3,1,P5], [P1,P2,P3,2,P5]) :- (P3==1;P3==2).
phil_4([P1,P2,P3,2,P5], [P1,P2,P3,3,P5]) :- (P5==1;P5==2).
phil_4([P1,P2,P3,3,P5], [P1,P2,P3,1,P5]).
phil_4(St, St).

%lefty Phil...
phil_5([P1,P2,P3,P4,1], [P1,P2,P3,P4,2]) :- (P1==1).
phil_5([P1,P2,P3,P4,2], [P1,P2,P3,P4,3]) :- (P4==1;P4==2).
phil_5([P1,P2,P3,P4,3], [P1,P2,P3,P4,1]).
phil_5(St, St).

%-------------------------------------------------------
parTrans([],St,St).
parTrans([Pi|P_rest],Si,Sf) :-
    (trans(Pi,Si,Sfi), parTrans(P_rest,Sfi,Sf);
     parTrans(P_rest,Si,Sfi), trans(Pi,Sfi,Sf)).

trans(1,Si,Sf) :- phil_1(Si,Sf).
trans(2,Si,Sf) :- phil_2(Si,Sf).
trans(3,Si,Sf) :- phil_3(Si,Sf).
trans(4,Si,Sf) :- phil_4(Si,Sf).
trans(5,Si,Sf) :- phil_5(Si,Sf).
```

Table 5.2. Code for Dining Philosopher's problem (Deadlock-free) - Continued

```
:- coinductive(driver/2).

driver(Si, [Sf|Rest]) :-
    parTrans([1,2,3,4,5], Si, Sf),
    driver(Sf, Rest).

%-----------------------------------------
validTrace(Trace) :-
    Init_State = [1,1,1,1,1],
    driver(Init_State, OutTrace),
    Trace = [Init_State|OutTrace].

%-----------------------------------------
:- table reach/2.

reach(Si, Sf) :- trans(_,Si,Sf).
reach(Si, Sf) :- trans(_,Si,Sfi), reach(Sfi,Sf).

deadlock :- reach([1,1,1,1,1], [2,2,2,2,2]).

%-----------------------------------------
:- coinductive(str_driver/2).

str_driver(Si, SL) :-
    (Si = SL -> Sf = SL; true),
    parTrans([1,2,3,4,5], Si, Sf),
    str_driver(Sf, SL).

%-----------------------------------------
starved(X) :-
    X=1, str_driver([1,1,1,1,1], [2,_,_,_,_]);
    X=2, str_driver([1,1,1,1,1], [_,2,_,_,_]);
    X=3, str_driver([1,1,1,1,1], [_,_,2,_,_]);
    X=4, str_driver([1,1,1,1,1], [_,_,_,2,_]);
    X=5, str_driver([1,1,1,1,1], [_,_,_,_,2]).
```

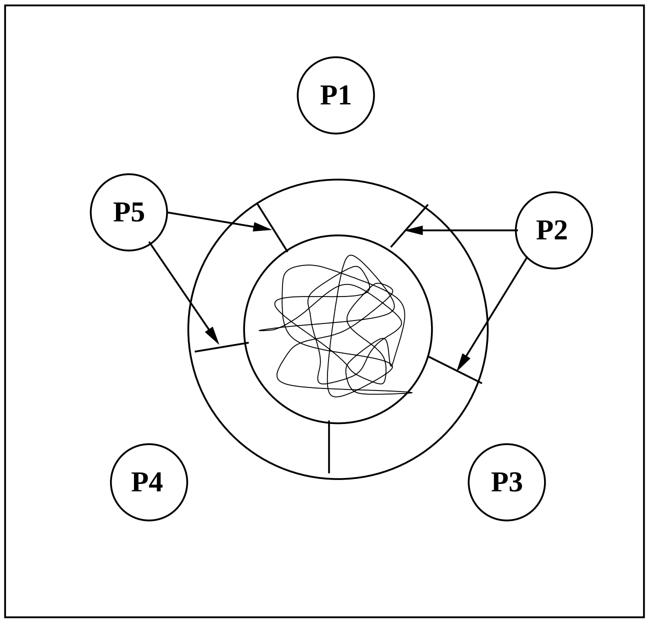

Figure 5.6. Starvation

5.1.2 A starvation-free solution

Starvation is the state of the system where one or more philosophers may never get a chance to acquire both forks and be able to eat. Figure 5.6 depicts this situation.

In the solution presented in Table 5.1 and 5.2, the predicate `starved/1` checks for a trace of a system in which a particular philosopher is starved. When called, the `starved(X)` predicate finds a trace for each philosopher, implying that there exists at least one trace of the system for each philosopher, in which that philosopher is starved.

In this section, we present a starvation free solution to the Dining Philosopher's Problem. This solution is obtained by adding a global clock and a pair of wait clocks for every philosopher to the deadlock free solution, mentioned in the previous section. Note that, to add and compare clocks, without discretizing time, we need the capability of CLP(R) (i.e., constraint logic programming over real domains) in our framework. This justifies the need of integrating CLP(R) with Tabled LP (for checking safety properties) and co-LP (for checking the liveness properties).

In this solution, each philosopher maintains a pair of clocks, one for each fork he has access to, which tells the timestamp (of the global clock) of when he accessed that fork last. When two philosophers contend to acquire the same fork, then the philosopher with the lower clock (timestamp) value gets its access. His clock value for that fork is reset to the current value of the global clock. When a philosopher acquires access to both his forks, he eats for some time. When he is done eating, he drops both the forks, making it available for other philosophers. At this time, both his clocks are reset to the current value of the global clock.

Table 5.3 and 5.4 present the encoding of the above mentioned starvation free solution in our framework. Eating event for a particular philosopher is captured by the `eat/1` predicate. Note that, a philosopher is not allowed to eat for an infinite amount of time. So the `eat/1` predicate is an ‘inductive’ predicate, as it terminates in a finite amount of time. To model this predicate, we need tabling in our framework as this can elegantly be modeled as a ‘tabled’ predicate in Tabled LP.

The global clock advances in every call to the `str_driver/2` predicate. This is achieved by setting up a constraint that the current clock value is greater than the previous one. It is this new value of the global clock that is used in the next (recursive) call to the `str_driver` predicate.

The `deadlock/0` predicate checks for a trace of the system in which deadlock occurs. When called, it fails, implying that there is no trace of the system in which a deadlock occurs, and hence the solution is deadlock is free. The predicate `starved/1` checks for a trace of a system in which a particular philosopher is starved. When called, the `starved(X)` predicate fails for every philosopher, implying that there exists no trace of the system for each philosopher, in which that philosopher is starved. Hence, this solution is starvation free.

Table 5.3. Code for Dining Philosopher's problem (Starvation-free)

```
:- use_module(library(clpr)).
:- [colp].
%-----------------------------------------------------------
init_Gvars :-
     nb_setval(wC, 0),
     nb_setval(p1C, (0,0)), nb_setval(p2C, (0,0)),
     nb_setval(p3C, (0,0)), nb_setval(p4C, (0,0)),
     nb_setval(p5C, (0,0)).
:- init_Gvars.

%-----------------------------------------------------------
% 1 - think; 2 - wait; 3 - eat;
%-----------------------------------------------------------
phil_1([1,P2,P3,P4,P5], [X,P2,P3,P4,P5]) :-
    P5==1, b_getval(p1C,(PL,PR)), b_getval(p5C,(_,CW)),
    {PL=<CW},b_getval(wC,W),b_setval(p1C,(W,PR)),!,X=2; X=1.
phil_1([2,P2,P3,P4,P5], [X,P2,P3,P4,P5]) :-
    P2==1, b_getval(p1C,(PL,PR)), b_getval(p2C,(CW,_)),
    {PR=<CW},b_getval(wC,W),b_setval(p1C,(PL,W)),!,X=3; X=2.
phil_1([3,P2,P3,P4,P5], [1,P2,P3,P4,P5]) :- eat(p1C).

phil_2([P1,1,P3,P4,P5], [P1,X,P3,P4,P5]) :- (P1==1), !, X=2;
    P1==2, b_getval(p2C,(PL,PR)), b_getval(p1C,(_,CW)),
    {PL=<CW},b_getval(wC,W),b_setval(p2C,(W,PR)),!,X=2; X=1.
phil_2([P1,2,P3,P4,P5], [P1,X,P3,P4,P5]) :-
    P3==1, b_getval(p2C, (PL,PR)), b_getval(p3C, (CW,_)),
    {PR=<CW},b_getval(wC,W),b_setval(p2C,(PL,W)),!,X=3; X=2.
phil_2([P1,3,P3,P4,P5], [P1,1,P3,P4,P5]) :- eat(p2C).

phil_3([P1,P2,1,P4,P5], [P1,P2,X,P4,P5]) :- (P2==1), !, X=2;
    P2==2, b_getval(p3C, (PL,PR)), b_getval(p2C, (_,CW)),
    {PL=<CW},b_getval(wC,W),b_setval(p3C,(W,PR)),!,X=2; X=1.
phil_3([P1,P2,2,P4,P5], [P1,P2,X,P4,P5]) :-
    P4==1, b_getval(p3C, (PL,PR)), b_getval(p4C, (CW,_)),
    {PR=<CW},b_getval(wC,W),b_setval(p3C,(PL,W)),!,X=3; X=2.
phil_3([P1,P2,3,P4,P5], [P1,P2,1,P4,P5]) :- eat(p3C).

phil_4([P1,P2,P3,1,P5], [P1,P2,P3,X,P5]) :- (P3==1), !, X=2;
    P3==2, b_getval(p4C, (PL,PR)), b_getval(p3C, (_,CW)),
    {PL=<CW},b_getval(wC,W),b_setval(p4C,(W,PR)),!,X=2; X=1.
phil_4([P1,P2,P3,2,P5], [P1,P2,P3,X,P5]) :- (P5==1), !, X=3;
    P5==2, b_getval(p4C, (PL,PR)), b_getval(p5C, (CW,_)),
    {PR=<CW},b_getval(wC, W),b_setval(p4C,(PL,W)),!,X=3; X=2.
phil_4([P1,P2,P3,3,P5], [P1,P2,P3,1,P5]) :- eat(p4C).
%-----------------------------------------------------------
```

Table 5.4. Code for Dining Philosopher's problem (Starvation-free) - Continued

```
%lefty Phil...
phil_5([P1,P2,P3,P4,1], [P1,P2,P3,P4,X]) :-
    P1==1, b_getval(p5C, (PL,PR)), b_getval(p1C, (CW,_)),
    {PR=<CW},b_getval(wC,W),b_setval(p5C,(PL,W)),!,X=2; X=1.
phil_5([P1,P2,P3,P4,2], [P1,P2,P3,P4,X]) :- (P4==1), !, X=3;
    P4==2, b_getval(p5C, (PL,PR)), b_getval(p4C, (_,CW)),
    {PL=<CW},b_getval(wC,W),b_setval(p5C,(W,PR)),!,X=3; X=2.
phil_5([P1,P2,P3,P4,3], [P1,P2,P3,P4,1]) :- eat(p5C).

%-------------------------------------------------------------
parTrans([],St,St).
parTrans([Pi|P_rest],Si,Sf) :-
    (trans(Pi,Si,Sfi), parTrans(P_rest,Sfi,Sf));
     parTrans(P_rest,Si,Sfi), trans(Pi,Sfi,Sf)).

trans(1,Si,Sf) :- phil_1(Si,Sf).
trans(2,Si,Sf) :- phil_2(Si,Sf).
trans(3,Si,Sf) :- phil_3(Si,Sf).
trans(4,Si,Sf) :- phil_4(Si,Sf).
trans(5,Si,Sf) :- phil_5(Si,Sf).

%-------------------------------------------------------------
:- table eat/1.

eat(PhC) :- eat(PhC).
eat(PhC) :- b_getval(wC, W), b_setval(PhC, (W,W)).

%-------------------------------------------------------------
:- table reach/2.

reach(Si, Sf) :- trans(_,Si,Sf).
reach(Si, Sf) :- trans(_,Si,Sfi), reach(Sfi,Sf).

deadlock :- reach([1,1,1,1,1], [2,2,2,2,2]).

%-------------------------------------------------------------
:- coinductive(str_driver/2).

str_driver(Si, SL) :-
    (Si = SL -> Sf = SL; true), b_getval(wC, T),
    parTrans([1,2,3,4,5], Si, Sf), {TA > T},
    b_setval(wC, TA), str_driver(Sf, SL).

%-------------------------------------------------------------
starved(X) :- X=1, str_driver([1,1,1,1,1], [2,_,_,_,_]);
              X=2, str_driver([1,1,1,1,1], [_,2,_,_,_]);
              X=3, str_driver([1,1,1,1,1], [_,_,2,_,_]);
              X=4, str_driver([1,1,1,1,1], [_,_,_,2,_]);
              X=5, str_driver([1,1,1,1,1], [_,_,_,_,2]).
```

5.2 Conclusions

In this chapter, we presented how our integrated framework, that combines the power of CLP, Tabled LP and co-LP in one system, can elegantly encode a model checking application. The application presented is the encoding of two different solutions for the classical Dining Philosopher's Problem. The ability to check for safety properties (deadlock - using Tabled LP), to check for liveness properties (starvation - using co-LP) and to encode and compare clocks (to prevent starvation - using CLP) in a single system, justifies the need and power of our integrated system.

CHAPTER 6
APPLICATION (ANSWER-SET PROGRAMMING): TIMED PLANNING

Planning has been an active area of research since the early days of AI and cognitive science. According to the MIT Encyclopedia of Cognitive Science [67], ”Planning is the process of generating (possibly partial) representations of future behavior prior to the use of such plans to constrain or control that behavior. The outcome is usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.” Research in planning has yielded many useful tools for real-world applications, and has led to significant insights into the organization of behavior and the nature of reasoning about actions [2].

In planning, a domain description *D* is given along with a set of observations about the initial state *O* and a collection of fluent literals *G* = $\{g_1, \ldots, g_l\}$, which is referred to as a goal. The problem is to find a sequence of actions $a_1, \ldots, a_n$ such that $\forall i,\ 1 \leq i \leq l$, *D* entails g_i from initial state *O*, after actions $a_1, \ldots, a_n$. The sequence of actions $a_1, \ldots, a_n$ is called a plan for goal *G* w.r.t. *(D,O)* [5].

In the field of logic programming, action description languages have been shown as a way to represent actions and change [25]. These are (high-level) languages that can be used to reason about actions and change in dynamic environments. These languages are useful in solving planning problems, as they can be used to specify, verify and diagnose plans. They are widely used for planning with domain specific constraints [5]. Given a (partial) description of the state of the world, and a sequence of actions with their properties, it may be required to deduce the resulting state after applying the action sequence. Also, given a resultant state (induced by a sequence of actions), it may be needed to deduce the sequence of actions that led to that state or deduce information about past states. We describe the action description language $\mathcal{A}$ [25] in Section 6.1. The language has a simple syntax to specify properties of actions and an automata-theoretic semantics to reason about sequences of actions [70].

Timed planning applications involve systems that have real-time constraints. The occurrence of an action is as important as the time at which the action occurs. To model such real-time systems or to encode timed planning problems, an extension of the action description language $\mathcal{A}$ with real-time clocks and constraints has recently been proposed [73]. The syntax and semantics of the extended language $\mathcal{A}_T$ and its implementation using logic programming are presented in [73].

In this chapter, we present the concept of Timed Planning. The problems in timed planning are planning problems that involve real-time constraints. These constraints could be on both actions or fluents (described in Section 6.1). We present examples of planning domains described using action description language $\mathcal{A}$ and timed planning domains described using real-time action description language $\mathcal{A}_T$. Then we show how elegantly we can encode these planning domains (described using $\mathcal{A}$) and timed planning domains (described using $\mathcal{A}_T$) using CLP(R) and ASP in our integrated framework.

6.1 The Action Description language $\mathcal{A}$

The action description language $\mathcal{A}$ was proposed by Michael Gelfond and Vladimir Lifschitz to represent actions and change using logic programs [25]. $\mathcal{A}$ is a simple, high-level declarative language that provides a mechanism for describing action domains. The language has the notion of *fluents* for temporal reasoning. A *fluent* is something that may depend on the situation, as, for instance, the location of a soccer ball. In particular, *propositional* fluents are assertions that can be true or false depending on the situation or the state of the world [25]. The language provides two kinds of propositions: (i) a *value proposition*, that describes the truth value of a fluent in a particular state (which can either be an initial state or a state induced by a sequence of actions); (ii) an *effect proposition*, that describes the effect of a given action on a particular fluent.

The language $\mathcal{A}$ has two sets of symbols, *fluent names* and *action names*. *Fluent expressions* are fluents that may be preceded by a $\neg$ symbol. A *value proposition* has the following syntax

$$F \textbf{ after } A_1; \ldots ; A_m,$$

where F is a fluent expression and $A_1, \ldots, A_m$ $(m \geq 0)$ are action names. If $m = 0$, the above value proposition is written as

$$\textbf{initially } F.$$

An *effect proposition* has the syntax

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n,$$

where A is an action name, and each of F, $P_1, \ldots, P_n$ $(n \geq 0)$ is a fluent expression. The effect proposition describes the effect that the action A has on the fluent F, subject to the *preconditions* $P_1, \ldots, P_n$. If $n = 0$, the above effect proposition is written as

$$A \textbf{ causes } F$$

A *domain description*, or simply *domain*, consists of a (possibly infinite set) of value propositions and a finite set of effect propositions. A state consists of a set of fluents. A fluent name F holds in state σ if $F \in \sigma$ and $\neg F$ holds in σ if $F \notin \sigma$. A *transition function* is a mapping Phi from the set of pairs (A, σ) to states. A *structure* is a tuple (σ_0, Φ), where σ_0 is called the *initial state* and Φ is a transition function.

A structure (σ_0, Φ) is a *model* of a domain description D if every value proposition in D is true in (σ_0, Φ), and for every action A, every fluent F and every state σ, the following hold [25]:

1. if D includes an effect proposition describing the effect of A on F, whose preconditions are valid in σ, then $F \in \Phi(A, \sigma)$.
2. if D includes an effect proposition describing the effect of A on $\neg F$ whose preconditions are valid in σ, then $F \notin \Phi(A, \sigma)$.
3. if D does not include such effect propositions, then $F \in \Phi(A, \sigma)$ iff $F \in \sigma$.

6.1.1 Example: Yale Shooting Domain

Here we present an example encoding of the classic Yale Shooting domain[25] in language $\mathcal{A}$. This domain consists of the fluents $Loaded$ and $Alive$. It has actions $Load$, $Shoot$ and $Wait$. The propositions constituting the domain are shown in Table 6.1.

Table 6.1. Yale Shooting Domain in language $\mathcal{A}$

initially $\neg Loaded,$
initially $Alive,$
$Load$ **causes** $Loaded,$
$Shoot$ **causes** $\neg Alive$ **if** $Loaded,$
$Shoot$ **causes** $\neg Loaded.$

Language $\mathcal{A}$ deals implicitly with time in a qualitative manner. So one can reason only about temporal properties of sequences of actions in $\mathcal{A}$. But for real-time domains, one needs to be able to reason about time in a quantitative manner, as the system may have real-time constraints on actions that must be satisfied for that action to happen. So in addition to reasoning about sequences of actions, it is also essential to reason about the deadlines that these actions have to meet. For example, in the Yale shooting problem, one may want to reason that $\neg Alive$ will become true only if the shot is fired with a loaded gun, within 30 seconds of loading it (otherwise the person may get away, or the ammunition did not work). Similarly, one may want to reason that the drop action will cause a fragile object to break, unless it is caught within one second, thus preventing it from hitting the ground and breaking. Language $\mathcal{A}$ lacks the capability to reason about real-time in this manner. There are many situations, as shown above, where this capability is needed.

In real-life situations, like specifying controllers and developing plans for machines, plants, and robots [58], most actions will have severe time constraints attached. Action description languages can be used for these applications if they are augmented with the capability to reason with time. In the next section, we present an extension to the action description language $\mathcal{A}$, called $\mathcal{A}_T$ [73], which has the extended capability to specify and reason about actions that have real-time constraints associated with them. This extended language is well-suited for timed planning and has significantly more applications; e.g., in safety-critical systems.

6.2 A Real-time Action Description language $\mathcal{A}_T$

Real-time systems are computing systems in domains where response within a hard time bound is critical for success. Controllers for aircraft, industrial machinery and robots are few such domains. These domains, by their very nature, have certain real-time constraints, which if violated, can cause the system to fail. Therefore it is essential to have a systematic framework to reason about properties of actions in real-time domains. Traditionally, the design of real-time systems has been studied from the perspective of scheduling [37]. However, in this approach one can only reason about properties that can be formulated in terms of task execution times. The scheduling strategies used are usually conservative, due to the safety critical nature of the domain. These strategies assume that all task execution times are known in advance, thereby making them inadequate for applying them in dynamic environments.

These problems provide motivation for looking at real-time systems from the planning perspective and to extend action description languages to reason about properties of actions with real-time constraints. We need the ability to model time as a continuous entity, because the delay between events in most real-time systems can be arbitrarily small. So we need to extend action description languages with the notion of *continuous time* [73]. To be able to apply action description languages such as language $\mathcal{A}$ to real-time systems, we need to augment an action (in $\mathcal{A}$) with the time at which the action occurs. Generally speaking, we need to specify clock constraints describing when the action occurs. The real-time action description language $\mathcal{A}_T$, which extends language $\mathcal{A}$ by augmenting action with real-time constraints, is presented in [73]. $\mathcal{A}_T$ defines a complete real-time action α by pairing its name with a list of clock constraints associated with it. In $\mathcal{A}_T$, this is written as

$$A \textbf{ at } T_1, \ldots, T_n$$

where $T_1 \ldots, T_n$ $(n \geq 0)$ are clock constraints of the form $C \leq E$, $C \geq E$, $C < E$, and $C > E$, where C is a clock name and E is a clock name or a clock name plus or minus a real valued constant, and when $n = 0$ the $\textbf{at}$ clause can be dropped.

With the ability to explicitly state when an action occurs, value propositions can be easily extended to include it. Given fluent expressions $F_1, \ldots, F_m$ $(m > 0)$ and real-time actions $\alpha_1, \ldots, \alpha_n$ $(n \geq 0)$, a real-time value proposition can be written as:

$$F_1, \ldots, F_m \textbf{ after } \alpha_1; \ldots; \alpha_n$$

When the sequence of actions is empty ($n = 0$), a real-time value proposition is written as

$$\textbf{initially } F_1, \ldots, F_m$$

These forms of real-time value propositions are used to describe the start state of a real-time system by asserting which fluents are true or false in the start state. All clocks are assumed to be reset when initially entering the start state of a real-time system. $\mathcal{A}_T$ also extends the effect proposition of language $\mathcal{A}$ to work with real-time constraints. The effect propositions must be extended so that they are able to describe the fluent preconditions as well as the clock preconditions for the rule to apply. In addition to be able to describe how fluents are mutated, real-time effect propositions also need the ability to reset (all or some of) the clocks in order to describe how clocks are changed. So the real-time effect propositions (sometimes referred to as action rules) in $\mathcal{A}_T$ are written as

$$A \textbf{ causes } F_1, \ldots, F_m \textbf{ resets } C_1, \ldots C_n \textbf{ when } T_1, \ldots, T_k \textbf{ if } P_1, \ldots, P_i$$

for action name A, fluent expressions $F_1 \ldots, F_m,\ P_1, \ldots, P_i\ (m, i \geq 0)$, clock names $C_1, \ldots, C_n\ (n \geq 0)$, and clock constraints $T_1, \ldots, T_k\ (k \geq 0)$, where $m + n + k + i > 0$. As usual, when m, n, k, or i is zero the keywords **causes**, **resets**, **when**, or **if** respectively, can be dropped. The **resets** clause specifies clocks that are to be reset, assuming the **when** clause and fluent preconditions are satisfied. Clocks continue to advance, if they are not reset. A special action **wait** that denotes the action of waiting for time to elapse is also provided in language $\mathcal{A}_T$. This action acts as a wild-card that matches all other action names, and thus provides the ability to encode the passing of time in the current state of the system. We provide some example $\mathcal{A}_T$ encodings of domains with real-time constraints in the next section.

6.3 Encoding $\mathcal{A}_T$ programs using CLP(R) + ASP

Timed planning problems can easily be encoded using CLP(R) and ASP, and our integrated framework is capable of handling both these paradigms, thus making suitable for encoding and solving timed planning problems. In this section we present how the real-time action description language $\mathcal{A}_T$, described in the previous section, can be used for timed planning, by providing some example encodings of domains with real-time constraints in $\mathcal{A}_T$. We also show how these domains, described using $\mathcal{A}_T$, can be easily and elegantly encoded in our integrated framework (using CLP(R) and ASP), thus proving that our framework is capable of doing timed planning elegantly.

We first present an encoding of classical Yale Shooting Domain, described in Section 6.1. Note that domain presented does not have any real-time constraints on its actions. Table 6.2 presents the executable encoding of this domain in our framework. In this encoding, predicate `holds(F,S)` is true if fluent *F* holds in situation (or state) *S* and predicate `not_holds(F,S)` is true if fluent *F* does not hold in situation (or state) *S* [5]. The query `|?- not_holds(alive, S)` would produce the solution `S = res(shoot, res(load,s0))`, thus providing the sequence of actions `[load, shoot]` that would make the fluent `alive` false, starting with the initial state `s0`.

6.3.1 Example: Fragile Object Domain

Now we present a domain with real-time constraints on its actions. The domain presented is a modification of the Fragile Object domain from [25]. The real-time Fragile Object domain, extends the original example with the notion that a dropped object can be caught before it hits the ground. We assume the object takes 1 second to hit the ground. The assumption that units are in seconds is merely a convention we use in this example. In the language $\mathcal{A}_T$ all clocks are variables in the real number domain, i.e., they can take any arbitrary real values. Figure 6.1 depicts the real-time Fragile Object Domain.

Table 6.2. Encoding of Yale Shooting Domain without real-time constraints

```
%------------------------------------------------------
% Yale shooting problem...
%------------------------------------------------------

:- [aspNt].

%------------------------------------------------------
:- coinductive(holds/2).
:- coinductive(not_holds/2).
:- coinductive(ab/3).

holds(alive, s0).
holds(loaded, res(load, _S)).
holds(loaded, S) :-
    holds(alive, S), not_holds(alive, res(shoot, S)).

holds(F,res(A,S)) :- holds(F,S), nnot(ab(F,A,S)).
holds(F,S) :- holds(F,res(A,S)), nnot(ab(F,A,S)).

%------------------------------------------------------
not_holds(loaded, s0).
not_holds(loaded, res(shoot, _S)).
not_holds(loaded, S) :- holds(alive, res(shoot, S)).
not_holds(alive,  res(shoot, S)) :- holds(loaded, S).

not_holds(F,res(A,S)) :- not_holds(F,S), nnot(ab(F,A,S)).
not_holds(F,S) :- not_holds(F,res(A,S)), nnot(ab(F,A,S)).

%------------------------------------------------------
ab(loaded,  load, _S).
ab(loaded, shoot, _S).
ab(alive,  shoot,  S) :- nnot(not_holds(loaded, S)).

nt(ab(loaded,  load, _S)) :- fail.
nt(ab(loaded, shoot, _S)) :- fail.
nt(ab(alive,  shoot,  S)) :- not_holds(loaded, S).

%------------------------------------------------------
```

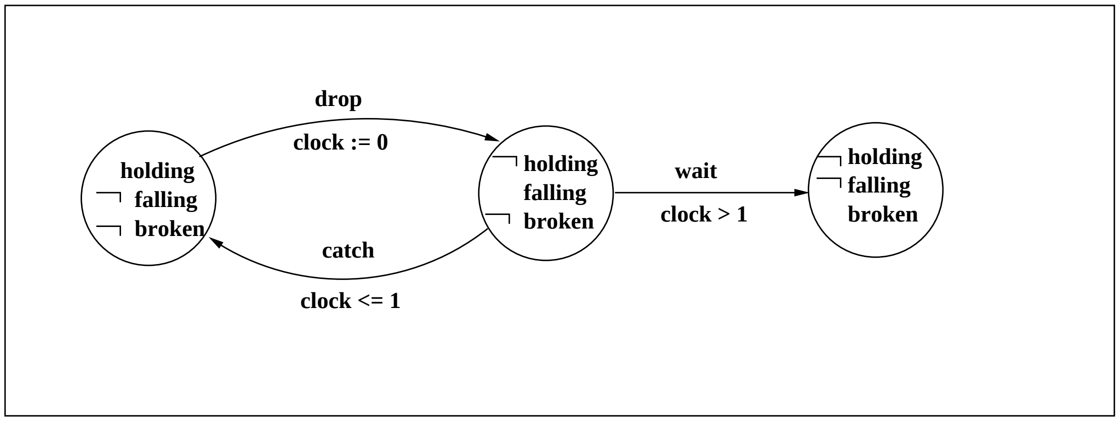

Figure 6.1. Real-time Fragile Object Domain

The language $\mathcal{A}_T$ describes many possible worlds, as is the case with language A. In one of these worlds **initially** $Holding, \neg Falling, \neg Broken$ is true, and therefore $Broken$ **after** $Drop$; **wait at** $Clock = 2$ also holds as the object is dropped and then allowed to fall to the ground. In that same world, if one takes too long to catch the object, then the object still shatters on the ground. Hence in the aforementioned world $Broken$ **after** $Drop$; $Catch$ **at** $Clock = 2$ is also true. However, if the object is dropped and then is successfully caught, say at half a second after dropping (i.e., before it hits the ground), then the object is not broken by the sequence of events, i.e., $\neg Broken$ **after** $Drop$; $Catch$ **at** $Clock = 0.5$ is true. Other possible worlds include the object starting out already in a falling state, while another world could even have the object already broken. The more information given in a description, the fewer possible worlds exist that satisfy the description. For example, assume that in addition to the original Real-time Fragile Object domain description, it is also given that $Broken$ **after** $Drop$; $Catch$ **at** $Clock = 0.5$ is true, then it can safely be deduced that the object was broken to begin with, i.e., according to this new description, **initially** $Broken$ is true [73]. Table 6.3 shows the encoding of the real-time Fragile Object Domain in language $\mathcal{A}_T$.

Table 6.4 presents the executable encoding of real-time Fragile Object domain using CLP(R) and ASP, in our framework. Real-time constraints on actions are represented using

Table 6.3. Fragile Object Domain in language $\mathcal{A}_T$

$Drop$ **causes** $\neg Holding, Falling$ **resets** $Clock$ **if** $Holding, \neg Falling$
$Catch$ **causes** $Holding, \neg Falling, \neg Broken$ **when** $Clock \leq 1$
  if $\neg Holding, Falling$
wait causes $Broken, \neg Falling$ **when** $Clock > 1$ **if** $\neg Holding, Falling$

clock variables in CLP(R), i.e., they can take values from real number domain. Predicates `holds` and `not_holds` have the same meaning as described in the previous example. Note that most of the rules have clock constraints in their body, and these constraints encode the real-time constraints associated with the action encoded by that rule.

The query `|?- holds(broken, S)` to the encoding shown in Table 6.4 would produce the solution `S = res(wait,res(drop,s0))`, thus providing the sequence of actions `[drop, wait]` that would make the fluent `broken` true, starting with the initial state `s0` in which `broken` was false. This corresponds to the situation that if we `wait` after the `drop` action (i.e., not `catch` the object in time), then it breaks. Note that the query `|?- not_holds(broken, S)` would produce the solution `S = res(catch,res(drop,s0))`, which corresponds to the situation that if we `catch` the object in time, after the `drop` action, then the object does not break. We can change the output produced by the program (i.e., the behavior of the actions encoded) by modifying or removing the clock constraints. This shows that the real-time constraints associated with the actions are encoded by the clock constraints.

6.3.2 Example: Soccer Playing Domain

Next we present another example of a real-time domain. The example presented is a modification of the Soccer Playing domain described in [80]. We extend the domain with some real-time constraints. In our extended real-time Soccer Playing domain, if a player with a

Table 6.4. Encoding of Fragile Object Domain with real-time constraints

```
:- [aspNt].
:- nb_setval(clock, 0).
:- use_module(library(clpr)).
:- coinductive(holds/2).
:- coinductive(not_holds/2).
:- coinductive(ab/3).

holds(holding, s0).                   not_holds(falling, s0).
not_holds(broken,  s0).
%-------------------------------------------------------------
holds(falling, res(drop,S)) :- {Clock>0}, holds(holding,S),
    not_holds(falling,S), b_setval(clock,Clock).

not_holds(holding, res(drop,S)) :- {Clock>0}, holds(holding,S),
    not_holds(falling,S), b_setval(clock,Clock).
%-------------------------------------------------------------
holds(holding, res(catch,S)) :- not_holds(holding,S),
    holds(falling,S), b_getval(clock,Clock),
    {Clock =< 1}, {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(falling, res(catch,S)) :- not_holds(holding,S),
    holds(falling,S), b_getval(clock,Clock),
    {Clock =< 1}, {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(broken, res(catch,S)) :- not_holds(holding,S),
    holds(falling,S), b_getval(clock,Clock),
    {Clock =< 1}, {NewClock > Clock}, b_setval(clock,NewClock).
%-------------------------------------------------------------
holds(broken, res(wait,S)) :- not_holds(holding,S),
    holds(falling,S), b_getval(clock,Clock),
    {Clock > 1}, {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(falling, res(wait,S)) :- not_holds(holding,S),
    holds(falling,S), b_getval(clock,Clock),
    {Clock > 1}, {NewClock > Clock}, b_setval(clock,NewClock).
%-------------------------------------------------------------
holds(F, res(A,S)) :-
    holds(F,S), nnot(ab(F,A,S)), b_getval(clock,Clock),
    {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(F, res(A,S)) :-
    not_holds(F,S), nnot(ab(F,A,S)), b_getval(clock,Clock),
    {NewClock > Clock}, b_setval(clock,NewClock).
%-------------------------------------------------------------
ab(holding,  drop, _S).               ab(falling, catch, _S).

nt(ab(holding,  drop, _S)) :- fail.
nt(ab(falling, catch, _S)) :- fail.
```

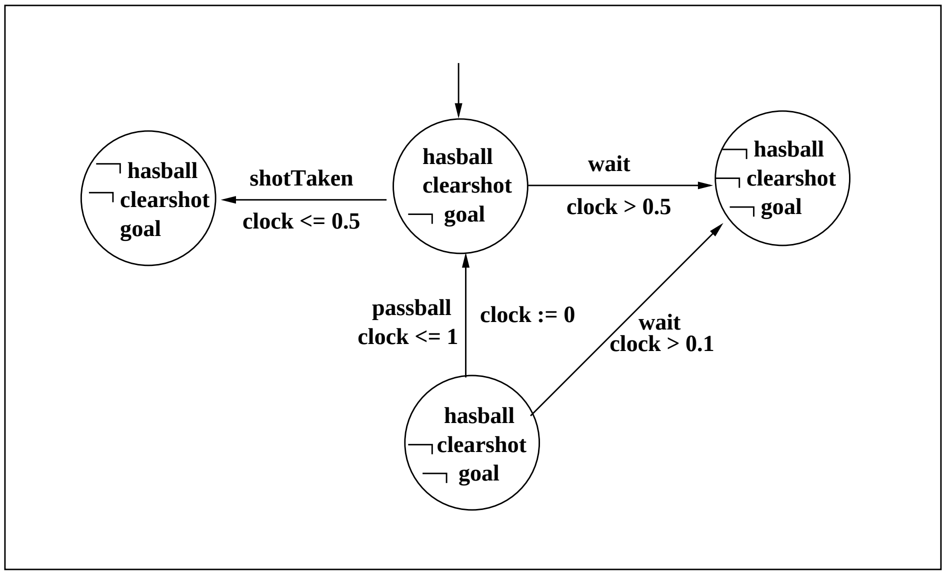

Figure 6.2. Real-time Soccer Playing Domain

clear shot at the goal has the ball, then it is assumed he scores the goal if he can take the shot within 0.5 time units. If the player does not take the shot within the specified time constraint, the ball is taken away by a player of the opposite team. Also, if a player does not have a clear shot at the goal but has the possession of the ball, then he can pass the ball to a teammate who has a clear shot, but the pass has to be completed within 1 time unit. If the specified time constraint is not met, then it is assumed that the ball has been intercepted by a player of the opposite team. Figure 6.2 depicts the real-time soccer playing domain.

In one possible world, **initially** $HasBall, ClearShot, \neg Goal$ holds and therefore $Goal$ **after** $ShotTaken$ **at** $Clock = 0.2$ is also true. However, in an alternative world, if the player does not have a clear shot, and the ball is passed to a teammate who has a clear shot, then if the teammate does not take the shot within 0.5 time units, possession of the ball is lost. Therefore the statement $\neg HasBall$ **after**$PassBall; wait$ **at** $Clock = 1$ is true. Table 6.5 shows the description of the real-time soccer playing domain, in the language $\mathcal{A}_T$.

Table 6.6 presents the executable encoding of real-time Soccer Playing domain using CLP(R) and ASP, in our framework. Real-time constraints on actions are represented,

Table 6.5. Soccer Playing Domain in language $\mathcal{A}_T$

ShotTaken **causes** $\neg HasBall, \neg ClearShot, Goal$ **when** $Clock \leq 0.5$
    if $HasBall, ClearShot, \neg Goal$
$PassBall$ **causes** $ClearShot$ **resets Clock when** $Clock \leq 1$
    if $\neg ClearShot$
wait causes $\neg HasBall, \neg ClearShot$ **when** $Clock > 0.5$
    if $HasBall, ClearShot$
wait causes $\neg HasBall,$ **when** $Clock > 1$ **if** $HasBall$

as mentioned in the previous example, using clock variables in CLP(R) (i.e., they can take values from real number domain). These clock constraints encode the real-time constraints associated with the actions in the domain. The query `|?- holds(goal, S)` to the encoding shown in Table 6.6 would produce the solution `S = res(shotTaken,s0))`, thus providing the sequence of actions `[shotTaken]` that would make the fluent `goal` true, starting with the initial state `s0` in which fluent `clearShot` was true. This corresponds to the situation that if the player has the ball and has a clear shot, then fluent `goal` would be true if the player performs the `shotTaken` action within the time constraint specified (0.5 seconds in this case). Note that if we make the fluent `clearShot` false in the initial state, then the sequence of actions produced to make the fluent `goal` true is `[passBall, shotTaken]`. This corresponds to the situation that if the player does not have a clear shot at the goal, then he can pass the ball to his team-mate within the time constraint specified for the `passBall` action (1 sec. in this case). Then his team-mate, who has the possession of the ball, would have a clear shot, making the `clearShot` fluent true, and if he takes the shot by performing `shotTaken` action within the specified time constraint, then the fluent `goal` would become true. In this example as well we can change the output produced by the program (i.e., the behavior of the actions encoded) by modifying or removing the clock constraints thus proving that the real-time constraints associated with the actions are encoded by the clock constraints.

Table 6.6. Encoding of Soccer Playing Domain with real-time constraints

```
:- [aspNt].
:- nb_setval(clock, 0).
:- use_module(library(clpr)).
:- coinductive(holds/2).
:- coinductive(not_holds/2).
:- coinductive(ab/3).

holds(hasBall,s0).   holds(clearShot,s0).   not_holds(goal,s0).
%--------------------------------------------------------------
holds(goal, res(shotTaken,S)) :- holds(hasBall,S),
    holds(clearShot,S),not_holds(goal,S),b_getval(clock,Clock),
    {Clock=<0.5}, {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(hasBall, res(shotTaken,S)) :- holds(hasBall,S),
    holds(clearShot,S),not_holds(goal,S),b_getval(clock,Clock),
    {Clock=<0.5}, {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(clearShot, res(shotTaken,S)) :-  holds(hasBall,S),
    holds(clearShot,S),not_holds(goal,S),b_getval(clock,Clock),
    {Clock=<0.5}, {NewClock > Clock}, b_setval(clock,NewClock).
%--------------------------------------------------------------
holds(clearShot, res(passBall,S)) :- not_holds(clearShot,S),
    b_getval(clock,Clock), {Clock =< 1},
    {NewClock > 0}, b_setval(clock,NewClock).
%--------------------------------------------------------------
not_holds(hasBall, res(wait,S)) :- holds(hasBall,S),
    holds(clearShot,S), b_getval(clock,Clock), {Clock > 0.5},
    {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(clearShot, res(wait,S)) :- holds(hasBall,S),
    holds(clearShot,S), b_getval(clock,Clock), {Clock > 0.5},
    {NewClock > Clock}, b_setval(clock,NewClock).

not_holds(hasBall, res(wait,S)) :- holds(hasBall,S),
    b_getval(clock,Clock), {Clock > 1},
    {NewClock > Clock}, b_setval(clock,NewClock).
%--------------------------------------------------------------
holds(F, res(A,S)) :- holds(F,S), nnot(ab(F,A,S)),
    b_getval(clock, Clock),{NewClock > Clock},
    b_setval(clock, NewClock).

not_holds(F, res(A,S)) :- not_holds(F,S), nnot(ab(F,A,S)),
    b_getval(clock, Clock), {NewClock > Clock},
    b_setval(clock, NewClock).
%--------------------------------------------------------------
ab(hasBall,shotTaken,_S).             ab(hasBall,wait,_S).
ab(clearShot,wait,_S).

nt(ab(holding,drop,_S)) :- fail.
nt(ab(falling,catch,_S)) :- fail.
```

The clock constraints used in the above mentioned examples are realized using global variables in Prolog. This makes the encoding of the real-time domains in our framework not purely declarative. Note that this is only because of a limitation in our implementation of co-LP and in turn ASP (as ASP is implemented by extending our co-LP implementation). The limitation is not being able to explicitly declare the arguments of a coinductive predicate on which the predicate behaves coinductively. So currently, only predicates that behave coinductively on all of their arguments can be implemented in our framework. In the above examples both the `holds(F,S)` and `not_holds(F,S)` predicates are coinductive on both their arguments, but not on the clock variables associated with the actions encoded by them, as the clocks advance continuously in real-time. So the clocks variables are implemented using global variables instead of passing them as arguments to the `holds(F,S)` and `not_holds(F,S)` predicates. This limitation can be taken care of easily by storing the predicate calls in the coinductive hypothesis, only with their coinductive arguments. This modification can be incorporated easily in a low-level (WAM level) implementation of our co-LP engine, which is part of our future work.

Action description languages have been used in past for reasoning about the effects of actions and change in various domains [42, 26, 27, 8]. They have also been extended to reason about concurrent actions [7]. They have been applied to various aspects of the planning problem [40, 41]. An action description language known as $\mathcal{B}$ [79] has been developed for reasoning about static causal laws and related to logic programming and default theories. However these languages do not provide the capability to reason about the hard real-time constraints that are found in most safety-critical domains and timed planning problems. Considerable work has been done on scheduling for real-time systems [37]. But most of these techniques do not provide the ability to reason about the effects of sequences of actions that occur in real-time. Approaches to extend model checkers and temporal logics with the ability to reason with time have also been proposed in the past [19]. But most of these extensions consider time to be a discrete entity, and so they lack ability to model systems in which actions may be separated by arbitrarily small time intervals. Methodologies that use continuous real-time have been considered as well. But most of these systems are based on some variation of timed automata [53, 3], and so they can only reason about temporal projection of the effects of real-time actions, but not about the past, given the current state of the system. So they are unsuitable for describing dynamic systems or systems whose description is incomplete, as is the case in most real world systems. So in our opinion, real-time action description language $\mathcal{A}_T$ is a novel approach to describe such systems. The ability to encode $\mathcal{A}_T$ programs using CLP(R) and ASP, in our integrated framework, provides us with a way to execute these programs and reason about real-time domains, thus providing us an executable framework to encode timed planning problems.

6.4 Conclusions

In this chapter, we presented how our integrated framework, that combines the power of CLP, and ASP in one system, can elegantly encode real-time domains and thus timed planning problems. The application presented is the executable encoding of planning domains described using the real-time action description language $\mathcal{A}_T$. The ability to do non-monotonic reasoning (ASP) in presence of time constraints (CLP) in a single system, justifies the need and power of our integrated system.

CHAPTER 7
CONCLUSIONS

The research presented in this dissertation contributes a big step towards the realization of the next generation logic programming system. The research presented motivates the need for such a system, which is realized by combining all powerful extensions of Logic programming into a single system. The extensions combined in this dissertation are *coinductive logic programming*, *tabled logic programming*, *constraint logic programming* and *answer set programming*. Further, applications justifying the need and power of various combinations are also presented.

This dissertation presents the design and implementation of Co-Logic Programming, atop YAP Prolog. The technique described is simple yet elegant and is easy to incorporate into any existing prolog engine. Future work involves extending co-LP's operational semantics with alternating OLDT and co-SLD, so that the operational behavior of inductive predicates can be made to more closely match their declarative semantics. Future work also involves extending the operational semantics of co-LP to allow for finite derivations in the presence of irrational terms and proofs, that is, infinite terms and proofs that do not have finitely many distinct subtrees. Our current approach is to allow the programmer to annotate predicate definitions with pragmas, which can be used to decide at run-time when a semantic cycle in the proof search has occurred, however, in the future we intend to infer these annotations by using static analysis.

This dissertation also presents a Top-Down mechanism for goal directed execution of Answer Set programs. This is realized by extending Coinductive Logic Programming to execute Answer Set Programs. Using this goal directed execution of ASP programs, we can do non-monotonic reasoning with general first order predicates. With this approach, we can also avoid the inherent limitation of grounding the ASP programs (before execution) in the bottom-up approach. Our current implementation works for propositional ASP. Future work involves implementing a top-down interpreter for ASP with general predicates [49].

This dissertation outlines a path towards achieving Kowalski's ideal perception of a logic programming language—one which provides complete flexibility in the control regime chosen. It relies on combining deterministic coroutining, CLP, tabled LP, co-LP, ASP and parallelism into a single framework with deterministic coroutining and CLP providing flexible ordering of subgoals while tabling and or-parallelism provide flexible ordering of rules. The biggest hurdle in combining all these techniques into one is the complexity of the implementation techniques devised for realizing them. We outlined simple techniques that facilitate quick implementation of some of these features as well as their integration in a single system.

REFERENCES

[1] XSB Inc. `www.sourceforge.net` and `www.xsb.com`.

[2] AAAI. Planning and Scheduling. `http://www.aaai.org/AITopics/html/planning.html`, 2007.

[3] R. Alur and D. L. Dill. A theory of timed automata. In *Theoretical Computer Science (TCS)*, pages 183–235, 1994.

[4] Y. Babovich. CModels. `www.cs.utexas.edu/users/tag/cmodels.html`, 2002.

[5] C. Baral. *Knowledge Representation - Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[6] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. In *Journal of Logic Programming*, volume 19–20, pages 73–148. Elsevier, New York, 1994.

[7] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. In *Journal of Logic Programming*, volume 31, pages 85–118, 1997.

[8] C. Baral, M. Gelfond, and A. Provetti. Reasoning about actions: laws, observations and hypotheses. In *Journal of Logic Programming*, volume 31, pages 201–244, 1997.

[9] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, 1996.

[10] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 23–42. MIT Press, 1993.

[11] M. Carlsson. The SICStus Prolog System. `http://www.sics.se`.

[12] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. In *Journal of Logic Programming*, volume 24(3), pages 161–199, September 1995.

[13] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. In *Journal of the ACM*, volume 43(1), pages 20–74. Springer Verlag, January 1996.

[14] T.Y. Chew, M. Henz, and K.B. Ng. A Toolkit for Constraint-based Inference Engines. In *Practical Aspects of Declarative Languages*, volume 1753. Springer Verlag, 2000.

[15] M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. In *International Journal of Software Tools for Technology Transfer*, volume 2(1), pages 29–45. Springer Verlag, November 1998.

[16] P. Codognet and D. Diaz. Compiling Constraints in clp(FD). In *Journal of Logic Programming*, volume 27(3), pages 185–226, 1996.

[17] V. S. Costa and R. Rocha. The YAP Prolog System.

[18] V. S. Costa, R. Yang, and et al. Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism. In *(PPoPP)*, pages 83–93, April 1991.

[19] E. Clarke and O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999.

[20] J. A. Fernandez and J. Lobo. A proof procedure for stable theories. In *Technical Report CS-TR-3034, University of Maryland*, 1992.

[21] P. W. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore Discovery Using the Inductive Logic Programming System PROGOL. In *Machine Learning*, volume 30(2-3), pages 241–270, 1998.

[22] M. Gebser and T. Schaub. Tableau Calculi for Answer Set Programming. In *International Conference on Logic Programming (ICLP)*, pages 11–25. Springer, 2006.

[23] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *International Conference and Symposium on Logic Programming*. MIT Press, 1988.

[24] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Conference and Symposium*, pages 1070–1080, 1988.

[25] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.

[26] M. Gelfond and V. Lifschitz. Action Languages. In *Electronic Transactions on AI*, volume 3, 1998.

[27] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: preliminary report. In *AAAI*, pages 623–630, 1998.

[28] J. Goguen and K. Lin. Behavioral Verification of Distributed Concurrent Systems with BOBJ. In *Conference on Quality Software*, pages 216–235. IEEE Press, 2003.

[29] D. Goldin and D. Keil. Interaction, Evolution and Intelligence. In *Congress on Evolutionary Computing*, 2001.

[30] A. Gordon. A Tutorial on Co-induction and Functional Programming. In *Workshops in Computing (Functional Programming)*, pages 78–95. Springer, 1995.

[31] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications (tutorial paper). In *International Conference on Logic Programming (ICLP)*, September 2007.

[32] G. Gupta and E. Pontelli. Constraint-based Specification and Verification of Real-time Systems. In *IEEE Real-time Symposium*, pages 230–239, 1997.

[33] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. In *ACM Transactions on Programming Languages and Systems*, pages 1–126.

[34] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *International Workshop Programming Languages Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[35] B. Jacobs. Introduction to Coalgebra: Towards Mathematics of States and Observation. In *Draft Manuscript*.

[36] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Principles of Programming Languages (POPL)*, pages 111–119, 1987.

[37] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.

[38] S. Peyton Jones and et al. Haskell 98 Language and Libraries, the Revised Report. In *CUP*, April 2003.

[39] H. Ait Kaci. *Warren Abstract Machine: A Tutorial*. MIT Press, 1991.

[40] V. Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, 1999.

[41] V. Lifschitz. Answer set planning. In *ICLP*. Springer Verlag, 1999.

[42] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *LPNMR*, 1999.

[43] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of Logic Programs by SAT Solvers. In *AAAI/IAAA*, pages 112–117. AAAI/MIT Press, 2002.

[44] X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed-points. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[45] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.

[46] A. Mallya. Deductive Multi-valued Model Checking. In *Ph.D. thesis, UT Dallas*, 2006.

[47] W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In *Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[48] S. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. In *Intelligent Systems*, volume 16(2). IEEE, 2001.

[49] R. Min. Coinduction in monotonic and non-monotonic reasoning. In *Ph.D. thesis forthcoming, UT Dallas*.

[50] R. Min, A. Bansal, and G. Gupta. Goal-directed SAT Solvers. In *In preparation*, 2007.

[51] S. Muggleton. Inductive Logic Programming. In *Academic Press*, 1992.

[52] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. In *Journal of Logic Programming*, volume 19-20, pages 629–679, 1994.

[53] S. Mukhopadhyay and A. Podelski. Model Checking for Timed Logic Processes. In *Computational Logic*, pages 598–612, 2000.

[54] G. Pfeifer N. Leone and W. Faber. DLV. `http://www.dbai.tuwien.ac.at/proj/dlv`.

[55] I. Niemel and P. Simons. Smodels: an implementation of the stable model and well-founded semantics for normal logic programs. In *International Conference on Logic Programming and non-monotonic reasoning*, pages 420–429, 1997.

[56] I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Programming and Non-monotonic Reasoning*, pages 421–430. Springer Verlag, 1997.

[57] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog Decision Support System for the Space Shuttle. In *Practical Aspects of Declarative Languages (PADL)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer Verlag, 2001.

[58] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL*, pages 169–183. Springer Verlag, 2001.

[59] B. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[60] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. In *Principles of Programming Languages (POPL)*, pages 132–144. ACM Press, 2005.

[61] E. Pontelli. Experiments in Parallel Execution of Answer Set Programs. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2001.

[62] E. Pontelli and T.C. Son. Smodels with CLP: A Simple Treatment of Aggregates in ASP. In *Logic Programming and Nonmonotonic Reasoning*. Springer Verlag, 2004.

[63] Y. S. Ramakrishna and et al. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification (CAV)*, pages 143–154, 1997.

[64] C. R. Ramakrishnan, S. Dawson, and D. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems: A Case Study. In *ACM PLDI*, 1996.

[65] Y.S. Ramakrishnan and et al. Efficient Model Checking using Tabled Resolution. In *Computer Aided Verification (CAV'97)*, 1997.

[66] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Effciently Computing WFS. In *Logic Programming and Nonmonotonic Reasoning*. Springer Verlag, 1997.

[67] Robert A. Wilson and Frank C. Keil. *The MIT Encyclopedia of the Cognitive Sciences (MITECS)*. The MIT Press, 1999.

[68] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. In *ACM TOPLAS*, volume 20(3), pages 586–635, May 1998.

[69] V. Schuppan and A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. In *ENTCS*, volume 149(1), pages 79–96, 2006.

[70] L. Simon. Extending Logic Programming with Coinduction. In *Ph.D. thesis, UT Dallas*, 2006.

[71] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-Logic Programming. In *International Colloquium on Automata, Languages and Programming (ICALP)*. Springer Verlag, July 2007.

[72] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming. In *International Conference on Logic Programming (ICLP)*, volume 1145 of *Lecture Notes in Computer Science*, pages 330–344. Springer Verlag, August 2006.

[73] L. Simon, A. Mallya, and G. Gupta. Design and implementation of A_T: A real-time action description language. In *International Workshop on Logic-based Program Synthesis and Transformation*. Springer Verlag, 2005.

[74] T.C. Son, E. Pontelli, D. Ranjan, B. Milligan, and G. Gupta. An Agent-based Domain Specific Framework for Rapid Prototyping of Applications in Evolutionary Biology. In *Declarative Agent Languages and Technologies*. Springer Verlag.

[75] L. Sterling and S. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[76] P. Stuckey. *Constraint Programming*. MIT Press, 1998.

[77] T. Syrjnen. Lparse 1.0 Users Manual. `http://www.tcs.hut.fi/Software/smodels`, 2000.

[78] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *International Conference on Logic Programming (ICLP)*, pages 84–98, 1986.

[79] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. In *Journal of Logic Programming*, volume 31, pages 245–298, 1997.

[80] U.D. Ulusar and H.L. Akin. Design and Implementation of a Real Time Planner for Robots. In *TAINN*, pages 263–270, 2004.

[81] P. van Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.

[82] M. Vardi. Verification of Concurrent Programs: The Automata-Theoretic Framework. In *Logic in Computer Science (LICS)*, pages 167–176. IEEE, 1987.

[83] D. H. D. Warren. An Abstract Instruction Set for Prolog. In *Tech. Note 309, SRI International*, 1993.

[84] D. S. Warren. The XWAM: A machine that integrates Prolog and deductive database query evaluation. In *TR 89/25 SUNY at Stony Brook*, 1989.

[85] P. Wegner and D. Goldin. Mathematical models of interactive computing. In *Brown University Technical Report CS 99-13*, 1999.

[86] N. Zhou and et. al. Implementation of a Linear Tabling Mechanism. In *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag, 2000.

VITA

Ajay Bansal was born in Barnala, Punjab, India on October 7th, 1976, the son of Usha Bansal and Ramchander Bansal. After completing his work at Shree Daulatram Nopany Vidyalaya, Kolkata, India, in 1995, he entered National Institute of Technology at Warangal (formerly Regional Engineering College - Warangal). He received the Bachelor of Technology degree in Computer Science and Engineering in May 1999. During the following six months he was employed as a software engineer at Siemens, Bangalore. In January 2000, he entered Texas Tech University and received the Master of Science degree in Computer Science in May 2002. He worked as a software engineer at Tyler Technologies from June 2001 to June 2003. In December 2007, he received his Doctorate degree from the University of Texas at Dallas for his research titled "Towards Next Generation Logic Programming Systems". His research interests are in Logic Programming, Non-monotonic reasoning, Programming Languages, and Semantic Web Services.

Conference Articles:

1. Ajay Bansal, Srividya Kona, Luke Simon, Ajay Mallya, Gopal Gupta, and T.D. Hite; Universal Service-Semantics Description Language, in *Proc. of European Conference On Web Services (ECOWS)*. IEEE Computer Society, Nov. 2005 (Received Best Paper Award)
2. Ajay Bansal, Kunal Patel, Gopal Gupta, et al; Towards Intelligent Services: A case Study in Chemical Emergency Response System. in *Proc. of International Conference on Web Services (ICWS)*. IEEE Computer Society, July 2005
3. Srividya Kona, Ajay Bansal, and Gopal Gupta; Automatic Composition of Semantic

Web Services, in *Proc. of International Conference on Web Services (ICWS)*. IEEE Computer Society, July 2007.

4. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta; Co-Logic Programming: Extending Logic Programming with Coinduction, in *Proc. of International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, July 2007.
5. Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya; Coinductive Logic Programming and its Applications, in *Proc. of International Conference on Logic Programming (ICLP)*. Springer, Sep. 2007.
6. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta; Coinductive Logic Programming, in *Proc. of International Conference on Logic Programming (ICLP) at FLoC*, Aug. 2006.
7. Srividya Kona, Ajay Bansal, Gopal Gupta, and T.D. Hite; Semantics-based Web Service Composition engine (Short Paper), in *Proc. of Conference on E-Commerce Technology and Conference on Enterprise Computing, E-Commerce and E-Service (CEC/EEE)*. IEEE Computer Society, July 2007.
8. Srividya Kona, Ajay Bansal, Gopal Gupta, and T.D. Hite; Web Service Discovery and Composition using USDL (Short Paper), in *Proc. of Conference on E-Commerce Technology and Conference on Enterprise Computing, E-Commerce and E-Service (CEC/EEE)*. IEEE Computer Society, June 2006.
9. Srividya Kona, Ajay Bansal, Gopal Gupta, and T.D. Hite; Efficient Web Service Discovery and Composition using Constraint Logic Programming, in *Proc. of International Workshop on Applied Logic Programming Semantic Web and Services (ALPSWS) at FLoC*, August 2006.
10. Luke Simon, Ajay Bansal, Ajay Mallya, Srividya Kona, Gopal Gupta, and T.D.Hite; Towards a Universal Service Description Language, in *Proc. of Next Generation Web Services Practices (NWeSP)*, Sep. 2005.
11. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta; Combining Traditional

and Coinductive Logic Programming, in Proc. of International Workshop on Software Verification and Validation (SVV) at FLoC, Aug. 2006.

Technical Reports:

1. Ajay Bansal, Richard Min, Gopal Gupta; Goal-directed execution of Answer Set Programs, Internal Report; under review for the European Joint Conferences on Theory and Practice of Software (ETAPS), 2008
2. Srividya Kona, Ajay Bansal, Gopal Gupta, and T.D. Hite; USDL: Formal definitions in OWL, Internal Report UTDCS-23-07
3. Ajay Bansal, Kunal Patel, Gopal Gupta, et al; Towards Intelligent Services: A case Study in Chemical Emergency Response System, Internal Report UTDCS-22-07 (Submitted to SP&E)
4. Srividya Kona, Ajay Bansal, Gopal Gupta; Generalized Semantics-based Service Composition, Internal report; under review for Software Composition Symposium (SC), 2008
5. Srividya Kona, Ajay Bansal, Luke Simon, Ajay Mallya, Gopal Gupta, and T.D. Hite; USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition, Internal Report UTDCS-18-06 (Submitted to IJWSR)
6. Ajay Bansal, L Pyeatt; Monte Carlo Localization of Mobile Robots in Dynamic Environments, Masters Thesis.
7. Ajay Bansal, Sanjeeva Reddy; Design and Implementation of Network Conferencing in Multimedia Systems, B.Tech Project Report.

Permanent Address: 150, Flat 2A,
Manicktalla Main Road,
Kolkata - 700054,
West Bengal,
INDIA.