National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

AVIS

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

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

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

Canada

TEMPORAL DEDUCTIVE DATABASES: QUERY FINITENESS AND A QUERY LANGUAGE

Daniel A. Nonen

A Thesis

in

The Department

of

Computer Science

Presented in the Partial Fulfillment of

the Requirements for the Degree of

Master of Computer Science at

Concordia University

Montréal, Québec, Canada

September 1993

ISBN 0-315-90921-8

ABSTRACT

Temporal Deductive Databases: Query Finiteness and a Query Language

Daniel A. Nonen

Temporal databases have been proposed in the literature to provide a uniform framework for modeling information that changes over time. Temporal relational databases append time information to tuples in the form of either valid intervals or event times. They also define special operators and integrity constraints to handle temporal data. We extend this approach to temporal deductive databases. Interesting temporal rules need function symbols, but the use of function symbols raises the possibility that queries may have an infinite number of answers. Unfortunately, detecting such queries is undecidable in general. A recent proposal is to test for a stronger condition than finiteness, called superfiniteness. However, the complexity of the only known algorithm for testing this property is exponential. We develop polynomial time decision procedures to detect superfiniteness for certain classes of linear programs.

We propose a Horn clause based query language called TKL (Temporal Knowledge-base Language). TKL incorporates a screen-based editor which visually associates attribute names with Horn clause arguments. In addition to the usual built-in **Date** data type for representing date and time, user-defined temporal data types are allowed. Recognizing that real-world measurement of time is inherently imprecise, we introduce two interpretation policies called *broad* and *narrow* to allow reasoning with such knowledge. TKL is implemented within a DBMS architecture which is modular and extendible.

Contents

Chapter 1

Introduction

Temporal deductive databases are informally introduced in this chapter by first discussing temporal databases and then extending the discussion to temporal deductive databases. This is followed by a description of the organization of the thesis.

1.1 Temporal Relational Databases

Information about the objects modeled by databases changes over time in real-world applications. However, time is not managed in a uniform manner in most databases. Temporal databases have been introduced to fill this gap. A survey of works on historical and temporal databases can be found in Soo [Soo 91]. The idea of representing time as an attribute was suggested by Clifford and Warren [CW 82]. They treat time as a linear sequence of instants which are isomorphic to the natural numbers. An event is .epresented by a

single time instant and an interval is represented by a sequence of consecutive instants. Temporal databases have to model objects in complex ways. For example, a student may be enrolled at a university, then drop out, and then enroll again at a later time. The student should be allowed to register in a course only if the course is offered for registration at the same time as the student is enrolled.

Temporal databases which associate an event time or an interval with each piece of temporal data are considered primarily in this work. This association can be achieved by adding a *from* and a *to* attribute to data objects. For example, if t is a tuple in a temporal relation, then $t.From$ is the first time that t is valid and $t.To$ is the final time t is valid. (If t is currently valid, the value of $t.to$ is the special variable NOW which evaluates to the current system time.)

The following problems arise in the context of temporal databases:

1. Writing queries is more difficult because even 'simple' queries may require many temporal comparisons;
2. Maintaining database consistency is problematic: for example, since data is not deleted when updates are made, many tuples will have the same 'key';
3. Query evaluation may be inefficient unless the properties of temporal attributes are recognized and exploited.[1]

Temporal database languages usually have special features to adapt them for use with temporal data. Two such languages for relational databases are

[1] Efficient temporal query processing techniques are not discussed in this work; see *e.g.*, [GS 91].

TQuel [Snod 87], a superset of Quel, and HSQL [Sard 90], a superset of SQL. Both these proposals, which are described in detail in Chapter 2, include database management system (DBMS) support for maintaining database consistency. They offer useful operators for joining relations that *precede*, *contain*, or *overlap* by performing the appropriate comparisons of the temporal intervals of the tuples being joined. TQuel extends Quel's aggregate functions to the temporal domain. However, since both HSQL and TQuel are based on relational algebra, they lack the expressive power of a deductive database language supporting recursion.

1.2 Temporal Deductive Databases

Deductive databases, with their increased expressive power over relational databases have been recognized as one of the important data models for next generation applications [T 91]. However, Datalog, a vehicle query language for deductive databases which is based on Horn clauses, lacks the power of function symbols. Function symbols are important for (*i*) better data structuring ability, (*ii*) applications involving streams or list constructors, and (*iii*) applications like temporal deductive databases (*e.g.*, see [CI 88] or [LN 92b]).

A query to a deductive database is expressed using a set of Horn clause rules, often called a query program. The query is then answered against the least fixpoint model (*e.g.*, see [Ull 89]) of this program, which intuitively is the set of facts derived by the rules from the facts in the database. In

the presence of function symbols, answers to certain queries can be infinite.[2] Detection of finite queries is fundamental to the design of query systems: finiteness analysis is an integral part of such systems as NAIL! [Ull 89], LDL [Chim 89], and SYGRAF [KL 88]. Recent works (*e.g.*, see Brodsky and Sagiv [BS 89, BS 91], Sohn and Van Gelder [SG 91]) also show applications of finiteness analysis to the detection of termination of top-down evaluation of logic query programs.

1.2.1 Finiteness of Query Answers

Function symbols are needed to write interesting temporal queries. However, when they are allowed, some difficult questions about the finiteness of query answers have to be answered. Consider the following example in which the successor function is used.

Example 1.1 An airline stores facts about daily inter-city flights in the following relation.

```
daily_flight(Dep, Arr, Date).
```

The intended meaning is that there is a daily flight from the departure airport `Dep` to the arrival airport `Arr` on the date `Date`. The following two rules define the `can_fly` relation. A Prolog-like notation is used for expressing rules.

```
can_fly(Dep, Arr, Date)
      :- daily_flight(Dep, Arr, Date).
```

[2] As a notational shorthand, we say a query is infinite (finite) when the answer to the query has an infinite (finite) number of tuples.

```
can_fly(Dep, Arr, Date + 1)
      :- can_fly(Dep, Arr, Date),
         daily_flight(Dep, Arr, Any_Date).
```

The first rule initializes the `can_fly` relation to the contents of the `daily_flight` relation. The intended meaning of the second rule is that if one can fly between two destinations on a particular date, and there is a daily flight between those two destinations, then one can fly on the same flight on the following day. Suppose the `daily_flight` relation contains the tuple `<montreal, toronto, Jan 1,1992>`. Consider the query `?- can_fly( montreal, toronto, T)`, in which `montreal` and `toronto` are interpreted as constants and `T` is interpreted as a variable to be matched to flight dates. This query asks for all flights from Montreal to Toronto. The set of answers to this query is clearly infinite. □

In the context of temporal deductive databases, two approaches have been considered for handling finiteness of queries. The first approach is to *allow* certain classes of "meaningful" infinite query answers, and to develop finite representations for their answers. This is the approach taken by Chomicki and Imieliński [CI 88]. The second approach, which is pursued in Chapter 3, is to detect and disallow infinite queries where possible.

Shmueli [Sh 87] showed that query finiteness is in general undecidable for programs with function symbols. References to other decidability results on query finiteness for different classes of queries can be found in Kifer *et al.*[KRS 88]. Ramakrishnan *et al.* [RBS 87] developed a framework for finiteness analysis in which programs with function symbols are approximated by

function-free programs with *infinite* base relations satisfying *finiteness constraints* (FCs). Intuitively, FCs assert that if certain columns in a relation have a finite number of values, so will some other column(s). For example, consider an infinite relation $s(A, B, C)$. The relation s satisfies the FC $AB \rightarrow C$ exactly when it associates a finite number of C-values with any given AB-value. Among other things they also showed that finiteness is decidable for monadic programs, *i.e.*, programs where all IDB predicates are monadic.

Sagiv and Vardi [SV 89] showed that finiteness (in this framework) is the conjunction of a property called *weak finiteness* and *termination.* They showed that while weak finiteness is decidable, termination is in general undecidable. They also furnished a polynomial time algorithm for detecting finiteness for monadic programs. It should be noted that, for general programs, the decidability of finiteness in the presence of infinite base relations and FCs is still open [KRS 88]. Kifer *et al.* [KRS 88] proposed a stronger notion of finiteness called *superfiniteness* which refers to a query answer being finite in all fixpoint models of the program, as opposed to only in the least fixpoint model. Intuitively, a model M of a program P is a fixpoint model if for every fact in M there is a rule in P that justifies this fact. They have developed a complete axiom system, and a decision procedure for superfiniteness. They also extend their procedure to detect finiteness for certain classes of query programs. While their contribution is fundamental and significant, the time complexity of their algorithm for detecting superfinite query programs is exponential in the size of a set of constraints generated[3]. The size

[3]The main concern of that paper was proving decidability and axiomatizability of superfiniteness.

of the latter set is polynomially larger than the original program size.

1.2.2 Temporal Deductive Database Systems

The extension of temporal databases to temporal deductive databases is introduced using three proposals for them as examples. Two of these examples, TEMPLOG and STATELOG, are discussed in Chapter 2. Our own proposal, TKL, is the subject of Chapter 4.

Temporal logic is the formalism used for the logic programming language TEMPLOG [AM 87]. This formalism gives the language a great deal of elegance and expressive power. However, the user is limited to using the abstract temporal operators *next*, *always*, *eventually*, *until*, and *precedes* as modifiers to first-order logic predicates. Such operators may be unfamiliar and difficult for the uninitiated to use as a query language. Also, in the context of (temporal) databases, the expressiveness offered by these primitives is severely limited. TEMPLOG has a Prolog-style execution model, and therefore, it is not truly declarative.

STATELOG [CI 90] is based on Datalog extended by the successor function to handle temporal queries. Its strength is the capability to determine a finite representation for infinite query answers when the data is periodic in nature. Unfortunately, due to the high computational complexity of the problem, a finite representation can be determined for only certain periodic programs. Also, temporal database programming using only the successor function can be cumbersome.

TKL (*T*emporal *K*nowledge-base *L*anguage) is also based on Datalog. A proposal for TKL is the subject of Chapter 4. Horn clause query programs are written in TKL by filling in *forms* using a special-purpose screen editor. Temporal queries that express even simple temporal relationships may be long and difficult to write because they can involve many comparisons of temporal attributes. A design objective of TKL is to reduce these problems by (*i*) incorporating the temporal predicates alluded to in Section 1.1 to make temporal queries concise, and (*ii*) associating attribute names visually to Horn clause arguments in a manner reminiscent of the relational database language QBE [Zlo 77]. Many relationships between temporal attributes can be expressed neatly using forms. For example, if the answer to a query must be restricted to tuples of relation R_1 and relation R_2 beginning at the same time, the same variable is entered into the *From* fields of the forms for R_1 and R_2. When it is more convenient, temporal relationships may be stated as expressions in the "Conditions" section of the query.

The approach taken in TKL to detecting query finiteness is to test for the superfiniteness of certain classes of programs. In order to express notions involving a progression in time, the successor function must be applied to temporal values. For example, a TKL user may wish to add a rule to infer that if a condition is true today, then some condition will also be true tomorrow. However, it is necessary to assure that queries using this rule are finite.

1.3 Contributions made by this Thesis

The contributions made by this thesis fall into two categories: query finiteness and temporal deductive database design. Our results for query finiteness are listed first.

- A conceptual aid for understanding superfiniteness called *chased R/G trees*;
- Identification of a class of programs for which superfiniteness analysis can be carried out efficiently. This is formalized using the notion of compositionality;
- The concept of *permissive* non-deterministic finite automata, a polynomial time algorithm for determining when automata have this property, and a characterization of superfiniteness in terms of permissiveness for compositional linear programs and for linear programs with one recursive rule.

In the area of temporal deductive database design, we make the following contributions.

- An extension of temporal relational databases to temporal deductive databases that resolves the question of query finiteness for certain classes of programs by means of superfiniteness analysis;
- The introduction of user-defined temporal data types;

- A mechanism for reasoning with imprecise temporal data and temporal null values using *broad* and *narrow* interpretation policies;
- A query language that uses a graphic user-interface in an essential manner to express query programs clearly and concisely.

1.4 Organization of the Thesis

The necessary background for the development of the thesis is presented in Chapter 2. The fundamentals of relational and deductive databases are presented, and a brief discussion of the temporal relational database systems TQuel and HSQL and of the temporal deductive systems TEMPLOG and STATELOG is given as well. In Chapter 3, polynomial time algorithms are presented for detecting superfiniteness for certain classes of programs. The details of the design and the implementation of TKL are given in Chapter 4 and, in Chapter 5, TKL is compared and contrasted to the systems introduced in Chapter 2. Conclusions are given in Chapter 6.

Chapter 2

Background

This chapter contains a review of the basic concepts needed for the development of the thesis. Notions from relational databases are described first. These notions are then extended to temporal relational databases and to temporal deductive databases.

2.1 Relational Databases and Temporal Relational Databases

The relational model was introduced in 1970 by Codd [C 70]. Discussions of the basics of relational databases can be found in this work as well as in text books by Date [Da 86], Desai [D 90], and Ullman [Ull 89]. A relational database can be considered to be a collection of tables. The column headings, or attribute names, are given in a fixed order so that each row, or tuple, in the table is a fixed sequence of values. The values that can appear in a column

are taken from a domain of values such as the integers, the set of strings of a certain length, *etc.* A relation is a set of tuples.

An *n-ary* relation is any subset of the Cartesian product of n domains. For each relation, there is a set of attributes called a *key* that uniquely identifies each tuple in the relation. A set of attributes S is a key for a relation R if (i) there are no two tuples r_1 and r_2 in R such that they have the same values for the attributes in S and yet $r_1 \neq r_2$; (ii) there is no subset of S that has property (i).

The relational algebra is briefly reviewed next. Let R and S be any two relations.

$R \cup S$ **(Union):** $t \in R \cup S$ if $t \in R$ or $t \in S$. Union is defined only when R and S have the same arity.

$R - S$ **(Set difference:)** $t \in R - S$ if $t \in R$ and $t \notin S$. Like union, set difference is defined only when the arities of R and S are the same.

$R \times S$ **(Cartesian product:)** Let the arity of R be k_1 and the arity of S be k_2; $t \in R \times S$ if t is a $(k_1 + k_2)$-ary tuple such that the first k_1 attribute values of t form a tuple in R and the last k_2 attribute values form a tuple in S.

$\pi_{i_1,\ldots,i_k} R$ **(Projection):** $t \in \pi_{i_1,\ldots,i_k} R$ iff t is a k-tuple and there is a tuple $s \in R$ such that the first value of t is equal to the $i_1{}^{th}$ value of s, ..., and the k^{th} value of t is equal to the $i_k{}^{th}$ value of s.

$\sigma_F R$ **(Selection):** Let F be a predicate expression such that

(i) its operands are either constants or attributes of R;

(*ii*) its arithmetic comparison operators are taken from $<, =, \leq, \neq$, and $\geq$;

(*iii*) its logical connectives are $\wedge$ and $\vee$ and its negation operator is $\neg$.

Then $t \in \sigma_F R$ if $t \in R$ and F is true when the values in t are substituted for the corresponding attributes in F.

The Cartesian product and the other relational algebra operators can be used to derive a new relation, called a *view*. Informally, a view can be thought of as a window through which the user can observe data that may actually be contained in several different relations. Usually, but not always, data shown in a view can be modified if all the modified columns are from the same underlying relation.

2.1.1 TQuel: A Temporal Extension to Quel

TQuel, developed by Snodgrass [Snod 87], is a superset of the relational database language Quel that is designed to model both data that changes over time and to allow the state of the database to be "rolled-back" to its state at some time in the past. A useful taxonomy of databases is also given in [Snod 87] which partitions them into four categories. *Snapshot* databases do not model time in a uniform way. In such a database, updating a tuple may cause some data to be lost. *Rollback* databases are snapshot databases whose data is extended to include a unique representation for the transaction times using transaction identifiers which have a many-to-one correspondence to the system date and time. A transaction identifier is added to the data when

it is physically inserted in the database and another transaction identifier is added if the data is ever considered incorrect, perhaps due to an input error. Such a feature can be extremely useful for queries about, for example, why a particular action was taken in the past, which may have been based on information that was later considered to be incorrect. *Historical* databases are an extension to snapshot databases to model data that changes over time. This can be achieved adding *valid from* and *valid to* information that is considered to be true over an interval. This is referred to as an *interval* relation in a relational database. Data that is considered to be true at a particular time instant can be modeled by adding *valid at* information indicating that the data is considered to be true at that particular time instant. This is referred to as an *event* relation in a relational database. Finally, according to this taxonomy, a *temporal* database is one that combines the time-handling features of roll-back databases and of historical databases.[1]

TQuel extends Quel by adding to Quel's *range, retrieve*, and *where* clauses new clauses for handling the temporal domain. The new clauses, *valid from ... to, when*, and *as of ... through*, are formally described using the tuple relational calculus on which Quel is based. The definitions of tuple variables are extended to allow them to represent the intervals or event times of historical relations. The *when* clause is the temporal analogue of the *where* clause in a Quel query: it is used to determine if a tuple satisfies the historical constraints of a query. A tuple variable by itself is used to represent an interval or an event time in the *when* clause. The operators *begin of* and *end of* can

[1] The use of the word temporal was revised in [JCGSS] to refer to databases that support some interpreted aspect of time. Thus, both historical databases and roll-back databases can be called temporal databases. Databases which combine the two are referred to as *bitemporal* databases.

be used to extract the end points of an interval represented by an interval tuple variable.

TQuel provides the predicates *precede, overlap,* and *equal,* as well as *and, or,* and *not* for expressions comparing intervals and events. Events are coerced into intervals that begin and end at the event time. Let α and β be interval or event tuple variables. Then α *precede* β is true if *end of* $\alpha \leq$ *begin of* β. The intervals represented by α and β are said to overlap if *begin of* $\alpha \leq$ *end of* β and *begin of* $\beta \leq$ *end of* α. Finally, $\alpha = \beta$ if α and β are represent events that occurred at the same time, or they represent intervals that began and ended at the same time. Overlap is overloaded in the sense that it returns the "intersection" of the intervals of its operands if they do overlap. In the same spirit, the operator *extend* is provided to return the "union" of the intervals if they overlap, that is, α *extend* β returns the interval containing every instant of time contained in α and β.

The semantics of TQuel are given in terms of tuple relational calculus. Statements in tuple relational calculus are expressions of the form $\{t^{(i)} \mid \psi(t)\}$ where t is a variable denoting a tuple of arity i and $\psi(t)$ is a first-order predicate calculus expression whose only free tuple variable is t. A TQuel query has the following form. The tuple variable t_i, $1 \leq i \leq k$, ranges over the relation R_i and D_{j_m} is used to represent the attribute name of the j_m^{th} column of relation R_{i_m}.

range of t_1 is R_1

$\vdots$

range of t_k is R_k

retrieve $(t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r})$

valid from ν to χ

where ψ

when τ

as of α through β

The *range of, retrieve,* and *where* clauses are unchanged from Quel. The contents of the *when* clause, τ are essentially made up of conditions on tuples stated using tuple variables and the temporal predicates and operators described previously.

The *as of* α *through* β clause determines the time-slice in which the query is to be evaluated. If the values of the expressions α and β are τ_α and τ_β respectively, then for all t_l, $1 \leq l \leq k$, t_l must be such that τ_α is before $t_l[stop]$ and $t_l[start]$ is before τ_β, where $t_l[start]$ is the transaction identifier referring to when t_l was inserted into the database and $t_l[stop]$ refers to the transaction identifier for when t_l was marked as incorrect.

Default values are assigned to TQuel clauses to provide the same semantics as Quel when the temporal clauses are not used. For example, if the tuple variables t_1 and t_2 are used in a query in which the *when* clause is unspecified, the default value of the *when* clause is (t_1 *overlap* t_2) *overlap* NOW), that is, tuples t_1 and t_2 are used in the calculation of the answer for the query only if the intersection of their valid intervals overlaps and that overlap interval also overlaps the current time. Other defaults are defined for the *valid from* and *as of* clauses. When a query uses some, but not all of the default values, the user must verify that the system supplied values are appropriate.

TQuel makes a significant contribution to temporal database technology by providing a simple syntax and semantics for both rollback and historical databases. In addition, it has been largely implemented in Quel. However, the meaning of a few of its predicates may be confusing to some users. For example, the temporal predicate expression t_1 *precedes* t_2 is true even when t_1 and t_2 have a time in common, *i.e.*, when *end of* t_1 is equal to *begin of* t_2. This seems to be the result of the design decision to use $\leq$ instead of $<$ to determine when one time instant precedes another time instant. Also, t_1 *overlap* t_2 is true even when t_1 is entirely contained in t_2, *i.e.*, there is no overlap in the non-technical sense.

TQuel is not consistent in its treatment of temporal predicates and temporal operators. For example, the predicate *overlap* has an operator counterpart which returns the intersection of its operands, while *equal* and *precede* do not have operator counterparts. In an attempt to increase the usefulness of this approach, the *extend* operator returns the union of overlapping intervals. While the values returned by *overlap* and *extend* are useful for writing queries, the choice of these operators (and not all predicate/operator pairs) is arbitrary.

Temporal attributes are allowed to have different granularities. However, no formal methodology is provided for resolving the problems associated with them, such as how to compare temporal terms having different granularities and how to interpret the imprecision inherent in time units that are longer than one tick.

2.1.2 HSQL: A Temporal Extension to SQL

HSQL is a proposal to extend the relational query language SQL to include facilities for queries about the history of relations. It was put forward by Sarda [Sard 90] as a complete historical database management system (HDBMS). HSQL relations can be used to model "real-world" time in two ways:

1. Relations called *state* relations have a *FROM* and a *TO* attribute associated with them, which are used to indicate the time interval in which a database object "prevails".

2. Relations called *event* relations have a *FROM* and a *TO* attribute associated with them also. However, they are actually degenerate interval relations in the sense that the *FROM* and *TO* values must be the same. Alternately, an event relation can have an *At* attribute associated with it. Event relations are used to model objects which prevail for one time unit.

Attributes are divided into two types. The *timing* attributes are *FROM*, *TO*, and *AT*. All the other attributes are called *visible* attributes. The timing attributes are automatically added to historical relation schemes if they are defined to be either state or event relations. The database is divided into two segments. The *current* segment corresponds roughly to a non-historical database. The value of the *TO* attribute of all tuples in the current section is NOW, a special 'moving' variable whose value is the current system time. All other historical tuples are considered part of the *historical* segment. When queries are written, the segment to which they apply can be specified.

When an historical relation is defined, the *granularity* of the timing attributes must also be specified by the database designer. The allowed granularities are prefixes of the format yy:mm:dd:hh:MM:ss, where yy is a year, mm is a month, dd is a day, hh is an hour, MM is a minute, and ss is a second. Operators are given to inspect fields of formatted time values and the successor/predecessor function for time values is defined. See [Sard 90] for the details.

Historical databases require integrity constraints (ICs) in addition to those needed for traditional relational databases. The HSQL proposal puts forward the following additional ICs.

1. Let R be an historical relation; let r and s be tuples belonging to R; let t be any instant of time. Then a set of attributes A taken from R is a key for R if $r.\bar{A} \neq s.\bar{A}$ and the intervals in r and s both contain t, for all s, r, and t.

2. Tuples with the null interval are not stored.

3. Concurrent tuples with the same visible attributes are coalesced into one tuple having the least FROM value and the maximum TO value of the concurrent tuples.[2]

The predicates $<, <=, =, >=, >$, and */= (not equal)* are provided for comparing time values. The following predicates are provided for comparing intervals. Let t be a time instant and let p, p_1, and p_2 be intervals.

- t *in* p is true iff t is included in p;

[2]The FROM and the TO attribute values, representing real-world time, come from an internal clock.

- $p_1 = p_2$ is true iff p_1 and p_2 contain the same time instants;
- p_1 *overlap* p_2 is true iff p_1 and p_2 have at least one time in common;
- p_1 *contains* p_2 is true iff all instants in p_2 are also in p_1;
- p_1 *meets* p_2 is true iff p_1.TO+1 = p_2.FROM;
- p_1 *adjacent* p_2 is true iff either p_1 *meets* p_2 or p_2 *meets* p_1;
- p_1 *precedes* p_2 is true iff p_1.TO< p_2.FROM.

Operators are defined for making an interval from two time instants, for concatenating overlapping or consecutive intervals, and to extract the common parts of overlapping intervals. Provision is made for comparing time instants with different granularities by first converting both times to intervals having the finer granularity of the two times, and then comparing these intervals.

Although HSQL provides the five operations usually needed to show a relational database language is complete [Ull 89], *i.e.*, union, set-difference, selection, projection, and Cartesian product, they are not sufficient to show completeness for a relational language extended with valid intervals. The reason is that query answers may depend on time instants while tuples are considered to be valid during an interval. For example, a tuple may have a valid interval of Sept. 1, 1993 to Sept. 30, 1993 and a query may ask if it is valid on Sept. 15, 1993. Since this date is not explicitly represented, the query will fail for the tuple unless the valid interval is interpreted correctly.

To handle this mismatch, the relational algebra is extended with two new operators: *expand*, denoted by e, which takes an historical relation for its argument and returns a set of tuples, one tuple for each instant that tuples

in the relation are valid, and *coalesce*, denoted by c, which also takes a relation for its argument and combines tuples with the same visible attributes into one tuple with the appropriate interval if they are concurrent or consecutive. Now the historical variant of the Cartesian product, called the *concurrent product*, denoted by $\times_t$, can be defined to be

$$R_1 \times_t R_2 = c(\pi_A(\sigma_F(e(R_1) \times e(R_2))))$$

where R_1 and R_2 are historical relations with visible attributes $\bar{X}_1$ and $\bar{X}_2$ respectively, $A = \bar{X}_1 \cup \bar{X}_2 \cup \{R_1.\text{FROM}, R_2.\text{TO}\}$, and F is $R_1.\text{FROM} = R_2.\text{FROM}$. Note that, unlike other definitions of the Cartesian product, one set of the join attributes is projected out of the resulting relation. The *time-slice* of R_1, w.r.t. the interval $[t_1, t_2]$, is defined to $R_1 \times_t \{< t_1, t_2 >\}$, which is the set of tuples of R_1 which are valid between times t_1 and t_2.

Using these extensions to the relational algebra, the syntax and semantics can be stated concisely. Queries in HSQL have the following syntax. Optional elements are indicated by square brackets.

[FROMTIME ... TOTIME ...]
SELECT [COALESCED] ...
FROM [CONCURRENT] ...
[WHERE ...]
[EXPAND BY ...]
[GROUP BY ...]
[HAVING ...]

This differs from SQL in the following ways. FROMTIME and TOTIME denote a time-slice, whose scope is all relations in the query. If COALESCED, the result of the query is coalesced using c. If CONCURRENT, the concurrent product is applied to the relations on the FROM line instead of the Cartesian product. If EXPAND BY, apply e with the appropriate granularity: that is, the level of granularity up to which tuples of relations on the FROM line are expanded. The key-words *historical* or *current* can be placed after relation names on the FROM line to indicate the appropriate segment.

HSQL offers a uniform extension to SQL to handle historical data. Among its contributions are (*i*) it provides an intuitive extension to SQL syntax, (*ii*) it develops the integrity constraints needed to maintain consistency in historical relational databases, and (*iii*) it provides an extension to relational algebra that is complete when historical tuples are stored with valid intervals while queries can be asked concerning valid time instants.

2.2 Deductive Databases

Deductive databases can be described with reference to logic programming which is treated formally in Lloyd [Ll 87]. Datalog, which is described in Ceri *et al.* [CGT 89] and in Ullman [Ull 89], is a function-free and negation-free logic programming language that has been developed to describe deductive databases. Let us first review some important preliminaries from logic programming. A *term* is defined recursively to be (*i*) a constant symbol, denoted by a string beginning with a lower-case character, or a variable symbol, denoted by a string beginning with an upper-case character, or (*ii*) a function

symbol applied to a tuple of terms. Examples of terms are the variable symbol X, the constant symbol *bob*, and the functional term $f(X)$, in which f is a function symbol. An *atom* is a predicate symbol followed by a (possibly empty) list of terms. For example, *rich(bob)* is an atom with predicate symbol *rich* of arity 1 and a singleton list of terms *bob*. A *literal* is an atom or its negation. A *Horn clause* has the form L_0 :- $L_1, \ldots, L_n$. where L_0 is an atom and each L_i, $1 \leq i \leq n$ is a literal. L_0 is called the *head* $L_1, \ldots, L_n$ is called the *body*. If $n = 0$, the Horn clause is called a *fact*. For example, *rich(bob)* is a fact asserting that the property *rich* holds for Bob. If $n > 0$, the Horn clause is called a *rule*. A predicate which is the head of a rule is called a *derived* predicate. For example, *rich(X)* :- *republican(X)*. is a rule asserting that if *republican(X)* is true, then *rich(X)* must also be true. The predicate *rich* is a derived predicate.

The *dependency graph* of a logic program Π is a graph with nodes labeled with the predicates in Π and, for all predicates p and q in Π, there is an arc from q to p if p is the head of a rule in Π and q occurs in its body. Π is *non-recursive* if there are no directed cycles in the dependency graph associated with Π, otherwise Π is *recursive* and q and p are *mutually* recursive predicates. Π is linear if there is at most one predicate q mutually recursive with p for every rule in Π. A recursive program is *linear* if no predicate P occurs more than once in any cycle in its associated dependency graph. An example of a linear recursive program is given in Example 2.2 below.

Logical predicates and relations in a relational database have a natural correspondence. Let p be a predicate symbol of arity n; let P be a relation having n attributes; let $\bar{t}$ be a tuple of n terms. We say p *corresponds* to P

(and vice versa) iff $\forall \bar{t} : p(\bar{t}) \Leftrightarrow < \bar{t} > \in P$. A relation is said to be in the *extensional database* (EDB) if the tuples in the relation are given data (fact relations). The *intensional database* (IDB) is made up of the relations corresponding to the derived predicates. It does not have an exact counterpart in relational databases. If all the rules defining p are non-recursive, then P corresponds to a view in a relational database. However, if p is defined by one or more recursive rules, P cannot be expressed in a relational database. It is this difference in expressive power that essentially accounts for the difference between deductive and relational databases.

The following example makes these definitions concrete.

Example 2.2 Consider a database for an airline which uses the EDB relation FLIGHT to store information about its flight schedule.

flight(Num, Depart_city, Arrive_city, At).

Its attributes are interpreted as *Num* being the flight number, *Depart_city* the departure city, *Arrive_city* the destination city, and *At* the arrival date and time represented as an integer. The corresponding EDB predicate is *FLIGHT*. The IDB predicate *can_fly*, which determines the transitive closure of the FLIGHT relation, is defined by the following two rules.

can_fly(D, A, At) :- flight(N, D, A, At).
can_fly(D, A, At) :- can_fly(D, A', At'), flight(N, A', A, At), At' < At.

The FLIGHT relation contains the following tuples.

<#001, montreal, quebec, 1>

<#002, quebec, chicoutimi, 2>

<#003, chicoutimi, montreal, 3>

The rules for *can_fly* imply the following tuples for the CANFLY relation.

<montreal, quebec, 1>

<quebec, chicoutimi, 2>

<quebec, montreal, 3>

<chicoutimi, montreal, 3>

<montreal, chicoutimi, 2>

<montreal, montreal, 3> □

The tuples belonging to an IDB relation P are obtained by substituting constants for the variables in the RHS of each rule for p that make the rule body true. We then infer that the derived tuples, formed from the head of p by uniformly replacing the variables in p with these constants, also belongs to P. Details of the procedure are presented on page 27.

Only programs which satisfy a condition called *safety* are considered here. We do not allow programs with rules such as *P(X, Y) :- Q(X)* or *P(X, Y) :-* $X \neq Y$ because they do put any 'restrictions' on Y, in the first rule, or on X and Y, in the second rule, except that their values be taken from their respective domains. We define *limited* variables ([Ull 89]) to disallow such programs as follows. A variable X is limited if (i) it occurs in an ordinary predicate (*i.e.*, not a built-in predicate such as $\neq$, $=$, or $<$) in the body, (ii) it occurs in a subgoal $X = c$ where c is a constant, or (iii) it occurs in a

subgoal $X = Y$ where Y is limited. A program Π is *safe* if (i) every fact in Π is variable-free, (ii) every variable occuring in the head of any rule also occurs in its body, and (iii) every variable X in Π is limited.

There is a simple correspondence between non-recursive Horn clause programs and relational algebra. Informally, a rule r is transformed into a relational algebra expression as follows. The Cartesian product of the relations corresponding to the body predicates of r is taken and the attributes corresponding to predicate arguments with shared variables are equated. Attributes corresponding to arguments in the head of r are projected from this product. Some complications due to constants appearing in rules, repeated variables within a predicate, and due to built-in predicates are not treated here. The reader is referred to Ullman [Ull 89] for details. Let p be the head of a rule r. The relation P is the union of the relations contributed by each rule with head p. The procedure is illustrated next using the program from Example 2.2. The exit rule has the following relational algebra translation.

$$\pi_{2,3,4}(FLIGHT).$$

The recursive rule cannot be expressed in relational algebra, but the result of one application of it can be represented as follows.

$$\pi_{\$1,\$6,\$7}\sigma_F(CANFLY \times FLIGHT) \subseteq CANFLY$$

where $F = ((CANFLY.2 = FLIGHT.2) \wedge (CANFLY.3 < FLIGHT.4))$.

The above expression can be evaluated repeatedly, each time taking the union of the result and the previously determined $CANFLY$ relation and

using the union in the next iteration. After a possibly infinite number of iterations, no new tuples will be added when the union is taken. The result is called a *fixpoint* model of this program. If the *CANFLY* relation was originally empty, it is called the *least fixpoint* model of the program. This evaluation is called *bottom-up* because it starts with the facts in the database (as opposed to *top-down* evaluation which starts with the relation whose value is required). This evaluation procedure for logic programs is called *naive* evaluation [Ban 85]. A more efficient variant of this, which guarantees never to derive a tuple twice in the same way, is called *semi-naive* evaluation [BR 86]. We define the *least fixpoint (LFP) model* of a logic program Π w.r.t. a set of relations D as follows. Let $\Pi^i(D)$ represent the result of i iterations of bottom-up evaluation of Π applied to D. Then $LFP_{\Pi(D)}$ is defined inductively as follows: (i) $\Pi^0(D) = D$, (ii) $\Pi^{k+1}(D) = \Pi^k(D) \cup \Pi(\Pi^k(D))$, (iii) $LFP_{\Pi(D)} = \cup_{k=0}^{\infty}\Pi^k(D)$. Note that the number of iterations k is finite when D is finite and Π does not contain function symbols.

While recursion cannot be expressed in relational algebra, relational algebra can express negation (using the set-difference operator). It has been shown that the LFP of facts and rules of Horn clause programs is the same as the least model for these programs [AV 82]. However, this is not true when negation is added. The Closed World Assumption (CWA) [Rei 78] can be used to deduce a negative fact, $\neg f$, from a logic program by determining if the positive form of the fact f can be derived using some procedure such as naive evaluation. If f cannot be derived, then $\neg f$ is true by CWA. Unfortunately, when negated body predicates are allowed, there is no longer a unique

least model which can serve as the natural meaning for a program. For example, consider the program comprised of the trivial rule *cannotfly(quebec, montreal) :- ¬can_fly(quebec, montreal, 2).* This program has two minimal models, {*cannotfly(quebec, montreal)*} and {*can_fly(quebec, montreal, 2)*}.

There is simple strategy for determining a "canonical" model for programs with negation if they have a property termed stratifiable[3]. The following test can be used to determine when a program is stratifiable. A predicate p *depends* on a predicate q if (i) q is a predicate in the body of some rule with head p, (ii) there is a predicate r in the body of a rule with head p such that r depends on q. The dependency is *positive* (*negative*) if the occurrence of r or q in the above definition (in the appropriate rule body) is positive (negative). A program is *stratifiable* if no predicate negatively depends on itself. Stratifiable programs have a distinguished minimal model which is considered to be their intended model. The strata can be ordered for evaluation so that negated predicates are always completely determined in a lower stratum. Thus, CWA can be used local to each stratum.

Function symbols are needed in order to write many interesting rules for temporal databases. Consider the following example.

Example 2.3 In Example 2.2 *flight* was considered to be an EDB relation. Here, *flight* is an IDB relation defined by the next two rules which capture the notion of a weekly fight schedule.

flight(N, D, A, At) :- first_flight(N, D, A, At).

flight(N, D, A, At+7) :- flight(N, D, At), weekly_flight(N).

[3]See Apt et al. [ABW 88].

The predicate *first_flight* plays the roll of the EDB predicate played by *flight* in the previous example. The intended meaning of *weekly_flight(N)* is that N is the flight number of a weekly flight. The semantics of the date and time attribute At are still intended to mean the arrival time of the flight, but $At+7$ is intended to mean exactly 7 days from the date and time referred to by the At attribute.

Clearly, the query *can_fly(montreal, quebec, A)* has an infinite number of answers. □

2.3 TEMPLOG

TEMPLOG was developed by Abadi and Manna as a logic programming language based on a clausal subset of first-order temporal logic [AM 87]. Three higher-order operators are allowed, ○ (*next*), ◇ (*eventually*), and □ (*always*), which can be applied to first-order predicates. Time is considered to be discrete and extending infinitely into the future, but not into the past. Constants and functions do not change over time, but the interpretation of predicates can change over time.

TEMPLOG is an extension to logic programming languages such as Prolog [Col 73] to permit temporal logic programs using higher-order operators. An answer to a query in Prolog is the set of bindings to the free variables in the query that make the query true with respect to the logic program. If there are no such bindings, the answer 'no' is returned. In TEMPLOG, the answer to a query is a sequence of sets of bindings for the free variables in the query,

one set for each instant of time. Clearly, care must be taken if finite answers to TEMPLOG queries are required. The following are examples of fragments of TEMPLOG programs.

Example 2.4 Let S represent the entropy of a system. Then the TEMPLOG clause $\Box(\circ S > S)$ expresses the notion that the entropy of the system is increasing.

Example 2.5 Consider the TEMPLOG program fragment $\Box(backup(X) \leftarrow \circ\ maintenance(X))$. If X is a file system, this rule expresses the fact that X is always backed-up just before maintenance is done.

TEMPLOG programs have fewer arguments than temporal programs written in other logic languages because attributes representing time are not necessary. This makes some programs easier to state and to understand. In real-time systems non-termination and time extending infinitely into the future are desirable features. The rule for backups and system maintenance above illustrates this. However, temporal databases model the past as well as the future. Typically, query answers do not extend beyond the present. TEMPLOG predicates naturally model an event, however, it is not clear how they can be used to model an interval. Also, an uninitiated programmer may have difficulty writing programs using the three temporal operators. Finally, since the implementation of TEMPLOG is based on Prolog, it is not truly declarative.

2.4 STATELOG

STATELOG is a deductive database query answering system developed by Chomicki and Imieliński which is capable of determining finite representations for certain infinite query answers. A *situation* is a possibly infinite set of circumstances which includes changing time as a special case. Situations are constructed using function symbols. STATELOG models variations in the state of a database with respect to changing situations. Temporal STATELOG is a variant of STATELOG in which the only allowed function is the successor function.

The use of function symbols is restricted in STATELOG. A functional term is either (*i*) a functional constant such as 0 or a variable, or (*ii*) a function symbol applied to a functional term. A function can have non-functional term arguments but it can have only one functional term argument which must occur in a fixed position in the function. A further restriction is that a STATELOG predicate can have no more than one functional term argument which must be in a fixed argument position.

Let P be a STATELOG program. Since P can contain function symbols, the least fixpoint of P, M_P is infinite in general. In order to answer queries on P which may have an infinite number of answers, a finite representation for M_P is needed. Consider the following program which expresses the fact that Tony and Jan meet with their (common) advisor on alternate days.

```
meets(0, tony).
meets(1, jan).
```

```
next(jan, tony).
next(tony, jan).
meets(T+1, X) :- meets(T, Y), next(Y, X).
```

Clearly the least fixpoint of this program is infinite and the query **meets(T, tony)** has an infinite number of answers.

The essential notion in STATELOG is that allowed programs exhibit a periodicity in the states associated with each functional term. Infinite answers can be determined by finding the least fixpoint of each of a finite number of states and by finding a congruence relation that maps arbitrary functional terms to representative terms for each of these states. These two elements are referred to as the *relational specification* for the program.

In order to determine the relational specification for the example above, the equivalence classes and their representatives are first found to be the even integers, represented by [0], and the odd integers, represented by [1]. The state associated with the functional term 0 is $M_P[0]$, that is all tuples in M_P such that either their functional term is 0 or they do not have functional terms. Thus the state associated with [0] is simply *meets(0, tony)* together with the EDB relations. In the same way, the state associated with [1] is *meets(1, jan)* together with the EDB relations. The congruence relation can be expressed using a rewriting rule as $(1 + (1 + T)) \rightarrow T$, where $+$ is treated as an uninterpreted function symbol.

Consider the query *meets(4, tony)* which requires a yes/no answer. The functional term 4 can be represented as $(1 + (1 + (1 + (1 + (0)))))$ which rewrites to 0 in two steps and therefore, the representative for 4 is 0. We

then verify that *meets(0, tony)* is in the least fixpoint of the state associated with the equivalence class [0] and thus the answer is yes.

Queries which are not yes/no queries are answered by giving the relational specification.

The complexity of determining the least fixpoint for each state is polynomial, the same as for determining the least fixpoint for a datalog program since rules that construct new functional terms are not used within a state. However, the number of states can be exponential in the number of constants in the EDB. Thus the complexity of STATELOG query answering is exponential.

STATELOG provides a decision procedure for a syntactically defined class of datalog programs extended by function symbols. It addresses the need to provide finite representations for infinite query answers and proposes an innovative methodology for determining such a representation. The authors state that, to the best of their knowledge, Temporal STATELOG and STATELOG are the only syntactically defined decidable logic programming languages studied so far. However, because of their high complexity, they are impractical as general deductive database inference mechanisms. Until useful classes of STATELOG programs which are computable in polynomial time are discovered, STATELOG will probably be mainly of theoretical interest.

Chapter 3

Superfiniteness

3.1 Introduction

Deductive database systems that are based on pure Datalog cannot be used directly for temporal deductive databases because Datalog does not have function symbols. Function symbols create new terms from terms that already exist in a database. They allow better data structuring ability and applications involving set or list constructors. They are essential for building interesting temporal rules(*e.g.*, see [CI 88]).

A query to a deductive database is expressed using a set of Horn clause rules, often called a query program. The query is then answered against the least fixpoint model of this program, which intuitively is the set of facts derived by the rules from the base facts in the database. In the presence of function symbols, answers to certain queries can be infinite. Detection of finite queries is fundamental to the design of query systems. Some form of

finiteness analysis is an integral part of such systems as NAIL! [Ull 89], LDL [Chim 89], and SYGRAF [KL 88].

Shmueli [Sh 87] showed that query finiteness is in general undecidable for programs with function symbols. Ramakrishnan *et al.* [RBS 87] developed a framework for finiteness analysis. They approximate programs with function symbols by function-free programs with *infinite* base relations satisfying *finiteness constraints* (FCs). Intuitively, FCs assert that if certain columns in a relation have a finite number of values, so will other column(s). For example, let $s(A, B, C)$ be an infinite relation. Then s satisfies the FC $AB \rightarrow C$ exactly when it associates a finite number of C-values with any given AB-value. Among other things [RBS 87] also showed that finiteness is decidable for monadic programs. Additional background is given in Section 1.2.1.

Kifer *et al.* [KRS 88] proposed a stronger notion of finiteness called *superfiniteness* which refers to a query answer being finite in *all* fixpoint models of the program, as opposed to only in the least model. Intuitively, a model M of a program Π is a fixpoint model if for every fact in M there is a rule in Π that justifies this fact. Unlike the least fixpoint model, IDB relations are not assumed to be empty initially when determining an arbitrary fixpoint model. It turns out that superfiniteness, which is stronger than finiteness, is decidable. They have developed a complete axiom system and a decision procedure for superfiniteness. They also extend their procedure to detect finiteness for certain class of query programs. While their contribution is fundamental and significant, the time complexity of their algorithm for detecting the superfiniteness of query programs is exponential in the size of the

input program and constraints[1].

The methodology used by [RBS 87, KRS 88] is (*i*) to approximate a given logic program with function symbols by a Datalog program with infinite base relations together with FCs acting on them, and (*ii*) to use superfiniteness as a sufficient condition for detecting finiteness. It would thus be desirable to have an efficient algorithm for detecting superfiniteness. The main motivation for this chapter is the development of such an algorithm for linear programs with one IDB predicate[2].

The remainder of this chapter is organized as follows. The basic notions used are given in Section 3.2. A simple proof procedure using rule/goal (R/G) trees for reasoning about superfiniteness is developed in Section 3.3. In addition to shedding some light on superfiniteness analysis, this procedure is useful in many proofs. The notion of *compositionality* of (linear) programs is developed in Section 3.4 and it is shown that this property can be tested in polynomial time. The significance of compositionality is that it characterizes programs (together with FCs) for which superfiniteness analysis can be performed by using the local information at the nodes of a R/G tree. Both Sections 3.5 and 3.6 consider only unary FCs. In Section 3.5, the class of compositional linear programs is considered and an automata-theoretic technique for detecting superfiniteness of predicates defined by such programs is developed. This technique leads to a polynomial time decision procedure. In Section 3.6, this technique is extended for the class of linear single recursive rule programs (also called linear sirups) which need not be compositional.

[1] As was noted in Section 1.2.1, the main concern of that paper was proving decidability and axiomatizability of superfiniteness.

[2] Each recursive rule of a linear program has at most one subgoal recursive with the head.

Again, the complexity of the decision procedure is polynomial. A highlight of this analysis is the development of logic programs for (*i*) the construction of the automaton associated with linear sirups (together with FCs), and (*ii*) determining whether the automaton satisfies a property called "permissiveness" and hence whether the predicate is superfinite. In related work, Sagiv and Vardi [SV 89] make use of a technique based on tree automata for deciding finiteness of monadic programs (*i.e.* programs defining monadic IDB predicates). In comparison, the technique presented here is based on (nondeterministic) word automata which are used to decide superfiniteness. Also, the classes of programs considered here are compositional linear programs and (arbitrary) linear sirups. A summary of this chapter is given in Section 3.7.

To end this section, a motivating example is considered in the form of a puzzle.[3] Consider a hypothetical society with an infinite population, satisfying the following constraints. There are a finite number of *founding fathers* and a finite number of *founding mothers*. Each person can only engender a finite number of children. The society follows certain rules for forming mating partners: (*i*) every founding father mates every founding mother; (*ii*) for every mating couple, either the man is a founding father and the woman is born of a couple who mate, or the woman is a founding mother and the man is born of a couple who mate. The problem is to show that the number of mating couples is finite. The constraints in the puzzle can be formalized as the FCs (formal definitions in Section 3.2) $\phi \rightarrow founding\_father_1$, $\phi \rightarrow founding\_mother_1$, $father_1 \rightarrow father_2$ and $mother_1 \rightarrow mother_2$. The rules

[3]This is an adaptation of an example appearing in [KRS 88].

followed for mating can be expressed using the following Datalog program.

$$
\begin{aligned}
r_1\text{: } & mates(X,Y) \text{ :- } founding\_father(X), founding\_mother(Y).\\
r_2\text{: } & mates(X,Y) \text{ :- } founding\_father(X), father(U,Y),\\
& \qquad mother(V,Y), female(Y), mates(U,V).\\
r_3\text{: } & mates(X,Y) \text{ :- } founding\_mother(Y), father(U,X),\\
& \qquad mother(V,X), male(X), mates(U,V).
\end{aligned}
$$

Since the relations *father* and *mother* are infinite, and since in every mating couple one of the partners can be born of some mating couple (*i.e.* (s)he is a non-founding member of the society), it is not clear whether the number of pairs of mating partners is finite. It can be shown using the technique developed in this chapter that the relation *mates* is superfinite, and hence finite. Notice that because of the interaction between rules r_2, r_3 it is not obvious that the relation *mates* is indeed superfinite. This example typifies the kind of reasoning that must be performed in order to detect (super)finiteness of queries.

3.2 Basic Definitions

The basic notions of Datalog were introduced in Section 2.2. A program is linear if each rule body contains at most one predicate which is mutually recursive with the head predicate. A *sirup* is a program consisting of a single recursive rule. Variables appearing in the head of a rule are *output* variables; variables appearing only in the body are *local* variables. Throughout the chapter, only linear programs with one IDB predicate, p, are considered.

(1) Reflexivity: if $Y \subseteq X$ then $X \rightarrow Y$
(2) Augmentation: if $X \rightarrow Y$ then $XZ \rightarrow YZ$
(3) Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Figure 3.1: Armstrong's Axioms

The following conventions are observed: all heads of rules are assumed to appear as $p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct output variables and n is the arity of p; a and a' are arbitrary IDB/EDB predicates; a_i is the i^{th} argument of a; b, c, d, *etc.*, are EDB predicates; $X_1, \ldots, X_n$ are the output variables; U, V, W, *etc.*, are local variables; Z is either an output variable or a local variable. $\Pi = \{r_1, \ldots, r_m\}$ denotes a Datalog program. The relation for the body of rule r_i, R_i, is a relation having an argument for each distinct variable appearing in r_i. $R_{i,j}$ refers to the j^{th} argument of R_i.

A *finiteness constraint* (FC) [RBS 87] is an integrity constraint of the form $a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_l}$. This FC is satisfied by a (possibly infinite) relation a if and only if a associates a finite number of values for the argument a_{i_l} with a tuple of values for the arguments $a_{i_1}, \ldots, a_{i_k}$. Here k is the *arity* of the FC. If $k = 0$, then a satisfies the FC if it has a finite number of values in the column a_{i_l}. In case $k = 1$, the FC is called *unary*. Naturally, a satisfies a set of FCs if it satisfies every FC in the set. Ramakrishnan *et al.* [RBS 87] have shown that Armstrong's axioms (see Figure 3.1), originally proposed for functional dependencies (FDs) [Ull 89], completely characterize FCs.

The notion of *closure* for a set of arguments S, w.r.t. a given set of FCs C, is identical to the classical one corresponding to FDs. By regarding the

body of a rule r_i as a relation R_i (as in [KRS 88]), closure can be associated with a rule body as well. For example, consider a rule r: $p(X_1, X_2)$:- $a(U, X_1), b(X_1, V), p(V, X_2)$ and the FCs $\{a_1 \rightarrow a_2, b_1 \rightarrow b_2\}$. Then the closure of $\{a_1\}$ applied to the relation a is $\{a_1, a_2\}$ while the closure of $\{a_1\}$ applied to the body of r is $\{a_1, a_2, b_1, b_2, p_1\}$. Both notions of closure are used in this chapter. The particular notion used will be clear from the context.

The following notions were introduced in [KRS 88]. Let a be any relation and $a_1, \ldots, a_j$ any relations of the same arity. Then $a_1, \ldots, a_j$ form a *decomposition* of a, denoted $a = a_1 | \ldots | a_j$ if $a = a_1 \cup \ldots \cup a_j$. A *partial constraint* (PC) is a statement of the form $F_1 | \ldots | F_k$, where each F_l in the statement is a set of FCs on some relation a. A PC *holds* for a decomposition $a = a_1 | \ldots | a_j$ if for every i, $1 \leq i \leq j$, there is a j_i such that a_i satisfies F_{j_i}, where $1 \leq j_i \leq k$. The notation $a : \alpha$ is used to mean that α is a PC for the predicate a. Let $\Pi = \{r_1, \ldots, r_k\}$ be a program together with FCs C for its EDB predicates. Consider a fixpoint model M for Π, satisfying C. Let R_i denote the relation in M for the body of rule r_i, and let $p^{(i)}$ denote its projection onto the arguments corresponding to the head predicate p of r_i. M associates the decomposition $p = p^{(1)} | \ldots | p^{(k)}$ with p. For Π and C, the constraints *associated with* Π, denoted $\mathcal{C}(\Pi)$, were introduced in [KRS 88]. A program Π together with FCs C on EDB predicates *satisfies* a PC $p : \alpha$ for its IDB predicate, provided in every fixpoint model M of Π satisfying C, the decomposition of p w.r.t. M satisfies α. This is written as $\mathcal{C}(\Pi) \models \alpha$.

Let a and a' be any predicates of arity m and l respectively, $m \geq l$, such that $a' = a[\bar{X}]$ where $\bar{X}$ is a vector of l arguments of a. Then a mapping τ, relating the arguments of a to those of a', can be defined as follows:

$\tau(a_i) = a'_{j_i}$ provided that the $j_i{}^{th}$ argument in $\bar{X}$ is a_i, where $1 \leq i \leq m$, $1 \leq j_i \leq l$. In other words, this means that the i^{th} argument of a is projected onto the $j_i{}^{th}$ argument of a'. Let $f = a_{i_1}, \ldots, a_{i_s} \rightarrow a_{i_t}$ be a FC on a. By $\tau(f)$ we mean the projection of f onto a', *vis.*, $\tau(a_{i_1}), \ldots, \tau(a_{i_s}) \rightarrow \tau(a_{i_t})$. Note that if $\{\tau(a_{i_1}), \ldots, \tau(a_{i_s})\} \not\subseteq {}^{4}\bar{X}$, then f does not have a projection onto a'. The mapping τ is extended to sets of FCs and to PCs in the natural manner. Since the order of components of a PC is irrelevant ([KRS 88]), we write $\alpha = \beta$, for PCs α and β, to mean α and β have the same set of components. For PCs $a{:}\alpha$ and $a{:}\beta$, α and β are equivalent, $\alpha \equiv \beta$, provided every decomposition of any relation a satisfies α iff it also satisfies β.

Let Π be a program and C be a set of FCs on the EDB predicates of Π. Then an argument is *superfinite* [KRS 88] if and only if the relation for p has a finite set of values for the argument p_i in every fixpoint model of Π satisfying C. Kifer *et al.* [KRS 88] propose a sound and complete axiom system for reasoning about superfiniteness. The system consists of rules for PCs, rules for projection dependencies (PRDs), rules for inclusion dependencies (INDs), and rules for decomposition dependencies (DDs). The complete axiom system is given in Figure 3.2.

They also propose an exponential time algorithm for detecting superfiniteness of predicate arguments based on their axiom system. The intuition behind the decision procedure is quite simple. Feed in the PCs known to hold for each predicate into each rule body, close it w.r.t. the axioms for PCs, and project the result onto the head; whenever there are a number of

[4] Note that $\bar{X}$ is a vector. Clearly, it has an associated set of arguments. The containment here refers to containment in this set.

PC-Rules: Notation: For sets of FCs F and G, $F \vdash_{fc} G$ is used to mean the FCs in G are derivable by applying the FC-axioms on F.

(i) Let $\alpha = F_1|\ldots|F_n$ and $\beta = G_1|\ldots|G_m$ be any PCs. Then from α and $\{F_i \vdash_{fc} G_{j_i} : i = 1,\ldots,n,\ 1 \leq j_i \leq m\}$ infer β.

(ii) Let α be $F_1|\ldots|F_n$ and β be $G_1|\ldots|G_m$. Let γ denote the PC formed by taking all pairwise unions of the components of α and β, i.e., $\gamma = F_1 \cup G_1|\ldots|F_i \cup G_j|\ldots|F_n \cup G_m$. Then from α and β infer γ.

PRD-Rules: Let α be a PC, a and a' be predicates, let $a' = a[\bar{X}]$ and suppose τ is the mapping associated with this projection. Then

(i) PRD-OUTPUT: from $a:\alpha$ infer $a':\tau(\alpha)$;

(ii) PRD-INPUT: from $a':\tau(\alpha)$ infer $a:\alpha$

IND-Rule: IND-INHERIT: from $a' \subseteq a$ and $a:\alpha$ infer $a':\alpha$.

DD-Rules:

(i) from $a = a_1|\ldots|a_m$ and $a_1:\alpha_1,\ldots,a_m:\alpha_m$ infer $a:\alpha_1|\ldots|\alpha_m$;

(ii) from $a = a_1|\ldots|a_m$ and $a:\alpha$ infer $a_i:\alpha,\ i = 1,\ldots,m$.

Figure 3.2: Superfiniteness Axioms [KRS 88].

rules defining the same predicate, derive the resulting PC for that predicate as the disjunction of the PCs obtained from each of the individual rules. Repeat this process as long as a PC not equivalent to a known PC can be derived for some predicate of the program. Since the total number of non-equivalent PCs is finite, the procedure must terminate. Now, an argument p_i is superfinite if and only if the constraint $\phi \rightarrow p_i$ can be derived by the above procedure.

Next we extend Example 2.2 to illustrate the difference between the least fixpoint model and any fixpoint model.

Example 3.6 Let us update the *flight* relation from Example 2.2 by adding a new fourth tuple to express the fact that there is a flight from Knowlton to Knowlton. We now have the following tuples in the *flight* relation.

<#001, montreal, quebec, 1>
<#002, quebec, chicoutimi, 2>
<#003, chicoutimi, montreal, 3>
<#000, knowlton, knowlton, 0>

Suppose the IDB relation *can_fly* is not empty initially as it was previously and that it now contains the tuple

can_fly(knowlton, montreal, 0).

Note that the least fixpoint model of *can_fly* is not changed by the updates since it is derived by applying the rules to the EDB relation *flight*. However, now there is another model which is also a fixpoint of *can_fly*. It contains all the tuples in the least fixpoint model,

<montreal, quebec, 1>
<quebec, chicoutimi, 2>
<quebec, montreal, 3>
<chicoutimi, montreal, 3>

<montreal, chicoutimi, 2>
<montreal, montreal, 3>

and the following additional tuples resulting from the non-emptiness of the *can_fly* relation:

<knowlton, montreal, 0>
<knowlton, quebec, 1>
<knowlton, chicoutimi, 2>
<knowlton, montreal, 3> □

We illustrate that a query may be finite but not superfinite with the following example.

Example 3.7 Let us use the *flight* relation and the non-empty *can_fly* relation from the previous example. In this example, we use the *daily_flight* relation and rules from Example 1.1 from Section 1.2.1, *i.e.*,

```
can_fly(Dep, Arr, Date)
        :- daily_flight(Dep, Arr, Date).
can_fly(Dep, Arr, Date + 1)
        :- can_fly(Dep, Arr, Date),
           daily_flight(Dep, Arr, Any_Date).
```

Let us assume that the only tuple in the *daily_flight* relation is <knowlton, knowlton, 0>. By considering the query *?- can_fly(knowlton, C, T)*, we can see that its answer contains no tuples in the least fixpoint model and an infinite number of tuples in the non-least fixpoint model. Hence, it is finite but not superfinite. □

For a rule r, a set of arguments S, and FCs C, we define $cl(S, r, C)$ to be the closure of S on the body of r, w.r.t. the FCs in C. When FCs are unary, it makes sense to define the inverse closure, $revcl(S, r, C)$, which is obtained by reversing the FCs in C and applying the closure cl with the reversed FCs. Finally, the set of arguments $\{p_1, \ldots, p_n\}$ of p is denoted by ARGUMENTS(p).

3.3 A Simple Proof Procedure for Superfiniteness

As outlined in Section 3.2, the methodology of [KRS 88] for detecting superfiniteness is to try to prove a corresponding FC from the constraints in $\mathcal{C}(\Pi)$ using the SF-Axioms. In this section, it is first shown that for every deduction of a FC using the SF-Axioms, there is a deduction of this FC conforming to a certain normal form. The significance is twofold: the normal form deduction keeps the proof procedure focused, and it will be used later to develop a simple intuitive proof procedure based on rule/goal (R/G) trees for detecting superfiniteness. Even though the results hold for arbitrary programs, for simplicity, the algorithm presented in Figure 3.3 is for linear programs.

It is straightforward to show the following.

Fact: A FC $p\!:\!f$ is derivable from $\mathcal{C}(\Pi)$ using the SF-Axioms iff Algorithm 3.1 returns "$p\!:\!f$ is true" when the input is the program Π and the set of FCs C. □

Definitions for the notions of Rule/Goal (R/G) trees and witness trees follow.

Definition 3.1 Consider a program Π. A R/G tree T for a goal predicate $p(X_1, \ldots, X_n)$ defined in Π, is any tree satisfying the following conditions:

Algorithm 3.1 NFD (Normal-Form Deduction for FCs)

Input: a program Π, a set of FCs C and a FC $p{:}p_{i_1},\ldots,p_{i_j}{\rightarrow}p_{i_k}$ to be tested;

Output: a decision whether $p{:}p_{i_1},\ldots,p_{i_j}{\rightarrow}p_{i_k}$ is true;

begin

for all rules r_k in Π **do**
project the given FCs on the EDB predicates onto R_k, the relation for the body of r_k;
$p{:}\alpha$:= Φ; /* Φ *is the* PC *with no components.* */

repeat

(1) IND-INHERIT: **for all** predicates a and a' such that $a' \subseteq a$ **do**
inherit the FCs (or PCs) on a onto a' and hence, into the relation for the body of r, R;

(2) **for all** relations R_k for rule bodies **do**
combine all PCs on R_k into a single PC (using the union rule for PCs);
close each component of the PCs w.r.t. FC-Axioms;

(3) PRD-OUTPUT: **for all** relations R_k for rule bodies **do**
Project the PCs on R_k onto $p^{(k)}$;

(4) DD: Let $p^{(1)}{:}\alpha_1,\ldots$, and $p^{(k)}{:}\alpha_k$ be the PCs derived in step (3). Then infer $p{:}\alpha_1|\ldots|\alpha_k$;

(5) $p{:}\alpha$:= $p{:}\alpha \uplus p{:}\alpha_1|\ldots|\alpha_k$, where $\uplus$ denotes the pairwise union of components of the two PCs according to the PC-Rules;

until there is no change (logically) to the PC for p;

output:

for all components F_k in $p{:}\alpha$ **do**
if $F_k \not\vdash p{:}p_{i_1},\ldots,p_{i_j}{\rightarrow}p_{i_k}$
then return "$p{:}p_{i_1},\ldots,p_{i_j}{\rightarrow}p_{i_k}$ is false";
return "$p{:}p_{i_1},\ldots,p_{i_j}{\rightarrow}p_{i_k}$ is true";
end.

Figure 3.3: Normal Form Algorithm

(i) T has the goal node labeled $p(X_1, \ldots, X_n)$ as its root, (ii) if v is a goal node for predicate p, then it is either a leaf or it has k rule children $v_1, \ldots, v_k$, corresponding to each of the rules defining p; the label of v_i is the body of rule r_i after its head is unified with the occurrence of p at v. If u is a rule node labeled with a rule occurrence r, then it has a goal child corresponding to each IDB subgoal occurring in its body. (For the class of programs considered here, this subgoal is always unique and it is labeled by the occurrence of p in the body of r.)

Consider a branch B of a R/G tree T. The *chase* of B w.r.t. a set of arguments $\{p_{i_1}, \ldots, p_{i_k}\}$ is defined as follows: (i) every occurrence of the variables $X_{i_1}, \ldots, X_{i_k}$ in B is chased; (ii) a variable is chased if it occurs in an argument b_m of an EDB predicate b, b satisfies the FC $b_{j_1}, \ldots, b_{j_l} \rightarrow b_m$, and the variables occurring in $b_{j_1}, \ldots, b_{j_l}$ are already chased. The set of variables in B that are chased by applying rules (i) and (ii) finitely many times is called the *chase* of B w.r.t. $\{p_{i_1}, \ldots, p_{i_k}\}$.

Definition 3.2 The branch B **witnesses** a FC $p_{i_1}, \ldots, p_{i_k} \rightarrow p_{i_l}$, denoted B *wit.* $p_{i_1}, \ldots, p_{i_k} \rightarrow p_{i_l}$, if X_{i_l} is in the chase of B w.r.t. $\{p_{i_1}, \ldots, p_{i_k}\}$. B witnesses a set of FCs if it witnesses every FC in the set. Consider a FC $\alpha = F_1 | \ldots | F_m$ and a R/G tree T. The tree T *witnesses* α, denoted T *wit.* α, if for every branch B_i in T, there is a j_i, $1 \leq j_i \leq m$ such that B_i *wit.* F_{j_i}.

The following notation is introduced to refer to subtrees of R/G trees in a concise manner. The symbol T is used to denote any R/G tree for the IDB predicate p. For such a tree, we let T_i denote the tree obtained from

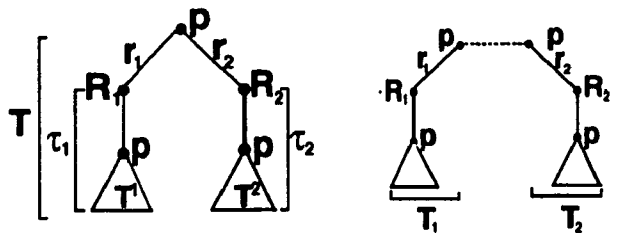


Figure 3.4: Structure of the R/G tree T and the trees T_i, τ_i, and T^i, $i = 1, 2$.

T by deleting the edge corresponding to r_j from the root, and its associated subtree, for all $j \neq i$. Clearly, T_i is a R/G tree for $p^{(i)}$. Note that the root of T_i is p, and p has a unique child u, which is a r_i-descendant, and that u is labeled with the body of r_i (after appropriate unification). The subtree rooted at u is denoted as τ_i. Finally, T^i is the subtree rooted at the first occurence of p below u (again, after appropriate unification). These conventions and notations are illustrated in Figure 3.4, where a two rule program is used for simplicity.

For each rule r_i with head p in Π, a *mapping* m_i is defined from the arguments of p (corresponding to the head) to the arguments of R_i, such that $m_i(p_j) = R_{i,k}$ if X_j appears in $R_{i,k}$. Similarly, the *inverse mapping* m_i^{-1} is the mapping from the arguments of R_i carrying output variables to the arguments of p. We define $m_i^{-1}(R_{i,k}) = p_j$ if $R_{i,k}$ carries X_j. Note that m_i^{-1} is defined for all rules. We extend m_i and m_i^{-1} to FCs and to PCs in the natural way.

A conceptually simple procedure for testing superfiniteness is to detect if there is a R/G tree T for the IDB predicate such that each branch of T witnesses the superfiniteness of the IDB argument. More formally, to decide if a FC $p: p_{i_1}, \ldots, p_{i_j} \rightarrow p_{i_k}$ holds, we simply test if there is a R/G tree witnessing

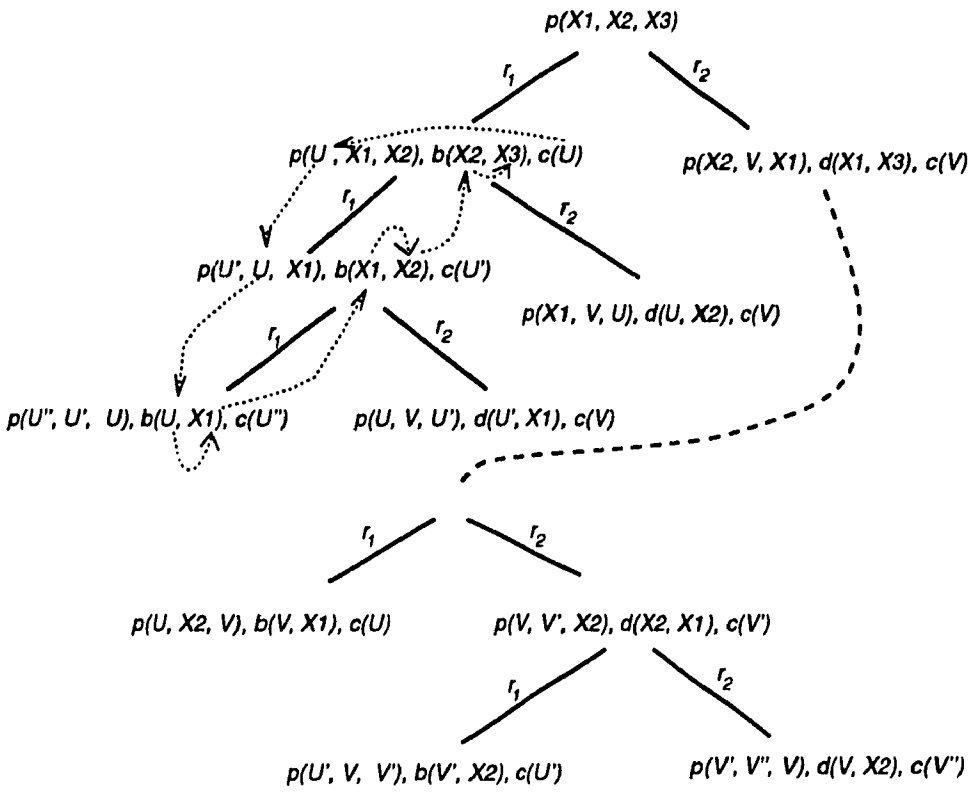


Figure 3.5: Chased R/G tree for the program in Example 3.8.

this FC.

Next, the notion of an R/G tree witnessing an FC is made concrete with an example.

Example 3.8 The first few levels of the R/G tree for the program

r_1: $p(X_1, X_2, X_3)$:- $p(U, X_1, X_2)$, $b(X_2, X_3)$, $c(U)$

r_2: $p(X_1, X_2, X_3)$:- $p(X_2, V, X_1)$, $d(X_1, X_3)$, $c(V)$

with FCs $b_1 \rightarrow b_2$, $d_1 \rightarrow d_2$, and $\phi \rightarrow c_1\}$ is shown in Figure 3.5. The chase of ϕ in the left-most branch of the tree is indicated by arrows and it is shown that $\phi \rightarrow p_3$, *i.e.*, p_3 is superfinite.

The following theorem establishes the equivalence of the two systems for verification of superfiniteness — SF-Axioms and witness trees.

Theorem 3.1 *Let* Π *be a Datalog program and let* α *be a* PC. *Then* $\mathcal{C}(\Pi) \models \alpha$ *iff there is a R/G tree* T *for* Π *(with root p) such that* T *wit.* α.

Remark: Only the PCs on the IDB predicate p are considered here. Also, to simplify the notation in the proof, assume without loss of generality that Π consists of just two (recursive) rules, r_1 and r_2.

Proof:

($\Leftarrow$): We prove sufficiency by induction on the height of T. We say there is a FC-*path* from X_i to X_j in a branch B of a R/G tree if X_j is in the chase of B w.r.t. $\{X_i\}$.

Base Case: Height of T is 1. Let α be a PC on p.

T *wit.* $\alpha \Rightarrow \exists$ a PC $F_1|F_2 \equiv \alpha$ such that

$\forall$ *branch* B *in* T, B *wit.* F_j for some $j \in \{1,2\}$ (by Def. 3.2).

Without loss of generality, suppose B_i *wit.* F_i, $i = 1,2$. Then

$\forall$ FC $p_j \rightarrow p_k \in F_i$,

$\exists$ a FC-path from X_j to X_k in B_i (by Def. 3.2).

Thus, $\forall$ FC $p_j \rightarrow p_k \in F_i$, we have $\mathcal{C}(\Pi) \vdash R_i : m_i(p_j) \rightarrow m_i(p_k)$ (by IND-INHERIT).

$\Rightarrow$ $\mathcal{C}(\Pi) \vdash p^{(i)} : p_j \rightarrow p_k$ (by PRD-OUTPUT)

Thus, since these arguments apply to B_1 and B_2, we have

$\mathcal{C}(\Pi) \vdash p^{(i)} : F_i$, $i = 1,2$, and hence $\mathcal{C}(\Pi) \vdash p : F_1|F_2$ (by DD).

$\Rightarrow$ $\mathcal{C}(\Pi) \vdash \alpha$. (by completeness of SF-Axioms).

Induction: Suppose for all R/G trees T with root p and height $\leq k$, if T *wit.* α then $\mathcal{C}(\Pi) \vdash \alpha$. Let T be a R/G tree of height $k+1$ with root p such that the subtree of the root corresponding to r_i is τ_i of height no more than k, $i = 1, 2$. Recall the notation developed for R/G trees above (see Fig. 3.4).

T *wit.* α $\Rightarrow$ $\exists \alpha_1, \alpha_2$ such that

(i) $\alpha \equiv \alpha_1 \,|\, \alpha_2$ and

(ii) T_i *wit.* α_i, $i = 1, 2$. (by Def. 3.2)

T_i *wit.* $\alpha_i \Rightarrow \tau_i$ *wit.* $m_i(\alpha_i)$ (by Def. of mapping m_i)

$\Rightarrow$ $\exists$ a set of FCs F_i on the body of r_i and a PC α_i' on p such that

(a) T^i *wit.* α_i', and

(b) the FC paths in T^i witnessing α_i' together with the FC-paths corresponding to F_i in the body of r_i will establish the FC-paths necessary for τ_i to witness $m_i(\alpha_i)$, $i = 1, 2$. This implies that $\mathcal{C}(\Pi) \cup \{R_i : F_i, p : \alpha_i'\} \vdash R_i : m_i(\alpha_i)$. (Note that the FC-rules and IND-INHERIT are used for this derivation.)

Since the height of T^i is at most k, by the induction hypothesis we have

$\mathcal{C}(\Pi) \vdash p : \alpha_i'$, $i = 1, 2$

$\Rightarrow$ $\mathcal{C}(\Pi) \vdash R_i : m_i(\alpha_i)$, $i = 1, 2$, (by IND-INHERIT and FC rules)

$\Rightarrow$ $\mathcal{C}(\Pi) \vdash p^{(i)} : \alpha_i$, $i = 1, 2$, (by Def. of mapping m_i and PRD-OUTPUT)

$\Rightarrow$ $\mathcal{C}(\Pi) \vdash \alpha_1 | \alpha_2$, (by DD).

and hence, $\mathcal{C}(\Pi) \vdash \alpha$ (by completeness of SF-Axioms).

(Note that the derivation allows unnecessary components to be included in α_1 and α_2, however the equivalence of $\alpha_1 | \alpha_2$ and α can be deduced using the

completeness of the SF-Axioms.)

($\Rightarrow$): We prove necessity by showing that if $\mathcal{C}(\Pi) \vdash F_1|\ldots|F_m$, then there is a R/G tree T such that every branch B in T witnesses F_i, for some i, $1 \leq i \leq m$. We first prove the following theorem.

Claim: $\mathcal{C}(\Pi) \vdash\ p : F_1|\ldots|F_m$ only if there exists an integer N such that for all strings $r_{j_1}\ldots r_{i_N}$ of length N over $\{r_1, r_2\}$, the following is true: if r_{j_1} is the only rule applied on iteration i of the Normal Form Deduction (Steps (1) to (5)), then a set of FCs F will be derived such that $F \vdash F_l$, for some l, $1 \leq l \leq m$. In this case, we say that the string $r_{j_1}\ldots r_{j_N}$ captures F_l.)

Proof of Claim: We use induction on the number of iterations. We use the notation $S \vdash_{\leq k} F$ to mean that F was derived from S in at most k iterations of Steps (1) to (5) Normal Form Deduction.

Let $\alpha = F_1|\ldots|F_m$ be derived by applying all rules on each iteration.

Base Case: The number of iterations is $k = 1$. The claim follows trivially by considering the strings r_1 and r_2 (of length 1).

Induction: Suppose the claim holds for some N after k iterations. Note that a PC α, derived on the $k+1^{th}$ iteration could have been derived from a PC γ, derived on the k^{th} iteration and a PC β, derived on the iteration $k+1$. (See Step (5) of NFD). Thus $\alpha = \gamma \uplus \beta$. We need to show that every string of some appropriate length captures some component of $\alpha = \gamma \uplus \beta$. By the induction hypothesis, there exists a N such that all strings of length N capture some component of γ. If we can show that there is such an integer N_{k+1} such

that all strings of length N_{k+1} capture some component of β, then it follows from the properties of FCs and the Normal Form Deduction that all strings of length $max(N, N_{k+1})$ do indeed capture some component of $\alpha = \gamma \uplus \beta$.

Suppose β is derived on iteration $k+1$.

$\Rightarrow$ $\exists \beta_1, \beta_2$ such that

(i) $\beta = \beta_1 | \beta_2$, (by DD)

(ii) $\mathcal{C}(\Pi) \vdash_{\leq k} p^{(1)} : \beta_1$ and $\mathcal{C}(\Pi) \vdash_{\leq k} p^{(2)} : \beta_2$, (by NFD)

$\Rightarrow$ $\exists \beta_1', \beta_2'$, such that

(i) $\mathcal{C}(\Pi) \vdash_{\leq k} R_1 : \beta_1'$ and $\mathcal{C}(\Pi) \vdash_{\leq k} R_2 : \beta_2'$, and

(ii) $R_1 : \beta_1' \vdash p^{(1)} : \beta_1$ and $R_2 : \beta_2' \vdash p^{(2)} : \beta_2$, (by PRD-OUTPUT)

$\Rightarrow$ $\exists$ β'' such that

(i) $\mathcal{C}(\Pi) \vdash_{\leq k} p : \beta''$, and

(ii) $p : \beta'' \cup \{< \text{FCs } for\ R_1 >\} \vdash R_1 : \beta_1'$, and

$p : \beta'' \cup \{< \text{FCs } for\ R_2 >\} \vdash R_2 : \beta_2'$. (by IND-INHERIT).

Here, $<$ FCs $for\ R_i >$ refers to the set of FCs derived for the relation for the body R_i from the FCs C given for EDB predicates, via IND-INHERIT. By the induction hypothesis, this implies there is an integer N_k such that each string of length N_k captures some component of β''.

$\Rightarrow$ each string of length N_k captures some component of β_1' and some component of β_2'

$\Rightarrow$ each string of length $N_k + 1$ with prefix r_1 captures some component of β_1 and each string of length $N_k + 1$ with prefix r_2 captures some component of β_2.

$\Rightarrow$ each string of length $N_k + 1$ captures some component of β_1 or some component of β_2.

$\Rightarrow \exists$ an integer $N_{k+1} = N_k + 1$ such that each string of length N_{k+1} captures some component of β. This was to be shown to prove the claim above.

We have shown that if $\mathcal{C}(\Pi) \vdash F_1|\ldots|F_m$ then there is an integer N such that each string of rules r_1 and r_2 of length N captures some component F_i, where $1 \leq i \leq m$. It is straightforward to show that the complete R/G tree of height N witnesses the PC $F_1|\ldots|F_m$. □

The characterization above does not *seem* to suggest an effective procedure for superfiniteness. However, because of the direct correspondence to provability in the axiom system of [KRS 88], it can be shown that there is a bound h on the height of R/G trees, such that if there is a R/G tree witnessing a PC α, then there is a R/G tree of height at most h such that T *wit.* α. Thus, the proof procedure based on R/G trees is effective.

3.4 Compositional Programs

Compositionality for linear programs together with unary FCs on EDB predicates are considered in this section. The notion of *compositional programs* is proposed and it is shown that compositionality can be tested in polynomial time. Compositionality simplifies superfiniteness analysis somewhat. An algorithm for determining the superfiniteness of compositional multi-rule programs is developed in the next section.

Before discussing compositionality in detail, it is necessary to introduce SF-trees. SF-trees are an abstraction of R/G trees formed by replacing the

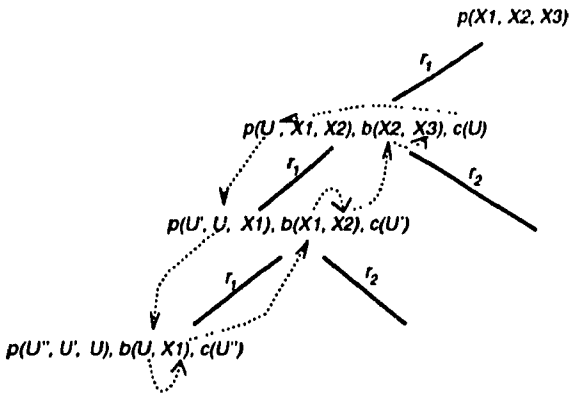


Figure 3.6: Chased R/G tree for the program in Example 3.9.

node labels of the R/G tree by the sets of arguments needed to show a certain FC holds. The idea behind SF-trees is illustrated by the following example.

Example 3.9 Consider the following program introduced in Example 3.8.

r_1: $p(X_1, X_2, X_3)$:- $p(U, X_1, X_2)$, $b(X_2, X_3)$, $c(U)$

r_2: $p(X_1, X_2, X_3)$:- $p(X_2, V, X_1)$, $d(X_1, X_3)$, $c(V)$

FCs: $\{b_1 \rightarrow b_2, d_1 \rightarrow d_2, \phi \rightarrow c_1\}$

It can be verified that $\phi \rightarrow p_3$ holds for this program using the R/G tree Figure 3.6. The arrows in the figure show the path taken when the chase procedure is applied. Note that superfiniteness information is propagated once downward and once upward in each branch: that is, the path, as indicated by the arrows, traverses each level of a branch at most once in each direction in order to chase a variable. Such programs are called *compositional* programs.

A formalism is presented next to make the kind of reasoning performed in Example 3.9, in order to detect compositionality automatically. An operator π_r is associated with a recursive rule r and a set of FCs C. Intuitively, π_r tells

us (i) which arguments of the subgoal occurrence of p[5] (in the context of a R/G tree) will become superfinite given that certain arguments in the goal occurrence are superfinite and (ii) which are the arguments of the subgoal p[6], at least one of which should be proved superfinite if the given argument (or one of a given set of arguments) is to be proved superfinite for the goal p.

Formally, let F and G denote sets of arguments of (the goal occurrence of) p. Associated with each rule r, there is a mapping ava from the head arguments to the body arguments of r. More precisely,

$$ava(\{p_{i_1}, \ldots, p_{i_k}\}) = \bigcup_{j=1}^{k} arg(X_{i_j})$$

Intuitively, ava identifies the set of body arguments where $X_{i_1}, \ldots, X_{i_k}$ appear. Then we define $\pi_r(F, G) = (F', G')$, where $F' = cl(ava(F), r, C) \cap$ ARGUMENTS(p), and (i) $G' = \phi$, if $cl(ava(F), r, C) \cap revcl(ava(G), r, C) \neq \phi$, ($ii$) otherwise, $G' = revcl(ava(G), r, C) \cap$ ARGUMENTS(p), if this set is non-empty, and (iii) $G' = \{\perp\}$, otherwise. Wherever $G' = \{\perp\}$, it signifies the original goal of proving any argument in G superfinite, given the arguments of F are superfinite, will fail. With each occurrence of the (sub)goal p in a R/G tree we can thus associate a pair of sets of arguments (F, G), where (i) F is the set of arguments of the occurrence of p which are known to be superfinite and (ii) G is the set of arguments of this occurrence at least one of which should be proved superfinite (in order to prove the original argument, say p_i, at the root of the tree superfinite). Note that in general, the pair

[5]Here, the subgoal occurrences are those obtained by an expansion using the rule r.

[6]By a goal (subgoal) p we mean the particular occurrence of p.

(F, G) associated with a node in a R/G tree makes use of the information available in the entire context of the associated branch.

A function is associated with each branch of a R/G tree T as follows. Let A denote ARGUMENTS(p). Let B be a branch in T corresponding to the sequence of rule applications $r_{i_1} \cdots r_{i_k}$. Then the function associated with B is $f_B : 2^A \times 2^A \rightarrow 2^A \times 2^A$. The image $f_B(F, G)$ is obtained by actually constructing branch B of the R/G tree and determining (i) which arguments of p (in the leaf occurrence) are superfinite given the arguments F are superfinite (in the root occurrence of p), and (ii) which are the arguments of p in the leaf occurrence at least one of which must be proved superfinite, if one of the arguments in G is to be proved superfinite (for the root occurrence of p).

Example 3.10 Consider the rule r: $p(X_1, X_2, X_3)$:- $p(X_2, V, X_1)$, $b(X_1, X_3)$, $c(X_2)$ with the FCs $C = \{\phi \rightarrow c_1, b_1 \rightarrow b_2\}$, and the branch $r \cdot r$. Then for the pair (ϕ, p_3), $f_{r \cdot r}(\phi, p_3) = (\{p_1, p_3\}, \phi)$. Indeed it can be seen in Figure 3.7 that the output variable X_3 is chased in the R/G tree corresponding to this branch. In this case it is easy to see that for any state (F, G), $f_{r \cdot r}(F, G) = \pi_r(\pi_r(F, G))$.

As another example, consider the rule s: $p(X_1, X_2, X_3, X_4)$:- $p(X_3, X_1, X_2, U)$, $b(X_3, X_4)$, $c(U, X_1)$ with FCs $C = \{b_1 \rightarrow b_2, c_1 \rightarrow c_2\}$ and the branch $s \cdot s \cdot s$. For the pair (p_2, p_4), it can be seen that $f_{s \cdot s \cdot s}(p_2, p_4) = (\{p_1, p_2, p_3, p_4\}, \phi)$. Note $\pi_s(\pi_s(\pi_s(p_2, p_4))) = (\{p_2\}, \{p_1, p_3\})$ which is not equal to $f_{s \cdot s \cdot s}(p_2, p_4)$.

□

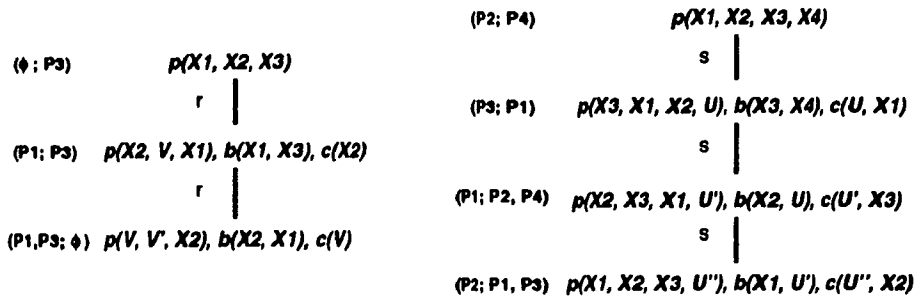


Figure 3.7: R/G trees with the associated F and G sets for the programs in Example 3.10.

The observations made in Example 3.10 are formalized as follows. A *decomposition* of a branch B is any pair of branches B_1, B_2 such that $B = B_1 \cdot B_2$. A branch is *compositional* if for any decomposition of $B = B_1 \cdot B_2$, we have $f_B = f_{B_2} \cdot f_{B_1}$. A proof tree is compositional if every branch of the tree is compositional. A program is *compositional* w.r.t. a set of FCs if every proof tree generated by the program is compositional. *E.g.*, consider the program consisting of the rules $r_1 : p(X_1, X_2, X_3) \text{ :- } p(U, X_1, X_2), b(X_2, X_3), c(U)$ and $r_2 : p(X_1, X_2, X_3) \text{ :- } p(X_2, V, X_1), d(X_1, X_3), c(V)$, together with the FCs $\{b_1 \rightarrow b_2, d_1 \rightarrow d_2, \phi \rightarrow c_1\}$ from Example 3.9. Figure 3.8 shows the F and G sets associated with each node in parentheses with the sets separated by a semi-colon. The set associated with the root is (ϕ, p_3) which corresponds to the FC $\phi \rightarrow p_3$. It can be shown that since every branch is compositional, the program is also compositional.

Intuitively, compositionality implies that superfiniteness analysis can be carried out using information locally available at each node in the R/G tree. This means that even though the pair of sets of arguments associated with a node could in principle make use of the information available in the entire branch, this information can be captured just by using the local information

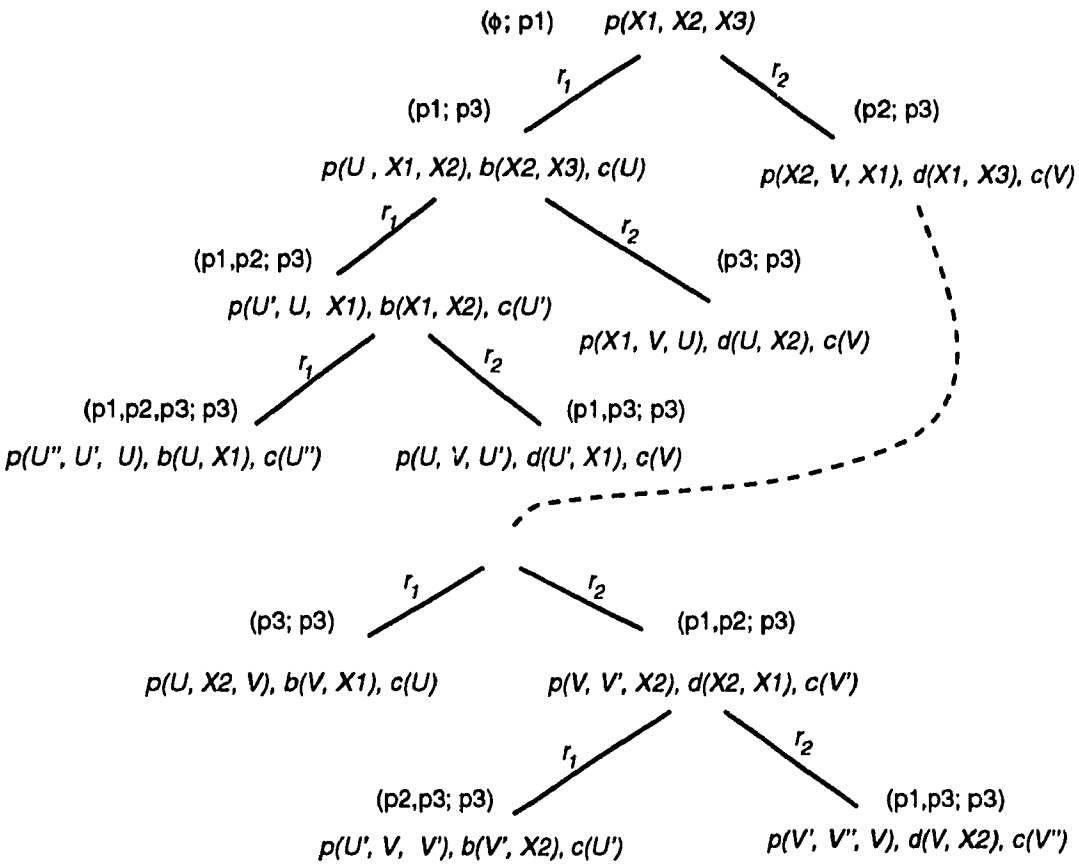


Figure 3.8: R/G tree for the program from Example 3.9 with the associated F and G sets.

at the node in question.

Before algorithms can be given to detect superfiniteness in compositional programs, it is necessary to show that compositionality can be efficiently detected. Since compositionality is a semantic property, a syntactic characterization is needed to obtain an efficient test. The first step is to define AVF graphs[7]. In the following $< a, b >$ indicates a directed node from a to b and (a, b) indicates an undirected node incident on nodes a and b. It is assumed, without loss of generality, that the local variables in the different rules of a program are distinct. The AVF graph G for a program Π has three types of nodes: *variable* nodes corresponding to the local and output variable in Π; *argument* nodes corresponding to the arguments in Π; and FC nodes corresponding to the given FCs. For each variable Z that occurs in an argument a_i in the body of a rule r in Π, there is an *identity* edge (a_i, Z) labeled r in G. For each argument p_i there is a *unification* edge $< p_i, X_i >$. For each FC $a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_l}$, let f be the FC node associated with this FC in G. Then G has *fc* edges $< a_{i_1}, f >, \ldots, < a_{i_k}, f >$, and $< f, a_{i_l} >$ in G. For example, Figure 3.9 shows the AVF graph for the single rule program $p(X_1, X_2, X_3, X_4)$:- $p(X_3, X_1, X_2, U)$, $b(X_3, X_4)$, $c(U, X_1)$ and the FCs $\{b_1 \rightarrow b_2, c_1 \rightarrow c_2\}$. Since there is only one rule, the rule labels have been omitted in the diagram.

Next, an AVF graph traversal algorithm is given for determining if Π is compositional. The AVF graph of the two-rule program $\{r_1, r_2\}$ is used in the following algorithm and theorems, although the results hold for multi-rule programs. In this context, the notion of a path is generalized as follows. A *path* in an AVF graph G is a sequence of nodes $< v_1, \ldots, v_m >$ such that for

[7] AVF graphs are an extension of Naughton's AV graphs [N 89].

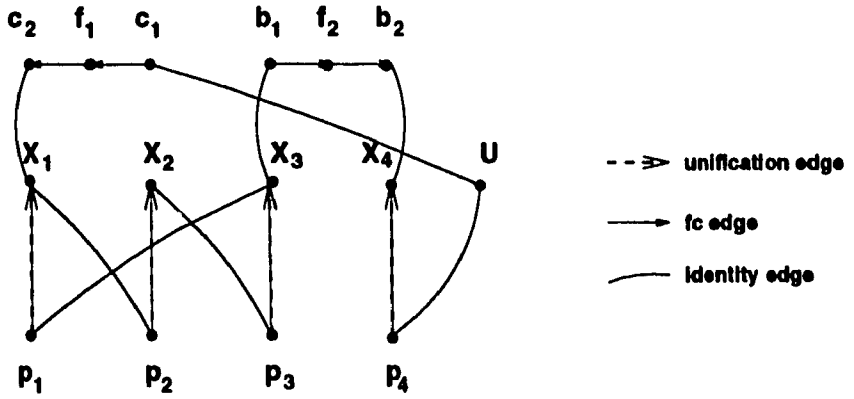


Figure 3.9: AVF graph for $p(X_1, X_2, X_3, X_4)$:- $p(X_3, X_1, X_2, U)$, $b(X_3, X_4)$, $c(U, X_1)$ and the FCs $\{b_1 \rightarrow b_2, c_1 \rightarrow c_2\}$.

i, $1 \leq i < m$, G has an edge incident on v_i and v_{i+1} satisfying the following conditions: (i) the edge is the unification edge $< v_i, v_{i+1} >$ or the unification edge $< v_{i+1}, v_i >$; (ii) or it is an identity edge (v_i, v_{i+1}); (iii) or it is an FC edge $< v_i, v_{i+1} >$. Thus, unification edges (and, essentially, identity edges) can be traversed in either direction, while FC edges must always be traversed in the forward direction. The number of *alternations* associated with a path is the number of times the path alternates the directions it traverses any unification edge(s). A path is said to be *k-alternating* if it has an alternation number k. The following theorem, which is proved in [LN 92a], shows that compositionality of a program can be characterized in terms of the number of alternations in any path connecting two output variables in its AVF graph.

Theorem 3.2 *[LN 92a] Let* Π *be a two-rule program. Then* Π *is compositional iff its AVF graph has no 2-alternating path connecting any output variables.* □

Example 3.11 To illustrate the concept of compositionality, consider a simple program Π, with just one rule r.

$r : p(X_1, X_2, X_3, X_4) :\text{-} p(X_3, X_1, X_2, U), b(X_3, X_4), c(U, X_1).$

The given FCs are $\{b_1 \rightarrow b_2, c_1 \rightarrow c_2\}$. Figure 3.9 shows the AVF graph for this sirup. There is a 2-alternating path $< X_3 - b_1 \rightarrow f_2 \rightarrow b_2 - X_4 \leftarrow p_4 - U - c_1 \rightarrow f_1 \rightarrow c_2 - X_1 - p_2 \rightarrow X_2 - p_3 \rightarrow X_3 - b_1 \rightarrow f_2 \rightarrow b_2 - X_4 \leftarrow p_4 - U - c_1 \rightarrow f_1 \rightarrow c_2 - X_1 >$. The unification edges traversed corresponding to the alternations are $X_4 \leftarrow p_4$, $p_2 \rightarrow X_2$, and $X_4 \leftarrow p_4$ (again). It can be seen that for the branch $r \cdot r \cdot r$, $f_{r \cdot r \cdot r}(\phi, p_4) = (\phi, \{p_1, p_2, p_3, p_4\})$, while $\pi_r(\pi_r(\pi_r(\phi, p_4))) = (\phi, \{p_1, p_3\})$. Thus, Π is not compositional. □

The following result, established in [LN 92a], shows that compositionality can be tested in polynomial time.

Theorem 3.3 *[LN 92a] Given a linear program Π together with unary* FCs *C on its EDB predicates, it can be decided in polynomial time whether Π is compositional in the presence of C.* □

The proof makes use of two-alternating paths and shows that Π is compositional in the presence of C iff the corresponding AVF-graph is free from two-alternating paths. This latter property is shown to be decidable in polynomial time.

It should be remarked here that when there is an alternating path with a higher number of alternations than 2, there necessarily exists a 2-alternating path. Thus, while the root cause of compositionality is an alternating path

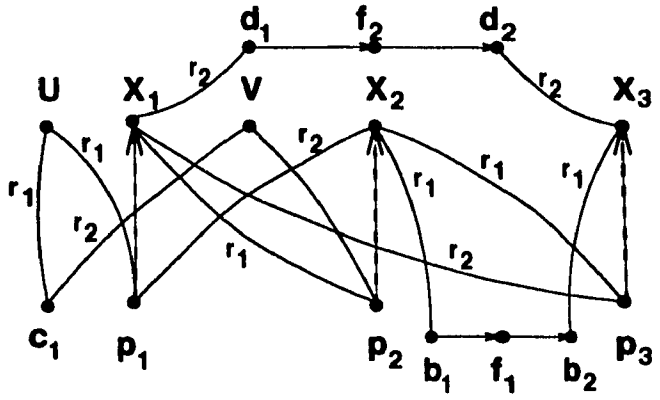


Figure 3.10: AVF graph for the program II in Example 3.12.

with number of alternations > 1, it is sufficient to test for the existence of 2-alternating paths when testing compositionality.

Example 3.12 Consider the two-rule program $\Pi = \{r_1, r_2\}$, where

r_1: $p(X_1, X_2, X_3)$:- $p(U, X_1, X_2), b(X_2, X_3), c(U)$.

r_2: $p(X_1, X_2, X_3)$:- $p(X_2, V, X_1), d(X_1, X_3), c(V)$.

Let the FCs be $C = \{\phi \rightarrow c_1, b_1 \rightarrow b_2, d_1 \rightarrow d_2\}$. The AVF graph for II is given in Figure 3.10. The AVF graph for this program has no 2-alternating paths connecting any distinguished variables and the program is compositional. □

3.5 Multi-Rule Programs

In this section, the class of linear programs with unary FCs satisfying the compositionality property is considered. A technique based on finite state automata for superfiniteness analysis is developed. It is shown that an automaton can be associated with each program (together with FCs) and that this automaton completely captures the superfiniteness characteristics of the

program w.r.t. its FCs. The main result in this section is that superfiniteness of predicates defined by programs in this class can be tested in polynomial time by appealing to a property termed *permissiveness* of finite state automata. Both here and in Section 3.6, attention is restricted to programs containing only recursive rules for the following reasons. Firstly, non-recursive (also called exit) rules can be handled with only minor modifications (see the end of Sections 3.5 and 3.6). Thus ignoring these rules allows attention to be focused on the heart of the problem. Secondly, notice that predicates defined by programs without exit rules are trivially empty and hence finite under the least fixpoint semantics. However, they are not necessarily superfinite. *E.g.*, in $p(X)$:- $p(X)$, the relation p is *not* superfinite.

The property of permissiveness of (nondeterministic) finite state automata (nfa) is formulated as follows. Intuitively, a nfa is permissive if, for every string over its alphabet, either the string is accepted by the nfa or there is a way of completing the string such that the resulting string is accepted. More formally, let M be a nfa with an input alphabet Σ. A string $w \in \Sigma^*$ is called *good* if (i) w is accepted by M, or (ii) w is the prefix of some good string in Σ^*. (iii) w is good iff it so follows from finitely many applications of (i)-(ii). Finally, a nfa M is *permissive* if every string over the alphabet Σ is good w.r.t. M.

Permissiveness of a nfa can be tested in time polynomial in the size of the description of the nfa. This can be accomplished by the following pebbling algorithm. Consider a nfa $M = < S, \Sigma, I, F, \delta >$, where S is the set of states, $\Sigma = \{a_1, \ldots, a_m\}$ is the alphabet, $I \subset S$ is the set of initial states, $F \subset S$ is the set of final states, and $\delta \subseteq S \times \Sigma \times S$ is the transition relation, describing

the behavior of M. Suppose that for each letter $a_i \in \Sigma$, there are $|S|$ pebbles of type i, $1 \leq i \leq m$. Initially, each final state is pebbled with m pebbles, one from each of the m types above. At any stage, a state is pebbled with at most one pebble of each type. The goal is to pebble as many states of the nfa with as many pebbles (of different types) as possible. The criterion used for pebbling is the following: a (non-final) state s is pebbled with a pebble of type i iff there are (not necessarily distinct) states $s_1, \ldots, s_m$ such that (i) $\delta(s, a_i, s_j)$ holds for all $1 \leq j \leq m$, and (ii) s_j is pebbled with a pebble of type j, $1 \leq j \leq m$. The algorithm above can be expressed concisely by a Datalog program Π_1. The program makes use of EDB predicates *final*(X) meaning X is a final state of M, *letter*(L) asserting that L is letter in Σ, and *delta*(X, L, Y) asserting that the transition $\delta(X, L, Y)$, where X, Y are states and L is a letter from the alphabet, holds for M. Finally, *distinct*$(L_1, \ldots, L_m)$ asserts that the letters $L_1, \ldots, L_m$ are distinct. The program recursively defines an IDB predicate *pebbled*(X, L) meaning that state X of the nfa has been pebbled with a pebble corresponding to the letter L. The program assumes that the alphabet has m letters and that the EDB relations are properly initialized. For convenience, a letter of the alphabet and its corresponding pebble type are used interchangeably. For example, in rule r_1, L is both a letter and a pebble of the corresponding type. The program Π_1 follows.

r_1: *pebbled*(X, L) :- *final*(X), *letter*(L).

r_2: *pebbled*(X, L) :- *letter*(L), *delta*$(X, L, Y_1), \ldots,$ *delta*(X, L, Y_m),
pebbled$(Y_1, L_1), \ldots,$ *pebbled*(Y_m, L_m),
letter$(L_1), \ldots,$ *letter*(L_m), *distinct*$(L_1, \ldots, L_m)$.

A state s of the nfa is *L-pebbled* where L is a letter from the alphabet, when s is pebbled with a pebble of type corresponding to the letter L. Suppose that the program Π_1 is run on the input consisting of the EDB relations properly initialized to correspond to the states and transitions of the nfa, and its least fixpoint is computed. A state s is L-pebbled *according* to Π_1 if the least fixpoint model contains the tuple *pebbled*(s, L). We have the following theorem.

Theorem 3.4 *For any nfa* $M = < S, \Sigma, I, F, \delta >$, *with* $\Sigma = \{a_1, \ldots, a_m\}$, M *is permissive if and only if there is an initial state* $s \in I$ *such that* s *is* L*-pebbled for all* $L \in \Sigma$, *according to* Π_1.

Proof: Call a state s of the nfa L-good, for a letter L, iff for each string with a prefix L, either the string will take the nfa to a final state starting from state s, or it can be completed into a string which will do so. We prove that a state is L-good iff it is L-pebbled, by an induction on the number of steps n in the bottom-up evaluation of Π_1 needed to infer *pebbled*(s, L). This will prove the theorem. We begin by proving sufficiency.

Base Case: $n = 1$. Since r_1 is the only applicable rule, s must be $L - good$.

Induction: Assume the result for all states s and and letters L such that *pebbled*$(s, L) \in \Pi_1^k(D)$, where $\Pi_1^k(D)$ is the result of k iterations of bottom-up evaluation of Π_1 applied to D (defined on page 27), which represents the EDB relations corresponding to *delta*, *letter*, *final*, *etc.* Suppose *pebbled*(s, L)

$\in \Pi_1^{k+1} - \Pi_1^k$. Then there exists $s_1, \ldots, s_m$ such that $\delta(s, L, s_1) \wedge \ldots \wedge \delta(s, L, s_m)$ is true. Beginning at state s, consider an input string $\sigma \in \Sigma^*$ such that $\sigma(1) = L$, where $\sigma(i)$ is the i^{th} letter in σ. Let $\sigma(2) = L_i$. Then, by the above, $\delta(s, L, s_i)$ is true and, by rule r_2 and the induction hypothesis, s_i is L_i-pebbled. It follows that s_i is L_i-good and there exists an accepting path for the string obtained as follows:
$< s, s_i, P_i >$ where P_i is the accepting path for the string $\sigma(2) \ldots \sigma(n)$ starting from s_i. Note that since s_i is L_i-good, there must be such an accepting path. Thus, s is $L - good$. Since σ is an arbitrary string, the result holds for all strings σ and it follows that M is permissive.

The proof of necessity is obtained by a similar argument, essentially obtained by reversing the above steps. □

It is well known that the least fixpoint model of a Datalog program can be computed in polynomial time in its input size. Thus it follows trivially that permissiveness of nfa can be decided in polynomial time using the pebbling algorithm. Direct efficient implementations of the pebbling algorithm are possible, however, using a logic program highlights the reasoning behind the characterization. This is best done using a Datalog program for implementing the pebbling strategy. The next goal in this section is to relate superfiniteness of predicates defined by compositional linear programs to the permissiveness of the corresponding nfa.

Definition 3.3 *Nfa associated with a program.* Consider a compositional linear program consisting only of the recursive rules $\Pi = \{r_1, \ldots, r_k\}$, together with a set C of unary FCs for its EDB predicates. Assume that p

is the IDB predicate defined by Π and that it is of arity n. Suppose it is desired to find if the predicate p satisfies a FC $p_i \rightarrow p_j$ in all fixpoint models of Π satisfying the FCs C. Then the nfa associated with the program and the FCs is defined as follows. The nfa is given by $N_\Pi = < S, \Sigma, \{s_0\}, F, \delta >$, where $S = \{(p_l, p_h) \mid 1 \leq l, h \leq n\} \cup \{(\phi, p_l) \mid 1 \leq l \leq n\}$ is the set of states, $\Sigma = \{r_1, \ldots, r_k\}$ is the alphabet, $s_0 = (p_i, p_j)$ is the initial state, $F = \{(p_l, p_l) \mid 1 \leq l \leq n\}$ is the set of final states, and the transition relation δ is defined as follows[8]. We say $\delta(p_u, p_v, r_q, s)$ holds provided at least one of the following holds:

- s is of the form (p_l, p_m), $\pi_{r_q}(p_u, p_v) = (F', G')$, and $p_l \in F'$, $p_m \in G'$, or
- s is of the form (ϕ, p_m), $\pi_{r_q}(p_u, p_v) = (\phi, G')$, and $p_m \in G'$, or
- s is of the form (p_m, p_m) for some m, and $\pi_{r_q}(p_u, p_v) = (F', \phi)$.

For the purpose of states, ϕ is viewed just as a symbol, corresponding to the empty set of arguments. Note that in the special case where $\pi_{r_q}(p_u, p_v) = (F', \{\perp\})$, we essentially leave δ unspecified, and this may be conceptually regarded as leading to a trap state in such cases. □

Additional notation is needed before presenting the next result. Let $\Pi = \{r_1, \ldots, r_k\}$ be a linear program defining a predicate p, and suppose that rule r_i is of the form $p :\!\!- a_{i1}, \ldots, a_{il_i}, p$. Then the program Π^{r_i} is defined to be the program obtained by transforming Π as follows: (i) introduce a new predicate p' of arity n, the same as that of p; (ii) rewrite each rule r_j

[8] The notation $\delta(p_u, p_v, r, p_l, p_m)$ is used to mean $\delta(s_1, r, s_2)$ where $s_1 = (p_u, p_v)$ and $s_2 = (p_l, p_m)$. When convenient, the details and form of the states s_i are suppressed.

in Π into a rule r'_j, by replacing occurrences of the predicate p in the head and body of r_j by p'; and (*iii*) let r_0 be the rule $p :- a_{i1}, \ldots, a_{il_i}, p'$, which is obtained simply by replacing the predicate occurrence p in the body of r_i by p'. Finally, Π^{r_i} is the program $\{r_0, r'_1, \ldots, r'_k\}$. Indeed Π^{r_i} is the program corresponding to the relation $p^{(i)}$ in the decomposition of p, introduced in Section 3.2. Our next result is

Theorem 3.5 *Let $\Pi = \{r_1, \ldots, r_k\}$ be a compositional linear program defining a predicate p, together with a set of unary* FCs *for its EDB predicates. Then p satisfies a* FC $p_i \rightarrow p_j$ *in every fixpoint model of Π satisfying the* FCs *C, i.e. $C(\Pi) \models p_i \rightarrow p_j$, iff the nfa N_Π associated with this program and set of* FCs*, with an initial state (p_i, p_j), is permissive.*

The following technical lemma is needed to prove the theorem.

Lemma 3.6 *Let Π be a linear compositional program defining an IDB predicate p together with unary* FCs *on its EDB predicates. Suppose that in the nfa N_Π associated with Π there is a state $s = (p_i, p_j)$ such that $\delta(s, s_1), \ldots, \delta(s, s_m)$ holds for exactly the states $s_1 = (p_{i_1}, p_{j_1}), \ldots, s_m = (p_{i_k}, p_{j_l})$. Suppose also that T_h is a R/G tree for the program Π^{r_h} and T^h is the subtree of T_h associated with rule r_h, according to the notation introduced in Section 3.3 (see also Figure 3.4). Then T_h witnesses the* FC $p_i \rightarrow p_j$ *if and only if T^h witnesses the* PC $p_{i_1} \rightarrow p_{j_1} | \cdots | p_{i_k} \rightarrow p_{j_l}$.

Proof. Recall the operator π_{r_h} associated with rules. Suppose that $\pi_{r_h}(p_i, p_j) = (F, G)$. By construction of the nfa N_Π, we have that $F = \{p_{i_1}, \ldots, p_{i_k}\}$ and $G = \{p_{j_1}, \ldots, p_{j_l}\}$.

T_h witnesses $p_i \rightarrow p_j$

$\Leftrightarrow \forall$ branch B of T^h, the branch $r_h \cdot B$ witnesses $p_i \rightarrow p_j$ (from Def. 3.2)

$\Leftrightarrow \forall$ branch B of T^h, B witnesses one of the FCs $p_{i_1} \rightarrow p_{j_1}, \ldots, p_{i_k} \rightarrow p_{j_l}$ (by compositionality of Π)

$\Leftrightarrow T^h$ witnesses the PC $p_{i_1} \rightarrow p_{j_1} | \cdots | p_{i_k} \rightarrow p_{j_l}$. □

The proof for Theorem 3.5 can now be given.

Proof of Theorem 3.5. Since the nfa is permissive iff its initial state is successfully pebbled with pebbles for each letter of the alphabet (Theorem 3.4), it is convenient to relate the program to the pebbling algorithm. In this context, we shall show the stronger result that a state (p_i, p_j) of the nfa is r_q-pebbled iff $\mathcal{C}(\Pi^{r_q}) \models p_i \rightarrow p_j$. (Here, r_q is a letter in the nfa's alphabet, and corresponds to a rule in Π.) To show the above, we note that by Theorem 3.1 it suffices to show that the state (p_i, p_j) is r_q-pebbled iff for the program Π^{r_q} there is a R/G tree T_q witnessing the FC $p_i \rightarrow p_j$.

($\Rightarrow$): Suppose that a state (p_i, p_j) is r_q-pebbled. We should show that $\mathcal{C}(\Pi^{r_q}) \models p_i \rightarrow p_j$. We prove this by induction on the number of steps needed to pebble a state, and we induce simultaneously on all rules in the program.

Base Case: The state (p_i, p_j) is pebbled in 0 steps. This is possible iff it is a final state, *i.e.* iff $i = j$. The trivial R/G tree consisting of one node labeled $p(X_1, \ldots, X_n)$ witnesses this trivial FC.

Induction: Assume that for all states (p_i, p_j) and for all rules $r_q \in \Pi$, if the state is r_q-pebbled in m or fewer steps, then there is a R/G tree T_q for Π^{r_q} which witnesses the FC $p_i \rightarrow p_j$. Suppose that the state $s = (p_i, p_j)$ is r_q-pebbled in $m + 1$ steps. This implies there are states $s_1 = (p_{l_1}, p_{h_1}), \ldots, s_k =$

(p_{l_k}, p_{h_k}) such that for each $g = 1, \ldots, k$: (i) $\delta(s, r_g, s_g)$ holds, and (ii) s_g is r_g-pebbled. Consider any $s_g = (p_{l_g}, p_{h_g})$. Clearly, s_g is r_g-pebbled in at most m steps. By inductive hypothesis, we have that there is a R/G tree T_g witnessing the FC $p_{l_g} \rightarrow p_{h_g}$. Consider the R/G tree T' obtained by taking the trees T_g, $g = 1, \ldots, k$, and identifying their roots. Clearly, T' is a R/G tree for the program Π and by construction, it witnesses the PC $p_{l_1} \rightarrow p_{h_1} | \cdots | p_{l_k} \rightarrow p_{h_k}$. Now, construct a R/G tree T_q as follows. (1) The root of T_q is a node u labeled $p(X_1, \ldots, X_n)$ (2) Node u has a unique rule child v corresponding to the rule r_q and labeled by the body of r_q. (3) Node v has a unique goal child which is the root of the tree T' above. It is easy to see that T_q is a R/G tree for the program Π^{r_q}. An application of Lemma 3.6 will now reveal that T will necessarily witness the FC $p_i \rightarrow p_j$. This completes the induction.

($\Leftarrow$): We prove the sufficiency by induction on the height of the R/G trees T_q for the program Π^{r_q} witnessing FCs. For convenience, we define the height of a R/G tree as the number of goal nodes in the longest branch of the tree, minus one. Let s denote the state (p_i, p_j).

Base Case: If the height of T_q is 0, clearly T_q witnesses only trivial FCs. Trivial FCs correspond to the final states of the nfa and they are r-pebbled, for all rules $r \in \Pi$. So consider a tree T_q of height 1. Let T_q witness the FC $p_i \rightarrow p_j$.

Since the height is 1, T_q has a unique branch B_q, and B_q *wit.* $p_i \rightarrow p_j$.

$\Rightarrow cl(ava(p_i), r_q, C) \cap revcl(ava(p_j), r_q, C) \neq \phi$.

$\Rightarrow \exists$ a final state $s_f = (p_l, p_l)$ for some l,

such that $\delta(s, r_q, s_f)$ holds for N_Π.

$\Rightarrow$ s will be r_q-pebbled (see the pebbling algorithm expressed by the Datalog program Π_1 in the beginning of this section).

Induction: Suppose that for all R/G trees T_q for programs Π^{r_q}, of height no more than m, whenever T_q witnesses a FC $p_i \rightarrow p_j$, the state $s = (p_i, p_j)$ is r_q-pebbled. Assume that T_q is a R/G tree of height $m+1$ for Π^{r_q} and that it witnesses the FC $p_i \rightarrow p_j$. Suppose that $\pi_{r_q}(p_i, p_j) = (\{p_{l_1}, \ldots, p_{l_u}\}, \{p_{h_1}, \ldots, p_{h_v}\})$. Then by Lemma 3.6, the subtree T^q of T_q witnesses the PC $p_{l_1} \rightarrow p_{h_1} | \cdots | p_{l_u} \rightarrow p_{h_v}$ (recall the notation for various subtrees of R/G trees, developed in Section 3.3).

By Theorem 3.1 this means that every component $p^{(g)}$ of the decomposition $p = p^{(1)} | \cdots | p^{(k)}$, associated with the relation p defined by the program Π^{r_q}, satisfies some component, say $p_{l_g} \rightarrow p_{h_g}$, of the above PC.

$\Rightarrow$ the subtree $(T^q)_g$ of the tree T^q (see Section 3.3 and Fig. 3.4) witnesses the FC $p_{l_g} \rightarrow p_{h_g}$, for $g = 1, \ldots, k$.

By induction hypothesis, this implies $\exists$ states $s_g = (p_{l_g}, p_{h_g})$, such that:

(i) $\delta(s, r_g, s_g)$ holds, and (ii) s_g is r_g-pebbled, for $g = 1, \ldots, k$.

$\Rightarrow$ the state s will be r_q-pebbled, according to the pebbling criterion (also see the Datalog program Π_1 at the beginning of this section). This completes the induction and the proof. □

Discussion: Theorem 3.5 reduces superfiniteness analysis of compositional linear programs in the presence of unary FCs to that of testing the permissiveness of an associated nfa, a property decidable in polynomial time. It is

possible to improve the efficiency of the algorithm for deciding superfiniteness significantly, by limiting the construction of the nfa to those states that are reachable from the initial state, and by having only one final state, say (p_l, p_l), for any l. Finally, the effect of exit rules is to add extra transitions to the nfa. For example, an exit rule r_e: p :- e will cause a transition from a state (p_i, p_j) to a final state iff $cl(ava(p_i), r_e, C) \cap revcl(ava(p_j), r_e, C) \neq \phi$. Otherwise it causes a transition into a trap state. This minor detail can be easily incorporated into our technique above. To conclude this section an example is given demonstrating the technique.

Example 3.13 Consider the program Π consisting of the rules

r_1: $p(X_1, X_2, X_3)$:- $p(U, X_1, X_2)$, $b(X_2, X_3)$, $c(U)$, and

r_2: $p(X_1, X_2, X_3)$:- $p(X_2, V, X_1)$, $d(X_1, X_3)$, $c(V)$.

We have already seen in Section 3.4 that Π is compositional. Suppose we want to test if the argument p_3 is superfinite. The nfa associated with this program is the following. For simplicity, we indicate only those states which are reachable from the initial state, and retain only one final state. The state set is $S = \{s_0 = (\phi, p_3), s_1 = (p_1, p_3), s_2 = (p_2, p_3), s_f = (p_3, p_3)\}$, where s_0 is the initial state and s_f is the final state. The transition relation is $\delta = \{(s_0, r_1, s_1), (s_0, r_2, s_2), (s_1, r_1, s_1), (s_1, r_1, s_2), (s_1, r_2, s_f), (s_2, r_1, s_f), (s_2, r_2, s_1), (s_2, r_2, s_2)\}$. (See Figure 3.11.) It can be verified by running the pebbling program Π_1 on this nfa, that the initial state s_0 will be r_i-pebbled, $i = 1, 2$. This shows that the argument p_3 is superfinite. It may also be verified that neither of p_1, p_2 is superfinite. □

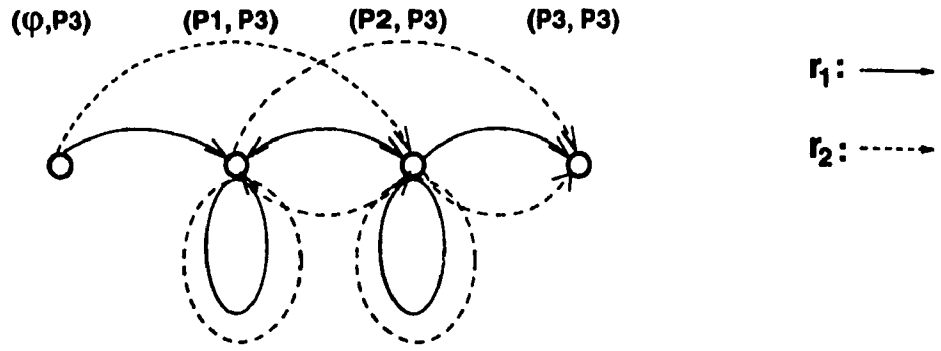


Figure 3.11: NFA used in Example 3.13

3.6 Linear Sirups

Programs consisting of one linear recursive rule, in the presence of unary FCs are considered in this section. The restriction to compositional programs is relaxed for this class of programs. A direct application of the technique of Section 3.5 will lead to incompleteness: arguments of (the subgoals in) the rule bodies have to be incorporated in the analysis based on nfa. A complication arises from the fact that with non-compositional programs, there may be some "communication" between nodes at different levels of a branch (in a R/G tree), and this cannot be captured using the present construction of the nfa. This problem is solved by introducing *basic* and *derived* transitions of the nfa, which will now incorporate the arguments of rule bodies in its states. The details are outlined next.

Construction of the nfa: Let r be a linear sirup with head predicate p and let C be a set of unary FCs on the EDB predicates of r. Then the automaton N_r associated with the sirup is defined as follows. $N_r = < S, \Sigma, s_0, F, \delta >$, where $S = \{(a_i, a'_j) \mid a, a'$ are subgoals of r and a_i, a'_j are any arguments in them$\}$, $\Sigma = \{r\}$ is the unary alphabet, s_0 is the initial state, determined

the FC required to be verified, $F = \{(a_i, a_i) \mid a$ is any subgoal of r and a_i is any argument of $a\}$, and finally, the transition relation $\delta \subseteq S \times S$, defined as follows[9].

The argument a_k is defined to be *fc-reachable* from a_j w.r.t. a rule r and (unary) FCs C if a_k is in $cl(\{a_j\}, r, C)$. The transition relation δ consists of a set of *basic* transitions δ_B and a set of *derived* transitions δ_D, *i.e.* $\delta = \delta_B \cup \delta_D$. In the following, we let $\alpha, \beta, \gamma, \epsilon, \theta, \tau$ denote any subgoals of r. For $s_1 = (\alpha_i, \beta_j)$ and $s_2 = (\gamma_k, \epsilon_l)$, a basic transition $\delta_B(s_1, s_2)$ holds exactly when the following conditions are satisfied:

1. $\exists$ arguments p_{m_i}, p_{m_j} of the IDB predicate p such that p_{m_i} is FC-reachable from α_i, and β_j is FC-reachable from p_{m_j};
2. $\exists$ arguments θ_{m_k}, τ_{m_l} in the body of r carrying the variables X_{m_i} and X_{m_j} respectively;
3. γ_k is FC-reachable from θ_{m_k}, and τ_{m_l} is FC-reachable from ϵ_l.

The derived transitions are defined as follows. A derived transition $\delta_D(s, t)$ holds exactly when the following conditions are satisfied:

1. $t = (\gamma_k, \epsilon_l)$ and $\exists$ a state $s_1 = (\alpha_i, \beta_j)$ such that $\delta(s, s_1)$ holds;
2. there is a transition $\delta(s_1, s_3)$, where $s_3 = (\theta_{n_k}, \tau_{n_l})$, for some subgoals θ, τ and some of their arguments;
3. θ_{n_k} and τ_{n_l} carry some output variables X_{m_k} and X_{m_l};

[9] Since the alphabet is unary, and the nfa here will not have any transitions on the null string, we can suppress the alphabet from the definition of the transition relation.

4. γ_k is FC-reachable from p_{m_k}, and p_{m_l} is FC-reachable from ϵ_l.

Notice that the definition of the derived transitions is recursive. Owing to the intricate details involved in the construction of the transition relation, it will be convenient to express the logic behind this construction in the form of a Datalog program. The following program, Π_2, assumes some of the details related to FC-reachability to be available in the form of EDB relations in its input. For an IDB argument p_i, let θ_k be any argument appearing in the body of r such that θ_k carries the output variable X_i. In this case, we say that θ_k is an *image* of p_i w.r.t. the rule r. The IDB predicate $d(\alpha_i, \beta_j, \theta_k, \tau_l)$ asserts that there is a transition (basic or derived) from state (α_i, β_j) to the state (θ_k, τ_l) (on input r). The details on the EDB predicates used by the program Π_2 are as follows. The predicate $fc(A, B)$ says that argument B is FC-reachable from argument A in the body of r; $image(A, B)$ says that an image of A (which thus denotes some argument of the IDB predicate p) w.r.t. r is B. The program follows.

$$
\begin{aligned}
r_1\colon\ & d(X,Y,Z,W) \text{ :- } fc(X,A), fc(B,Y), image(A,C), \\
& \qquad image(B,D), fc(C,Z), fc(W,D). \\
r_2\colon\ & d(X,Y,Z,W) \text{ :- } d(X,Y,X',Y'), d(X',Y',A,B), image(C,A), \\
& \qquad image(D,B), fc(C,Z), fc(W,D).
\end{aligned}
$$

The main result in this section is a characterization of superfiniteness of predicates defined by linear sirups. Suppose that r is such a sirup. Let N_r denote the nfa associated with r, constructed as described above, where the Datalog program Π_2 is used to construct the transitions. The following theorem is established in [LN 92a].

Theorem 3.7 *[LN 92a] Let r be a linear sirup defining a predicate p, C a set of unary* FCs *on the EDB predicates of r, and N_r the nfa constructed as above. Then r satisfies a* FC *p: $p_i \rightarrow p_j$ if and only if the language accepted by N_r with (p_i, p_j) as its initial state, is nonempty.* □

The pebbling algorithm for detecting permissiveness, developed in Section 3.5, is related to R/G trees witnessing PCs. The steps followed in the proof are similar to those of Theorem 3.5. The result follows on noting that on a unary alphabet, permissiveness of the nfa reduces to non-emptiness of the language accepted.

Discussion: Theorem 3.7 shows that using the automata theoretic technique developed here, it is possible to test the superfiniteness of predicates defined by linear sirups. Since each of the EDB relations used by the program Π_2 can be constructed trivially in polynomial time, it follows that the nfa N_r can be constructed in this time.

Theorem 3.8 *Superfiniteness of predicates defined by linear sirups (in the presence of unary* FCs*) can be decided in polynomial time.*

Proof. Follows from the above discussion and Theorem 3.4. In fact, by reducing non-emptiness of the nfa to graph reachability, this problem can be solved in nondeterministic logspace [Var 89]. □

Discussion: As in Section 3.5, the technique may be made more efficient by avoiding the inclusion of states of the nfa which are not reachable from the initial state, and by limiting the set of final states to just one state (also

see the discussion at the end of Section 3.5). A direct implementation of the techniques developed here may lead to an improved efficiency in testing. However, since the main goal here was to emphasize the logic behind the techniques, the algorithms are presented in the form of Datalog programs. Finally, the effect of exit rules on the nfa is to add extra transitions. As discussed in Section 3.5, the transitions can be either to a final state or to a trap state. Incorporation of exit rules into our overall technique is thus trivial. Example 3.14 illustrates our technique.

Example 3.14 Consider the linear sirup

r: $p(X_1, X_2, X_3, X_4)$:- $p(X_3, X_1, X_2, U)$, $b(X_3, X4)$, $c(U, X_1)$

from Example 3.10. Recall that r is *not* compositional. Suppose we want to test if p satisfies the FC $p_2 \rightarrow p_4$ in every fixpoint model of r which satisfies the FCs $C = \{b_1 \rightarrow b_2, c_1 \rightarrow c_2\}$. The associated nfa is the following. For simplicity, we only show the part of the nfa relevant to the problem on hand. (See Figure 3.12.) The state set of the nfa includes the states $s_0 = (p_2, p_4)$ (the initial state), $s_f = (p_4, p_4)$ (the final state), $s_1 = (p_3, p_1)$, $s_2 = (b_2, p_4)$, $s_3 = (p_2, p_1)$, and $s_4 = (p_3, p_4)$. The basic transitions include $(s_0, s_1), (s_1, s_2), (s_3, s_4), (s_4, s_f)$. It can be seen that the transition (s_0, s_3) is derived. (*E.g.*, using the automaton program Π_2 it can be seen that these basic and derived transitions hold for the nfa.) Now, clearly the language accepted by the nfa is nonempty, as all strings with the prefix $r \cdot r \cdot r$ are accepted by the nfa. It follows that the FC $p_2 \rightarrow p_4$ is indeed satisfied by p.

□

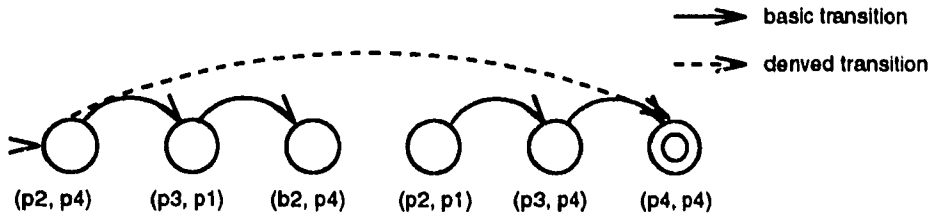


Figure 3.12: NFA used in Example 3.14

3.7 Summary

The problem of proving finiteness of query answers for deductive database languages with function symbols was considered in this chapter. Function symbols are an intrinsic part of temporal deductive databases. While detecting query finiteness is undecidable in general, Ramakrishnan *et al.* [RBS 87] and Kifer *et al.* [KRS 88] proposed a methodology consisting of (*i*) approximating the original program with function symbols by a function-free program while allowing infinite base relations with finiteness constraints acting on them, and (*ii*) testing the stronger property of superfiniteness on the resulting program. Kifer *et al.* have shown that superfiniteness is decidable, and provided a complete axiom system for this purpose. However, their procedure takes an exponential time in general. It would be desirable to have an efficient proof procedure for this problem.

The first question addressed in this chapter was the development of a simple intuitive proof procedure for superfiniteness based on the well understood notion of R/G trees. Then the problem of efficient detection of superfiniteness was addressed by developing an automata-theoretic characterization of superfiniteness for certain classes of programs under unary FCs. From this, polynomial time algorithms for detecting superfiniteness were developed.

Chapter 4

A Temporal Deductive Database Language

The fundamental notions of deductive databases and of relational temporal databases have been discussed in Chapter 2. One of the primary problems involved in combining these two approaches, that of detecting when a query has a finite number of answers, was discussed in Chapter 3. In this chapter, a proposal for a temporal deductive database language, TKL is presented as well as the general DBMS architecture in which TKL is implemented. This architecture is capable of supporting object-oriented and heterogeneous databases as well as TKL.

4.1 Introduction

The temporal deductive database language TKL (*T*emporal *K*nowledge-base *L*anguage) offers a uniform medium for queries and updates in temporal deductive databases. In Section 4.2, the graphic user interface used by TKL is described. The interface improves clarity in two basic ways: (*i*) attribute names are visually associated with predicate arguments in *forms* which are presented on a computer terminal screen for editing by the user, and (*ii*) an area is provided to write *conditions* on statements in a way similar to domain calculus or to tuple calculus. The temporal data type, defined in Section 4.3, includes the "Date" predefined temporal data type which is found in other temporal database proposals and introduces the user-defined temporal data type. Included in its definition is a mechanism for handling temporal null values in stored relations and in queries, as well as a mechanism for modeling imprecise temporal data. TKL has much of the expressive power of a Horn clause language with bottom-up evaluation and stratified negation. A more formal description of TKL is given in Section 4.4. The approach taken to determine the superfiniteness of query programs, as well as other implementation issues, are discussed in Section 4.6. Conclusions for this chapter are presented in Section 4.7.

4.2 Overview of TKL

The basic concepts of deductive databases and Datalog were introduced in Section 2.2. TKL supports recursive queries similar to queries supported

by Datalog and also supports features for the the temporal domain. It has much of the power of Horn clause languages with bottom-up evaluation and stratified negation (see, *e.g.*, [ABW 88]). The language provides the built-in temporal predicates *precedes, contains, overlaps, etc.*, that are usually included in temporal database query languages as well as support for valid intervals and valid events. In this section, TKL is introduced informally by the use of examples. In Section 4.4, its semantics are developed more formally.

In TKL, the *temporal attribute* names `From`, `To`, and `At` are reserved for *temporal* relations. There are two types of temporal relations: (*i*) *interval* relations, which have a closed valid interval in the form of the `From` attribute and the `To` attribute associated with them; and (*ii*) *event* relations, which have associated with them the `At` attribute, indicating the time an event occurs. Relations which are not associated with temporal attributes are called *non-temporal* relations.

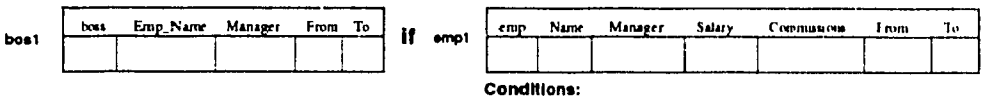
Toward the goal of making the expression of a query suggest its meaning, TKL provides a graphic form-based interface in the spirit of QBE [Zlo 77]. The language is introduced informally by the next two examples.

Example 4.15 A company maintains information about its employees using the relation schema

```
emp(Name, Manager, Salary, Commissions, From, To)
```

to record the name of each employee, the name of the employee's manager, and the employee's earnings from salary and from commissions. The `From` and `To` attribute values give the valid interval of each tuple in this relation.

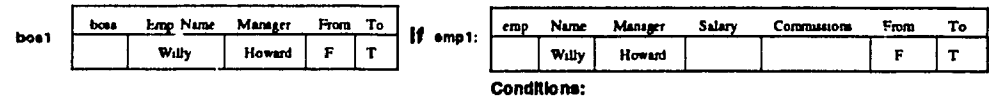
A query program to determine all the bosses of each employee can be written in TKL as follows. A menu allows the selection of rules, queries, or integrity constraints to be written, edited, browsed, or run. We select the action of writing a rule. Then we select the head of the rule from a list of IDB predicates in the database, or define a new IDB predicate interactively, if necessary. In this case, we define a new IDB predicate by supplying the predicate name **boss** and its attribute names and types: *i.e.*, **Emp_Name**(*string*), **Manager**(*string*), **From**(DATE *yy:mm:dd*)[1], and **To**(DATE *yy:mm:dd*). Then the first rule for the query program is written by requesting a **boss** form and a **emp** form using a menu. The system will then display on the screen empty forms similar to the following figure.



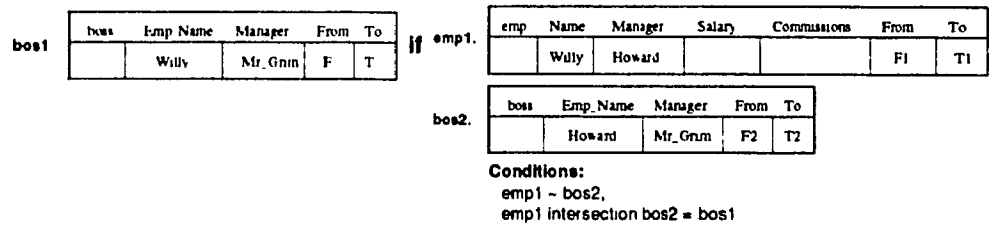
TKL variables are unquoted strings beginning with an uppercase character or an underscore. Constants are (*i*) strings which are either quoted or begin with a lowercase character, (*ii*) numbers, (*iii*) the boolean constants *true* and *false*, or (*iv*) temporal constants (see Section 4.3). The strings **bos1** and **emp1** are called *form identifiers*. Unique form identifiers are automatically generated by the system for every displayed form. They can be used with an attribute name in the **Conditions** section, which appears after the last form, to refer to the corresponding field value. *E.g.*, **emp1.Name** refers to the field below **Name** in the form to the right of **emp1**. When a form identifier is used by itself, it refers to the valid interval or the valid event of the

[1] Note the use of temporal data types for (temporal) attributes. More details on these data types can be found in Section 4.3.

corresponding form. The empty forms are completed as follows to create the base rule for the query program.



This rule is equivalent to the SQL query Select *Name, Manager, From, To* From *emp*. Note that empty attribute value fields are interpreted as 'don't care' variables in TKL. The recursive rule is created by requesting a boss form, an emp form, and another boss form which are then completed as shown below. Notice that an employee's valid interval should be concurrent with the valid interval of each of his/her bosses.



□

The rule for the boss relation cannot be expressed in a conventional relational query language because it is recursive. Notice that the first condition uses the form identifiers emp1 and bos2 and the built-in TKL predicate $\sim$ to specify that the corresponding valid intervals are concurrent, that is, they have some time instant in common. The second condition is used to extract the interval they have in common. (See Section 4.3.3 for the precise definition of the built-in temporal predicates.)

In the following example, we illustrate how a query is written using TKL.

Note that a P. in an attribute value field of a form indicates that the value of that field is part of the query answer. If P. is entered in the first value field of a form, that is, the field under the form name, all attribute values of the form are part of the query answer.

Example 4.16 *Query*: Using the relations in Example 4.15, give the details of each employee whose earnings are more than 10% greater than his/her boss at some time since 1987, where an employee's earnings are his/her salary plus commissions.

Two emp forms and one boss form are needed to write this query. After displaying them on the screen, they can be completed as follows.

emp1:

emp	Name	Manager	Salary	Commissions	From	To
	Willy	Howard	Sw	Cw		

emp2:

emp	Name	Manager	Salary	Commissions	From	To
	Howard		Sh	Ch		

bos1:

boss	Emp_Name	Manager	From	To
P.	Willy	Howard		T

Conditions:

(Sw + Cw) > (Sh + Ch) * 1.1;
emp1 - emp2;
emp1 intersection emp2 = bos1;
T >= 87:01:01;

□

4.3 The Temporal Data Type

The temporal data type is used to represent the time of an event or to indicate the end points of a temporal interval. Temporal information may be

stored using other data types such as string or integer, but it would not take advantage of the interpretations, optimizations, and integrity checks that have been developed for the temporal data type [GS 91, Sard 90]. Multiple temporal data types are allowed in TKL, unlike other temporal systems which allow only one temporal data type in a database.

The built-in *DATE* temporal data type is useful for many common applications. However, there are some applications that do not naturally fit into the 'date and time of day' format of the DATE data type: *e.g.*, it is difficult and unnatural to capture the idea that something occurs every Monday in the DATE format. Also, the time-frames needed for scientific databases are frequently specialized: *e.g.*, different units for measuring time may be required for particle physics and for astronomy. The user-defined temporal data types is provided for cases like these.

An important feature of the temporal data type is that imprecise temporal data is accommodated implicitly by using temporal units that represent a time interval (*flexible granularity*). A simple method for interpreting these intervals is provided which is similar in spirit to the treatment of null values in relational databases [Bisk 81]. As is usual in temporal database systems, a number of temporal comparison predicates for common temporal relationships are provided in order to simplify queries and they are extended to handle temporal imprecision.

The temporal data type may have one or more subfields. The built-in DATE data type, in the format yy:mm:dd:hh:MM:ss, is predefined with subfields for years, months, and days following the Gregorian calendar and for

hours, minutes and seconds as usual using a 24-hour clock. Similarly, a user-defined temporal data type may be separated into subfields by one or more formatting characters such as the colon. The following conventions will be used in this section. Let U be a temporal data type with n subfields separated by colons, denoted $< \textit{subfield } n >:< \textit{subfield } n-1 >: \ldots :< \textit{subfield } 1 >$; we say that G is a *valid temporal subtype* of U with m subfields, $m \leq n$, numbered from n down to $n-m+1$, if G has m subfields and it is a prefix of U. Notice that each type G implicitly induces a set of constants of this type. They are referred to as temporal constants of type G.

A *temporal term* of type G is defined as follows: (i) all temporal constants of type G are temporal terms of type G; (ii) all variables representing temporal constants of type G are temporal terms of type G; (iii) all strings of the form $sf_n : \ldots : sf_{n-m+1}$, where each sf_i, $n-m+1 \leq i \leq n$ is either a constant which is allowed in the i^{th} field of G or a variable symbol, is a temporal term of type G. For example, the DATE subtype term *92:M:D* may appear in a query about which month and day, denoted by the variables M and D, in 1992 something happened.

Two important properties of a temporal data (sub)type are its *tick* size and *granularity*. The selection of U determines the tick size, which is the shortest time period that can be represented by constants of type U (*i.e.*, a tick is the interval represented by one unit of the rightmost subfield of U). It is the atomic time unit for the data type. For example, the tick determined by the DATE data type is a second, but it may be more suitable to have a tick size of a minute for a business application; a tick of 100,000 years may be suitable for a database of geological samples; a tick of 10^{-12}

seconds may be suitable for a database of elementary particles for a physics application. We define the *granularity* of a subtype G to be the number of ticks in its rightmost subfield. Let p be a temporal relation having temporal attribute(s) of type G. We define the *granularity of* p, denoted *gran*(p), to be the granularity of the type associated with the temporal attribute(s) of p. Although multiple temporal data types are allowed, only one subtype (of one of them) may be used in p.[2]

The granularity of a temporal relation reflects the imprecision of the valid interval or event of the data modeled by the relation. This imprecision may be a matter of convenience or due to a lack of information because of theoretical and/or practical limitations on the accuracy of time measurements. Later in this section, a proposal is made for reasoning using imprecise temporal information.

The cardinality of the set of ticks is the largest integer that can be naturally represented on the target machine of the implementation, *viz.*, MAXINT. MAXINT is typically $2^{32} - 1$ for an unsigned integer on a machine with a 32-bit word-size. The product of the tick size of the temporal data type and MAXINT determines the maximum time interval that can be represented using the data type. For example, if the tick size is one day and MAXINT is $2^{32} - 1$, the maximum time interval that can be represented is approximately 11,759,301 years.

The user must supply the *start-up* time for each temporal data type when

[2] This restriction is imposed to simplify the description of this version of TKL. Multiple temporal data types within the same predicate may be incorporated in later versions.

a new database is defined. Temporal constants are mapped to integers internally. The start-up time maps to 0, represented as a signed integer if time is counted both forward and backward from the start-up time. If time is counted forward (backward) only, then the start-up time maps to 0 (MAXINT, resp.) represented as an unsigned integer.

4.3.1 User-Defined Temporal Data Types

User-defined temporal data types, just like the predefined DATE data type, have a fixed format. Associated with all data types, there is an interpretation function, which maps constants of that type to the integers. It is convenient to distinguish between the *base* of a numbering system, such as octal, decimal, hexadecimal, *etc.*, and the base of a temporal subfield which we call the *radix* of the subfield. For example, the radix of the hour subfield in the DATE data type is 24 while its base is decimal.[3]

For user-defined temporal data types, the following specifications must be supplied by the user:

1. the radix and base for subfields $n-1$ to 1 and the base for the n^{th} subfield[4];
2. whether the subfield represents an ordinal integer as it does in the year, month and day subfield of DATE or a cardinal as in the hour, minute, and second subfields of DATE;

[3] Our use of the word 'radix' is, perhaps, non-standard and it may be confusing to some readers Nevertheless, we use it for lack of a better word.

[4] The radix of the n^{th} subfield may be considered MAXINT, MAXINT/2 − 1 or MAXINT/2, depending on the how time is counted, however, incrementing this value may be considered an overflow error.

3. if the subfield represents a subrange of values then each string that may be a member of the subrange must be given in order beginning with the least member (*e.g.*, "January", ..., "December"); in this case, there is an ordinal number associated with each member of the subrange.

Example 4.17 Designers of a database for school schedules might want to tailor the temporal data type S to fit their needs as follows:

subfield 1: $1, \ldots, 8$; /* the number of periods in a school day */
subfield 2: mon, tues, wed, thurs, fri; /* school days */
subfield 3: winter, summer, fall; /* school terms */
subfield 4: base = 10, ordinal = yes; /* any integer representing a year */
start-up time: 92:winter:mon:1:
sense in which time is counted. Forward;

Thus, 92:fall:mon:1 refers to the first period on Monday during the fall term, 1992 and [92:fall:mon:1, 92:fall:fri:8] is the interval that includes all periods in a school week during the fall term of 1992. □

Some definitions are necessary before presenting the temporal interpretation function for user-defined temporal data types. Let t be a constant of type U. We define $SF(t, k)$ to be the value of the k^{th} subfield of t. We define $Ord(U, k)$ to be 1 if the k^{th} subfield of U is defined to have ordinal numbering, otherwise it is defined to be 0. Then the function $t\_to\_i(U, t)$, which maps a constant t of type U to an integer, is described as follows. Let $t_{start-up_U}$ be

the start-up time given for U.

$$Let\ j_o = \sum_{j=1}^{n}[SF(t_{start-up_U}, j) - Ord(U,j)] * rf(j);$$

$$Then\ t\_to\_i(U, t_{start-up_U}) = 0$$

$$t\_to\_i(U,t) = \begin{cases} (\sum_{j=1}^{n}[SF(t,j) - Ord(U,j)] * rf(j)) - j_o & \text{if sign } t \text{ is } + \\ -(\sum_{j=1}^{n}[SF(t,j) - Ord(U,j)] * rf(j)) - j_o & \text{if sign } t \text{ is } - \end{cases}$$

where $rf(1) = 1$,

$rf(j+1) = rf(j) * radix(j+1),\ j = 1, \ldots, n-1$, and

$j_o = t\_to\_i(U, t_{start-up_U})$.

The sign of t will be negative when the n^{th} subfield is less than zero (*e.g.*, Aristotle was born in the year -384 (*i.e.*, 384 B.C.), in the DATE data type.). It is possible to support arithmetic functions on temporal data (sub)types by transforming them to corresponding functions on integers. It is also possible to construct the inverse mappings which can convert integers back to the corresponding temporal constants. The same inverse mapping is used to interpret variables in temporal terms. Temporal terms can be thought of as templates to be matched to integers. We say that field j of temporal term t *matches* an integer i if the following relation holds:

$$SF(t,j) = [((i + j_o) \bmod rf(j+1)) \operatorname{div} rf(j)] + Ord(j), 1 \leq j \leq n,$$

where $rf(j)$ and j_o are as defined above and we define $rf(n+1)$ to be the maximum value allowed in the n^{th} subfield. If $SF(t,j)$ is the variable X_j, then X_j is assigned a value equal to the RHS of the above equation. The use of the temporal interpretation function is illustrated by the following example.

Example 4.18 With reference to Example 4.17, $rf(1) = 1$, $rf(2) = 8$, $rf(3) = 40$, $rf(4) = 120$, and $rf(4+1) =$MAXINT. The start-up time for temporal data type S is 92:winter:mon:1. Thus, the value of j_0 is $t\_to\_i(S, 92{:}winter{:}mon{:}1) = 11040$. It follows that $t\_to\_i(S, 92{:}fall{:}mon{:}1) = 11120 - 11040 = 80$. To determine the value for the variable T such that the temporal term $92{:}T{:}mon{:}1$ will match 80, we evaluate the expression $(((80 + 11040) \bmod 120) \text{ div } 40) + 1$ and get the value 3 for T, the ordinal corresponding to fall in the third subfield of S. □

4.3.2 Interpretation of Imprecision in Intervals and Events

Temporal relations must be assigned a temporal data type and a granularity when they are defined. Temporal relations are allowed to have different granularities, as appropriate for the precision of the available temporal information. Indeed, real-world temporal information is inherently imprecise. However, allowing flexible granularities introduces a complication in interpreting intervals which is illustrated in the following example.

Example 4.19 The relation *hours_worked(Name, From, To)* is used by a company to record the interval employees work each day with granularity of hour. A tuple s in this relation, <*Bob, 92:10:15:8, 92:10:15:17*>, can be interpreted as meaning Bob worked anywhere from 8 hours and 1 second to 9 hours, 59 minutes, and 59 seconds on October 15, 1992. □

To resolve the above problem, it is necessary to make precise what exactly is meant by a valid interval or a valid event in the context of flexible granularities. This is achieved by specifying an *interpretation policy.* Two interpretation policies are provided; they are *narrow*, denoted by $\Box$, and *broad*, denoted by $\Diamond$. Applying the narrow interpretation to tuple s in Example 4.19 gives [92:10:15:8:59:59, 92:10:15:17:00:00], meaning that s holds for all times t in this closed interval. Likewise, the broad interpretation is [92:10:15:8:00:00, 92:10:15:17:59:59] meaning that there is a non-zero probability that s holds for all times t in this closed interval. The interpretation policies *broad* and *narrow* have interesting connections to the modalities *possibly* and *necessarily* arising in modal logic [Chel 80] as well as the concepts of *maybe answers* and *sure answers* studied in the context of relational databases with null values [Bisk 81, Laks 89]. Within the context of the temporal data type, it is convenient to think of *necessity* as a narrow interpretation and *possibility* as a broad interpretation.

The temporal *null* character '/', is introduced to indicate an unknown value when it appears in a temporal term. Referring once again to Example 4.19, the valid interval of tuple s could be represented using nulls as [92:10:15:8:/:/, 92:10:15:17:/:/]. A broad interpretation of an interval will include every tick that could possibly be in the interval. This is obtained by replacing every null in the *From* term with the minimum value for the field in which it appears and by replacing every null in the *To* term by the maximum value for the field in which it appears. Similarly, in a narrow interpretation, every null in the *From* field is replaced by the maximum value for that field and every null value in the *To* field is replaced by the minimum value for

that field. It is now a simple matter to allow null values in stored relations: all that is needed is to extend the definition of temporal terms to allow '/' to be the value of a subfield. Nulls appearing in terms that are arguments of the built-in temporal predicates, which are defined formally below, are interpreted according to the interpretation policy associated with the temporal predicate.

The temporal attributes of interval relations can be represented internally by expanding each of the *To* and *From* attribute values into two integers, one for narrow interpretation and the other for broad interpretation. More formally, let R be a temporal interval relation with temporal data type U and $r \in R$ be a tuple where $r =< \bar{x}, f, t >$. Then the internal representation of r is $r_I =< \bar{x}, f^-, f^+, t^-, t^+ >$, where

$$
\begin{aligned}
f^- &= t\_to\_i(U, f); \\
t^- &= t\_to\_i(U, t); \\
f^+ &= f^- + gran(R) - 1; \\
t^+ &= t^- + gran(R) - 1.
\end{aligned}
$$

Event attribute values are treated much like intervals by setting the single event temporal attribute value to both f and t and then proceeding in the same way as for interval relations.[5] Thus, the internal representation for events is the same as for intervals. This allows a uniform semantics for the built-in temporal predicates with either event or interval operands. More formally, let E be a temporal event relation with temporal data type U and $e \in E$ be a tuple, where $e =< \bar{x}, a >$. Then $e_I =< \bar{x}, a^-, a^+, a^-, a^+ >$

[5]The word *event*, as it is used here, does not refer to a single tick in general, but to a sequence of ticks.

is the internal representation of e, where $a^- = t\_to\_i(U, a)$ and $a^+ = a^- + gran(E) - 1$.

4.3.3 Built-in Temporal Predicates

The built-in temporal predicates are defined in terms of the internal representation of their operands. Let r_1 and r_2 be tuples from any interval or event relation in the database and let the internal representation of their temporal intervals be $I_1 = < f_1^-, f_1^+, t_1^-, t_1^+ >$ and $I_2 = < f_2^-, f_2^+, t_2^-, t_2^+ >$ respectively. In the following, by $\tau \in [f, t]$ we mean τ is a member of the set of ticks corresponding to the interval $[f, t]$.

- (*congruent*) Intuition: $I_1 \equiv I_2$ *iff*
  $(\forall \tau_1 \in [f_1^+, t_1^-]\ \exists \tau_2 \in [f_2^+, t_2^-]\ \tau_1 = \tau_2)\ \wedge\ (\forall \tau_2 \in [f_2^+, t_2^-]\ \exists \tau_1 \in [f_1^+, t_1^-]\ \tau_1 = \tau_2)\ \wedge\ (\forall \tau_1 \in [f_1^-, t_1^+]\ \exists \tau_2 \in [f_2^-, t_2^+]\ \tau_1 = \tau_2)\ \wedge\ (\forall \tau_2 \in [f_2^-, t_2^+]\ \exists \tau_1 \in [f_1^-, t_1^+]\ \tau_1 = \tau_2)$, *i.e.* I_1 is identical to I_2;
  Formal Definition:
  $I_1 \equiv I_2$ *iff* $(f_1^- = f_2^-)\ \wedge (f_1^+ = f_2^+)\ \wedge\ (t_1^- = t_2^-)\ \wedge (t_1^+ = t_2^+)$.

- (*equivalent*$_\square$) Intuition: $I_1 =_\square I_2$ *iff* $(\forall \tau_1 \in [f_1^+, t_1^-]\ \exists \tau_2 \in [f_2^+, t_2^-]\ \tau_1 = \tau_2)\ \wedge\ (\forall \tau_2 \in [f_2^+, t_2^-]\ \exists \tau_1 \in [f_1^+, t_1^-]\ \tau_1 = \tau_2)$, *i.e.* every tick that must necessarily be in I_1 must necessarily also be in I_2, and vice versa;
  Formal Definition: $I_1 =_\square I_2$ *iff* $(f_1^+ = f_2^+) \wedge (t_1^- = t_2^-)$.

- (*equivalent*$_\circ$) Intuition: $I_1 =_\circ I_2$ *iff* $(\forall \tau_1 \in [f_1^+, t_1^-]\ \exists \tau_2 \in [f_2^-, t_2^+]\ \tau_1 = \tau_2)\ \wedge\ (\forall \tau_2 \in [f_2^+, t_1^-]\ \exists \tau_2 \in [f_2^-, t_2^+]\ \tau_1 = \tau_2)$, *i.e.* every tick that must necessarily be in I_1 (I_2) might possibly also be in I_2 (I_1);

Formal Definition:

$I_1 =_\diamond I_2$ *iff* $(f_1^- \leq f_2^+) \wedge (f_2^- \leq f_1^+) \ \wedge \ (t_1^- \leq t_2^+) \wedge (t_2^- \leq t_1^+)$.

- (*precedes*$_\square$) Intuition: $I_1 <_\square I_2$ *iff* I_1 must necessarily end before I_2 might possibly begin;

  Formal Definition: $I_1 <_\square I_2$ *iff* $t_1^+ < f_2^-$.

- (*precedes*$_\diamond$) Intuition: $I_1 <_\diamond I_2$ *iff* I_1 might possibly end before I_2 must begin;

  Formal Definition: $I_1 <_\diamond I_2$ *iff* $t_1^- < f_2^+$.

- (*contains*$_\square$) Intuition: I_1 *in*$_\square$ I_2 *iff* $\forall \tau_1 \in [f_1^-, t_1^+]$ $\tau_1 \in [f_2^+, t_2^-]$, *i.e.*, every tick that might possibly be in I_1 must necessarily be in I_2;

  Formal Definition: I_1 *in*$_\square$ I_2 *iff* $(f_2^+ \leq f_1^-) \wedge (t_1^+ \leq t_2^-)$.

- (*contains*$_\diamond$) Intuition: I_1 *in*$_\diamond$ I_2 *iff* $\forall \tau_1 \in [f_1^+, t_1^-]$ $\tau_1 \in [f_2^-, t_2^+]$, *i.e.*, every tick that must necessarily be in I_1 might possibly be in I_2;

  Formal Definition: I_1 *in*$_\diamond$ I_2 *iff* $(f_2^- \leq f_1^+) \wedge (t_1^- \leq t_2^+)$.

- (*overlaps* $-$ *before*$_\square$) Intuition: $I_1 <|_\square I_2$ *iff* I_1 must necessarily begin before I_2 might possibly begin, I_1 must necessarily end before I_2 might possibly end, and I_1 and I_2 must necessarily have some tick in common;

  Formal Definition: $I_1 <|_\square I_2$ *iff* $(f_1^+ < f_2^-) \wedge (t_1^+ < t_2^-) \wedge (f_2^+ \leq t_1^-)$.

- (*overlaps* $-$ *before*$_\diamond$) Intuition: $I_1 <|_\diamond I_2$ *iff* I_1 might possibly begin before I_2 must necessarily begin, I_1 might possibly end before I_2 must necessarily end, and I_1 and I_2 might possibly have some tick in common;

  Formal Definition: $I_1 <|_\diamond I_2$ *iff* $(f_1^- < f_2^+) \wedge (t_1^- < t_2^+) \wedge (f_2^- \leq t_1^+)$.

- ($consecutive - before_{\Box}$) Intuition: $I_1 <\|_{\Box} I_2$ *iff* $\forall \tau_1 \in [t_1^-, t_1^+]\ \exists \tau_2 \in [f_2^-, f_2^+]\ \tau_1 + 1 = \tau_2$, *i.e.*, one tick after I_1 must necessarily end, I_2 must necessarily begin;
  Formal Definition: $(t_1^- + 1 = f_2^-) \wedge (t_1^+ + 1 = f_2^+)$.

- ($consecutive - before_{\Diamond}$) Intuition: $I_1 <\|_{\Diamond} I_2$ *iff* $(\exists \tau_1 \in [t_1^-, t_1^+]\ \exists \tau_2 \in [f_2^-, f_2^+]\ \tau_1 + 1 = \tau_2$; *i.e.*, one tick after I_1 might possibly end, I_2 might possibly begin; Formal Definition: $I_1 <\|_{\Diamond} I_2$ *iff*
  $(t_1^- + 1 \leq f_2^- \wedge f_2^- \leq t_1^+ + 1) \vee (t_1^- + 1 \leq f_2^+ \wedge f_2^+ \leq t_1^+ + 1) \vee$
  $(f_2^- \leq t_1^- + 1 \wedge t_1^- \leq f_2^+) \vee (f_2^- \leq t_1^+ + 1 \wedge t_1^+ \leq f_2^+)$.

In a similar fashion, we define $>_{\Box}$, $>_{\Diamond}$, $|>_{\Box}$, $|>_{\Diamond}$, *etc.* The following derived predicates are defined next. The symbol $\circ$ is used to mean either $\Box$ or $\Diamond$.

- ($overlap_{\circ}$) $I_1 |_{\circ} I_2$ *iff* either $I_1 <|_{\circ} I_2$ or $I_1 |_{\circ}> I_2$ holds;

- ($concurrent_{\circ}$) $I_1 \sim_{\circ} I_2$ *iff* either $I_1 |_{\circ} I_2$ or $I_1\ in_{\circ}\ I_2$ or $I_2\ in_{\circ}\ I_1$ holds.

Let t_1 and t_2 be temporal terms. The value of $max(t_1, t_2)$ is defined to be the maximum of t_1 and t_2 and the value of $min(t_1, t_2)$ to be the minimum of t_1 and t_2. When I_1 and I_2 are concurrent, the value of $I_1 \cap I_2$ is defined to be the interval $[max(I_1.From, I_2.From), min(I_1.To, I_2.To)]$ and the value of $I_1 \cup I_2$ to be the interval $[min(I_1.From, I_2.From), max(I_1.To, I_2.To)]$. When I_1 and I_2 are not concurrent, the value of $I_1 \cap I_2$ and $I_1 \cup I_2$ is left undefined.

To end this section an example is given using the temporal data type in a Horn clause program. The symbol $*$ is used instead of $\Box$ and $+$ is used instead of $\Diamond$.

Example 4.20 This example shows the differences in philosophy implied by the choice of temporal interpretation policy.

The tick-size of a company's database is a day for the DATE temporal data type.

The following schemas are used to store accounts receivable and accounts payable information.

```
receivable(Client, Amount, At)
payable(Creditor, Amount, At)
```

Both of these schemas have a granularity of month and the value of their At attribute indicates when payment should be received or made. The times when payments are actually received and payments are actually made by the company are stored using the **receipts** schema and the **payments** schema respectively, each of which have a granularity of a day.

```
receipts(Name, Amount, At)
payments(Name, Amount, At)
```

The following rules are used to find clients who are late in making their payments and to find creditors to which the company is late in making its payments at some time T.

```
late_receipt(C,T,T1):-receivable(C,A,T), not receipts(A,C,T1),
                                T1 <* T.
late_paying(C,T,T1) :-payable(C,A,T), not payments(C,A,T1),
                                T1 <+ T.
```

The only essential difference between these queries is in their temporal interpretation policies. As a result of this difference, it appears to be company policy to consider a client to be in arrears if he/she has not paid by the first day of the month payment is due. On the other hand, the company considers itself to be in arrears only if it has not paid a creditor by the last day of the due month. □

4.4 Semantics of TKL

In this section, the semantics of TKL is given in terms of Datalog extended with function symbols and stratified negation. This is followed by a discussion of the integrity constraints (ICs) needed to maintain data consistency in a temporal (deductive) database. Lastly, some aspects of the TKL implementation are discussed.

4.4.1 Semantics of TKL Forms

The meaning of TKL queries, rules and ICs can be given by (*i*) translating them into Datalog extended with stratified negation and function symbols and (*ii*) applying the semantic interpretation for this extended Datalog [Ull 89] to the translated rules. We show how to translate a TKL query into its corresponding Horn clause r and then we illustrate the procedure using the query from Example 4.16. A TKL query is translated into the corresponding Horn clause by performing the following steps. (Rules and ICs are handled in much the same way.)

1. Replace the reserved word NOW with the current system time and date using the appropriate granularity; uniformly replace TKL variables with new variable symbols; replace each empty field with a new variable symbol;

2. Replace form identifiers in the `Conditions` section with

   (i) the corresponding attribute value when they appear in the *dot* field accessor notation,

   (ii) the corresponding valid event or valid interval when they are used with built-in temporal predicates;

3. Map constant temporal terms to the corresponding integers;

4. Create the head predicate of r, $query(X_1, \ldots X_k)$, where $query$ is a reserved predicate symbol, as follows. Let $A_1, \ldots, A_k$ be the distinct attribute value fields which either contain 'P.' or the first field of whose form contains 'P.'. Then we assign to each X_i the value appearing in A_i, $1 \leq i \leq k$. This value can be a variable or a constant.

5. For each form in the query, append a subgoal p to r where p is the predicate symbol corresponding to the form and the arguments of p are taken from the corresponding attribute values of the form;

6. For each arithmetic or temporal expression x *op* y in the `Conditions` section, where x and y are integers, reals, strings, or temporal terms, create a subgoal using a built-in predicate as follows. Let p_{op} be the built-in predicate symbol corresponding to *op*. Then $p_{op}(x, y, z)$ is the

corresponding subgoal where z corresponds to the value of the arithmetic expression. Nested arithmetic expressions are handled by replacing x *op* y by z in the parse tree for the expression and repeating the above procedure. Finally, append the resulting subgoals to the body of r;

7. For each condition x τ y in the Conditions section, where x and y are integers, reals, strings, or temporal terms and τ is <, <=, =, >=, ! =(*not equal*) or a built-in temporal predicate, create a subgoal as follows: let p_τ be the built-in predicate symbol corresponding to τ; append $p_\tau(x, y)$ to r;

8. If any form or condition is negated, negate the corresponding subgoal.

The above procedure is illustrated by applying it to the query from Example 4.16.

Example 4.21 The TKL query from Example 4.16 is transformed into the corresponding Horn clause by first performing variable replacements and insertions, substituting intervals for form identifiers, and mapping temporal terms to constants. For simplicity, we assume *87:01:01* maps to 0. This yields the following equivalent TKL query.

emp1:

emp	Name	Manager	Salary	Commissions	From	To
	V01	V02	V03	V04	T01	T02

emp2:

emp	Name	Manager	Salary	Commissions	From	To
	V02	V05	V06	V07	T03	T04

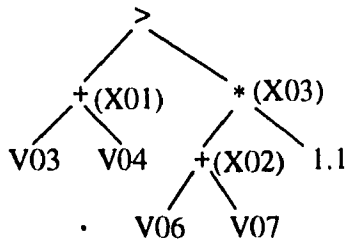
bos1:

boss	Emp_Name	Manager	From	To
P.	V01	V02	T05	T06

Conditions:

(V03 + V04) > (V06 + V07) * 1.1;
[T01, T02] ~ [T03, T04],
[T01, T02] intersection [T03, T04] = [T05, T06],
T06 >= 0,

Since P. appears only in the first value field of the `bos1` form, the only output values are `V01`, `V02`, `T05`, and `T06`. Thus, the head predicate is *query*($V01$, $V02$, $T05$, $T06$). The predicate symbols corresponding to the form names `boss` and `emp` are *boss* and *emp*, respectively. Subgoals corresponding to the forms are created using these predicate symbols, where their arguments are taken from the corresponding field values. The tree corresponding to the first line in the `Conditions` section is shown below.



The built-in predicates corresponding to +, *, and > are *plus*, *times*, and *gt*. The subgoals corresponding to the subexpressions with atomic operands are *plus*($V03$, $V04$, $X01$) and *plus*($V06$, $V07$, $X02$). According to Step 6, the subtrees that these subgoals represent are replaced in the expression tree by $X01$ and $X02$. These replacement values are shown in parentheses in the

diagram above. We continue in this way to the root of the tree. The remaining conditions are handled similarly. The built-in predicates corresponding to $\sim$, `intersection`, and $>=$ are *concurrent*, *intersection*, and *gte*. Finally, the corresponding Horn clause is

$query(V01, V02, T05, T06)$:-
$emp(V01, V02, V03, V04, T01, T02)$, $emp(V02, V05, V06, V07, T03, T04)$,
$boss(V01, V02, T05, T06)$, $plus(V03, V04, X01)$, $plus(V06, V7, X02)$,
$times(X02, 1.1, X03)$, $gt(X01, X03)$, $concurrent(T01, T02, T03, T04)$,
$intersection(T01, T02, T03, T04, T05, T06)$, $gte(T06, 0)$. □

4.4.2 Integrity Constraints

Integrity constraints can be expressed in TKL in much the same manner as rules in query programs. However, the user has to perform the appropriate menu selection. In the context of temporal databases, in addition to the normal ICs arising in any database application, certain ICs are implicitly needed for ensuring consistency. TKL provides for the following types of implicitly needed ICs. The first IC concerns the notion of *keys* appropriate for temporal relations. The *key* for temporal relations is defined with respect to valid intervals or valid events. Let R be a temporal relation and let $\bar{A}$ be a set of non-temporal attributes of R. Then $\bar{A}$ is a key of R provided (i) for every pair of tuples t_i and t_j in R, either $t_i[\bar{A}] \neq t_j[\bar{A}]$ or t_i and t_j have non-concurrent valid intervals, *i.e.*, $t_i \sim t_j$ is false, and (ii) $\bar{A}$ is minimal with respect to this property.

The second IC concerns the notion of *duplicates* appropriate for temporal relations. A temporal relation R contains duplicate tuples if there are two tuples in R whose non-temporal values are identical and whose valid times are concurrent. In TKL, duplicate elimination is performed as follows. Consider the set of tuples $\{t_1, \ldots, t_k\} \subseteq R$ such that they have identical non-temporal values. We say that $t_1, \ldots, t_k$ are *connected* if for every pair of tuples t_i, t_j, $1 \leq i, j \leq k$, (i) either t_i and t_j are concurrent, or (ii) there is a t_l, $1 \leq l \leq k$, such that t_i is concurrent with t_l and t_l and t_j are connected. Clearly, the set $\{t_1, \ldots, t_k\}$ form duplicates exactly when they are connected. To eliminate duplicates, we combine the tuples $t_1, \ldots, t_k$ into one tuple t such that (i) the non-temporal values of t match those of t_i and, (ii) $t.\texttt{From} = \min(\{t_i.\texttt{From} \mid 1 \leq i \leq k\})$, $t.\texttt{To} = \max(\{t_i.\texttt{To} \mid 1 \leq i \leq k\})$.

The third IC comes from the fact that if a tuple t in a relation R is such that $t.\texttt{From} > t.\texttt{To}$ then t is never valid. TKL enforces this IC as well by disallowing such tuples. This is in keeping with the closed world assumption (see [Rei 78]) approach to implicitly represent negative information in base relations as customarily adopted in (non-temporal) databases. In addition, any tuples with this property arising in the intermediate stages of query answer generation are discarded.

4.5 Object-Oriented Architecture

The TKL implementation uses a modular architecture which was conceived both as a formal organizing structure for this project and as a unified framework for incorporating other projects from the SOFTEKS Research Group at

Concordia University. This architecture is proposed to enhance extendability and maintainability for the TKL project as well as to provide a uniform framework into which other SOFTEKS projects, such as object-oriented deductive databases and a heterogeneous federation of databases, can be integrated. The general layout is shown in Figure 4.1.

By a *FM* or *Federation Manager* we mean a central authority regulating the access of a group of users to a group of databases (referred to as file systems). An *Affiliate* is a database querying facility that is part of the federation and a *File System* is a (possibly external) database system which may be as simple as a file server or as complex as a complete DBMS. Each affiliate is required to translate its native language into a *lingua franca* (lf) for communicating with the federation and also to translate query answers from the federation (in lf) into its native language. The lingua franca is also used for communications between the federation and the file systems.

The Affiliates can be individual database language projects which depend on external File Systems for many essential functions. The architecture parallels the dependency of TKL on the LDL deductive database system. The LDL language is the lingua franca of this mini-federation. It is natural to generalize this dependency into a general architecture which may be useful for other SOFTEKS projects. The database engine is factored out of TKL into the FM. Other important aspects of database management, such as logging, crash-recovery, and concurrency control, which are not a part of TKL at this stage of its development, can also be factored out. Thus, the designers of each Affiliate (*e.g.* TKL, object-oriented databases, probabilistic databases, *etc.*) can concentrate on the aspects of database design important to their

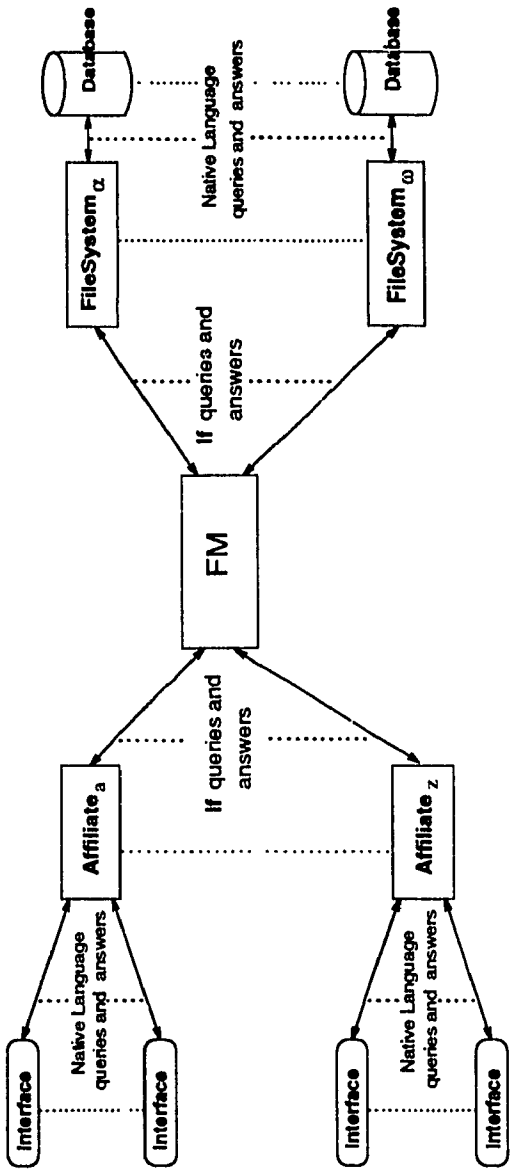


Figure 4.1: Modular Architecture for TKL and other SOFTEKS projects.

project and, at the same time, they can legitimately claim to have a 'real' database system because other elements of the federation provide default implementations for the missing capabilities.

Unfortunately, use of If alone is not sufficient to integrate the file systems into the federation. The translations must take into account differences in how data is represented. The problem is illustrated by the following example taken from [LSS 93].

Example 4.22 [LSS 93] Three universities maintain essentially the same information concerning their staff regarding average salary (*avg_sal*) according to department (*dept*) and skill category (*category*), in the following ways.

University A: relation *pay_info* with attributes *dept*, *category*, and *avg_sal*;

University B: relation *pay_info* with attributes *category*, *dept1*, *dept2*, ... where the domain of *dept1*, *dept2*, ... is the same as that of *avg_sal* for University A;

University C: relation *dept1* with attributes *category* and *avg_sal*,
relation *dept2* with attributes *category* and *avg_sal*,
⋮ ⋮

A strategy for querying these three databases uniformly has been proposed by Lakshmanan, Sadri, and Subramanian ([LSS 93]) using a simple logic called SchemaLog which provides a unified view of different schemas. We propose that the federation maintain a set of schemas which will be provided to the users for composing If queries and that SchemaLog be used by

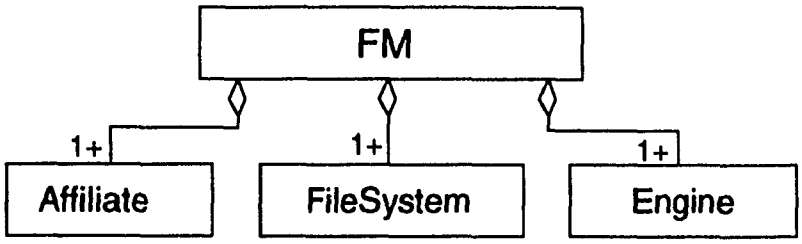


Figure 4.2: Overview of Classes.

the file systems as part of the translation from If into their native languages. The details of schema updates, insertions and deletions are not addressed at this stage of the development of the architecture.

The class structure of the architecture is shown in Figure 4.2 using the Object Model Notation. The principal class is the *FM* class which provides operations for creating, accessing and maintaining affiliates, file systems and engine-class objects (described below). This class controls access to the collection of file systems that comprise the heterogeneous database. It implements the function *query* which has a parameter for a query statement in If and a parameter for the set of file systems to which the query will be posed. In addition the query function has a parameter for the data stream or the file to which the answer is to be written. (Related functions are *cost*, which returns a cost estimate for running a query, and *hits*, which returns the number of tuples in the answer to a query.) This class is also responsible for combining answers from the various file systems into a single answer.

A FM class object poses queries to files systems in If. It is the responsibility of each *FileSystem* object to translate If into its native language and to translate answers back into If. This class provides operations for querying its associated database as well as operations for obtaining the cost of a query

and the number of hits.

Queries are posed to the FM class object by *Affiliate* class objects. The Affiliate class provides the same translation operations as does the FileSystem class. It is also responsible for creating an interface for interacting with agents posing queries. The agent may be as simple as a file which is read by the Affiliate, or as complex as a person using a sophisticated computer interface. The interface itself provides operations for sending tokens to the Affiliate object from which a query can be compiled, as well as operations for returning the answer to the agent.

The *Engine* class is the FM component responsible for processing queries. It develops a processing strategy and queries the file systems for the information it needs for determining query answers. Communication with the file system is via the FM object in order to centralize control of access to the file systems. There may be many engine objects in each FM object. Individual engines may be separate computers, computer networks, or other processing entities. If the FM object maintains priority levels for users, engines can be assigned according to priority. In addition, the FM object may be able to determine from the query which engine is most appropriate and dispatch the query accordingly. (E.g. If a query which is known to be I/O bound, it will not benefit much from a vector processor.)

4.6 The Implementation of TKL

TKL is currently being implemented as an interface to the deductive database system LDL. (See [Chim 89] for the syntax and the features of LDL.) LDL

is used as the underlying inference engine for TKL. It is also used as a 'rapid prototyping' language for implementing aspects of TKL. For example, TKL database schema definitions are stored as data in LDL. This allows type-consistency checking on TKL constants and variables in queries to be implemented easily using rules. TKL schemas are stored in LDL relations of the following form.

```
schema(Pred_Name, Arg_Position, Attr_Name, Attr_Type).
```

The value of the attribute *Pred_Name* is the predicate symbol corresponding to a TKL form; *Arg_Position* is the relative position of an argument in the predicate corresponding to *Pred_Name*; *Attr_Name* and *Attr_Type* are the attribute name and attribute type associated with that argument position.

Information about forms used in statements is stored in the *literal* relation in LDL. There is a tuple in this relation for each attribute field in the corresponding form.

```
literal(Sid, Pred_Name, Pred_Position, Arg, Value, Type).
```

The value of *Sid* is a unique statement identifier for each rule, query, and IC defined by the user; *Pred_Name* is the predicate symbol corresponding to a TKL form; *Pred_Position* is the position of the predicate in the statement, relative to the other subgoals; *Arg* is an argument position, and *Value* and *Type* are its value and type. The type of the field value is determined by lexical analysis of the tokens in the corresponding fields. The allowed types are *integer, real, string, boolean,* DATE, and user-defined temporal data types.

Two rules are used for type checking. The first rule type checks constants by verifying if the lexical type of each constant corresponds to the type

declared for that argument in the schema declaration. Underscores are used to indicate unique 'don't care' variables.

```
constant_type_error(Sid, Pos, Arg, Value, Type, Right_Type)
  :- literal(Sid, P_Name, Pos, Arg, Value, Type),
     not equal(Type, 'variable'),
     schema(P_Name, Arg, _, Right_Type),
     not equal(Type, Right_Type).
```

Variables are type checked for consistency in their usage. We define consistent usage as follows. Let A_i and B_j be arguments of literals used in the same statement such that the same variable X appears in both arguments. Then the usage of X is type consistent if the type of A_i is the same as the type of B_j. The following rule performs this check. The atom `literal(_, _, _, _, _, 'variable')` is true whenever the corresponding TKL field contains a variable.

```
variable_type_error(Sid, Pos, Arg, Value)
  :- literal(Sid, Pred, Pos, Arg, Value, 'variable'),
     schema(Pred, Arg, _, Type),
     literal(Sid, Other_Pred, _, Other_Arg, Value, 'variable'),
     schema(Other_Pred, Other_Arg, _, Other_Type),
     not equal(Type, Other_Type).
```

A type error is reported to the user whenever either of the relations `constant_type_error` or `variable_type_error` is non-empty.

4.6.1 Other Issues

Typically, a DBMS provides for detailed access control of its users. However, this creates a problem in a federated system from a software-engineering point of view. The root of the problem is the absence of agreement among managers of file systems on a uniform treatment of access control. If each file system is allowed to control the access of each agent, the resulting cross-linkage would severely limit the extendibility and maintainability of the system: whenever a new agent is added, every file system to which the agent requires access would need to be informed. If a new file system is added, it would be necessary to determine access privileges for each user, interface, terminal, and agent. For these reasons, we require each file system to negotiate a global level of access privileges with the federation. It is the responsibility of the federation to delegate access privileges to its users, agents, *etc.* This approach allows each file system to maintain its autonomy without hampering the manageablility of the federation.

There is a natural hierarchy of access control within the architecture: each file system determines how much it will trust the federation, the federation determines how much it will trust each user, and each user decides how much it will trust each agent, interface, terminal, *etc.* We note that a probable consequence of the policy of global authorization is that the level of authorization granted by file systems may be at a low level since they cannot control how authorization is delegated within the federation.

Authorizations for queries and operations for communicating with the file systems within the federation are centralized in the Federation class. Users

dispatch queries to the Federation object to which they belong where they are authorized and passed to an Engine class object for processing. The engine itself obtains the data needed by dispatching queries to the Federation object where the authorization process is repeated. Only the Federation class has operations to access the file systems. This assures that User and Engine objects cannot side-step authorization.

The topics of validation, transaction control, and crash recovery have not been included in this version of the architecture. In the current implementation of TKL, the separation of file server and engine is not observed because they are provided as a unit in LDL, the deductive DBMS that is used as a rapid prototyping tool. However, a future version of TKL could use LDL as a server for unprocessed relations and the engine could be implemented as a separate unit. TKL is implemented in C++ using the InterViews class library to build the graphic user interface. (InterViews, revised and renamed Fresco, is scheduled to become part of the X-Windows Release 6 in 1994.)

The principal operations of the User class in TKL are to create an Interface object, read queries written on the interface in the TKL language, translate the queries to the LDL language[6], dispatch queries to LDL and get answers from LDL, and return answers to the interface.

The interface, as it appears on a computer display is shown in Figure 4.3. The principal operations of the Interface class are to create a window showing the activity of LDL, which runs as a background process, to display choosers for the queries, rules, and EDB/IDB relations that have been entered into

[6]The language of LDL is the *lingua franca* of this system.

the system, and to display forms corresponding to agents' selections from the choosers. It also provides scanning operations to get tokens from forms and from the conditions section, as well as for type checking of tokens. The schema and type data needed for these operations are obtained by querying LDL.

Answers from LDL are read into a buffer which is dynamically allocated in order to avoid placing an artificial bound on the size of query answers, as do statically allocated buffers. (In this same spirit, a dynamically allocated buffer is used for queries.) The required functionality is synthesized using C ++ libraries. Answers are optimized for rapid access to attribute values by using pointers. They are also optimum in the sense that values are never copied unnecessarily when they can be referenced using pointers.

LDL is accessed via its public interactive interface: *stdin* and *stdout*, in UNIX parlance. This presents a problem with detecting the end of output from LDL since it does not send a standard signal when it is finished writing an answer. The solution we use is to detect the text patterns it typically writes when its output is finished. There are five such patterns: the prompts ":-) " and ":-(", and, when LDL requires instruction from the agent, the patterns "[y/n/?] ","(yes/no) ", and "(yes/no/all) ". The pattern matching procedure is optimized for long query answers.

InterViews was selected for building the interface after experimenting with three other X-Windows toolkits[7]. The first one we tried was the Xt-Intrinsics toolkit with Athena widgets[8]. This toolkit was rejected because it

[7]Motif was not considered for use because it was not available to us.

[8]A widget is an object that appears on a computer screen, such as a push-button or a scroll-bar, which is used to construct graphic user interfaces.

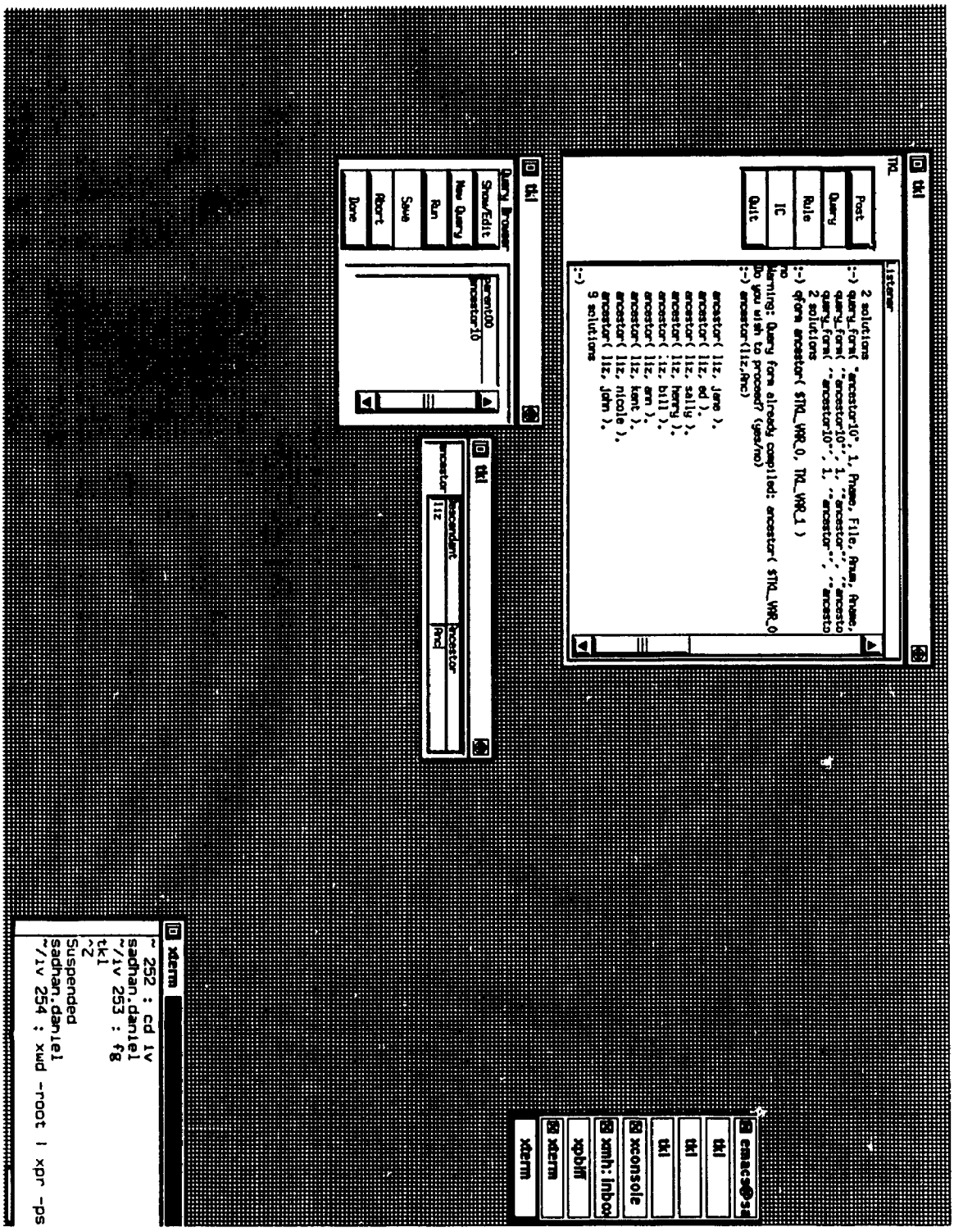


Figure 4.3: The TKL Interface. Shown answering a query.

requires programming at an unnaturally low level in comparison to C++, the general programming language we used, and because Athena widgets are not adequate for building TKL forms.

The next toolkit we tried was Olit using OpenLook widgets. While this toolkit is at the same programming level as Xt, the widgets are powerful enough for TKL forms. However, the administrators of the computer system on which the implementation was being done decided to withdraw support for OpenLook because of its inefficient use of memory. Thus, we decided not to pursue use of this toolkit.

The third toolkit we tried was XView. This toolkit has sufficiently powerful widgets and it 'appears' to allow a higher level of programming than the other toolkits because comparable functions have fewer parameters in XView than they do in Xt or Olit. However, in many cases global variables must be set before calling these functions. This has an undesirable effect on the clarity of code and it goes against object-oriented programming style where almost all data is encapsulated in a class.

InterViews was finally selected because it provides widgets suitable for TKL forms and it has a high-level C++ interface to X-Windows. Further, as a bonus to our system administrators, its widgets are very efficient in their use of memory. Unfortunately, this library has other deficiencies which slowed the progress of the TKL project. The only documentation available is a very terse manual. For example, in the important area of geometry management, only three sentences are written to explain the basic operations (Section 2.1 of [L 92]). A set of examples is provided to demonstrate how the library

should be used. Some InterViews classes do not function as documented: for example, the dialog box function *dismiss* does not work. Finally, it does not provide some widgets which are commonly found in widget sets. For example, it does not provide a text window with a scroll-bar or a string chooser. Both of these widgets had to be constructed for use in TKL. On the other hand, InterViews widgets have an attractive appearance, use memory efficiently, and work quickly and dependably.

4.7 Conclusion

A uniform treatment of temporal information is a natural extension to database technology that can aid in the design and the use of databases maintaining the history of objects they model. We take the temporal extension one step further to temporal deductive databases. The temporal deductive database query language, TKL, offers some important innovations: its graphic user interface is designed to make the meaning of queries as transparent as possible; user-defined temporal data types help adapt the representation of temporal data to fit the user's needs; finally, a simple semantics for temporal imprecision is provided using *broad* and *narrow* interpretation policies. A spin-off of our treatment of flexible granularity is that null values can be stored in temporal terms and used effectively in queries that reason about time.

Chapter 5

TKL and Other Temporal Query Systems

Several query systems were introduced in Chapter 2 for temporal databases. TQuel and HSQL extend the relational database languages Quel and SQL (resp.) to temporal relational database languages. Two temporal deductive systems were also introduced: TEMPLOG and STATELOG. TEMPLOG is based on temporal logic and STATELOG is a deductive query answering system. In this chapter, we compare TKL to these systems.

5.1 TKL & Temporal Relational Query Systems

The temporal relational database systems TQuel and HSQL are extensions to the relational database query languages Quel and SQL respectively. They are based on relational calculus. On the other hand, TKL is based on a

deductive query system similar to Datalog. Thus, TKL features increased expressive power over relational database systems.

5.1.1 TQuel and TKL

TQuel is a functioning temporal database system that provides a simple syntax and semantics for roll-back and historical databases. It differs from TKL somewhat regarding its choice of fundamental temporal predicates and operators. For example, TQuel defines *precedes* using the $\leq$ comparison operator while TKL uses the $<$ operator. Also, in TQuel the intervals I_1 and I_2 are said to *overlap* when (*i*) I_1 (I_2) begins before I_2 (I_1, *resp.*) and both I_1 and I_2 have some time instant in common, or (*ii*) I_1 (I_2) is contained in I_2 (I_1, *resp.*). Thus, an interval I_1 can *precede* an interval I_2 even if they have a point in time in common and they can *overlap* even when there is no overlap in a non-technical sense, *i.e.*, one interval *contains* the other interval.

Since TKL provides for flexible granularities together with the *broad/-narrow* interpretation policies, operators in TKL and TQuel can only be compared concisely on intervals which have a granularity of one tick. In this case, differences in interpretation policies degenerate to produce equivalent models of query programs. It can be seen from the definitions in Section 4.3.3 that the TKL $precedes_{\square}$ predicate is defined using the $<$ comparison operator and that $I_1 <_{\square} I_2$ is true only when I_1 occurs entirely before I_2. Also $overlaps - before_{\square}$ is true for $I_1 <|_{\square} I_2$ only if I_1 begins before I_2 and both intervals have some point in time in common. For one-tick granularity, it is easy to verify that substituting $\diamond$ for $\square$ in the above will give the same

interpretation. Thus, in TKL I_1 *precedes* I_2 is true when I_1 ends strictly before I_2 begins and I_1 *overlaps* I_2 is never true when I_1 *contains* I_2 is true.

Some TQuel temporal primitives such as *overlap* and *extend* return intervals as their value, while others such as *precedes* are predicates which are either true or false. In TKL, the division between operators returning values and predicates is clear: all the temporal primitives are predicates.

5.1.2 HSQL and TKL

The design of TKL has been influenced by the the design of HSQL with respect to integrity constraints and its treatment of flexible granularity. The notion of a unique key in the context of temporal intervals and the notion of coalescing concurrent tuples with the same visible attributes from HSQL are integrated into the design of TKL. The methodology for comparing intervals with different granularities also originated there, although we have extended it with broad and narrow interpretation policies.

5.1.3 Summary: Temporal Relational Systems

Both TQuel and HSQL provide the system support needed to model valid events and valid intervals. They provide useful temporal operators, a sophisticated temporal data type, and the integrity constraints needed to maintain data consistency. TKL includes these features also, and, in addition, it features a graphic user-interface, user-defined temporal data types, interpretation policies for handling flexible granularity correctly, and allows null-values in temporal data types.

In both TQuel and HSQL, queries are written using a plain text editor. However, temporal queries are often more complex than similar non-temporal queries because the underlying relations have additional temporal attributes and because the temporal conditions are often intricate and subtle. For these reasons, TKL incorporates a graphic user-interface as part of its definition as an aid to writing queries clearly and concisely. TKL forms visibly associate attribute names and their values. Logical joins between forms can be expressed concisely by entering the same variable symbol in the value fields to be joined. When a plain text editor is more convenient than forms, the TKL conditions-section can be used. Form identifiers are provided for use in this section to represent the temporal intervals of the corresponding forms. This is useful in expressions involving the temporal operators.

The DATE data type which is provided by TQuel and HSQL is unsuitable for specialized temporal databases such as databases of geological data or for particle physics experimental data. Also, it cannot capture periodicity of data naturally. For these reasons, user-defined temporal data types are provided in TKL. Both TQuel and HSQL allow flexible granularities, but they rely on default interpretations of the imprecision that flexible granularities introduce. TKL provides broad and narrow interpretation policies to give the database user control over how much imprecise data is interpreted. In addition, null values can be stored as TKL temporal data values and used effectively in answering queries - something that cannot be done using TQuel or HSQL.

5.2 TKL & Temporal Deductive Query Systems

The two temporal deductive query systems described in Sections 2.3 and 2.4, TEMPLOG and STATELOG, are experimental query systems based on logic programming. TEMPLOG uses a higher-order syntax to express the notions of *next, eventually,* and *always* and it can easily model processes that extend infinitely into the future. It handles infinite answers by giving them one at a time. However, users of temporal databases may find it difficult to express queries using its primitives when intervals are the natural choice for representing temporal validity.

The great strength of STATELOG is its ability to finitely represent certain query answers that are periodic in nature. However, it will probably be of mainly theoretical interest until tractable classes programs are discovered. Some of its power for periodic query answers in captured in TKL by the user-defined temporal data type, as illustrated in Example 4.17, which can be tailored by the user to capture the periodic nature of some query answers.

TKL is designed for historical data and, by using superfiniteness analysis, it provides a practical solution to the problem of detecting finiteness for certain classes of query programs. Since TEMPLOG handles infinite query answers by returning them one at a time, it is restricted to modeling real-time systems when possibly infinite queries are involved. TKL cannot model such real-time systems in its present form, although an extension for this purpose may be possible. In any case TKL is not limited to such applications.

Neither TEMPLOG nor STATELOG were conceived to be general purpose temporal deductive database query answering systems. In their current form,

they promise interesting capabilities. However, they do not provide sophisticated temporal data types (user-defined or otherwise), the usual set of temporal operators, or the integrity constraints needed for a practical temporal deductive database system.

5.3 Summary

Neither HSQL nor TQuel provide facilities for conveniently handling temporal data that cannot be concisely represented using the DATE data type, such as the user-defined temporal data type found in TKL. Also they do not provide support for imprecise temporal data or for temporal null values which is provided in TKL.

The user-interface, which is an integral part of TKL, associates attribute names and values graphically. Relationships between intervals, between intervals and events, and between events can be long and complex in temporal queries. Therefore, we provide two methods for representing them:

(*i*) temporal domain variables and values can be used in both forms and in the conditions section of query programs, and

(*ii*) interval identifiers associated with literals can be used in the conditions section.

With regard to the choice of primitive operators, we believe the choices made in TKL are intuitive and useful. However, the relative advantages of design decisions such as these depend on the individual user and the particular application.

TQuel and HSQL are based on relational algebra in which negation can be

expressed naturally by the *set difference* operator. While negation is problematic in general in logic programming (see page 28), it should be noted that stratified negation, which is used in TKL, does have effective semantics and proof-procedure. In fact, negation in TKL is strictly more expressive than negation in TQuel and HSQL. Note also that because TKL uses logic programming, it can express recursive queries — something that cannot be expressed in TQuel or HSQL. In summary, we have extended temporal database technology by introducing user-defined temporal data types, the broad and narrow interpretation policy for imprecise temporal data, a query language defined in terms of a graph user-interface, and by moving from a relational database framework to a deductive database framework.

Chapter 6

Conclusions

Temporal databases are a natural extension to database technology that can aid in the design and the use of databases which maintain the history of objects they model. In this thesis, the temporal extension is taken one step further to temporal deductive databases. After introducing the topic of temporal deductive databases in Chapter 1, Chapter 2 was devoted to the fundamentals of relational and deductive databases, as well a discussion of the temporal relational database systems TQuel and HSQL and of the temporal deductive systems TEMPLOG and STATELOG. In Chapter 3, polynomial time algorithms were presented for detecting superfiniteness for certain classes of programs. Finally, the details of the design and the implementation of TKL were given in Chapter 4.

Finiteness is known to be undecidable in general for Datalog with function symbols. Ramakrishnan, *et al.* [RBS 87] and Kifer, *et al.* [KRS 88] have developed a methodology which consists of (i) approximating the logic

program with function symbols, with a function-free program together with infinite base relations satisfying *finiteness constraints* (FCs) and (*ii*) testing a stronger notion called *superfiniteness* on the resulting program. They have developed a complete axiom system and a decision procedure for superfiniteness. Unfortunately, the time complexity of their algorithm is exponential in the size of the program and the FCs.

For relational temporal database systems, it was noted that Tquel makes a significant contribution to temporal database technology by providing a simple syntax and semantics for both rollback and historical databases. It has been largely implemented in Quel and it is the only temporal database system known to be operational at this time. HSQL offers a uniform extension to SQL to handle historical data. Among its contributions, in addition to its intuitive extension to SQL syntax, are the development of the integrity constraints needed to maintain consistency in historical relational databases and an extension to relational algebra for completeness when historical tuples are stored with valid intervals or event times.

TEMPLOG and STATELOG are important proposals towards the introduction of temporal deductive databases systems. TEMPLOG is based on temporal logic and is implemented in Prolog. It can easily handle queries about processes that extend for an indefinite time into the future and for which there is a non-finite number of answers. In the case of such processes, each answer is handled one at a time and, hence, non-finiteness is not a problem. However, temporal databases model the past as well as the future and, in general, more sophistication is necessary to detect which queries have a finite number of answers. STATELOG is the only known syntactically

decidable logic languages with function symbols. It develops an innovative methodology for determining a finite representation for infinite query answers. However, because of its very high computational complexity, it is impractical as a general deductive database inference mechanism.

6.1 Contributions of the Thesis

Contributions to the study of temporal deductive databases were made in the detection of queries which are superfinite and in the design of temporal deductive database languages. A summary follows.

6.1.1 Superfiniteness

The following contributions to the theory of superfiniteness have been presented in this thesis.

- Chased R/G tree were introduced as a conceptually simple method for determining superfiniteness based on rule/goal trees. Besides being a useful tool for developing an intuitive understanding for superfiniteness, this method has been useful in proofs.

- The class of compositional linear programs was identified. Superfiniteness information is passed at most one time up and down each branch of the R/G tree for compositional programs.

- A polynomial time algorithm was developed for compositional programs based on transforming the programs together with the given FCs

for (EDB) predicates into a non-deterministic finite automaton (nfa) when the FCs are unary. An nfa is said to be *permissive* if every string over the alphabet of the nfa is a prefix of some string that is accepted by the nfa. An algorithm based on pebbling was developed to decide this property in polynomial time. Superfiniteness reduces to permissiveness of the nfa associated with the program and, therefore, it can be tested in polynomial time.

- For the class of single rule linear recursive programs, superfiniteness reduces to the non-emptiness of the nfa associated with the program and, again, this can be tested in polynomial time.

6.1.2 Temporal Deductive Database Systems

The temporal deductive database language TKL offers a uniform medium for queries and updates in temporal deductive databases. In addition to incorporating algorithms for detecting query finiteness based on superfiniteness analysis, the following features are included in the design of the language.

- The clarity of query exposition is enhanced by means of a graphic user interface which improves clarity in two basic ways:
  - attribute names are visually associated with predicate arguments in forms, and
  - an area is provided to write conditions on statements in a way similar to domain calculus or to tuple calculus.

- The temporal data type is extended beyond that which is found in other temporal database proposals. The following extensions are proposed.
  - User-defined temporal data types are introduced to allow a natural representation for time when the traditional 'date and time of day' representation is inappropriate, (*e.g.*, for a weekly school schedule);
  - The use of time values variable granularities introduces imprecision into temporal data. The temporal interpretation policies broad and narrow are provided as a means for making such values precise; and,
  - Temporal null values can be stored in base relations and used in queries in a meaningful manner.

6.2 Future Work

There remains work to be done on the finiteness problem in deductive databases due to the introduction of function symbols. The immediate work is to extend the work presented here, which uses the permissive property of nfas, to new classes of programs. A general algorithm with acceptable complexity may be achievable by applying algorithm to chase in the R/G tree. It is not clear at this time what the bound on the height of the tree and the number of folds should be. While it is known that finiteness is undecidable for logic programs with function symbols, the decidability of this class of programs with finiteness constraints is open and investigating this question is an interesting avenue for future research.

In the of direction of extending the capabilities of TKL, an interesting problem is to allow user-defined temporal interpretation policies in addition to the predefined *narrow* and *broad* policies. For example, it may be known that the probability that a temporal tuple holds in the imprecise temporal intervals which the `From` and the `To`, and `At` attribute values can represent, follows a normal distribution. In such cases where the probability distribution is known, it should be possible to associate a probability with a tuple that is in the temporal join of two temporal relations.

Bibliography

[AM 87] Abadi, M., Z. Manna. "Temporal logic programming," *Proc. Symposium on Logic Programming*, 1987, pp. 4-16.

[ABW 88] Apt, K., H. Blair, A. Walker. "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, Ca., pp. 89-142, 1988.

[AV 82] Apt, K. R., M. H. Van Emden. "Contributions to the theory of logic programming," J. ACM, Vol.29, No. 3, 1982.

[Ban 85] Bancilhon, F. "Naive evaluation of recursively defined relations," in *On knowledge based management systems,* Brodie and Mylopoulos, Eds. Springer-Verlag, New York, 1985.

[BR 86] Bancilhon, F. and R. Ramakrishnan. "An amateur's introduction to recursive query-processing strategies," ACM SIGMOD Intl. Conf. on Management of Data, pp. 16-20, 1986.

[Baud 89] Baudinet, Marianne. "Temporal logic is complete and expressive," ACM Symposium on Principles of Programming Languages, Austin, Texas, 1989, pp. 267-280.

[Bisk 81] Biskup. J. "Null values in database relations," in *Advances in Data Base Theory*, Vol. 1, H.Gallaire, J.Minker, J.M. Nicolas, eds., Plenum Press, New York, 1981.

[BS 89] Brodsky A. and Y. Sagiv. "Inference of monotonicity constraints in datalog programs," *Proc. ACM PODS, 1989*, pp. 190-199.

[BS 91] Brodsky, A. and Y. Sagiv. "Inference of inequality constraints in logic programs," *Proc. ACM PODS, 1991*, pp. 227-240.

[CGT 89] Ceri, S., G. Gottlob, L. Tanca. "What you always wanted to know about datalog (and never dared to ask)," IEEE Trans. on Knowledge and Data Eng. Vol. 1, No. 1, March 1989, pp. 146-166.

[Chel 80] Chellas, B. F. *Modal logic: an introduction*, Cambridge Univ. Press, 1980.

[Chim 89] Chimenti, D. *et al.* "The LDL system prototype," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 2. No. 1, 1989, pp. 76-90.

[CI 88] Chomicki, J., T. Imieliński. "Temporal deductive databases and infinite objects," *ACM PODS*, 1988.

[CI 90] Chomicki, J., T. Imieliński. "Finite representation of infinite query answers", Tech. Report TR-CS-90-10, Dept. of Comp. and Information Sciences, Kansas State Univ., 1990.

[CW 82] Clifford, J., D. S. Warren. "Formal semantics for time in databases," ACM Trans. on Database Systems, Vol. 6 No. 2, 1983, pp. 214-254.

[C 70] Codd, E. F. "A relational model of data for large shared data banks," Comm. ACM Vol. 13, No. 6, pp. 377-387.

[Col 73] Colmerauer, A., *et al. Un système de communication homme-machine en français,* Tech. Report, Université d'Aix, France, 1973.

[Da 86] Date, C. J. *Relational databases: selected writings,* Addison-Wesley, Reading, Mass., 1986.

[D 90] Desai, B. C. *An introduction to database systems,* West Publishing Co., St. Paul, Minn., 1990.

[GS 91] Gunadhi, H., A. Segev. "Query processing for algorithms for temporal intersection joins," *Proc. of the 7th International Conf. on Data Eng.*,Kobe, Japan, 1991, pp. 336-344.

[I 89] Ioannidis, Y. "Commutativity and its role in the processing of linear recursion," *IEEE VLDB*, 1989.

[JCGSS] Jensen, C. S., J. Clifford, S. K. Gadia, A. Segiv, R. T. Snodgrass. "A glossary of temporal database concepts," *in preparation.*

[KL 88] Kifer, M., and E.L. Lozinskii. "SYGRAF: Implementing logic programs in a datalog style," in *IEEE Transactions on Software Engineering*, Vol. 14 No.7, 1988, pp. 922-934.

[KRS 88] Kifer, M., R. Ramakrishnan and A. Silberschatz. "An axiomatic approach to deciding finiteness of queries in deductive databases," *Proc. ACM PODS*, 1988. (Extended version available as Tech. Report, Department of Comp. Sci., SUNY, Stony Brook, NY, 1991.)

[KRS 88a] Krishnamurthy, R., R. Ramakrishnan, and O. Shmueli. "A framework for testing safety and effective computability," *Proc. ACM SIGMOD Conf.*, 1988, pp. 154-163.

[Laks 89] Lakshmanan, V.S. "Query evaluation with null values: how complex is completeness?," *Proc. 6th Foundations of Software Technology and Computer Science*, Springer-Verlag 1989, pp. 204-222.

[LN 92a] Lakshmanan, V.S., D. Nonen. "Superfiniteness of query answers in deductive databases: an automata-theoretic approach," *Proc. 9th Foundations of Software Technology and Computer Science*, 1992. *(Extended version in preparation.)*

[LN 92b] Lakshmanan, V.S., D. Nonen. "On querying temporal deductive databases" *Workshop on Formal Method in Databases & Software Engineering*, Springer-Verlag, London, 1993.

[LSS 93] Lakshmanan, V.S., F. Sadri, I. N. Subramanian. "On the logical foundation of schema integration and evolution in heterogeneous database systems," *to appear in* Third International Conference on Deductive and Object-Oriented Databases, Phoenix, Az., Dec. 1993.

[Leun 90] Leung, T., R. Muntz, "Query processing for temporal databases," *Proc. 6th International Conference on Data Engineering*, Los Angles, Calif. Feb. 1990, pp. 200-208.

[L 92] Linton, Mark A., *et al.* "InterViews reference manual version 3.1," available as part of the InterViews 3.1, Stanford Univ. Ca. Dec. 1992.

[Ll 87] Lloyd, J. W. *Foundations of logic programming,* second extended ed. Springer-Verlag, New York, 1987.

[N 89] Naughton, J.F. "Data independent recursion in deductive databases," *JCSS*, 38 (1989), pp. 259-289.

[RBS 87] Ramakrishnan, R., F. Bancilhon, and A. Silberschatz. "Safety of recursive Horn clauses with infinite relations," *ACM Principles of Databases Systems*, 1987 pp. 328-339.

[Rei 78] Reiter, R. "On closed world databases," in *Symposium on logic and data bases, Centre d'etudes et de recherches de Toulouse, 1977,* edited by H. Gallaire and J. Minker, 1978, pp. 55-76.

[SV 89] Sagiv, Y. and M. Vardi. "Safety of datalog queries over infinite databases," in *Proc. ACM PODS, 1989*, pp. 160-171.

[Sard 90] Sarda, N. "Extensions to SQL for historical databases," *IEEE Transactions on Knowledge and Data Engineering*, 2, No. 2, June 1990, pp.220-230.

[Seth 89] Sethi, R. *Programming languages concepts and constructs*, Addison-Wesley Publishing Co. Reading, Mass., 1989.

[Sh 87] Shmueli, O. "Decidability and expressiveness aspects of logic queries," *ACM PODS*, 1987, pp. 237-249.

[Snod 87] Snodgrass, R. "The temporal query language TQuel," *ACM Transactions on Database Systems*, 12, No. 2, June 1987, pp. 247-298.

[SG 91] Sohn, K., A. Van Gelder. "Termination in logic programs using argument size," *Proc. ACM PODS, 1991*, pp. 216-226

[Soo 91] Soo, M.D. "Bibliography of temporal databases," *Sigmod Record*, Vol.20, No.1, March, 1991, pp. 14-23.

[T 91] Tsur S. "Deductive databases in action" in *Proc. ACM PODS, 1991*, pp. 142-153.

[Ull 89] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, Vol. I & II, Computer Science Press, 1989.

[Var 89] Vardi, M. "Automata Theory for Database Theoreticians," *Proc. ACM PODS*, 1989.

[Zlo 77] Zloof, M. "Query-by-example: a database language," *IBM Systems J.* 16:4 pp. 324-343.