Integration of Active and Deductive Database Rules

DISSERTATION

zur Erlangung des Doktorgrades
der Fakultät für Angewandte Wissenschaften
der Albert-Ludwigs-Universität Freiburg im Breisgau

von

Bertram Ludäscher

Dekan: Prof. Dr.-Ing. Hans Burkhardt
Erstreferent: Prof. Dr. Georg Lausen
Zweitreferent: Prof. Dr. Bernhard Nebel

Datum der Promotion: 25.2.1998

Foreword

Rules are a very popular and widely used paradigm in database programming. Depending on the intended application area, one can distinguish between different rule paradigms:

Active rules extend the query capabilities of traditional (passive) database systems by supporting (re)active behaviour like automatic triggering of updates as a response to external or internal events. The dynamic features of active rules are used in non-standard applications, for example, workflow management and for the implementation of advanced database functionality like automatic integrity enforcement and view maintenance. Despite their flexibility and numerous application areas, active rules have some drawbacks, most notably the highly complex and often intricate rule semantics. As a consequence, understanding or predicting the behaviour of active rules frequently becomes a very difficult if not impossible task.

Deductive rules, on the other hand, allow a very natural and intuitive modelling of the static aspects of database applications, i.e., (recursive) views and integrity constraints. The quest for "reasonable" semantics of deductive rules in the presence of negation and recursion has produced some generally accepted declarative semantics for deductive rules and solid theoretical foundations (like expressiveness and complexity).

Although active rules and deductive rules reflect quite different intentions, i.e., active rules have an imperative flavour, whereas deductive rules mirror a reasoning process or express normative statements in the form of integrity constraints, a unified framework integrating both paradigms has several benefits: From the viewpoint of active rules, the important issue of assigning a precise and unambiguous semantics is resolved. Conversely, the capabilities of deductive rules are extended by means to define updates and dynamic behaviour, thereby addressing a central problem in the area of deductive databases.

Thus, by providing a unified logic-based framework for active and deductive rules, the present work is an important contribution to both problems, i.e., formal semantics of active rules and dynamic extensions of deductive rules. Apart from the practical gain for the rule designer, who now can express recursive views, static and dynamic integrity constraints, and event-driven updates in a single language, a number of theoretical results are established in this work, shedding light on such fundamental rule properties as expressiveness, complexity, and termination behaviour. The proposed language *Statelog* has an intuitive set-oriented semantics, which lends itself to a bottom-up evaluation and, therefore, can be implemented using standard database techniques. The author shows that certain deductive rules and production rules have a very natural representation in Statelog and hence are just special classes of Statelog rules. Despite its simplicity, the language allows to express many procedural features of active rules including different coupling modes and composite event detection in a precise and elegant way. The reader, whether s/he comes from the background of active databases or deductive databases, will find in this work valuable information on both fields.

Freiburg, June 1998 *Prof. Dr. Georg Lausen*

Acknowledgments

This work was performed while I was a full time research assistant in the database research group of Prof. Dr. Georg Lausen, first at the University of Mannheim and—to the larger extent—at the Institute of Computer Science of the newly founded Faculty for Applied Sciences at the University of Freiburg. I am deeply indebted to Prof. Dr. Georg Lausen, my supervisor, for his support and guidance during the evolution of this thesis. I would also like to thank my second advisor Prof. Dr. Bernhard Nebel, who provided valuable comments from the AI perspective. A special thanks to all my current and former colleagues, especially the folks in the database group: Jürgen Frohn, Rainer Himmeröder, Paul-Thomas Kandzia, Wolfgang May, Klaus Ries, Christian Schlepphorst, and Heinz Uphoff; the communication in this circle has often been encouraging, refreshing and fun. Ulrich Hamann implemented a prototype for (pure) Statelog in the course of his Master's Thesis and was a supportive collaborator in the "early Statelog days". For illuminating insights into the "realm of the finite", I thank Prof. Dr. Jörg Flum and Max Kubierschky. Clarifying discussions and exchange of ideas with other researchers in the field have been very valuable and stimulating; here, I want to thank Jan Chomicki, Jürgen Dix, and Victor Vianu, in place of many others.

Last, but not least, I am also very grateful for some rather different and special kind of support, both from my parents, and from Brigitte, who provided me with inspiring and complex non-database events.

Thank you all!

Freiburg, June 1998 *Bertram Ludäscher*

Abstract

Rules are an important paradigm in database programming. In this work, a state-oriented extension of Datalog called *Statelog* is proposed, which integrates dynamic capabilities of *active rules* and the declarative semantics of *deductive rules* in a unified logical framework. A basic idea of the language is to incorporate state terms, thereby allowing to refer to different database states within the rule language. In Statelog, database transactions are modeled as a sequence of intermediate transitions, where each transition is declaratively specified by deductive rules.

After briefly reviewing basic notions and the state of the art in active, production, and deductive rules, respectively, a number of theoretical properties of Statelog are investigated: The expressive power of Statelog is characterized wrt. single transitions and transactions using several normal forms of the language. Moreover, it is shown that certain deductive and production rules correspond to special classes of Statelog programs. The complexity of transitions and transactions is determined based on the established expressiveness results: The problem of deciding whether a given tuple is contained in the result of a transition or transaction is complete for **PTIME** and **PSPACE**, respectively. Results on the complexity of detecting termination conclude the first part of the work.

In the second more practically oriented part, a framework for active rules is developed, which uses pure Statelog as the underlying core language. This extended framework provides the user with a set of logically specified, predefined operations, thereby facilitating the handling of low-level procedural aspects. For example, the semantics of primitive updates and transaction control is given by system-defined rules. Using so-called Δ-monotonic rules, a method for enforcing rule termination for an efficient subclass of Statelog rules (i.e., with **PTIME** transaction complexity) is presented. Finally, the handling of static and dynamic integrity constraints, and the possibilities and limitations of composite event detection are investigated.

Contents

Figures

Definitions

Theorems

Examples

1 Introduction

Relational databases, popularized in the early seventies by the pioneering work of Codd [Cod70], have become enormously influential ever since, and are todays most widespread and commercially successful database technology. One advantage of the relational model is the flexibility and ease by which the contents of the database can be queried using expressions of the relational calculus, the relational algebra, or the more verbose analogue of the latter, the Structured Query Language (SQL).

Motivated by the need for increased expressiveness and the advent of new applications, *rules* have become very popular as a paradigm in database programming since the late eighties (cf. [Min96]). Today, there is a plethora of quite different application areas and semantics for rules. From a bird's-eye view, deductive and active rules may be regarded as two ends of a spectrum of database rule languages:

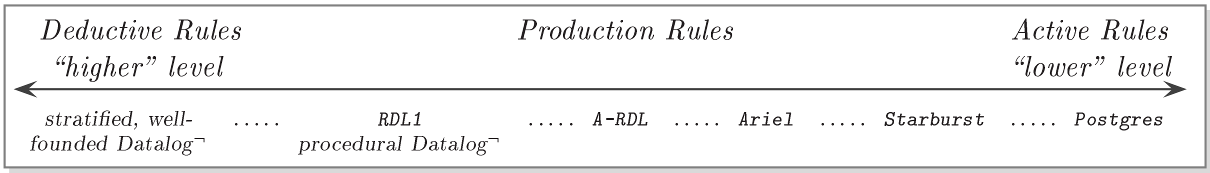


Figure 1.1: Spectrum of database rule languages (adapted from [Wid93])

On the one end of the spectrum, *deductive rules* provide a concise and elegant representation of intensionally defined data. Recursive views and static integrity constraints can be specified in a declarative and uniform way using deductive rules, thereby extending the query capabilities of traditional relational languages like SQL. Moreover, the semantics developed for deductive rules with negation are closely related to languages from the field of knowledge representation and nonmonotonic reasoning, which substantiates the claim that deductive rules are rather "high-level" and model a kind of natural reasoning process. However, deductive rules do not provide enough expressiveness or control to directly support the specification of updates or active behavior. Since updates play a crucial role even in traditional database applications, numerous approaches have been introduced to incorporate updates into deductive rules.

In contrast to deductive rules, *active rules* support (re)active behavior like triggering of updates as a response to external or internal (database) events. Conceptually, most existing active rule languages are comparatively "low-level" and often allow to exert explicit control on rule execution. While such additional procedural control increases the expressive power of the language considerably, this is also the reason why the behavior of active rules is usually much more difficult to understand or predict than the meaning of deductive rules. Not surprisingly, researchers continue to complain about the unpredictable behavior of active rules and the lack of a uniform and clear semantics.[1]

[1] *"The unstructured, unpredictable, and often nondeterministic behavior of rule processing can become a*

Production rules constitute an intermediate family of languages, which provide facilities to express updates and some aspects of active behavior, yet avoid overly detailed control features of active rules at the right end of the spectrum.

Contributions

In this thesis it is shown how the seemingly discordant paradigms of active and deductive rules, respectively, can be reconciled using *Statelog* (= ***State**s+Data**log***), a state-oriented extension of Datalog. A key idea of the approach is to incorporate an explicit notion of state into deductive rules. The benefits of the integrated Statelog framework are twofold. First, from the viewpoint of deductive rules, the possibility to specify (re)active update rules increases the practical applicability of deductive rules. For example, in addition to checking static integrity constraints, it is now possible to express dynamic constraints and constraint enforcement policies. Second, seen from the point of view of active rules, a clear, intuitive, and logical rule semantics is obtained, thereby allowing to investigate theoretical properties of active rules like expressive power, complexity, and termination in a rigorous way. These issues are dealt with in the first main part of the work:

Using normal forms for Statelog and a novel distinction between transition and transaction expressiveness, the expressive power of the language is determined and its complexity is characterized: The main variant of the language is equivalent to S-DATALOG wrt. transitions and to WHILE wrt. transactions. This implies that, in general, the evaluation of transitions and transactions is **PTIME**-complete and **PSPACE**-complete, respectively. The main normal forms are

- *NF-Statelog*, where each rule corresponds to either a deductive rule, *or* to a (Datalog$^\neg$) production rule, and
- *P-Statelog*, where only production rules occur.

Due to these normal forms, Statelog is readily implementable using standard bottom-up techniques.[2] Several production rule semantics and declarative semantics for Datalog$^\neg$ turn out to be special cases of Statelog, formally substantiating the claim that Statelog integrates these different paradigms. Finally, borrowing from similar techniques for Datalog$^\neg$, results on the complexity of detecting termination of Statelog rules at run-time (**PSPACE**-complete) and at compile-time (undecidable) conclude the first main part of the work. Most of the presented results carry over to the related but independently proposed language *XY-Datalog* [Zan93, ZS94] and yield new insight into the precise complexity, expressiveness, and termination behavior of XY-Datalog.

In the second main part of the work, a framework for active rules is developed, using Statelog as the underlying "core" language. With this extension, Statelog not only encompasses deductive and production rules, but also essential features of active rules. In this framework, several variants of so-called Δ-*monotonic* rules are proposed, which yield a tractable (**PTIME**-evaluable) class of terminating Statelog programs. Moreover, using techniques from [Cho95b], it is shown how dynamic integrity constraints and composite event detection can

nightmare for the database rule programmer" [AWH95]; similar statements can be found in [Wid94], [WC96a], [PCFW95], [DHW95], and [FT95].

[2] A prototype based on the deductive database system *XSB-Prolog* has been implemented in the course of a diploma-thesis [Ham95].

be expressed in Statelog. New insight into composite events is gained by a result showing the inexpressibility—in the pure framework—of event detecting with chronicle context. By incorporating timestamps into the active domain, also event detection with chronicle context can be handled in the way described in [MZ95].

Chapter Outline

The necessary formal background and notational conventions for subsequent chapters are given in Chapter 2. Chapters 3, 4, and 5 have introductory character and briefly review the state of the art in active, production, and deductive rules, respectively.

Chapter 6 develops Statelog in its "pure" form and develops the main technical results on normal forms, expressive power, complexity, and termination.

In Chapter 7, the Statelog framework for active rules is presented taking into account specific features like delta relations, transaction control, integrity constraints, and complex event detection.

Chapter 8 proposes some extensions of the basic framework, and Chapter 9 contains a brief summary and some concluding remarks. Finally, in Appendix A, two prototypical applications of active rules, i.e., incremental view maintenance and maintenance of referential integrity, are presented.

2 Preliminaries and Notation

We assume familiarity with the basic notions of relational and deductive databases. The notation and terminology is close to [AHV95]; further standard references are [Ull88], [CGT90], and [Kan90]. In this chapter, for convenience and to fix our notation, we briefly introduce some formal background needed in Chapter 4 and onward. We will use the following notational conventions:

Constants	$a, b, c, \ldots$
Variables	$X, Y, Z, \ldots$
Terms	$t_1, t_2, \ldots$
Relation symbols	$p, q, r, \ldots$
Atoms	$A, B, C, \ldots$
First-order formulas	$\varphi, \psi, \ldots$
Database schemas	$\mathbf{R}, \mathbf{S}, \ldots$
Database instances, structures	$\mathcal{D}, \mathcal{I}, \mathcal{J}, \ldots$

2.1 First-Order Logic

Syntax

The signature of a first-order language consists of disjoint countable sets $\mathbb{R}$ of *relation* (or *predicate*) *symbols*, $\mathbb{F}$ of *function symbols*, and $\mathbb{C}$ of *constants*.[1] To each $p \in \mathbb{R}$ and $f \in \mathbb{F}$ we associate an *arity* such that $arity(p) \geq 0$ and $arity(f) \geq 1$. In addition, the first-order vocabulary contains a countable set of variable symbols $\mathbb{V}$.

The set of *terms* over $\mathbb{F}$, $\mathbb{C}$, and $\mathbb{V}$ is the least set such that (i) constants and variables are terms, and (ii) for every k-ary $f \in \mathbb{F}$: $f(t_1, \ldots, t_k)$ is a term if $t_1, \ldots, t_k$ are terms.

If $p \in \mathbb{R}$ is k-ary and $t_1, \ldots, t_k$ are terms, then $p(t_1, \ldots, t_k)$ is an *atomic formula* (or *atom*). A *formula* is either an atomic formula, or of the form $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \leftarrow \psi)$, $(\neg \varphi)$, $(\exists X \varphi)$, or $(\forall X \varphi)$, where φ and ψ are formulas, and $X \in \mathbb{V}$. Parentheses are omitted when understood from the context. A *literal* is an atom A or its negation $\neg A$. A formula without free variables (i.e., where all variable occurrences are bound by a quantifier) is called a *sentence*. A formula or term is *ground* if it involves no variables.

The notation $\varphi(\bar{X})$ indicates that all free variables of φ are among $\bar{X} = X_1, \ldots, X_n$; if we write φ, nothing is said about the free variables of φ.

[1] We chose to distinguish constants due to their special role in databases; often they are viewed as 0-ary function symbols.

Semantics

A first-order *structure* $\mathcal{I} = (U, I)$ comprises a *universe* (or *domain*) U and an interpretation function I assigning to every constant c, k-ary function symbol f, and n-ary relation symbol p, a domain element $c^{\mathcal{I}}$, a k-ary function $f^{\mathcal{I}} : U^k \to U$, and an n-ary relation $p^{\mathcal{I}} \subseteq U^n$, respectively. If $|U| < \infty$ then $\mathcal{I}$ is called a *finite structure* (or *database instance*, see below).

The *satisfaction relation* between structures and sentences is defined as usual: Let $v : \mathbb{V} \to U$ be a *valuation* (or *variable assignment*) mapping variables to domain elements; v extends to terms in the obvious way. Then:

- $\mathcal{I}, v \models p(t_1, \ldots, t_n)$, if $(v(t_1), \ldots, v(t_n)) \in p^{\mathcal{I}}$, (*for terms t_i*)
- $\mathcal{I}, v \models \neg\varphi$, if not $\mathcal{I}, v \models \varphi$,
- $\mathcal{I}, v \models \varphi \wedge \psi$, if $\mathcal{I}, v \models \varphi$ and $\mathcal{I}, v \models \psi$, (*similar for $\vee, \leftarrow$*)
- $\mathcal{I}, v \models \exists X \varphi$, if there is some $a \in U$ s.t. $\mathcal{I}, v_{X/a} \models \varphi$, (*similar for $\forall$*)

Here, $v_{X/a}$ is defined like v, but maps X to a. If $\varphi = \varphi(X_1, \ldots, X_k)$, i.e., $X_1, \ldots, X_k$ are the free variables of φ, then we may simply write $\mathcal{I} \models \varphi(x_1, \ldots, x_k)$ instead of $\mathcal{I}, v \models \varphi$, where $v(X_i) := x_i$ for all $i = 1, \ldots, k$. In particular, if φ is a sentence, we just write $\mathcal{I} \models \varphi$, and say that $\mathcal{I}$ *satisfies* φ, or $\mathcal{I}$ is a *model* of φ. A structure $\mathcal{I}$ is a model of a set Φ of sentences, if $\mathcal{I} \models \varphi$ for all $\varphi \in \Phi$.

Herbrand Structures

As usual in logic programming and deductive databases, we will only consider *Herbrand structures*, i.e., structures which interpret constants and function symbols in a canonical "syntactic" way. More precisely, if P is a set of expressions from a first-order language—most notably a logic program (possibly including a database as a set of facts)—then the *Herbrand universe* U_P is the set of ground terms which can be built from constants and function symbols in P. Now, every *Herbrand interpretation* $\mathcal{I}$ interprets terms "syntactically", that is, $t^{\mathcal{I}} := t$.

The *Herbrand base* $\mathcal{B}_P$ of P comprises all ground atoms which can be constructed from the relation symbols of P and terms from U_P. Since Herbrand interpretations only vary in the interpretation of relation symbols, they can be conveniently represented as subsets $\mathcal{I} \subseteq \mathcal{B}_P$: Atoms in $\mathcal{I}$ are regarded as *true* whereas those in $\mathcal{B}_P \setminus \mathcal{I}$ are *false*.

2.2 Specialization to the Database Context

Relational Databases

A relational *database schema* $\mathbf{R}$ is a finite set of relation symbols (with associated arities). Fix some underlying domain U. A relational *database instance* $\mathcal{I}$ (or *database* for short) over a schema $\mathbf{R}$ is a mapping which assigns to every k-ary relation symbol $p \in \mathbf{R}$ a *finite* k-ary relation $p^{\mathcal{I}} \subseteq U^k$. Thus, a database can be conceived as a finite first-order Herbrand structure, where the Herbrand universe comprises only constants but no terms with function symbols. In particular, constants and domain elements are identified. The set of all database instances over $\mathbf{R}$ is denoted *inst*($\mathbf{R}$). The *active domain* of a database $\mathcal{I} = (U, I)$, denoted *adom*($\mathcal{I}$), is the set of domain elements occurring in I. If the schema of $\mathcal{I}$ contains a binary relation symbol "$<$" which is interpreted as a linear order[2] then $\mathcal{I}$ is called an *ordered database* (or *ordered structure*).

Database Transformations

Fix two database schemas $\mathbf{R}$ and $\mathbf{S}$. A *database transformation* (*db-transformation*) is a mapping

$$\begin{array}{rccc} q: & \mathit{inst}(\mathbf{R}) & \rightarrow & \mathit{inst}(\mathbf{S}) \\ & \mathcal{I} & \mapsto & \mathcal{J} \end{array}$$

assigning to each instance $\mathcal{I}$ over the input schema $\mathbf{R}$ an instance $\mathcal{J}$ over the output schema $\mathbf{S}$. The relations in $\mathbf{R} \cap \mathbf{S}$ are *updated* by q, while those in $\mathbf{S} \setminus \mathbf{R}$ are *intensionally defined* by q. In particular, if $\mathbf{R} = \mathbf{S}$ then q defines an *update*. On the other hand, if $\mathbf{R} \cap \mathbf{S} = \emptyset$, then q is called a *query* since it does not change any input relation of $\mathcal{I}$. If $\mathbf{S}$ contains a single relation ans where $\mathit{arity}(\mathsf{ans}) = k$, then q is called a k*-ary query*; if $k = 0$ then q is a *boolean* or *yes/no* query.

Often[3], when we consider the expressive power and complexity of db-transformations, the distinction between queries and updates is insignificant, and we simply use the more traditional term *query* instead of db-transformation.

First-Order Queries

An important class of db-transformations is FO, the class of *first-order queries*. Let $\varphi(\bar{X})$ be a first-order formula with free variables $\bar{X} = X_1, \ldots, X_k$ and relations over the schema $\mathbf{R}_\varphi$. Then φ induces a k-ary query $q_\varphi : \ \mathit{inst}(\mathbf{R}_\varphi) \rightarrow \mathit{inst}(\{\mathsf{ans}\})$ as follows:

$$q_\varphi : \quad \mathcal{I} \mapsto \{\mathsf{ans}(x_1, \ldots, x_k) \mid \mathcal{I} \models \varphi(x_1, \ldots, x_k)\}.$$

In the database community, FO is also referred to as the *relational calculus*. $q_\varphi(\mathcal{I})$ is sometimes called the *answer set* (or just *answer*) of φ in $\mathcal{I}$.

Domain Independence

Certain first-order formulas give rise to *domain dependent* (also called *unsafe*) queries. For example, $\varphi(X) := \neg\, p(X)$ is domain dependent, since for a given database $\mathcal{I}$, $q_\varphi(\mathcal{I})$ yields all elements which are not in $p^{\mathcal{I}}$. Clearly, the result depends on the underlying domain U. In particular, if U is infinite then q_φ is infinite as well, so the answer to such a query cannot be directly represented as a relational database. One solution to this problem is to use the *active domain semantics* in which quantifiers do not range over the entire underlying domain (as in the *natural semantics*) but only over the active domain $\mathit{adom}(\mathcal{I})$ (cf. [AHV95]). This guarantees that even domain dependent formulas always yield finite answers. Syntactically, this semantics can be enforced by adding a predicate $\mathsf{adom}(X)$ at appropriate places, and defining adom as the union of all domain elements of $\mathcal{I}$. However, this method may lead to

[2] For all $a, b, c \in \mathit{adom}(\mathcal{I})$: (i) not $a <^{\mathcal{I}} a$, (ii) if $a \neq b$ then $a <^{\mathcal{I}} b$ or $b <^{\mathcal{I}} a$, and (iii) if $a <^{\mathcal{I}} b$ and $b <^{\mathcal{I}} c$, then $a <^{\mathcal{I}} c$.

[3] ... but not always. Some languages have different expressiveness when considered as query or as update languages, see Example 4.5.

unexpected results: e.g., given an infinite domain U, the formula $\psi(X) := \forall Y\, p(X, Y)$ yields an empty answer for any finite database relation p. On the other hand, under the active domain semantics, $\psi(X)$ produces a nonempty answer provided all values of the active domain occur in the second column of p. Thus, it is desirable to restrict to *domain independent* formulas, which produce the same answers when evaluated wrt. any domain $U \supseteq adom(\mathcal{I})$. In particular, the active domain semantics coincides with the natural semantics on domain independent formulas. Although domain independence is undecidable [DP69], there are well-known decidable classes of domain independent formulas which yield the same expressive power as FO; most notably the so-called *safe*, *allowed*, *range-restricted*, and *evaluable* formulas (see [VGT91]). The framework of *constraint databases* provides another very elegant and general solution by allowing finite representations of infinite answers using so-called *generalized tuples* [KKR95].

Genericity

Database transformations are usually required to be *generic*. Informally, this means that solely information at the conceptual level, i.e., provided by the schema $\mathbf{R}$ may be used. More precisely, q is generic if it is preserved under isomorphisms, i.e., for every isomorphism π of $\mathcal{I}$, $q(\pi(\mathcal{I})) = \pi(q(\mathcal{I}))$.

Example 2.1 (Genericity) Consider the database instance $\mathcal{I} = \{e(a, b), e(b, a), e(c, d)\}$ over the schema $\mathbf{R} = \{e\}$ and the query q: *"return all edges $(x, y) \in e^{\mathcal{I}}$ without symmetric edge (y, x) in $e^{\mathcal{I}}$"*. Thus,

$$q(\mathcal{I}) := \{\mathsf{ans}(x, y) \mid \mathcal{I} \models e(x, y) \wedge \neg\, e(y, x)\},$$

which yields $\{\mathsf{ans}(c, d)\}$ for the given database. Now consider the permutation $\pi(a) := b, \pi(b) := a, \pi(c) := d$, and $\pi(d) := c$. π extends to tuples and database instances in the obvious way. Then $\pi(\mathcal{I}) = \{e(b, a), e(a, b), e(d, c)\}$, so π is an isomorphism on $\mathcal{I}$. Genericity requires that $q(\pi(\mathcal{I})) = \{\mathsf{ans}(d, c)\} = \pi(q(\mathcal{I}))$ which is indeed the case.

Note that no generic query can distinguish between a and b in $\mathcal{I}$: Let $\pi(a) := b, \pi(b) := a$, and $\pi(x) := x$ for all other constants. Observe that $\pi(\mathcal{I}) = \mathcal{I}$, i.e., π is an *automorphism* of $\mathcal{I}$. Now for every generic query q over $\{e\}$ we have $\pi(a) \in q(\mathcal{I}) \Leftrightarrow a \in q(\pi(\mathcal{I}))$ (by genericity) $\Leftrightarrow a \in q(\mathcal{I})$ (since $\pi(\mathcal{I}) = \mathcal{I}$). Therefore, either both a and b belong to $q(\mathcal{I})$, or neither does. □

In general, if $\pi(x) = y$ for an automorphism π on $\mathcal{I}$, then *no* generic query q can distinguish between x and y. Genericity is a desirable property originating from the data independence principle. However, there is a price to pay for it: there are certain easy queries (wrt. their Turing complexity) which are very hard to compute using generic computations. For example, the following simple query is not a WHILE query [AHV95, Ch. 17.3]:

$$even(p) := \begin{cases} true, & \text{if } |p^{\mathcal{I}}| \text{ is even,} \\ false, & \text{otherwise.} \end{cases}$$

On ordered structures, *even* can be expressed easily, e.g., in S-DATALOG (cf. Figure 4.2). However, an order allows access to individual elements of the domain, so the advantage of genericity is lost.

Expressive Power

A relational *database language* L is a set of expressions together with a semantics which associates with every expression $P \in L$ a db-transformation q_P. Usually, P will be a Datalog or Statelog program, and the input and output schemas $\mathbf{R}$ and $\mathbf{S}$, respectively, are implicitly given by P.

The *expressive power* of a database language L is the class of all db-transformations definable in L. $P_1 \in L_1$ is *equivalent to* $P_2 \in L_2$ if they express the same db-transformation. We say that L_1 is *at most as expressive as* L_2, denoted $L_1 \leq L_2$, if for every expression in L_1 there is an equivalent expression in L_2. Both languages have the *same expressive power*, written as $L_1 \equiv L_2$, if $L_1 \leq L_2$ and $L_2 \leq L_1$. We say that L_1 is *strictly more expressive* then L_2, denoted $L_1 < L_2$ if $L_1 \leq L_2$ but not $L_1 \equiv L_2$. L may denote both the language and the class of db-transformations definable in it.

One of the classic results on expressive power, also known as *Codd's Theorem*, is FO $\equiv$ RA, i.e., the equivalence between first-order queries, and queries expressible in the *relational algebra* RA [Cod70].[4]

Complexity of Database Transformations

The complexity of db-transformations is measured by relating them to classical complexity classes. There are two main kinds of complexity which have to be distinguished (cf. [Var82], [AHV95]): The *data complexity* is the complexity of evaluating a fixed db-transformation for variable databases as inputs, whereas the *expression complexity* assumes that the database is fixed, and the expression representing the db-transformation is variable. In the database context, the size of the database usually dominates by far the size of the expression, so the data complexity measure is more important here. Therefore by *complexity* we mean data complexity unless otherwise stated.

The complexity of a db-transformation q is given by the following *recognition problem*: Given an instance $\mathcal{I}$ and a tuple t, decide whether $t \in q(\mathcal{I})$. For a Turing complexity class $\mathbf{C}$, the class of all db-transformations whose recognition problem lies in $\mathbf{C}$ is denoted by DB-$\mathbf{C}$.

A database language L is said to be *complete wrt.* $\mathbf{C}$, or just $\mathbf{C}$-*complete*, if (i) each db-transformation expressible in L is in DB-$\mathbf{C}$, and (ii) L can express some db-transformation whose recognition problem is complete wrt. $\mathbf{C}$ (in the classical sense). Note that this does *not* imply that L captures all of DB-$\mathbf{C}$, since there may be db-transformations in DB-$\mathbf{C}$ whose reduction to the $\mathbf{C}$-complete problem is computationally easy, yet inexpressible in L (see [AHV95, Ch. 16.3] for further details).

[4] There are several equivalent versions of RA around, e.g., the *named relational algebra* based on the operators σ (*select*), π (*project*), $\bowtie$ (*join*), $\cup$ (*union*), δ (*rename*), and $-$ (*set difference*) [AHV95].

3 Active Rules

3.1 Introduction

Traditional database systems are *passive*, i.e., they only execute queries or transactions explicitly submitted by the user or an application program. In contrast, *active database systems* (ADBs) allow to define (re)active behavior by means of *active rules* resulting in a quite flexible and powerful formalism. ADBs support automatic triggering of data manipulating operations or more general actions as a response to external or internal events, and are attractive for several reasons: A number of advanced database features like checking and enforcing integrity constraints, incremental view maintenance, (view) updates, access and version control, and many others, can be expressed naturally and uniformly by active rules. From a software engineering perspective, it is advantageous that functionality can be moved from application programs to the ADB. Thus, more code can be shared by different application programs, thereby facilitating development and maintenance of applications. Moreover, an ADB can be more efficient for monitoring tasks compared to a passive system which would have to be polled frequently by the application program. Finally, active rules can be used in advanced applications like workflow management and data warehousing applications, the latter requiring maintenance of huge materialized views.

3.2 ECA Rules

Active rules are typically expressed as *Event-Condition-Action (ECA)* rules of the form

$$\textbf{on } \langle \textit{event} \rangle \textbf{ if } \langle \textit{condition} \rangle \textbf{ then } \langle \textit{action} \rangle .$$

The intuitive meaning of such a rule is: Whenever the specified event occurs, the rule is triggered and the condition is checked on the current database state. If it evaluates to true, the specified action is executed. Thus, rule processing in an ADB involves the tasks *event detection*, *condition evaluation*, and *action execution*; see Figure 3.1 for the basic architecture of an active database system. *Condition-action rules*, i.e., ECA rules without the event part are sometimes called *production rules*, *event-action rules* are sometimes referred to as *triggers*.

The following illustrative example (adopted from [WF90]) gives a first impression of programming with active rules using an already "classical" application of active rules, i.e., enforcement of referential integrity:

Example 3.1 (Integrity Enforcement) Consider a relational schema involving the relations $\mathsf{emp}(\mathsf{EmpNo}, \mathsf{Salary}, \ \mathsf{DeptNo})$ and $\mathsf{dept}(\mathsf{DeptNo}, \mathsf{MgrNo})$ with their obvious meaning. We want to enforce the inclusion dependencies

$$\mathsf{dept}[\mathsf{MgrNo}] \subseteq \mathsf{emp}[\mathsf{EmpNo}] \quad \text{and} \quad \mathsf{emp}[\mathsf{DeptNo}] \subseteq \mathsf{dept}[\mathsf{DeptNo}] \ ,$$

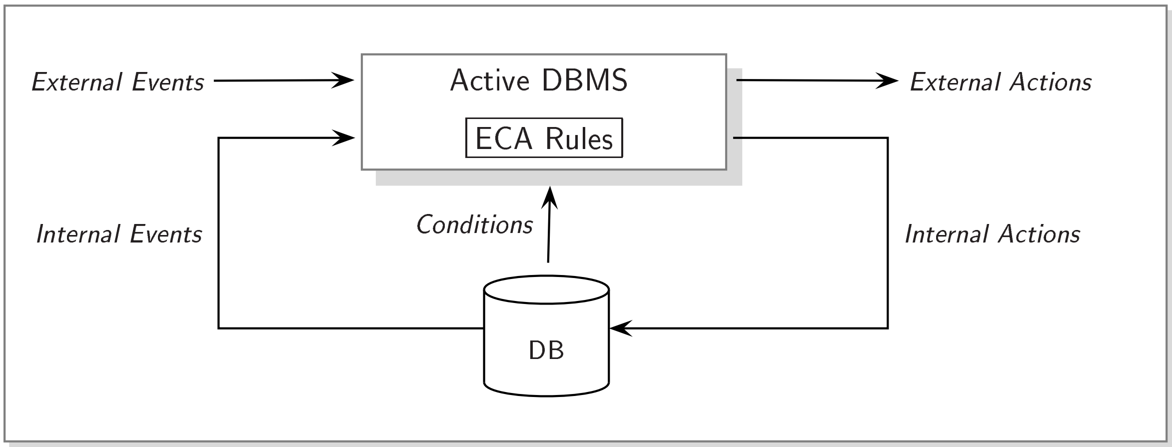


Figure 3.1: Basic ADB architecture

i.e., every manager referred to in relation dept must also occur in relation emp, and every department referred to in emp must occur in dept. If an employee tuple is deleted from emp, the first inclusion dependency will get violated provided this employee is a manager. To maintain integrity, we can *cascade* the deletion to the department relation and delete the corresponding department(s). Note that this in turn may violate the second constraint. Thus, we also have to delete all employees working in the department of the "fired" manager. This policy can be implemented by the following (informal) ECA rule:

on	delete from emp
if	E in emp.deleted **and** E manages D **and** E1 works for D
then	delete D from dept **and** delete E1 from emp .

Assuming the concrete semantics corresponds to the intuitive semantics, this rule enforces the given constraints in case of deletions to emp. However, it is an important and—as will be seen—rather involved task to precisely define the semantics of ECA rules. For instance, the rule above may trigger itself recursively, since the specified action can raise new deletion events. Moreover, the interaction with other rules (e.g., which specify the reaction on different updates to emp and dept) has to be taken into account. □

3.2.1 Events

In the context of active rules, an *event* is a noteworthy happening which occurs at a certain point of time. Thus, the occurrence of events can be modeled as a function

$$ocd: \; E \times T \to \{true, false\}$$

which assigns to every event symbol $e \in E$ and every instant t from a temporal domain T a boolean value indicating whether e has occurred at t, or not. The *event history* up to some time point t_0 (e.g., the current instant) is then simply given by $ocd|_{t \leq t_0}$.

Primitive Events

In active databases, primitive events are usually classified as follows:

- *Internal events* (or *database events*) are caused by database operations like retrieval or update of relations (`insert`, `delete`, `modify`), transactional events (e.g., `commit`, `abort`), or procedure calls. In object-oriented systems, internal events may also be raised by method invocations.

- *External events* (or *abstract events*) are events occurring outside the database system (e.g., `mouse-button-hit`, `temperature-exceeds-threshold`, etc.); these are explicitly signaled to the ADB by some application program.

- *Temporal events* are special system-generated events which are scheduled to occur at specific time points. Temporal events may be *absolute* (**on** `Jun 6 10:10:25 MET DST 1997`), *relative* (**on** `now + 1 hour`), or *periodic* (**on** `now every 2 days`).

Composite Events

Often, the situations in which active rules should be triggered are rather complex and may not be described solely by individual occurrences of primitive events like the ones given above. In this case, an *event algebra* can be used to specify the occurrence of *composite events*. The following is a list of some frequently used constructors for composite events in active databases (based on [CKAK94]):

- The *disjunction* $(E_1 \mid E_2)$ occurs when E_1 or E_2 occurs.

- The *sequence* $(E_1 \,; E_2)$ occurs when E_2 occurs provided E_1 has already occurred.

- The *conjunction* $(E_1 \,, E_2)$ occurs when both E_1 and E_2 occur, irrespective of the order of occurrence. Thus, $(E_1 \,, E_2) :\Leftrightarrow (E_1 \,; E_2) \mid (E_2 \,; E_1)$.

- The *simultaneous conjunction* $(E_1 \& E_2)$ occurs when both E_1 and E_2 occur simultaneously.

- The *within event*[1] $(E_2 \in [E_1; E_3])$ occurs whenever E_2 occurs within the interval marked by the occurrences of E_1 and E_3. The *cumulative* version $E_2 \in^* [E_1; E_3]$ occurs only once when E_3 occurs, provided E_2 has occurred within $[E_1; E_3]$.

- The *negation* $(\textbf{not}\ E_2 \in [E_1; E_3])$ occurs whenever E_3 occurs, provided E_2 does not occur within the interval $[E_1; E_3]$.

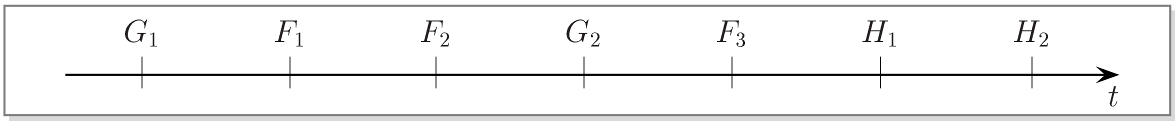
In practice, one often needs *parameterized events*: For example, a database event like `insert-into-`R has to supply the attribute values of the tuple which is inserted into relation R. Similarly, an external event like `temperature-exceeds-threshold` may be parameterized with the current value of the temperature and the timestamp when this value was observed. The parameters supplied by the event may then be referenced in the rule's condition and action.

[1]Called *aperiodic event* $\mathsf{A}(E_1, E_2, E_3)$ in [CKAK94] (to contrast it with periodic events).

Event Consumption Modes

A question arising from the use of composite events is which of the constituent events "take part" in the composite event, and how they are "consumed" by the composite event. This *event consumption policy* is elaborated using so-called *parameter contexts*, which were introduced for the SNOOP algebra in [CKAK94, CM94].

Example 3.2 (Composite Event)
Consider the composite event $E := ((F\,,G)\,;H)$, which occurs if H occurs after both F and G have occurred. Suppose the following event history is given:



Here, the F_j's denote several occurrences of the same primitive event F, similarly for G_k and H_l. Using a so-called *unrestricted* parameter context—as we have implicitly assumed above—there are twelve occurrences E_i of the composite event E, comprising all possible combinations of F_j, G_k, and H_l which make E occur, i.e., $E_i \in F_j \times G_k \times H_l$. Thus, constituent events are not consumed, but are reused arbitrarily many times. □

Alternative parameter contexts are motivated by applications where constituent events should be consumed by the composite event in a certain way. Apart from the unrestricted context, the following parameter contexts have been proposed [CKAK94]:

- *Recent:* In this context, only the most recent occurrences of constituent events are used; all previous occurrences are lost. In Example 3.2, E will be raised twice: for the constituent events $\{G_2, F_3, H_1\}$ and for $\{G_2, F_3, H_2\}$. Alternatively, one can define a slightly different semantics, which consumes the constituent events immediately when the composite event is detected. In this case, only $\{G_2, F_3, H_1\}$ is reported.

- *Chronicle (*or *chronological):* In this context, events are consumed in their chronological order. In a sense, this corresponds to a first-in-first-out strategy. In Example 3.2, E will be reported for $\{G_1, F_1, H_1\}$ and for $\{F_2, G_2, H_2\}$.

- *Continuous:* In this context, each event which may *initiate* a composite event starts a new instance of the composite event. A constituent event can be shared by several simultaneous occurrences of the composite event. In the example, each G_i and each F_j starts a new instance. Thus, the composite occurrences $\{G_1, F_1, H_1\}$, $\{F_1, G_2, H_1\}$, $\{F_2, G_2, H_1\}$, and $\{G_2, F_3, H_1\}$ are reported. The composite event initiated at F_3 is still to be completed.

- *Cumulative:* In this context, all occurrences of constituent events are accumulated until (and consumed when) the composite event is detected. In the example, E is raised once for the constituent events $\{G_1, F_1, F_2, G_2, F_3, H_1\}$.

3.2.2 Conditions

If the event of an ECA rule has been detected, the condition has to be evaluated. In general, conditions are queries executed against the current database state and thus may be formulated as SQL statements of the form `SELECT ... FROM ... WHERE ...`, or in similar languages like the relational algebra or calculus. If the event is parameterized, parameter values may be used in rule conditions. In order to allow for rules which react on changes in the database state, transition conditions need to be specified. This requires the ability to refer to *different database states* (e.g., to $R.\mathsf{old}$ and $R.\mathsf{new}$), or equivalently, to *delta relations* like $R.\mathsf{inserted}$, and $R.\mathsf{deleted}$ as in Example 3.1.

There are several reasons why event and condition part are distinguished in active rules: Conceptually, the event specifies *when* to check the rule, while the condition specifies *what* to check [Day95]. From a practical point of view, the main reason for the distinction is that rule processing can be optimized. In particular, conditions are only evaluated when the specified event occurs.

From a more abstract logical point of view, however, the event may be regarded as a special condition. This viewpoint is used for example in Starburst where events are viewed as changes in the database state [WF90]. The database evolution is modeled as a sequence of *transitions* between states. When rules are triggered, the so-called *net effect* of changes is used in rule conditions, i.e., the combined effect of changes which have been applied [WF90, Wid92]. For example, if a tuple is first modified and deleted afterwards, the modification is not considered. Similarly, if a tuple is inserted and then deleted, the net effect is void.

3.2.3 Actions

The action part of an ECA rule is executed after the rule's event has been detected and the condition has been satisfied. Similar to events, actions may be roughly classified as internal or external:

- *Internal actions* are database operations like updates (`insert`, `delete`, `modify`), or transactional operations (`abort`, `commit`).
- *External actions* are application-specific operations happening outside the database system which may be started through a procedure call of an application program (e.g., `send-mail`, `turn-on-sensor`).

Like the condition part, the action part may refer to values of event parameters. Note that the effect of an internal action may raise new internal events, which in turn may trigger rules, etc., possibly resulting in nontermination of rule execution (cf. Figure 3.1).

3.3 Evaluation of ECA Rules

Arguably the most critical issue in active rules is the precise specification of the overall rule execution semantics. While the meaning of each individual rule is usually easy to grasp, the global semantics of a set of interacting rules is much harder to comprehend.

The basic execution model of active rules is similar to the *recognize-act cycle* of OPS5 [BFKM85], a production rule language developed in the AI community: one or more triggered rules (i.e., whose triggering event and condition are satisfied) are selected and their action is executed. This process is repeated until some termination condition is reached, e.g., until no more rules are triggered, or a fixpoint is reached [WC96a]:

while there are triggered rules do	*% event detection*
1. select a triggered rule r	*% conflict resolution*
2. evaluate r's condition	*% condition evaluation*
3. if this condition is true, execute r's action	*% action execution*

Figure 3.2: Simplified ECA rule processing algorithm

Clearly, this basic algorithm leaves open many issues and there are a lot of possible choices and details which have to be elaborated in order to precisely specify the semantics of rule execution:

Processing Granularity

The rule processing granularity specifies the possible rule invocation points. The finest granularity is when rules may be executed at any time during the ADB's operation. Usually, however, rules may only be invoked after execution of database operations like insertion and deletion of tuples. This raises the question whether rules are executed in a *tuple-oriented* or *set-oriented* way. For example, if a relation R is updated, is a triggered rule invoked for each updated tuple, or only once for the whole delta set?[2] Set-oriented execution conforms more closely to the standard model of querying in relational databases, and is in a sense more "declarative" than tuple-oriented execution. In contrast, tuple-oriented execution adds a degree of nondeterminism to the language, since the outcome may now depend on the order in which individual rule instances are fired. A coarser granularity results if rules are only executed at statement boundaries, e.g., at the beginning or end of an SQL statement. Since in database programming, operations are grouped into transactions, transaction boundary execution yields another very coarse granularity of rule processing.

Conflict Resolution Policy

Another issue is the *conflict resolution policy* in step 1 (Figure 3.2): If several rules are triggered, which of them should be executed and in which order? Again, there are a number of possibilities:

- An arbitrary rule may be chosen. Clearly, this yields another source of nondeterminism, and confluence is in general not guaranteed.
- The rule with the highest priority may be chosen. This requires a set of system-defined or user-defined priorities. Assigning priorities—especially if done in an ad-hoc way—has the disadvantage of adding "non-logical" control to rule processing [ACL91].
- All triggered rules are evaluated simultaneously, using the *union of rule effects* as the result. This policy leads to a deterministic semantics and has the conceptual advantage

[2] Set-oriented updates are sometimes called *bulk-updates*.

that no priorities are assigned, i.e., all triggered rules are considered equally justified. However, this strategy is only applicable if rules are "logical" or "high-level" enough as is the case with deductive rules. In contrast, very procedural, low-level rules do not lend themselves to a natural union semantics.

Coupling Modes

In the database context, rule processing has to be related to the execution of transactions. To this end, *coupling modes* have been introduced in [HLM88] and further developed in the HiPAC project [DBC96]; see also [Buc94]. Coupling modes permit a low-level control over how a rule is processed relative to the transaction that triggered the rule. Suppose some transaction T executes an operation which raises event E, thereby triggering a rule r which executes action A provided condition C holds.

If E-C coupling is ...

- *immediate*, then C is evaluated within the same transaction T immediately after E was detected. The remaining operations of T are suspended until r has been processed.
- *deferred*, then C is still evaluated within T but just before T commits. This mode is useful for checking integrity constraints, since it allows to abort a transaction if the update results in a violation of integrity.
- *decoupled* (or *detached*), then C is evaluated in a separate transaction T'. Note that T' may commit, even if the triggering transaction T is aborted. Since this behavior may lead to the execution of an action whose cause (the triggering transaction) is invisible, the variant *decoupled causally dependent* has been proposed: in this mode, T' can only commit if T commits.

The same modes can be used to specify C-A coupling, i.e., between condition evaluation and action execution. It should be clear that although these coupling modes allow a rather flexible low-level control of rule execution, and may be quite useful in some applications, they also complicate design, understanding, and maintenance of rules.

3.4 Properties of Active Rules

Despite the great variety of execution models, the following properties come up repeatedly and are of practical and theoretical importance. It is assumed that a particular execution model for active rule programs is fixed.

Termination

Since active rules may trigger each other recursively, nontermination of rule processing is a permanent threat. In most languages, termination is undecidable at compile-time, and computationally very hard (if decidable at all) at run-time. In [AWH92, AWH95], conservative analysis techniques have been introduced for the Starburst system which guarantee termination by requiring that the rule *triggering graph* is acyclic. Using "semantic" properties of rules, an improved analysis has been developed in [BCW93a, BW94]. The basic idea is to look "closer" at rules when determining whether some rule r_i may trigger a rule r_j. For example, the rules

$$r_i: \quad \textsf{\textbf{del}:emp(E, \ldots)} \leftarrow \varphi(\textsf{emp}, \ldots)$$
$$r_j: \quad \textsf{\textbf{ins}:emp(E, \ldots)} \leftarrow \psi(\textsf{emp}, \ldots)$$

may trigger each other indefinitely. However, if one can show that the effect of r_i (deleting some employee tuples) together with the condition φ of r_i logically implies that the condition ψ of r_j is false, then r_j may *not* be triggered by r_i, so this edge should not be considered in the triggering graph. Other recent work on termination in active rules includes [BCP95a, BCP95b], [KU96], and [VGD97].

Confluence

A rule set is called confluent if—under the given semantics—there is at most one final state for a given database and rule program. In general, confluence is a highly desirable feature, since the behavior of rules is easier to grasp if there is a unique final result. Obviously, if the rule semantics is deterministic, confluence is guaranteed. For rules with nondeterministic semantics, confluence is usually undecidable and one must confine to sufficient criteria like those developed in [AWH95] for the Starburst rule semantics. On the other hand, in some contexts, nondeterminism is a desired feature increasing the expressive power of the language [GGSZ97].

Expressive Power and Complexity

Recall that the expressive power of a database language is the class of database transformations expressible in it (Section 2.2). While concrete ADBs often exhibit the full computational power of Turing machines (e.g., via procedure calls to a host language or a sublanguage), it is still interesting to investigate the impact of certain language constructs on expressive power. In particular, one can study the expressiveness of the "pure" language fragments excluding calls to the host language, arithmetics, etc., or the expressiveness of the sublanguages defining events, conditions, and actions, respectively.

The *complexity* of evaluating a set of rules in a certain language measures the computational cost involved in determining the final result. Not surprisingly, there is a trade-off between expressive power and complexity.

Expressive power and complexity will be studied in more detail in Chapters 4 and 6 for production rules and Statelog, respectively.

3.5 Summary

Active rules are a powerful and flexible paradigm for the specification of (re)active behavior. The main application areas can be roughly classified as (i) within the database system, to enhance "traditional" functionality (e.g., automatic integrity enforcement), and (ii) for non-standard applications with active requirements like monitoring applications or workflow management. There are numerous proposals for active rule languages and execution models including complex events and elaborated E-C and C-A coupling modes. On the downside, the semantics of active rules can become very complex and hard to understand and predict, especially, if the semantics is not formally defined. Indeed, unless simpler semantic frameworks are available *"... rule designers will have to be four-star wizards in order to ensure that the* *rule system does what they actually want"* [Wid94]. In contrast, production rules and deductive rules further to the left of the spectrum in Figure 1.1 have a precisely specified semantics based on first-order logic. Since these languages are more "high-level", it is usually easier to understand their behavior and to reason about properties of rules.

Bibliographic Notes

Active Database Systems

Early precursors of what is now referred to as ADBs include the `ON` ⟨*command-list*⟩ `CALL` ⟨*procedure*⟩ clauses of CODASYL as defined in the early seventies, the suggested *trigger subsystem* for *System R* [Esw76], and the AI production rule language OPS5 [BFKM85].

Nowadays, there are a number of ADB prototypes which are either based on the relational or an object-oriented model. Among the relational systems, Postgres [SK91], Starburst [WF90, Wid96], and Ariel [Han92, Han96] belong to the most prominent systems; A-RDL [SK96] is particularly interesting due to its close relationship to deductive databases. Other relational systems are AP5 [Coh89] and Heraclitus [GHJ+93, GHJ96]. The latter elevates delta relations to "first-class citizens" of the rule language.

The active object-oriented system HiPAC [DBB+88, DBC96] has been very influential on subsequent research by establishing the now widely accepted ECA-rule paradigm. Additionally, HiPAC developed concepts for time-constrained rules and introduced a number of sophisticated coupling modes. Follow-on projects are Sentinel [AMC93] (with its powerful event specification language SNOOP [CKAK94]) and REACH [BBKZ93]. Other object-oriented systems are ADAM/EXACT [DJP93], Chimera [CFPT96], ODE [GJS92], NAOS [CCS94], and SAMOS [GD93].

Commercial systems are now beginning to include ideas and concepts from active databases, e.g., triggers for maintenance of referential integrity [CW96]. Moreover, such triggers are also considered in the current SQL3 proposal [ISO97].

Further Reading

A good starting point for further reading is [WC96a] which contains a nice introduction to active rules [WC96b] and describes the essentials of a number of prototypes. [DHW95] is another introductory text, [Cha92] contains a special issue on active databases. [PDW+93] discusses dimensions of active behavior (such as structure and execution model of active rules) which allow to examine and classify ADBs according to their distinctive features. [FT95] contains another classification of ADBs. The different possible options in rule behavior are encoded using user-readable *Extended ECA* rules which are then translated into an internal core language. [PCFW95] surveys work on formal specification of active database functionality. [DGG95] presents the *active database management system manifesto*. In [Day95], a survey on the accomplishments of research in active databases is given. The workshops [WC94] and [BH95] were dedicated particularly to active rules; the workshop series [PW93, Sel95, GB97] also has major sections on active rules.

4 Production Rules

4.1 Introduction

From the viewpoint of active databases, production rules can be regarded as special ECA rules without the event part, i.e., as *condition-action rules* of the form

$$\textbf{if } \langle \textit{condition} \rangle \textbf{ then } \langle \textit{action} \rangle.$$

However, there is no clear-cut borderline between active and production rules in databases, and sometimes these terms are even used interchangeably. From a more abstract point of view, one can regard ECA rules also as production rules since the event detection part can be encoded in the condition as is illustrated in Example 4.1 below. Although such an encoding may be less efficient in an implementation, this abstraction is very useful as it allows to apply techniques and results developed for production rules to the context of active rules.

Example 4.1 (Integrity Enforcement, Cont'd)
In Example 3.1, active rules are used to enforce two inclusion dependencies on the relations emp(EmpNo, Salary, DeptNo) and dept(DeptNo, MgrNo). The following production rules accomplish the same task. However, in contrast to the informally specified semantics before, their semantics will be precisely defined by the fixpoint semantics of Datalog$^{\neg\neg}$ below.

del_emp(E,Sal,D) $\leftarrow$ $\triangleright$del_emp(E), emp(E,Sal,D).	*% detect external events*
$\neg$ emp(E,Sal,D) $\leftarrow$ del_emp(E,Sal,D)	*% interpret delete requests*
$\neg$ dept(D,E) $\leftarrow$ del_dept(D,E).	*% ... as deletions*
del_dept(D,E), del_emp(E1,S1,D) $\leftarrow$ del_emp(E,_,_), dept(D,E), emp(E1,S1,D).	

Here, $\triangleright$del_emp is a (auxiliary) relation for storing *external* delete requests. In this way, occurrences of external events are mapped to database events which can then be used in rule conditions as in the first rule. del_emp and del_dept denote *delta relations* used to store delete requests. Every rule body denotes a condition to be evaluated on the current database state. In contrast, rule heads correspond to (database) actions, in this case to insertions into delta relations (first and last rule) and to deletions from emp and dept (second and third rule). While the first three rules are typically system-defined, the last rule specifies the actual update propagation as given by the user and corresponds to the ECA rule in Example 3.1. □

A characteristic feature of production rule semantics is the *forward chaining* execution model: The conditions of all rules are matched against the current state. From the set of triggered rules (the *candidate set*) one ore more rules are selected and the corresponding actions are executed. This process is repeated until there are no more changes or no more triggered rules. Thus, the basic algorithm for evaluating production rules is very similar to the one given for active rules in Figure 3.2. Note, however, that production rules are conceptually much simpler than ECA rules, since they usually do not allow complex events, or coupling modes other than immediate or deferred triggering.

In the database community, forward chaining or *fixpoint* semantics have been studied extensively for a number of Datalog variants [AV88, AV91], thereby providing a logic-oriented formalization of production rule semantics.

In a sense, the Datalog fixpoint semantics are the "pure" database version of forward chaining production rules and will be investigated in the sequel. Other, more declarative semantics for Datalog (and more generally, logic programs) have been developed by the deductive databases and logic programming communities and will be presented in Chapter 5; these semantics behave quite differently from the more procedural fixpoint semantics below.

4.2 Datalog with Production Rule Semantics

4.2.1 Syntax

Definition 4.1 (Datalog, Datalog$^\neg$, Datalog$^{\neg\neg}$) A *Datalog$^\neg$ rule* r is an expression of the form

$$H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m$$

where the *head* H and all B_i, C_j in the *body* of r are first-order atoms. In the body, we also allow equality and inequality atoms of the form $X = Y$ and $X \neq Y$, respectively. If $m = 0$, then r is a *Datalog rule*. If $m = n = 0$, then r is called a *fact*. A *Datalog$^{\neg\neg}$ rule* is like a Datalog$^\neg$ rule, but the head H may also be a negated atom $\neg A$. A *Datalog$^\neg$ program* P is a finite set of Datalog$^\neg$ rules; analogously for Datalog and Datalog$^{\neg\neg}$.

A rule with multiple heads, i.e., of the form

$$H_1, \ldots, H_k \leftarrow \mathit{body}$$

is viewed as a shorthand notation for the set of rules $\{H_1 \leftarrow \mathit{body}\ ;\ \ldots\ ;\ H_k \leftarrow \mathit{body}\}$. □

Let P be a Datalog[1] program. The *schema* $\mathbf{P}$ of P is the set of all relation symbols occurring in P. An *extensional relation* is a relation occurring only in rule bodies of P; an *intensional relation* is one that occurs in some rule head of P. The *extensional schema* $\mathit{edb}(\mathbf{P})$ comprises all extensional relations, the *intensional schema* $\mathit{idb}(\mathbf{P})$ consists of all intensional relations of P.

4.2.2 Semantics

The production rule semantics of Datalog programs is intuitively very simple: Given some initial database $\mathcal{D}$, all rules of the program P are fired simultaneously (i.e., set-oriented and in parallel) until a fixpoint is reached. Atoms in the head of rules are interpreted as insertions whereas—in the case of Datalog$^{\neg\neg}$—negated atoms in the head are interpreted as deletions. The semantics of P and $\mathcal{D}$ is then simply the fixpoint, if it exists, or $\perp$ (undefined) otherwise.

[1] If clear from the context, like here, we use *Datalog* as a catch-all for Datalog, Datalog$^\neg$, and Datalog$^{\neg\neg}$.

Immediate Consequence Operator

For notational convenience, we assume in the sequel that the input database $\mathcal{D}$ is added to a given program P' as a set of facts, so $P = P' \cup \mathcal{D}$, unless stated otherwise. By $ground(P)$ we denote the set of ground instances of rules of P that can be constructed using the constants occurring in P. The operator T_P maps a given database instance $\mathcal{I}$ to its *immediate consequences* under P:[2]

$$T_P(\mathcal{I}) := \{head(r) \mid \mathcal{I} \models body(r), r \in ground(P)\} \ .$$

If $T_P(\mathcal{I}) = \mathcal{I}$ then $\mathcal{I}$ is a *fixpoint* of T_P. In general, a program may have zero, one, or more fixpoints.

Every program P $(= P' \cup \mathcal{D})$ induces a sequence of database instances $T_P^0, T_P^1, \ldots$ (called the *stages* of the fixpoint computation) by setting

$$T_P^0 := \emptyset \ , \qquad \text{and} \qquad T_P^{i+1} := T_P(T_P^i) \ . \tag{4.1}$$

If $T_P^0 \subseteq T_P^1 \subseteq \cdots$ then the operator T_P is called *inductive*, and—since we are dealing with finite structures—reaches a fixpoint $T_P^\omega = \bigcup_{i \geq 0} T_P^i$ after finitely many steps, i.e.,

$$T_P^\omega := T_P^n = T_P^{n+1} = \cdots$$

for some $n \in \mathbb{N}_0$. For programs involving negation, T_P^ω is generally *not* the least fixpoint, even if the sequence is inductive (see P'_{ntc} in Example 4.7 below). In case the sequence is not inductive, the fixpoint T_P^ω may not even exist. We consider the above Datalog variants in turn.

Datalog

If P is a Datalog program (hence, P contains no negated atoms) then T_P is *monotone*, i.e., $\mathcal{I} \subseteq \mathcal{J}$ implies $T_P(\mathcal{I}) \subseteq T_P(\mathcal{J})$. This in turn implies that T_P is inductive and guarantees the existence of the least fixpoint.[3]

Example 4.2 (Transitive Closure) A prototypical example for Datalog is the transitive closure query, mapping a given graph over the schema $\{\mathsf{e}\}$ to its transitive closure:

P_{tc} :

```
tc(X,Y) ← e(X,Y).
tc(X,Y) ← e(X,Z), tc(Z,Y).
```

Here, $edb(\mathbf{P}_{tc}) = \{\mathsf{e}\}$ and $idb(\mathbf{P}_{tc}) = \{\mathsf{tc}\}$. Note that this query requires recursion (or equivalently, iteration) in an essential way, and is not expressible in relational algebra or relational calculus. □

[2] The use of $ground(P)$ implies that rules are evaluated wrt. to the *active domain* (Section 2.2).

[3] See, e.g., [EF95, p. 167].

I-Datalog (*Inflationary Datalog*$^\neg$)

Datalog$^\neg$ programs allow negation in rule bodies. In order to guarantee the existence of a fixpoint, the following inflationary version T_P^+ of the T_P operator has been proposed:

$$T_P^+(\mathcal{I}) := \mathcal{I} \cup T_P(\mathcal{I}) \ .$$

Clearly, the set of inferred facts through consecutive applications of T_P^+ is increasing, i.e., $\mathcal{I} \subseteq T_P^+(\mathcal{I})$, which means that T_P^+ is *inflationary*. Since every inflationary operator is also inductive, convergence to the fixpoint is guaranteed.

Note that the effect of T_P^+ can be obtained using the standard T_P operator by adding a rule

$$p(X_1, \ldots, X_k) \ \leftarrow \ p(X_1, \ldots, X_k)$$

to the program, for every k-ary relation $p \in idb(\mathbf{P})$.

P-Datalog (*Partial* or *Noninflationary Datalog*$^\neg$)

When applied to arbitrary Datalog$^\neg$ programs, the original T_P operator is no longer inductive (and also not monotone) leading to a *noninflationary* semantics. Consider, for example, the rule $p(X) \leftarrow q(X), \neg\, r(X)$. Clearly, if new information is derived for $r(X)$, this may prevent the derivation of tuples for $p(X)$. In contrast to Datalog and I-Datalog, the P-Datalog[4] semantics is *partial* because the sequence $(T_P^n)_{n\in\mathbb{N}_0}$ may no longer reach a fixpoint:

Example 4.3 (Oscillation) The rule $p(X) \ \leftarrow \ q(X), \neg\, p(X)$ together with the database $\mathcal{D} = \{q(a)\}$ gives rise to sequence

$$\emptyset,\ \{q(a)\},\ \{q(a), p(a)\},\ \{q(a)\},\ \{q(a), p(a)\},\ \ldots$$

eventually oscillating between $\{q(a)\}$ and $\{q(a), p(a)\}$. □

Datalog$^{\neg\neg}$

The previous Datalog variants only allowed positive atoms in rule heads. Thus, viewed as condition-action rules, the only possible action is the insertion of tuples. In contrast, Datalog$^{\neg\neg}$ allows negated atoms in rule heads which are interpreted as deletions. More precisely, the fixpoint semantics of Datalog$^{\neg\neg}$ programs is defined by iterating an operator $T_P^{\neg\neg}$. Like above, where T_P is applied to Datalog$^\neg$ programs, $T_P^{\neg\neg}$ is not inductive, hence, also the Datalog$^{\neg\neg}$ semantics is noninflationary. Indeed, both languages have the same expressive power (see Figure 4.2).

First, immediate insert and delete "requests" are defined as follows:

$$\begin{aligned} T_P^{ins}(\mathcal{I}) \ &:= \{H \mid \mathcal{I} \models body, (H \leftarrow body) \in ground(P))\} \\ T_P^{del}(\mathcal{I}) \ &:= \{H \mid \mathcal{I} \models body, (\neg\, H \leftarrow body) \in ground(P))\}. \end{aligned}$$

Intuitively, tuples derived by T_P^{ins} or T_P^{del} should be inserted or deleted, respectively, in the next stage. In case of a conflict, i.e., if there is an atom $H \in T_P^{ins} \cap T_P^{del}$, several options exist. For example, one can define $T_P^{\neg\neg}$ as

$$T_P^{\neg\neg}(\mathcal{I}) \ := \ (\mathcal{I} \cup T_P^{ins}(\mathcal{I})) \setminus T_P^{del}(\mathcal{I}) \ ,$$

[4] The names I-Datalog and P-Datalog are adopted from [EF95].

$\mathcal{D}$:

emp

EmpNo	*Salary*	*DeptNo*
ed1	7000	d1
md1	10000	d1
md2	1000	d1
ed2	6000	d2
md2	8000	d2
md3	9000	d3

dept

DeptNo	*MgrNo*
d1	md1
d2	md2
d3	md3

⋮

$\mathcal{D}'$:

emp

EmpNo	*Salary*	*DeptNo*
md3	9000	d3

dept

DeptNo	*MgrNo*
d3	md3

Figure 4.1: Initial and final state for Example 4.4

in which case priority is given to deletions. In contrast, if $T_P^{\neg\neg}$ is defined as

$$T_P^{\neg\neg}(\mathcal{I}) \quad := \quad (\mathcal{I} \setminus T_P^{del}(\mathcal{I})) \cup T_P^{ins}(\mathcal{I}) \ ,$$

then priority is assigned to insertions. Further possibilities will be discussed in the context of Statelog in Section 7.4.2.

Example 4.4 (Integrity Enforcement, Cont'd) The rules of Example 4.1 constitute a valid Datalog$^{\neg\neg}$ program. Assume an initial database $\mathcal{D}$ as depicted in Figure 4.1. Additionally, there is an external request to delete the employee md1, modeled by adding the fact ▷del_emp(md1) to $\mathcal{D}$. Consider the rules of Example 4.1: By the first rule, the request del_emp(md1,10000,d1) is derived. Next, requests to delete dept(d1,md1) and all employees working for d1 are derived by the last rule. Since md2—the manager of d2—also works for d1, the last rule is triggered again, resulting in the deletion of d2 and all employees working there. Eventually, the final database state $\mathcal{D}'$ is reached. Note that the actual deletions are accomplished by rules with negative heads, i.e., the second and third rule in Example 3.1. □

Thus, the Datalog$^{\neg\neg}$ semantics has finally lead to a precisely defined semantics for our introductory Example 3.1.

4.3 Properties of Production Rules

Theoretical properties like expressive power, complexity, and termination behavior of various Datalog variants with fixpoint semantics have been studied in-depth. In this section, we briefly review the main results in order to classify and relate the corresponding results for Statelog developed in Chapter 6.

4.3.1 Termination

Rule termination is a property of the given rules together with the chosen execution model. In the case of bottom-up fixpoint semantics, termination corresponds to the existence of a fixpoint.

More precisely, given a database $\mathcal{D}$ and a program P, we say that P *terminates for* $\mathcal{D}$ if the sequence obtained by iterating the corresponding fixpoint operator reaches a fixpoint. We say that P *always terminates* if the fixpoint of $P \cup \mathcal{D}$ exists for *all* databases $\mathcal{D}$. While Datalog and I-Datalog programs always terminate, this is not the case for the noninflationary languages P-Datalog and Datalog$^{\neg\neg}$. Indeed, deciding termination for a *given* database $\mathcal{D}$ is **PSPACE**-complete wrt. the size of $\mathcal{D}$. Moreover, the class of P-Datalog programs which *always* terminates is undecidable. In other words, detecting nontermination at run-time (i.e., for a given database) may be very expensive, and impossible at compile-time (since there is no algorithm for the latter). However, one can find for every (possibly nonterminating) program P an equivalent program $P^{\downarrow}$ which always terminates. Note that equivalence here means: both programs define the same database transformation, *provided* P *terminates*. Whenever $P \cup \mathcal{D}$ does not terminate, $P^{\downarrow}$ yields a special result, say $\mathcal{D}' = \emptyset$ (see Section 6.9 for details on termination and the corresponding results for Statelog).

4.3.2 Expressive Power and Complexity

Recall from Section 2.2 that the expressive power of a language L is the class of database transformations expressible in L.

Remark on Expressiveness: Queries vs. Updates

There is a subtle difference whether one considers db-transformations as *queries*, i.e., as mappings $inst(\mathbf{R}) \rightarrow inst(\mathbf{S})$ between instances of disjoint schemas $\mathbf{R}$ and $\mathbf{S}$, or as *updates*, i.e., as mappings $inst(\mathbf{R}) \rightarrow inst(\mathbf{R})$:[5]

Example 4.5 (Alternative Definition of P-Datalog)
In the above definition of P-Datalog (p. 24), we have assumed that the program P includes a given database $\mathcal{D}$ as a set of facts. Consequently, tuples in $\mathcal{D}$ can never be removed. In order to account for deletions, P and $\mathcal{D}$ have to be distinguished, so in this case, $\mathcal{D}$ is *not* a part of P. Moreover, in contrast to equation (4.1), the initialization of the fixpoint computation for P-Datalog is given by $T_P^0 := \mathcal{D}$.

With these modifications, the tuples in $\mathcal{D}$ are not automatically rederived at every stage and deletions can be expressed. For example, by using a rule

$$p(\bar{X}) \leftarrow p(\bar{X}), \neg\, del\_p(\bar{X}) \ ,$$

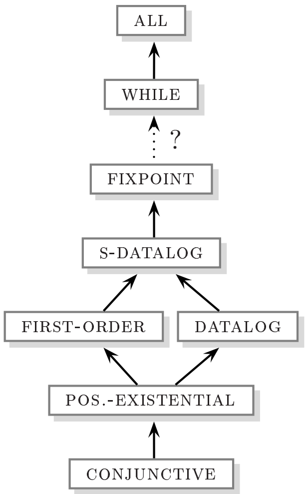
only those tuples are transferred to the next stage, which are not contained in the relation *del_p* (*del_p* may be part of the initial database or defined by other rules). □

Since the original definition did not allow to express deletions, the second variant is more expressive wrt. *updates*. On the other hand, it is easy to see that both P-Datalog variants have the same expressive power wrt. *queries*. For another example, consider the following db-transformation [Don88]:

[5] The author thanks Victor Vianu for clarifying discussions on this point [Via97a].

Example 4.6 (Symmetric Transitive Closure)
Given a binary relation e, compute $sym(tc(e))$, i.e., the transitive closure followed by the symmetric closure. It is straightforward to define this as a Datalog *query* using the usual rules for tc plus the rules $\{q(X,Y) \leftarrow tc(X,Y);\ q(X,Y) \leftarrow tc(Y,X)\}$. However, the same mapping cannot be expressed as an *update* to e, as shown in [Don88]. For example, by simply adding the rule $e(X,Y) \leftarrow q(X,Y)$ the output of q is "fed back" into e, resulting in an answer which is too large.[6] □

Note that the difference disappears for languages which can express some form of "assignment" like Datalog$^{\neg\neg}$, *While*, or Statelog: by "assigning" the result of a query to the input relations (without recomputing the query on the updated data!), an equivalent update is obtained. In the sequel, we do not elaborate on this technicality but adopt the (rather standard) point of view that the expressive power of a language is the class of mappings from instances of a given input schema $\mathbf{R}$ to instances of a disjoint output schema $\mathbf{S}$. In fact, when comparing the expressiveness of languages, it is common to restrict even further and consider only boolean queries, i.e., mappings $inst(\mathbf{R}) \rightarrow \{yes, no\}$.



Class	Languages	Data Complexity
CONJUNCTIVE	FO($\exists, \wedge$), RA($\sigma, \pi, \bowtie$)	$\subset$ **LOGSPACE**
POSITIVE-EXISTENTIAL	FO($\exists, \wedge, \vee$), RA($\sigma, \pi, \bowtie, \cup$)	$\subset$ **LOGSPACE**
FIRST-ORDER	FO, RA	$\subset$ **LOGSPACE**
DATALOG	Datalog	**PTIME**
S-DATALOG	S-Datalog	**PTIME**
FIXPOINT	FO(IFP), I-Datalog, FO(LFP), WF-Datalog	**PTIME**
WHILE	FO(PFP), P-Datalog, *While*, Datalog$^{\neg\neg}$	**PSPACE**
ALL	$While_{new}$	no bound

Figure 4.2: Database transformations: main classes and languages

Main Classes of Database Transformations

On the left of Figure 4.2 (adapted from similar figures in [AHV95]) the main classes of database transformations are depicted. A solid arrow $C_1 \rightarrow C_2$ indicates strict inclusion (i.e., C_1 is strictly less expressive than C_2; cf. Section 2.2). The inclusion for the dotted arrow is strict iff **PTIME** $\neq$ **PSPACE** [AV95]. In the table to the right of Figure 4.2, the most prominent languages capturing these classes together with their complexities are given.

Note that the data complexity, say for Datalog, only states that there are **PTIME**-complete problems which can be expressed in Datalog.[7] However, Datalog does not express all of DB-**PTIME** (cf. Section 2.2). If an order on the underlying domain is available, expressiveness and

[6]For e = "$\bullet \leftarrow \bullet \rightarrow \bullet$" we get "$\bullet \leftrightarrow \bullet \leftrightarrow \bullet$" instead of "$\bullet \leftrightarrow \bullet \leftrightarrow \bullet$".
[7]See Example 6.8 for such a problem.

data complexity are usually more balanced. For example, I-Datalog and P-Datalog express all of DB-**PTIME** and DB-**PSPACE** on ordered databases.

In Figure 4.2, FO and RA denote first-order calculus and relational algebra, respectively; arguments in parentheses denote restrictions to the corresponding operations. S-Datalog and WF-Datalog denote the stratified and well-founded semantics for Datalog$^\neg$ programs, respectively, and will be presented in Chapter 5. The class FIXPOINT can be obtained by the fixpoint extensions of first-order logic FO(IFP) (inductive fixpoint logic) and FO(LFP) (least fixpoint logic), which are well-known in finite mode theory [EF95]. Datalog$^\neg$ with inflationary or well-founded semantics also expresses FIXPOINT. The class WHILE is obtained by FO(PFP) (partial fixpoint logic), or the "procedural" language *While*, which extends the relational algebra RA by while-loops, assignment, and composition. In addition, While_{new} allows the invention of new values during the computation; see [AHV95] for the details. The hierarchy of classes depicted in Figure 4.2 is meant to illustrate only the most prominent classes, which are particularly important in the database context. For example, the class of languages S-Datalog$_k$ allowing at most k strata (cf. Section 5.3) form a strict hierarchy within S-Datalog. In particular, in S-Datalog$_1$, only *edb*-relations may occur negated; this class is also known as *semipositive Datalog* [AHV95] and *existential least fixpoint logic* $\exists$LFP [EF95].

4.3.3 Handling of Negation

The fixpoint semantics for the Datalog languages may yield unexpected and unintuitive results when negation in rule bodies is involved. Recall that under the fixpoint semantics, the condition in the rule body is evaluated wrt. the current stage. A rule $H \leftarrow \ldots, \neg A, \ldots$ fires and infers H, if A is not contained in the current stage $\mathcal{I}_n$ and the remaining literals of the rule condition are satisfied. However, if at a later step A *is* derived, the previously inferred atom H is *not* removed, possibly leading to unintended results.

Example 4.7 (Complement of Transitive Closure)
Consider the following Datalog$^\neg$ program

P_{ntc}:

r_1: $\mathsf{tc(X,Y)} \leftarrow \mathsf{e(X,Y)}.$
r_2: $\mathsf{tc(X,Y)} \leftarrow \mathsf{e(X,Z),\ tc(Z,Y)}.$
r_3: $\mathsf{ntc(X,Y)} \leftarrow \neg\, \mathsf{tc(X,Y)}.$

Note that, by definition, unsafe rules like r_3 are evaluated wrt. the active domain.[8] r_1 and r_2 compute in tc the transitive closure of e. Intuitively, one would expect that the complement of tc is computed in ntc by r_3. However, under the inflationary semantics, r_3 is applied "too early", i.e., before the transitive closure has been computed completely: Initially, tc is empty, so ntc contains all tuples (x, y) where x, y are constants from the active domain. Since no tuples are removed by the inflationary semantics, these "unjustified" tuples remain in the result.[9]

In contrast, under the noninflationary P-Datalog semantics, the desired result is obtained. However, by adding the seemingly harmless rule $\mathsf{ntc(X,Y)} \leftarrow \mathsf{ntc(X,Y)}$ to P_{ntc}, we obtain a program whose P-Datalog semantics coincides with the (unintended) I-Datalog semantics above. □

[8] Here, we could add the goals adom(X), adom(Y) to r_3 thereby obtaining a safe rule (adom is defined by adom(X)← e(X,_) and adom(X)← e(_,X)); cf. Section 2.2.

[9] The intended query can be expressed in I-Datalog using a somewhat involved delaying technique. The resulting program is less elegant and readable compared to P_{ntc}; see [AHV95, Ex. 14.4.2].

This somewhat surprising behavior results from the procedural bottom-up execution model of the presented fixpoint semantics. The focus of these semantics is not the computation of a “justified” or declarative static model of a set of rules. Instead, rules are interpreted “dynamically”, i.e., as condition-action rules in the style of production systems.

In contrast, the semantics developed by the logic programming and deductive databases communities do not deal with defining state-changing operations but focus on finding the “intended” or declarative model of rules with negation.

4.4 Summary

Production rules have a clear and conceptually simpler semantics than ECA rules. Since events can be encoded as part of rule conditions, active behavior of rules can be accomplished by production rules, although more elaborated concepts like coupling modes with nested transactions or complex event detection are not supported. The Datalog languages with fixpoint semantics may be conceived as pure database versions of production rules and allow to study fundamental properties like expressiveness, normal forms, termination and complexity of rules. Viewed as a query language, production rules extend the expressive power of traditional query languages like RA or SQL. A problem not addressed by these languages is the declarative handling of negation in rule bodies. Moreover, there is no conceptual framework incorporating complex events, delta relations, or the distinction between passive query rules, and (re)active update rules; such a framework will be developed for Statelog in Chapter 7.

Bibliographic Notes

The term *production rules* has its roots in the context of formal languages, where the rules defining a language are called productions. In AI parlance, a *production system* is a forward chaining rule-based system conisting of (i) a set of rules, (ii) a working memory, and (iii) an inference engine [BL96a]. The rule antecedent (the condition) consists of a conjunction of literals, the consequent (the action) is a conjunction of atoms of the form $del(A)$ or $add(A)$ for an atom A. A well-known early production system is OPS5 [BFKM85], which has been used, for example, to build industrial applications like XCON and XSEL at DEC [McD82], [BL96a].

In the database context, the term production rule usually refers to some kind of Datalog-like language with bottom-up (or fixpoint) semantics. The theoretical properties of such Datalog variants have been studied in-depth, e.g., in [AV91]. The inflationary semantics for Datalog$^\neg$ programs was first proposed in [AV88] and [KP88]. [AS91] study fundamental properties like termination of deterministic and nondeterministic Datalog extensions. In particular, they show how to obtain for every P-Datalog program an equivalent one which always terminates.

Early surveys on production rules in databases include [HW93] and [Bay93]. In [Via97b], a more recent survey is presented which nicely contrasts the production system (fixpoint) approach with the logic programming (model-theoretic) approach to Datalog semantics. [AHV95] is a very good source for learning more about expressiveness and complexity of database languages. [EF95] includes detailed results on the expressive power and complexity of fixpoint logics and Datalog with fixpoint semantics.

Several database prototypes have been implemented which are based on production rule semantics. As mentioned before, the borderline between production rules and ECA rules is not clear-cut. Among the well-known systems with production rules are RDL1 [dMS88b, dMS88a], its "active" successor A-RDL [SK96], Ariel [Han92, Han96], and Starburst [WF90, Wid96]. One of the already "classical" applications of production rules is the maintenance of views and integrity constraints [CW90, CW91, CFPT94].

5 Deductive Rules

5.1 Introduction

A characteristic feature of active and production rules is their procedural forward chaining (bottom-up) semantics. Such a "data-driven" semantics seems to be well-suited in order to specify reactive behavior—most notably updates—in response to previous changes in the database. The focus of deductive rules is rather different: Deductive rules are used to specify, in a concise and declarative way, knowledge about the current state of the modeled world. From the database perspective, deductive rules constitute a uniform representation for queries and static integrity constraints but do not directly lend themselves to the specification of updates or dynamic constraints. As in the case of production rules, the expressive power is increased compared to standard relational languages by the possibility to define recursive concepts.

In the sequel, we briefly review the main declarative semantics developed for deductive databases and logic programs. Since we are dealing with finite relational databases, we focus on Datalog$^\neg$ programs, although the presented semantics extend naturally to logic programs with function symbols.[1]

5.2 Minimal Models and Positive Programs

The first step towards a declarative semantics of rules is to view them as a set of universally quantified first-order sentences. This allows to speak about models of programs. In logic programming and deductive databases, one is not interested in arbitrary models, or truth in all models as in classical logic, but in certain "intended" Herbrand models—in the ideal case, there is one distinguished or *canonical* model.

Let P be a logic program. A *Herbrand model* $\mathcal{I} \subseteq \mathcal{B}_P$ of P is a Herbrand interpretation which makes true all rules of $ground(P)$; it is called *minimal* if there is no other such model $\mathcal{I}' \subseteq \mathcal{I}$. While programs involving negation may have several minimal models, it is easy to show that every positive logic program P has a *unique* minimal model $\mathcal{M}_P$ (also called the *least Herbrand model*) defined as the intersection of all Herbrand models of P. It is well known that for positive programs, the minimal model can also be obtained as the least fixpoint of T_P, i.e., $\mathcal{M}_P = T_P^\omega$. Thus, in the special case of positive programs, the production rule semantics and the declarative semantics of rules coincide. However, this is no longer the case when it comes to programs with negation.

[1]A *logic program* is defined similar to a Datalog$^\neg$ program, but terms may also include function symbols. Thus, for logic programs, Herbrand universe and Herbrand base are in general infinite.

5.3 Stratified Semantics

Recall Example 4.7, where the inflationary semantics did not yield the desired model since the negative condition $\neg\,\mathsf{tc}(\mathsf{X},\mathsf{Y})$ was applied "too early", i.e., before the final result of tc was computed. The basic idea of the stratified semantics is that a relation should be defined completely before its negation is used, thereby avoiding such semantical problems.

Definition 5.1 (Stratification, S-Datalog) Let P be a Datalog$^\neg$ program and r, r' rules of P of the form

$$\begin{array}{lll} r: & \cdots & \leftarrow \cdots\ p(\ldots)\ \cdots \\ r': & p(\ldots) & \leftarrow \cdots \end{array}$$

Then r is said to *depend positively* on r', denoted $r' \rightarrow r$. If instead p occurs negated in the body of r, then r *depends negatively* on r', denoted $r' \stackrel{\neg}{\rightarrow} r$. The *dependency graph* $\mathcal{G}_P$ of P consists of the positive and negative dependencies $r' \stackrel{(\neg)}{\rightarrow} r$. We write $r' \rightsquigarrow r$, if there is a path from r' to r in $\mathcal{G}_P$, and $r' \stackrel{\neg}{\rightsquigarrow} r$, if the path involves at least one negative dependency. P is called *stratified* if there exists a partition $P = P_1 \dot\cup \ldots \dot\cup P_n$ such that for all $i, j = 1, \ldots, n$ and all $r' \in P_i$, $r \in P_j$:

- if $r' \rightsquigarrow r$ then $i \le j$, and
- if $r' \stackrel{\neg}{\rightsquigarrow} r$ then $i < j$.

The sequence $P_1, \ldots, P_n$ is called a *stratification* of P with the *strata* P_i. By *S-Datalog* we denote the class of stratified Datalog$^\neg$ programs. □

In S-Datalog programs, a rule r depends negatively only on rules from strictly lower strata, whereas it may depend positively also on rules from the same stratum. One can show that P is stratified iff $\mathcal{G}_P$ contains no *negative cycle* $r \stackrel{\neg}{\rightsquigarrow} r$.[2] The strata P_i are given by the *strongly connected components* (*scc*) of $\mathcal{G}_P$.[3] Clearly, no scc contains a negative edge (otherwise P would not be stratified). Therefore, a topological sort of the scc's yields the desired stratification $P_1, \ldots, P_n$.

Let P be a S-Datalog program with n strata. For every database $\mathcal{D}$, $P \cup \mathcal{D}$ has a canonical model $\mathcal{S}_{P\cup\mathcal{D}}$ called the *stratified model* of $P \cup \mathcal{D}$ which is obtained by successively evaluating the fixpoints $T^\omega_{P_i}$ of the strata P_i as follows:

$$\begin{array}{lll} \mathcal{I}_0 & := & \mathcal{D}, \\ \mathcal{I}_i & := & \mathcal{I}_{i-1} \cup T^\omega_{P_i}(\mathcal{I}_{i-1}) \text{ for all } i = 1, \ldots, n, \\ \mathcal{S}_{P\cup\mathcal{D}} & := & \mathcal{I}_n. \end{array}$$

Example 5.1 (Complement of Transitive Closure, Cont'd) Let P be the program from Example 4.7. Clearly, P is a S-Datalog program with the stratification $P_1 = \{r_1, r_2\}$, $P_2 = \{r_3\}$. Moreover, for any database $\mathcal{D}$ with edge relation e, $\mathcal{S}_{P\cup\mathcal{D}} \models \mathsf{ntc}(\mathsf{x},\mathsf{y})$ iff x,y are nodes in e, and (x,y) is not in the transitive closure of e. □

[2] For infinite P, this condition is not sufficient; take, e.g., $P = \mathit{ground}(P_2)$ in Example 6.3.

[3] Recall that a strongly connected component of a directed graph $\mathcal{G}$ is a maximal set of vertices V s.t. for all $u, v \in V$: $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

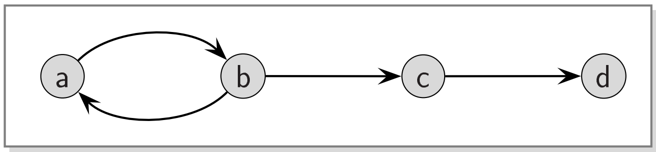


Figure 5.1: A simple move-graph

Note that a S-Datalog program may have several stratifications. However, one can show that $\mathcal{S}_{P\cup\mathcal{D}}$ is independent of the chosen stratification [ABW88]. For stratified programs, $\mathcal{S}_{P\cup\mathcal{D}}$ is now universally accepted as the intended declarative semantics. For non-stratified Datalog$^\neg$ programs there are two main competing approaches, the *well-founded* and *stable semantics*. Both semantics coincide with $\mathcal{S}_{P\cup\mathcal{D}}$ for stratified programs.

5.4 Well-Founded Semantics

The intuition of the well-founded semantics can be explained by the following well-known game example.

Example 5.2 (Win-Move Game) The game is given by a set of positions and a set of moves between them (cf. Figure 5.1). There are two players moving alternately on the given move-graph. A player who cannot move loses. Hence, a position x is won, if there is a move to some position y which is lost (since then the opponent has to move). Clearly, positions without outgoing moves are immediately lost. Games of this general type are described by the following non-stratified program in an intuitive and declarative way:

$$P_{game}: \quad \boxed{\mathsf{win(X)} \leftarrow \mathsf{move(X,Y)},\ \neg\,\mathsf{win(Y)}.}$$

The rule states that win(X) holds if there is a move to Y such that $\neg$ win(Y) holds. Consider, for example, the move-graph given by the database

$$\mathcal{D} = \{\mathsf{move(a,b)},\ \mathsf{move(b,a)},\ \mathsf{move(b,c)},\ \mathsf{move(c,d)}\}$$

in Figure 5.1: Clearly, d is lost since there are no possible moves from d. Thus, using P_{game} one can argue that win(c) is true since move(c,d) $\wedge\ \neg$ win(d) holds. Starting from b, however, one must not move to c (since c is won for the opponent). Therefore, in order to avoid losing, the only reasonable move from b is to a. In this case, the player moving from a can only move back to b, so the game ends up in the cycle a$\rightarrow$b$\rightarrow$a. Consequently, there is no "well-founded argumentation" establishing that b is won. On the other hand, the other player can also not "prove" that b is lost, since one can avoid losing from b by playing a game of infinite length. Using a similar argumentation, one shows that a is neither lost nor won. Thus, apart from the won and lost positions c and d, respectively, there are *drawn* positions a and b. □

The intuitive semantics of P_{game} is precisely captured by the well-founded model which assigns a unique 3-valued model to every program. In Example 5.2, win(a),win(b) will be assigned the truth-value *undefined*, win(c) and win(d) are *true* and *false* under the well-founded semantics, respectively.

Definition 5.2 (Well-Founded Semantics, WF-Datalog) Let P be a Datalog$^{\neg}$ program including a given database $\mathcal{D}$ as a set of facts. Fix some Herbrand interpretation $\mathcal{J} \subseteq \mathcal{B}_P$. The immediate consequences under the assumptions $\mathcal{J}$ are given by:

$$\begin{array}{rcl} T_P^{\mathcal{J}}(\mathcal{I}) & := & \{H \mid (H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m) \in \mathit{ground}(P), \\ & & \mathcal{I} \models B_1 \wedge \ldots \wedge B_n, \quad \mathcal{J} \models \neg C_1 \wedge \ldots \wedge \neg C_m \ \} \ . \end{array}$$

Since $\mathcal{J}$ is fixed, $T_P^{\mathcal{J}}$ is a monotone operator. Let $\Gamma_P(\mathcal{J}) := \mathrm{lfp}(T_P^{\mathcal{J}})$ be its least fixpoint. The operator Γ_P is antimonotone (observe how $\mathcal{J}$ is used in $T_P^{\mathcal{J}}$), i.e., $\mathcal{J}_1 \subseteq \mathcal{J}_2$ implies $\Gamma_P(\mathcal{J}_2) \subseteq \Gamma_P(\mathcal{J}_1)$. It follows that Γ_P^2 ($:= \Gamma_P \circ \Gamma_P$) is a monotone operator, so it has a least and a greatest fixpoint. These are used to define the 3-valued well-founded model $\mathcal{W}_P = \langle \mathcal{W}_P^{\boldsymbol{t}}, \mathcal{W}_P^{\boldsymbol{u}}, \mathcal{W}_P^{\boldsymbol{f}} \rangle$ of *true*, *undefined*, and *false* atoms as follows:

$$\begin{array}{rcl} \mathcal{W}_P^{\boldsymbol{t}} & := & \mathrm{lfp}(\Gamma_P^2) \ , \\ \mathcal{W}_P^{\boldsymbol{u}} & := & \mathrm{gfp}(\Gamma_P^2) \setminus \mathrm{lfp}(\Gamma_P^2) \ , \\ \mathcal{W}_P^{\boldsymbol{f}} & := & \mathcal{B}_P \setminus \mathrm{gfp}(\Gamma_P^2) \ . \end{array}$$

If $\mathcal{W}_P^{\boldsymbol{u}} = \emptyset$ then P is called *total* or *2-valued*. By *WF-Datalog* we denote the class of Datalog$^{\neg}$ programs evaluated under the well-founded semantics; *WF-Datalog*$_2$ is WF-Datalog restricted to total programs, i.e., where $\mathcal{W}_P$ is total for *all* databases $\mathcal{D}$. □

The construction of $\mathcal{W}_P$ given above is called the *alternating fixpoint* of the well-founded semantics [VG93] and corresponds to a nested fixpoint: the inner fixpoint is given by $\Gamma_P(\mathcal{J}) = \mathrm{lfp}(T_P^{\mathcal{J}})$. The outer fixpoints are obtained by iterating the antimonotone operator Γ_P. This sequence $\Gamma_P^0, \Gamma_P^1, \ldots$ which is given by

$$\Gamma_P^0 := \emptyset, \qquad \text{and} \qquad \Gamma_P^{i+1} := \Gamma_P(\Gamma_P^i), \tag{5.1}$$

alternates between underestimates of *true*, and overestimates of *true* or *undefined* atoms, respectively. More precisely, the subsequence Γ_P^{2k} converges to the least fixpoint (the set of true atoms) from below, while Γ_P^{2k+1} converges to the greatest fixpoint (the set of true or undefined atoms) from above. The key idea is that applying the antimonotone operator Γ_P to an underestimate yields an overestimate and vice versa.

Example 5.3 (Win-Move Game, Cont'd) For the program and database from Example 5.2 above, the alternating sequence is

i	Γ_P^i
0	$\boxed{\emptyset}$
1	$\boxed{\mathit{move} \cup \{\mathsf{win(a)}, \mathsf{win(b)}, \mathsf{win(c)}\}}$
2	$\boxed{\mathit{move} \cup \{\mathsf{win(c)}\}}$
3	$\boxed{\mathit{move} \cup \{\mathsf{win(a)}, \mathsf{win(b)}, \mathsf{win(c)}\}}$
4	$\boxed{\mathit{move} \cup \{\mathsf{win(c)}\}}$
...	...

Hence, $\mathrm{lfp}(\Gamma_P^2) = \mathit{move} \cup \{\mathsf{win(c)}\}$, $\mathrm{gfp}(\Gamma_P^2) = \mathit{move} \cup \{\mathsf{win(a)}, \mathsf{win(b)}, \mathsf{win(c)}\}$, and the set of undefined atoms is $\mathrm{gfp}(\Gamma_P^2) \setminus \mathrm{lfp}(\Gamma_P^2) = \{\mathsf{win(a)}, \mathsf{win(b)}\}$, as expected. □

The well-founded semantics is generally accepted as a skeptical semantics for non-stratified programs; in "controversial" cases like $\mathsf{win(a)}$ and $\mathsf{win(b)}$, the truth-value *undef* is assigned.

5.5 Stable Semantics

The stable semantics is rather different from the semantics considered so far: It assigns a (possibly empty) *set of models* to a program instead of one canonical model. Thus, by choosing an arbitrary stable model one obtains a nondeterministic semantics, whereas all previously considered semantics were deterministic. Intuitively, a model $\mathcal{I}$ of a program P is called *stable*, if it reproduces itself under a certain natural reduction:

Definition 5.3 (Stable Models, ST-Datalog)
Let P be the ground version of a Datalog$^\neg$ program including a given database. The set of stable models of P is given as follows:

Guess an interpretation $\mathcal{I} \subseteq \mathcal{B}_P$. Obtain the *reduced program* $P/\mathcal{I}$ by replacing every *negative* literal of P by its truth-value wrt. $\mathcal{I}$. Since $P/\mathcal{I}$ is a negation-free program, it has a unique minimal model $\mathcal{M}_{P/\mathcal{I}}$. $\mathcal{I}$ is a *stable model* of P if $\mathcal{I} = \mathcal{M}_{P/\mathcal{I}}$, i.e., if the original "set of beliefs" $\mathcal{I}$ reproduces itself. By *ST-Datalog* we denote the class of Datalog$^\neg$ programs evaluated under the stable semantics. □

Relation to Well-Founded Semantics

It can be shown that every stable model extends the well-founded model [VGRS91], i.e., every stable model agrees with the true and false atoms of $\mathcal{W}_P$, and assigns true or false to the remaining undefined atoms in such a way that a "self-reproducing" model is obtained. Note that not every program has a stable model.

Example 5.4 (Win-Move Game, Cont'd) The program from Example 5.2 has two stable models: either $\mathsf{win(a)}$ is true and $\mathsf{win(b)}$ false, or vice versa. If $\mathsf{move(e,e)}$ is added to the database, then no stable model exists: the ground instantiated program contains the rule $\mathsf{win(e)} \leftarrow \mathsf{move(e,e)}, \neg\,\mathsf{win(e)}$, so $\mathsf{win(e)}$ must be true in every 2-valued model. On the other hand, no rule supports $\mathsf{win(e)}$, so there is no stable model. □

Since stable models extend the well-founded model, the existence of a unique stable model is guaranteed if $\mathcal{W}_P$ is total (the converse does not hold in general). If $\mathcal{W}_P$ is not total, the well-founded model may be conceived as a skeptical approximation of the stable models. This approximation is mirrored by the alternating fixpoint computation of $\mathcal{W}_P$ in the following sense: An application of Γ_P corresponds to the above program reduction followed by a minimal model computation. More precisely:

$$\Gamma_P(\mathcal{I}) = \mathcal{M}_{ground(P)/\mathcal{I}} .$$

Thus, the alternating fixpoint computation Γ_P^i can be equivalently obtained as follows:

$$\Gamma_P^0 := \emptyset, \qquad \text{and} \qquad \Gamma_P^{i+1} := \mathcal{M}_{ground(P)/\Gamma_P^i}.$$

5.6 Properties of Deductive Rules

As for the production rule semantics, we briefly present the main results regarding theoretical properties of deductive rules.

Language	Expressiveness	Data Complexity
S-Datalog	S-DATALOG	**PTIME**
WF-Datalog	FIXPOINT	**PTIME**
WF-Datalog$_2$	FIXPOINT	**PTIME**
ST$_\exists$-Datalog	DB-**NP**	**NP**
ST$_\forall$-Datalog	DB-**coNP**	**coNP**

Figure 5.2: Declarative Datalog$^\neg$ semantics: expressiveness and complexity

5.6.1 Termination, Expressive Power, and Complexity

Termination depends on the underlying rule execution model. In contrast to production rule semantics with their natural bottom-up execution model, declarative semantics do not presuppose a specific evaluation strategy. Hence, the notion of termination of rules is meaningful only in connection with the corresponding evaluation algorithm.

In the database context, where finite structures are presupposed, termination can be guaranteed for bottom-up evaluation of S-Datalog and WF-Datalog: For S-Datalog, successively iterating T_{P_i} for each stratum P_i always yields a unique fixpoint. In the case of WF-Datalog, the fixpoint of the sequence $\Gamma_P^0, \Gamma_P^1, \ldots$ only exists if P is total. However, as mentioned above, the sequences $\Gamma_P^0, \Gamma_P^2, \Gamma_P^4, \ldots$ and $\Gamma_P^1, \Gamma_P^3, \Gamma_P^5, \ldots$ are guaranteed to converge to the least and greatest fixpoint of the doubled operator $\Gamma_P(\Gamma_P)$, respectively.

While WF-Datalog expresses precisely the FIXPOINT queries, S-Datalog is strictly less expressive (cf. Figure 4.2). It is easy to see that both semantics can be evaluated in polynomial time wrt. the size of the given database $\mathcal{D}$.

Let P be a Datalog$^\neg$ program (including a given database $\mathcal{D}$), and $A \in \mathcal{B}_P$ an atom. Then, for the stable semantics, the following types of inferences are commonly used:

- *Possibility semantics:* determine whether $\mathcal{I} \models A$ for *some* stable model $\mathcal{I}$ of P.

- *Certainty semantics:* determine whether $\mathcal{I} \models A$ for *all* stable models $\mathcal{I}$ of P.

Note that the possibility and certainty semantics are deterministic, and correspond to the union and intersection, respectively, of all stable models. A nondeterministic semantics is obtained by choosing an arbitrary stable model (if there exists one) as the intended model. It can be shown that the complexity of reasoning under the possibility and certainty semantics is **NP**-complete and **coNP**-complete, respectively [MT91]. In particular, deciding whether a stable model exists is already **NP**-complete [BF91]. Figure 5.2 summarizes the expressiveness and complexity results. ST$_\exists$-Datalog and ST$_\forall$-Datalog denote the stable semantics for Datalog$^\neg$ under possibility and certainty reasoning, respectively.

5.6.2 Handling of Negation

The focus of research on declarative semantics is on finding the "intended" model in the presence of rules with negation. The basic features of the semantics presented here can be illustrated with the following simple examples:

P_1: $\boxed{\mathsf{p} \leftarrow \neg\, \mathsf{q}.}$ P_2: $\boxed{\mathsf{p} \leftarrow \neg\, \mathsf{p}.}$ P_3: $\boxed{\begin{array}{l}\mathsf{p} \leftarrow \neg\, \mathsf{q}.\\ \mathsf{q} \leftarrow \neg\, \mathsf{p}.\end{array}}$

Declarative Semantics. The stratified semantics deals with programs without negative cycles like P_1 and assigns to them a unique canonical model. All three declarative semantics (stratified, well-founded, and stable) coincide on stratified programs. The stratified semantics excludes programs with negative cycles like P_2 and P_3. The skeptical well-founded semantics assigns *undef* to all atoms which (i) depend negatively on themselves and (ii) for which no independent, well-founded derivation exists. This is the case for p (p and q) in P_2 (P_3). Stable models are "less skeptical" and extend the well-founded model—if possible—to consistent, "self-reproducing" total models: P_3 has two stable models $\{\mathsf{p}, \neg\, \mathsf{q}\}$ and $\{\mathsf{q}, \neg\, \mathsf{p}\}$. P_2 has no stable models, since the only possible interpretations are $\{\neg\, \mathsf{p}\}$ and $\{\mathsf{p}\}$: the former is not a model of P_2, while the latter is not stable (no rule supports p).

Production Rule Semantics. The behavior of the inflationary and noninflationary semantics on these examples is quite different. For P_1, both the I-Datalog and the P-Datalog semantics yield the intended model $\{\mathsf{p}\}$. However, the I-Datalog model of P_2 is also $\{\mathsf{p}\}$ which is not supported by any rule (see property SUPP below). For P_3, I-Datalog yields the model $\{\mathsf{p}, \mathsf{q}\}$ which is neither minimal nor supported. The P-Datalog semantics does not yield a fixpoint for P_2 and P_3. The same result as for I-Datalog is obtained if one adds the tautological rules $\mathsf{p} \leftarrow \mathsf{p}$ and $\mathsf{q} \leftarrow \mathsf{q}$ to P_2 and P_3.

Classification

In order to formally compare and classify semantics of logic programs wrt. their "declarativity", numerous desirable properties of semantics have been introduced. Obviously, a model $\mathcal{I} \subseteq \mathcal{B}_P$ has to satisfy all rules of *ground*(P), which is the case iff $T_P(\mathcal{I}) \subseteq \mathcal{I}$. In the sequel, we focus on further important properties which allow to distinguish between declarative semantics and production rule semantics. We say that *a semantics satisfies a property*, if every intended model of the given semantics satisfies the property.

Minimality (MIN): A very natural property, already introduced in Section 5.2, is minimality. If a model $\mathcal{I}$ is not minimal, then there exists a strictly smaller model $\mathcal{I}' \subseteq \mathcal{I}$ which should then be preferred to $\mathcal{I}$.

Supportedness (SUPP): A model $\mathcal{I}$ should also be *supported*, i.e., every $A \in \mathcal{I}$ should be "justified" by a rule $(A \leftarrow \mathit{body}) \in \mathit{ground}(P)$ such that $\mathcal{I} \models \mathit{body}$. Clearly, this is the case iff $\mathcal{I} \subseteq T_P(\mathcal{I})$. It follows that the fixpoints of T_P are precisely the supported models of P. If a model is *not* supported, it contains atoms whose truth cannot be explained using only the rules of P.

Elimination of Tautologies (TAUT): A semantics allows elimination of tautologies, if rules of the form

$$A \leftarrow \ \ldots, A, \ldots$$

can be eliminated from P without changing the semantics.

	MIN	SUPP	TAUT	REL	UNFOLD
I-Datalog	–	–	+	+	–
P-Datalog	–	+	–	–	–
S-Datalog	+	+	+	+	+
WF-Datalog	+[4]	+	+	+	+
ST-Datalog	+	+	+	–	+

Figure 5.3: Classification of Datalog$^\neg$ semantics

Relevance (REL): Let P be a program over the schema $\mathbf{P}$. A semantics satisfies the principle of relevance, if the addition of rules in which only "irrelevant" relation symbols $p \notin \mathbf{P}$ occur does not change the semantics of P wrt. relations from $\mathbf{P}$.

Unfolding (UNFOLD): A semantics allows unfolding, if *positive* literals can be replaced by their definitions without changing the semantics: Let P be a ground program containing a rule

$$r: \quad A \leftarrow B_1, \ldots, B_i, \ldots, B_n, \neg C_1, \ldots, \neg C_m \ .$$

Moreover, assume that

$$\begin{array}{lcl} B_i & \leftarrow & body_1, \\ & \vdots & \\ B_i & \leftarrow & body_k, \end{array}$$

are all rules defining B_i in P. Then by unfolding of B_i we obtain

$$\begin{array}{rlcl} P' := P \setminus \{r\} \cup \{ & A & \leftarrow & B_1, \ldots, B_{i-1}, body_1, B_{i+1}, \neg C_1, \ldots, \neg C_m \ , \\ & & \vdots & \\ & A & \leftarrow & B_1, \ldots, B_{i-1}, body_k, B_{i+1}, \neg C_1, \ldots, \neg C_m \ \} \end{array}$$

whose semantics should coincide with the semantics of P.

The stratified, well-founded, stable, and numerous other semantics developed for logic programs (and disjunctive extensions) have been investigated and classified wrt. the above and several other properties. However, there seem to be no corresponding classifications for the production rule semantics of Datalog$^\neg$—presumably because (i) the focus of these semantics was more on expressiveness, complexity, normal forms (as in finite model theory), and updates, rather than on declarativity, and (ii) they were developed by different communities. Indeed, the production rule semantics I-Datalog and P-Datalog fail to satisfy many of the "nice" and desirable properties above, as can be seen from Theorem 5.1 and Figure 5.3. For the analogous results on the declarative semantics, see [Dix95], [Min96], [BD97]. Note that ST-Datalog (the stable semantics for Datalog$^\neg$) violates REL since adding an irrelevant rule $\mathsf{q} \leftarrow \neg\, \mathsf{q}$ prevents the existence of stable models.

Theorem 5.1 (Properties of I-Datalog, P-Datalog)
The I-Datalog semantics satisfies TAUT and REL, but not MIN, SUPP, and UNFOLD. The P-Datalog semantics satisfies SUPP, but not MIN, TAUT, REL, and UNFOLD.

[4]Here, by MIN we mean that $\mathcal{W}_P$ is the *smallest* 3-valued stable model: there is no *3-valued* (or *partial*) stable model $\mathcal{PS}_P \neq \mathcal{W}_P$ s.t. $\mathcal{PS}_P^{\boldsymbol{t}} \subseteq \mathcal{W}_P^{\boldsymbol{t}}$ and $\mathcal{PS}_P^{\boldsymbol{f}} \subseteq \mathcal{W}_P^{\boldsymbol{f}}$; cf. [Prz90].

PROOF I-Datalog violates MIN for the program P_{ntc} from Example 4.7, since the (intended) stratified model is strictly included in the I-Datalog model. The P-Datalog model of P'_{ntc} coincides with the I-Datalog model of P_{ntc}, so P-Datalog also violates MIN. Since the P-Datalog semantics differs for P_{ntc} and P'_{ntc}, the previous example also establishes that P-Datalog does not satisfy TAUT. In contrast, I-Datalog obviously satisfies TAUT: Due to the inflationary fixpoint definition, rules of the form $A \leftarrow \ldots, A, \ldots$ are superfluous and can be deleted.

The I-Datalog model of P_2 above is {p} which clearly violates SUPP. On the other hand, the P-Datalog model—if it exists—is a fixpoint of T_P and thus is supported. By the definition of I-Datalog, it is clear that REL is satisfied. For P-Datalog, adding an irrelevant rule q ← ¬ q , where q does not occur in the remaining rules, prevents the existence of the fixpoint; thus, REL is not satisfied. Finally, I-Datalog does not satisfy UNFOLD as can be seen from the following programs:

P:

```
p  ← .
p' ← p.
q  ← p', ¬ p'.
```

P_{UNF}:

```
p  ← .
p' ← p.
q  ← p, ¬ p'.
```

P_{UNF} is obtained from P by unfolding the positive occurrence of p' in the last rule. Note that P never derives q. In contrast, since p' is derived one stage later than p, P_{UNF} allows to derive q.

By adding the rule q ← q to both programs, the same argument shows that P-Datalog violates UNFOLD. ■

Remark. The P-Datalog semantics satisfies REL, if it is defined in a slightly more general way: For P ($=P' \cup \mathcal{D}$), the original P-Datalog semantics is T_P^ω—if the fixpoint exists—and $\perp$ otherwise (Section 4.2.2). The only way to violate REL is by adding irrelevant rules which cause the result to become undefined. This can be avoided by defining as the semantics the intersection of all stages T_P^n for sufficiently large n (i.e., when the stages become periodic).[5] This semantics extends the original definition and satisfies REL (since it is always defined).

5.7 Summary

Deductive rules are used to specify knowledge about the current state of the modeled world in a concise and declarative way. One important aspect of the proposed semantics is a "reasonable" and intuitive handling of rules with negation. Indeed, unlike the production rules semantics, the stratified, well-founded, and stable semantics satisfy many natural properties which any well-behaved[6] logical semantics should have. Roughly speaking, production rules and deductive rules have different "readings" of rules: While deductive rules correspond more to a reasoning process (**if** *premise* **then** *conclusion*), the fixpoint semantics of production rules can be viewed as a specific form of active rules (**if** *condition* **then** *action*). Since deductive rules do not directly lend themselves to the specification of updates, numerous extensions have been proposed to incorporate dynamic aspects (cf. Section 7.9.2).

[5]This definition is the analogue to the *eventually true* atoms p_t in Theorem 6.33.

[6][Dix95] formalizes this notion and shows that any well-behaved semantics coincides with the stratified model for stratified programs, and only extends the well-founded model for non-stratified programs.

Bibliographic Notes

The notion of stratification for logic programs has come up several times including [CH85], [VG88], [ABW88], and [Prz88]. In [ABW88], it was shown that the choice of stratification has no influence on the unique preferred model; this model has been termed *perfect model* by [Prz88] who also introduced the natural extension to ground programs called *local stratification*. S-Datalog is closely related to *bounded fixpoint logic* FO(BFP), known from finite model theory [EF95]. The fact that S-Datalog is less expressive than FIXPOINT was shown in [Kol91].

The well-founded model was first proposed in [VGRS88] using so-called *greatest unfounded sets*; a 3-valued reconstruction is presented in [Prz89], see also [BF91], [SZ91]. The constructive alternating fixpoint characterization is due to [VG89], where it is also shown that WF-Datalog expresses FIXPOINT. [AHV95] raised the question whether this still holds for total programs. This question was answered affirmatively in [FKL97], where it was shown that WF-Datalog $\equiv$ WF-Datalog$_2$, i.e., for every WF-Datalog program there is an equivalent total program (which is of the form of the win-move game from Example 5.2).

The stable semantics is due to [GL88], the reduction $P/\mathcal{I}$ is also called *Gelfond-Lifschitz transformation*. The stable semantics is closely related to nonmonotonic reasoning logics like *default logic* and *autoepistemic logic*; see, e.g., [MT91] and [Dix95] for an introduction and general overview. In addition to 2-valued stable models, several natural 3-valued extensions have been proposed [Prz90, ELS96, Sac97]. There are a number of surveys on deductive rules including [Bid91], [AB94] (with a focus on logic programming semantics), [Dix95] (which is more oriented towards nonmonotonic reasoning semantics), [Via97b] (comparing deductive with production rule semantics), and [Min96] (where almost all activities and historic developments in the area are covered). A comprehensive survey on complexity results for logic programming semantics is presented in [Sch95]; the expressive power of various 3-valued stable Datalog$^\neg$ semantics is characterized in [Sac97]. [GGSZ97] contains an introduction and overview on nondeterministic programming in deductive databases.

6 Statelog

6.1 Introduction

In the following, we show how to integrate the declarative semantics of deductive rules with the possibility to define updates in the style of production rules. Recall that the declarative Datalog semantics only specify the current state of the modeled world and therefore do not support updates within the logical language. In contrast, the production rule semantics for Datalog$^{\neg\neg}$ allows to express updates using a certain procedural fixpoint semantics, but leads to unintuitive results wrt. the handling of negation. As we will show, both seemingly discordant approaches can be reconciled, and one can obtain deductive and production rules as special cases of the more general Statelog rules. While for production rules the notion of state was implicitly given by the stages of the bottom-up fixpoint semantics, Statelog incorporates an *explicit* notion of state by extending the syntax of Datalog and then applying standard declarative semantics. In this way, it is possible to express that the truth-value of an atom varies over time, nicely mirroring the fact that tuples in the database are repeatedly modified, inserted, or deleted, respectively.

In this chapter, we present the "pure" version of Statelog and investigate in-depth theoretical properties like normal forms, expressive power, complexity, and termination. A logical framework integrating active and deductive rules is developed in Chapter 7 using pure Statelog as the underlying language.

6.2 Basic Execution Model

The basic execution model of Statelog is motivated by the distinction between queries and updates: Conceptually, a *query* posed against the database $\mathcal{D}$ is a question about the current state of the modeled world and should not give rise to a new database state $\mathcal{D}'$. In Statelog, queries are expressed using so-called *local rules* which only increase the information about the current state without defining a new one. In contrast, an *update* creates a new database state $\mathcal{D}'$, in order to mirror the changes which occurred in the modeled world.[1] Although, in principle, it would be sufficient to consider only two states at a time, i.e., the old state before the update and the new state after the update, a finer granularity is needed to model active rule processing *within* the logical language. In Statelog, this is accomplished by *progressive rules* which allow to refer to the current and past states, in order to define the new database state. A natural distinction is between virtual, *intermediate states*, which are used during rule processing to store intermediate results, and *final states*. The latter are materialized and may

[1] [KM91] elaborate on the distinction between update and revision. Basically, *updates* deal with keeping up-to-date the theory (here: database) in a dynamically changing world. In contrast, *revisions* are used when one incorporates new information, in order to revise the knowledge about a static world.

be queried by the user, while the former are not (directly) accessible to the user.

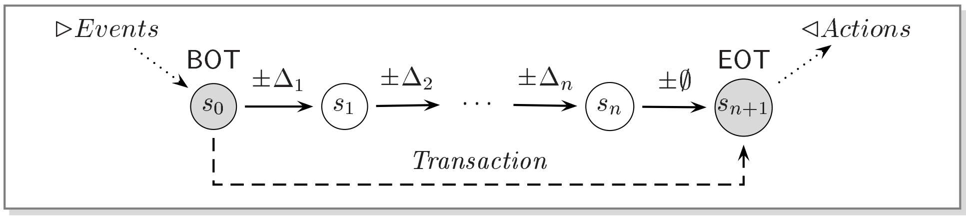


Figure 6.1: Database evolution: a transaction and intermediate transitions

The basic execution model of Statelog is illustrated in Figure 6.1: s_0 denotes the *initial state* of the database and may be queried using local Statelog rules. The occurrence of external events, like user requests to update the database, are mapped to the initial state, thereby "triggering" a given set of Statelog rules P.[2] This results in a sequence of *transitions* to the intermediate states $s_1, s_2, \ldots$ Eventually, when there are no more changes, the *final state* s_{n+1} is reached. The Statelog *transaction* is given by the overall effect of P on $\mathcal{D}$, i.e., the update from the initial database $\mathcal{D}$ at s_0 to the new final state $\mathcal{D}'$ at s_{n+1}.

6.3 Syntax

In order to refer to the different database states of the above execution model, we use the natural numbers as a "state index" for the database evolution. This is done by extending the syntax of Datalog with a unary function symbol "+1" which can be regarded as an operator mapping the current state to the next state. The resulting language *Statelog* is related to the temporal database query languages *Datalog$_{1S}$* and *Templog* (see Section 6.10.1) and corresponds to a restricted class of logic programs.

Definition 6.1 (State Terms, Statelog Atoms) In addition to the Datalog vocabulary comprising relation symbols $\mathbb{R}$, *data constants* $\mathbb{C}$, and *data variables* $\mathbb{V}$, the Statelog vocabulary contains a new *state constant* $0 \notin \mathbb{C}$ (denoting the initial state), a unary function symbol "+1" (mapping a state to the next state), and a distinguished *state variable* $S \notin \mathbb{V}$. The set $\mathbb{S}$ of *state terms* is the least set such that

- $\{\, [0], [S] \,\} \subseteq \mathbb{S}$, and
- if $[T] \in \mathbb{S}$, then $[T{+}1] \in \mathbb{S}$.

A *Statelog atom* is an expression of the form

- $[T]\, p(x_1, \ldots, x_n)$,

where $[T]$ is a state term, p is an n-ary relation symbol from $\mathbb{R}$, and $x_1, \ldots, x_n \in \mathbb{C} \cup \mathbb{V}$. A *Statelog literal* is a Statelog atom or its negation. □

The set of ground state terms can be identified with $\mathbb{N}_0$. Thus, we simply write k and $S{+}k$ to denote the k-fold application of "+1" to 0 and S, respectively. A ground state term is called a *state identifier* or, loosely speaking, a *state*.

[2]The precise notion and handling of events in Statelog will be discussed in Chapter 7.

Definition 6.2 (Statelog) A *Statelog rule* is an expression of the form

$$[S{+}k_0]\, H \leftarrow [S{+}k_1]\, B_1, \;\ldots\;, [S{+}k_n]\, B_n \;,$$

where $[S{+}k_0]\, H$ is a Statelog atom and $[S{+}k_1]\, B_1, \ldots, [S{+}k_n]\, B_n$ are Statelog literals. A *Statelog program* is a finite set of Statelog rules. The *schema* $\mathbf{P}$ of a Statelog program P is the set of all relation symbols occurring in rules of P, together with their arities.

A *Statelog database* $\mathcal{D}$ over a given schema $\mathbf{P}$ is a finite set of ground facts of the form $[0]\, p(x_1, \ldots, x_n)$, where $p \in \mathbf{P}$, and $x_i \in \mathbb{C}$ for $i = 1, \ldots, n$. By $\mathcal{D}[0]$ we denote the database instance at the initial state $[0]$:

$$\mathcal{D}[0] := \{\, p(\bar{x}) \mid [0]\, p(\bar{x}) \in \mathcal{D}\}.$$

□

Notation. Notice that, with slight abuse of notation, $\mathcal{D}$ may denote a Statelog database (facts of the form $[0]\, p(\bar{x})$), or—as in previous chapters—a "conventional" database instance (facts of the form $p(\bar{x})$). From the context it will always be clear which variant is meant.

In Definition 6.2, we excluded ground state terms $[k]$ from Statelog rules. This is no loss of generality, since such rules can be easily simulated using auxiliary relations (Lemma 6.8). As in the Datalog case, the rule $H_1, \ldots, H_m \leftarrow$ *body* is shorthand for the set of rules $\{H_1 \leftarrow$ *body* ; $\ldots$; $H_m \leftarrow$ *body*$\}$. Moreover, if in a sequence of literals all of them refer to the same state, all occurrences of state terms—apart from the first occurrence—can be omitted. For example instead of

$$[S{+}1]\, A, [S{+}1] \neg B, [S]\, C$$

we simply write

$$[S{+}1]\, A, \neg B, [S]\, C.$$

Example 6.1 (Integrity Enforcement, Cont'd) The following Statelog rules accomplish the same effect as the production rules from Example 4.1:

r_1:	$[S]$ del_emp(E,Sal,D) $\leftarrow [S]$ $\triangleright$del_emp(E), emp(E,Sal,D).
r_2:	$[S{+}1]$ emp(E,Sal,D) $\leftarrow [S]$ emp(E,Sal,D), $\neg$ del_emp(E,Sal,D).
r_3:	$[S{+}1]$ dept(D,E) $\leftarrow [S]$ dept(D,E), $\neg$ del_dept(D,E).
r_4:	$[S]$ del_dept(D,E), del_emp(E1,S1,D) $\leftarrow$ $[S]$ del_emp(E,_,_), dept(D,E), emp(E1,S1,D).

Note that r_1 and r_4 are *local*, i.e., for each value of S, they refer to one state only (see Definition 6.3). They derive delete requests from external events (r_1), or generate new requests from existing ones (r_4) within the state $[S]$. The non-local rules r_2 and r_3 are *frame rules* specifying which tuples are transferred to the next state during a transition.[3] Thus, logically, no real deletion occurs but a smaller "copy" of the old database state is created. □

A canonical model $\mathcal{M}_{P\cup\mathcal{D}}$ for Statelog program P with database $\mathcal{D}$ will be defined below. Nevertheless, it should be clear that under any "reasonable" declarative semantics, the above rules define the same transaction as the production rules from Example 4.1.

[3]Events and frame rules are further discussed in Chapter 7.

6.3.1 Reified Syntax

Let P be a Statelog program (possibly including a database $\mathcal{D}$ as a set of facts). P corresponds to a standard logic program $P^\star$ by the following mapping $(\,)^\star$ from n-ary Statelog atoms to $n{+}1$-ary "conventional" atoms:

$$([T]\,p(X_1,\ldots,X_n))^\star := p(T,X_1,\ldots,X_n)\ .$$

Here, $X_i \in \mathbb{C} \cup \mathbb{V}$, and p is an n-ary relation symbol occurring in P. The obvious extension of $(\,)^\star$ to literals and rules yields the desired *reified program* $P^\star$. Since there is a natural one-to-one correspondence between Herbrand models of $P^\star$ and Statelog models of P (see Proposition 6.2), the well-known semantics for logic programs can be applied to the Statelog framework.

6.3.2 Progressiveness

The basic idea of the Statelog approach is to use the current state and possibly previous states in order to define (i) a new state, or (ii) extend the current one. In the first case, a rule defines the changes which should be applied in the transition from the current state to the successor state, i.e., an *update*. In contrast, if rules only extend the current state by some derived information without changing the underlying fact base, a *view* is defined. In either case, nothing is said about the precise correspondence between states and some underlying notion of time, like *valid time* or *transaction time* [TCG+93]. The only requirement is that the (partial) order of the underlying time model corresponds with the order on states in the following sense: For all state identifiers s_1, s_2:

$$s_1 < s_2 \Rightarrow t(s_1) < t(s_2)\ ,$$

where $t(s_i)$ denotes the time associated with s_i in some underlying time model. Using this assumption, the order on state identifiers reflects the progression of time. Since Statelog rules are meant to define the database evolution progressing from the past to the future, we therefore disallow rules of the form

$$[S{+}k_0]\,H \leftarrow\ [S{+}k_1]\,B_1,\ldots,[S{+}k_n]\,B_n$$

where $k_0 < k_i$ for some i. Otherwise, such a rule would either require to make assumptions about a future state or to change a past state. Thus, in the sequel, we deliberately assume that all Statelog rules are *progressive*:

Definition 6.3 (Progressiveness) A Statelog rule r

$$[S{+}k_0]\,H\ \leftarrow\ [S{+}k_1]\,B_1,\ \ldots\ ,[S{+}k_n]\,B_n$$

is called *progressive*, if $k_0 \geq k_i$ for all $i = 1,\ldots,n$. A progressive rule is said to be *strictly progressive* if $k_0 > k_i$ for some $i = 1,\ldots,n$. If $k_0 = k_i$ for all $i = 1,\ldots,n$, then r is called *local*. Finally, if $k_0 = 1$ and $k_i = 0$ for $i \geq 1$, r is called *1-progressive*. A Statelog program P is *progressive*, if all rules $r \in P$ are progressive. □

As will be shown in Section 6.5, Statelog rules can be transformed into a certain normal form. Two interesting subclasses of programs in normal form are programs comprising only local rules, and programs consisting solely of 1-progressive rules. While the former correspond to the usual declarative Datalog$^\neg$ rules, the latter can be conceived as production rules. In this way, Statelog provides a seamless integration of both paradigms.

6.4 Semantics

Since in Statelog the database evolution is modeled as a linear sequence of states, Statelog structures may be viewed as certain temporal structures with $\mathbb{N}_0$ as the underlying temporal domain:

Definition 6.4 (Statelog Interpretation) Let $P = P' \cup \mathcal{D}$ be a Statelog program including a database $\mathcal{D}$ as a set of facts. A *Statelog structure* $(U_P, \mathcal{I}_P)$ *appropriate to* P consists of the finite *Herbrand universe* U_P, comprising the data constants of P, and a *Statelog interpretation* $\mathcal{I}_P$, where

$$\mathcal{I}_P = (\mathcal{I}_P[0], \mathcal{I}_P[1], \mathcal{I}_P[2], \ldots)$$

is an infinite sequence of *database states* or *snapshots* $\mathcal{I}_P[n] \subseteq \mathcal{B}_P$. The *Herbrand base* $\mathcal{B}_P$ consists of all ground atoms which can be constructed from U_P and relation symbols from P. If understood from the context, we simply write $\mathcal{I}$ instead of $\mathcal{I}_P$.

A *window* $(\mathcal{I}[m], \mathcal{I}[m{+}1], \ldots, \mathcal{I}[n])$ of snapshots is denoted as $\mathcal{I}[m..n]$. In particular, $\mathcal{I}[0..n]$ is the database *evolution* (or database *history*) up to state $[n]$. □

A Statelog interpretation $\mathcal{I}$ can be represented in several equivalent ways, similar to interpretations of temporal databases (cf. [Cho94]):

- In the *snapshot view*, $\mathcal{I}$ is a mapping $\mathcal{I} : \mathbb{N}_0 \to 2^{\mathcal{B}_P}$ assigning to each $n \in \mathbb{N}_0$ the snapshot database $\mathcal{I}[n]$ of tuples which are true in state $[n]$. Under this view, $\mathcal{I}$ can be represented as the infinite sequence $\mathcal{I} = (\mathcal{I}[n])_{n \in \mathbb{N}_0}$ as in Definition 6.4.

- The *timestamp view* is a mapping $\mathcal{I} : \mathcal{B}_P \to 2^{\mathbb{N}_0}$, where each atom is "stamped" with the set of timepoints for which it is true.

- The *model-theoretic* (or *reified*) *view* regards $\mathcal{I}$ as a subset of $\mathbb{N}_0 \times \mathcal{B}_P$, i.e., $\mathcal{I}$ is a set of tuples of the form $(n, p(\bar{x}))$. This is essentially the reified representation $p(n, \bar{x})$ (cf. Sections 6.3.1 and 6.4.2).

Definition 6.5 (Statelog Model) Let $P = P' \cup \mathcal{D}$ be a Statelog program P' with database $\mathcal{D}$. A Statelog *ground instantiation* is a function $gr : \{S\} \times \mathbb{V} \to \mathbb{N}_0 \times U_P$ mapping the state variable S to a state identifier and data variables to elements from U_P. gr extends to rules and programs in the obvious way. A Statelog interpretation $\mathcal{I}$ satisfies a ground instance of a Statelog rule, denoted

$$\mathcal{I} \models [n_0]\, H \leftarrow [n_1]\, B_1, \ldots, [n_k]\, B_k$$

if

$$\mathcal{I}[n_1] \models B_1, \ \ldots\ , \mathcal{I}[n_k] \models B_k \quad \text{implies} \quad \mathcal{I}[n_0] \models H\ .$$

A Statelog interpretation $\mathcal{I}$ is a *model* of a (non-ground) rule $r \in P$, denoted $\mathcal{I} \models r$, if $\mathcal{I}$ satisfies all ground instances of r. We write $\mathcal{I} \models P$, if $\mathcal{I} \models r$ for all $r \in P$. □

Range-Restriction. Due to Definition 6.5, unsafe rules like $p(X) \leftarrow \neg q(X)$ are always evaluated wrt. U_P, i.e., the active domain of constants from $P' \cup \mathcal{D}$. Although this guarantees finiteness of answers, such rules are domain dependent and may lead to unintended results (see Section 2.2). This problem can be avoided by restricting to *range-restricted* rules, where every variable must occur in a *positive* literal of the body. If all rules of a program P are range-restricted, then the outcome of P is independent of the underlying domain, and rules can be evaluated bottom-up using standard database techniques.

6.4.1 Periodicity and Finite Representation

In order to compute the transition from the current state $[n]$ to $[n{+}1]$, it is desirable to refer only to a fixed number of predecessor states, thereby avoiding the need to store the complete database evolution $\mathcal{I}[0..n]$. This is possible for interpretations which are *w-deterministic*:

Definition 6.6 (w-deterministic) A Statelog interpretation $\mathcal{I}$ is *w-deterministic*, if there exist $n_0, w \in \mathbb{N}_0$ such that for all $n \geq n_0$:

$$\mathcal{I}[n] = f(\mathcal{I}[n{-}w..n{-}1]) \ ,$$

and some computable function $f : (2^{\mathcal{B}_P})^w \to 2^{\mathcal{B}_P}$. □

Thus, starting from n_0, the whole sequence $\mathcal{I}[n_0], \mathcal{I}[n_0{+}1], \ldots$ can be effectively constructed from f in a state-by-state manner, looking at no more than w previous states. The iteration can be stopped if $\mathcal{I}$ becomes stationary, i.e., when there are no more changes and the final database state is reached. For the canonical model associated with a Statelog program (Section 6.4.4), such a unique final state may not exist and the model may become periodic:

Definition 6.7 (Periodicity, Termination) A Statelog interpretation $\mathcal{I}$ is called *ultimately periodic*, if there exist $n_0, T \in \mathbb{N}_0$, $T \geq 1$, such that

$$\mathcal{I}[n] = \mathcal{I}[n{+}T], \text{ for all } n \geq n_0.$$

The least such T is called the *period* of $\mathcal{I}$, the least n_0 describes the length of the *initial window* $\mathcal{I}[0..n_0{-}1]$. If $T = 1$, $\mathcal{I}$ is called *terminating*. In this case, we write $\mathcal{I}[\$]$ as a generic notation for the unique *final state* $\mathcal{I}[n_0]$ of $\mathcal{I}$.[4] If $\mathcal{I}$ does not terminate, we agree to set $\mathcal{I}[\$] := \emptyset$. □

By setting $\mathcal{I}[\$]$ to $\emptyset$ in case of nontermination, we guarantee that the expression $\mathcal{I}[\$]$ is always defined. This convenient notation is justified by Theorem 6.33, which says that termination for a given program P and database $\mathcal{D}$ can be detected within the language, so undefinedness can be avoided.

Note that the Herbrand universe U_P of a Statelog program P $(= P' \cup \mathcal{D})$ is finite. Therefore w-deterministic Statelog interpretations are ultimately periodic:

Proposition 6.1 (w-deterministic $\Rightarrow$ ultimately periodic)
Every w-deterministic Statelog interpretation is ultimately periodic.

Proof Let $\mathcal{I}_P$ be w-deterministic. Choose n_0 as in Definition 6.6. Since $\mathcal{B}_P$ is finite, there are at most $2^{w \cdot |\mathcal{B}_P|}$ many different windows of width w. Hence, there must be a window repetition in $\mathcal{I}_P$, i.e., there exist $n_2 > n_1 \geq n_0$ such that $\mathcal{I}_P[n_1{-}w..n_1{-}1] = \mathcal{I}_P[n_2{-}w..n_2{-}1]$. Thus, $\mathcal{I}_P[n_1{+}i] = \mathcal{I}_P[n_2{+}i]$ for all $i \in \mathbb{N}_0$, since $\mathcal{I}_P$ is w-deterministic. Finally, let $T = n_2 - n_1$, and the claim follows. ■

Every w-deterministic interpretation $\mathcal{I}$ has a natural finite representation in form of the initial window $\mathcal{I}[0..n_0{-}1]$ and the *periodic window* $\mathcal{I}[n_0..n_0{+}T{-}1]$:

$$\mathcal{I} = (\,\mathcal{I}[0..n_0{-}1], (\mathcal{I}[n_0..n_0{+}T{-}1])^\omega\,).$$

Thus, even if $\mathcal{I}$ is not terminating, a finite representation of the infinite model can be effectively computed. Termination will be further investigated in Sections 6.9 and 7.5.

[4] As a mnemonic, recall that the final outcome of a transaction is often measured in \$.

6.4.2 Reified Semantics

There is a natural one-to-one correspondence between Statelog interpretations of P and certain Herbrand interpretations of the reified program $P^\star$:

Let $\mathcal{I}$ be a Statelog interpretation appropriate to P. The Herbrand interpretation $\mathcal{I}^\star$ of $P^\star$ corresponding to $\mathcal{I}$ is defined by

$$\mathcal{I}^\star \models p(n, \bar{x}) \ :\Leftrightarrow \ \mathcal{I}[n] \models p(\bar{x}) \ ,$$

for all $n \in \mathbb{N}_0$ and $p(\bar{x}) \in \mathcal{B}_P$. Conversely, every such Herbrand interpretation $\mathcal{I}^\star$ of $P^\star$ induces a Statelog interpretation of P by using the same equivalence in the other direction. From this, the following is immediate:

Proposition 6.2 (Reification) *For every Statelog program P and database $\mathcal{D}$, $\mathcal{I}^\star$ is a model of $(P \cup \mathcal{D})^\star$ iff $\mathcal{I}$ is a Statelog model of $P \cup \mathcal{D}$, i.e.,*

$$\mathcal{I}^\star \models (P \cup \mathcal{D})^\star \ \Leftrightarrow \ \mathcal{I} \models P \cup \mathcal{D} \ .$$

Thus, in the sequel, we may use P and $P^\star$ synonymously. This allows to apply all standard notions developed for logic programs (e.g., *stratification* and *local stratification*) to Statelog. In particular, from the set of Statelog models certain distinguished ones can be selected by means of the well-known declarative semantics for logic programs. We will focus on so-called *state-stratified* Statelog programs P which have a single canonical $\mathcal{M}_{P \cup \mathcal{D}}$ for every database $\mathcal{D}$. Before turning to the definition of $\mathcal{M}_{P \cup \mathcal{D}}$, we briefly comment on the relation of Statelog rules to "classical" first-order temporal formulas.

6.4.3 Temporal Semantics

As mentioned before, Statelog structures can be regarded as special first-order temporal structures with a fixed underlying domain U, a temporal interpretation of relations $\mathcal{I}$, and time isomorphic to $\mathbb{N}_0$. This viewpoint allows the seamless integration of (some form of) temporal reasoning into Statelog. In Section 7.6.2, this temporal reading of Statelog structures will be used to incorporate dynamic integrity constraints into the Statelog framework.

We obtain a fragment of first-order temporal logic FOTL by adding the usual future operators "$\circ$" (*next time*) and "$\Box$" (*always in the future*) to first-order logic. Given a fixed universe U, and time isomorphic to $\mathbb{N}_0$, the semantics of FOTL formulas is defined as follows:

Let $\mathcal{I}$ be a Statelog interpretation and $v : \mathbb{V} \rightarrow U$ a valuation, which extends to terms as usual. Then:

- $\mathcal{I}[n]\,, v \models p(t_1, \ldots, t_n)$, if $(v(t_1), \ldots, v(t_n)) \in p^{\mathcal{I}[n]}$, (*for terms t_i*)
- $\mathcal{I}[n]\,, v \models \neg\, \varphi$, if not $\mathcal{I}[n]\,, v \models \varphi$,
- $\mathcal{I}[n]\,, v \models \varphi \wedge \psi$, if $\mathcal{I}[n]\,, v \models \varphi$ and $\mathcal{I}[n]\,, v \models \psi$, (*similar for $\vee, \leftarrow$*)
- $\mathcal{I}[n]\,, v \models \exists X \varphi$, if there is some $a \in U$ s.t. $\mathcal{I}[n]\,, v_{X/a} \models \varphi$, (*similar for $\forall$*)
- $\mathcal{I}[n]\,, v \models \circ\varphi$, if $\mathcal{I}[n{+}1]\,, v \models \varphi$,
- $\mathcal{I}[n]\,, v \models \Box\varphi$, if for all $l \in \mathbb{N}_0$: $\mathcal{I}[n{+}l]\,, v \models \varphi$.

Here, $v_{X/a}$ is defined like v, but maps X to a. Moreover, for a sentence φ we define: $\mathcal{I} \models \varphi$ if $\mathcal{I}[n] \models \varphi$ for all $n \in \mathbb{N}_0$. It is straightforward to verify that Statelog rules are equivalent to temporal formulas in the following way:

Proposition 6.3 (Temporal Reading) *Let $\mathcal{I}$ be a Statelog interpretation appropriate to P, and $([S{+}k_0]\, H \leftarrow [S{+}k_1]\, B_1, \ldots, [S{+}k_n]\, B_n) \in P$ with free variables $\bar{X} = X_1, \ldots, X_m$. Then the following are equivalent:*

- $\mathcal{I} \models [S{+}k_0]\, H \leftarrow [S{+}k_1]\, B_1, \ldots, [S{+}k_n]\, B_n$
- $\mathcal{I} \models \forall \bar{X}\, (\circ^{k_0} H \leftarrow \circ^{k_1} B_1, \ldots, \circ^{k_n} B_n)$
- $\mathcal{I}[0] \models \Box\, (\forall \bar{X}\, (\circ^{k_0} H \leftarrow \circ^{k_1} B_1, \ldots, \circ^{k_n} B_n))$.

Note that this temporal semantics of Statelog rules only specifies a weak criterion for rejecting interpretations $\mathcal{I}$, which do not satisfy all rules of a program. However, it does not allow to select a single intended model from the set of all models. In order to obtain a canonical model, one has to apply semantics and notions developed for logic programs and deductive databases.

6.4.4 State-Stratified Semantics

We wish to select a single "intended" or canonical model $\mathcal{M}_{P \cup \mathcal{D}}$ from the set of possible models of a Statelog program P with database $\mathcal{D}$. As shown in Sections 6.3.1 and 6.4.2, Statelog programs can be conceived as specific logic programs. Therefore, the standard declarative semantics for logic programs, i.e., the stratified, well-founded, and stable semantics can be applied. As it turns out, the usual definition for stratification based on relation names is too restrictive to be useful for Statelog:

Consider the program P from Example 6.1. P is not stratified since there are negative cyclic dependencies between rules (e.g., via del_emp we have $r_1 \stackrel{\neg}{\rightarrow} r_2 \rightarrow r_1$). On the other hand, the negative cycles are not present *within* a single state but rather *across* different states: in r_2, $[S{+}1]$ emp negatively depends on $[S]$ del_emp which in turn depends on $[S]$ emp by r_1. Thus, $[S{+}1]$ emp does not depend negatively on itself, but on the previous value at $[S]$.

In order to handle such situations, we extend stratification to *state-stratification*. Intuitively, a Statelog program is called state-stratified if there are no negative cyclic dependencies within a state. As will be shown below, together with progressiveness this implies local stratification. Although local stratification is undecidable for logic programs in general [CB94], for Statelog programs it is decidable, and a state-stratification can be computed in essentially the same way as the stratification of a Datalog$^\neg$ program.

As will be seen later, the use of the well-founded semantics instead of the state-stratified semantics does not yield an increase of expressive power for Statelog transactions.[5] Therefore, we choose the conceptually simpler 2-valued state-stratified semantics as the "standard" Statelog semantics. In Section 6.7, we investigate the well-founded semantics for Statelog programs, and also briefly comment on stable semantics.

Definition 6.8 (State-Stratification) The *labeled dependency graph* $\mathcal{LG}_P = (V, E)$ of a Statelog program P is defined as follows:

[5] However, wrt. *transitions* the expressiveness is increased; see Proposition 6.7.

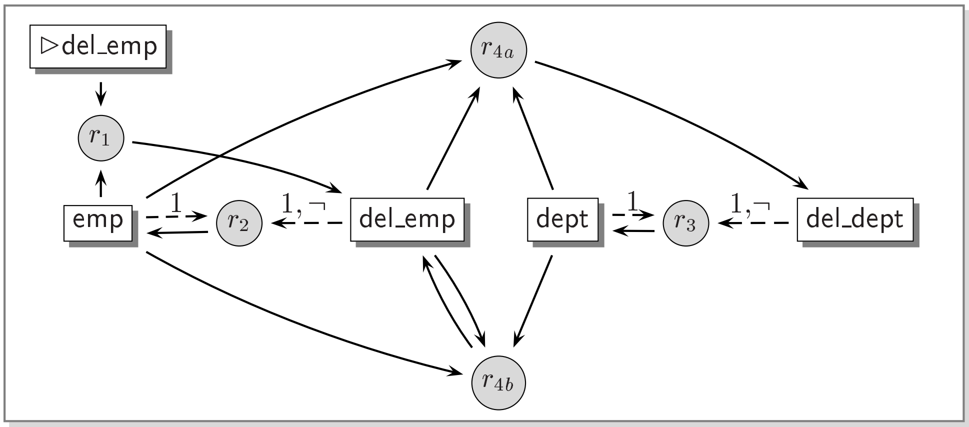


Figure 6.2: A labeled dependency graph

- For each rule $r \in P$ and each relation symbol $p \in \mathbf{P}$, there are vertices r, $\boxed{p} \in V$.
- For each rule $r \in P$ of the form

$$\begin{array}{lll} r: & [S{+}k_0]\, h(\bar{X}_0) \;\leftarrow & [S{+}k_1]\, b_1(\bar{X}_1),\; \ldots\;, [S{+}k_n]\, b_n(\bar{X}_n), \\ & & [S{+}k_{n+1}]\, \neg\, b_{n+1}(\bar{X}_{n+1}), \ldots, [S{+}k_m]\, \neg\, b_m(\bar{X}_m)\ , \end{array}$$

  E contains the labeled edges

  - $r \xrightarrow{0} \boxed{h}$,
  - $\boxed{b_i} \xrightarrow{l_i} r$, for all $i = 1, \ldots, n$,
  - $\boxed{b_i} \xrightarrow{l_i,\neg} r$, for all $i = n{+}1, \ldots, m$.

The *leap* $l_i := k_0 - k_i$ of the edge $\boxed{b_i} \xrightarrow{l_i,(\neg)} r$ is the difference between the state referred to in the head, and the state of the corresponding body literal. An edge marked with "¬" is a *negative* edge, otherwise a *positive edge*.

A cycle C of $\mathcal{LG}_P$ where the sum $\sum_{b \overset{l,(\neg)}{\to} r \in C} l$ of leaps equals 0 is called a *local cycle*. A rule r occurring in a non-local cycle is called *progressively recursive*; if r occurs only in local cycles, it is called *locally recursive*.

P is called *state-stratified*, if P is progressive and no local cycle of $\mathcal{LG}_P$ contains a negative edge. □

In order to determine whether P is state-stratified, one could define $\mathcal{LG}_P$ using rules (or alternatively relation symbols) only, as has been done in Definition 5.1 for S-Datalog. Here, we chose the more explicit variant which includes both rules and relation symbols.

Example 6.2 (Labeled Dependency Graph) Consider the Statelog program P from Example 6.1. The labeled dependency graph $\mathcal{LG}_P$ is shown in Figure 6.2. Edges with leap $l = 0$ are depicted by solid lines, whereas dashed lines are used for transitions to the successor state, i.e., for edges with $l = 1$. Notice that r_4 has multiple heads and is shorthand for two rules r_{4a}

and r_{4b}. Since $\mathcal{LG}_P$ contains negative cycles, P is not stratified. However, P is state-stratified since neither of these is a local cycle.[6] $\square$

Local stratification generalizes stratification by considering the ground instances of atoms rather than relation names (cf. Definition 5.1). To better distinguish local stratification from stratification based on relation names, the latter is sometimes called *predicate stratification*.

Definition 6.9 (Local Stratification)
Let P be logic program, $r, r' \in ground(P)$ of the form

$$\begin{array}{lll} r: & \cdots & \leftarrow \cdots\ (\neg)\ p(\bar{x})\ \cdots \\ r': & p(\bar{x}) & \leftarrow \cdots \end{array}$$

Then r is said to *depend positively* (*negatively*) on r', denoted $r' \to r$ ($r' \stackrel{\neg}{\to} r$). The *dependency graph* $\mathcal{G}_{ground(P)}$ consists of all positive and negative dependencies between rules in $ground(P)$. We write $r' \rightsquigarrow r$, if there is a path from r' to r in $\mathcal{G}_{ground(P)}$, and $r' \stackrel{\neg}{\rightsquigarrow} r$, if the path contains a negative dependency.

P is called *locally stratified* if there is a (generally infinite) partition $P_1, P_2, \ldots$ of $ground(P)$ such that for all $i, j \geq 1$ and all $r' \in P_i$, $r \in P_j$:

- if $r' \rightsquigarrow r$ then $i \leq j$, and
- if $r' \stackrel{\neg}{\rightsquigarrow} r$ then $i < j$. $\square$

For progressive rules, the notion of state-stratification "almost" captures local stratification. However, a rule like $[S]\,p(a) \leftarrow [S]\,\neg\,p(b)$ is locally stratified but not state-stratified, because state-stratification is defined using relation symbols and state terms only but does not consider data constants. Therefore, w.l.o.g., we assume that Statelog rules are *rectified*, i.e., do not contain any constants.[7] Note that we may identify a Statelog program P with its reified version $P^\star$ (Section 6.4.2). Now we are ready to prove:

Theorem 6.4 (State-Stratification vs. Local Stratification)
Let P be a rectified progressive Statelog program, including a database $\mathcal{D}$ as a set of facts. Then: P is state-stratified $\Leftrightarrow$ P is locally stratified.

Proof We can assume that all rules of P are progressive.

($\Rightarrow$) Suppose P be state-stratified. Let $P^n \subseteq ground(P)$ denote the (finite) set of ground rules of the form $[n]\,p(\bar{x}) \leftarrow \ldots$ defining state $[n]$. Notice that rules of P^n solely depend on rules from P^m for $m \leq n$. We show how to obtain a partition $P^n = P^n_1 \dot\cup \cdots \dot\cup P^n_{k_n}$ such that the rules in P^n_j depend positively (negatively) only on rules from P^n_i for $i \leq j$ ($i < j$): Consider the dependency graph $\mathcal{G}_{P^n}$. Let P_{scc} be a strongly connected component of $\mathcal{G}_{P^n}$, i.e., a maximal subset of rules from P^n such that for all $r_1, r_2 \in P_{scc}$, r_1 and r_2 are mutually dependent. Assume P_{scc} contains a negative edge. Then $r \stackrel{\neg}{\rightsquigarrow} r$ for some $r \in P_{scc}$. Due to progressiveness, all state terms occurring in the path $r \stackrel{\neg}{\rightsquigarrow} r$ must be equal to n. Hence, there is a negative local cycle in $\mathcal{LG}_P$, contradicting the assumption that P is state-stratified. Therefore, no scc

[6] It should be clear that the notion "local" is overloaded and used in different—albeit related—ways, i.e., to refer to the well-known notion of *local stratification* [Prz88], and on the other hand, when we speak of *local rules* and *local cycles*, to indicate that a rule or a cyclic dependency occurs *within* a state.

[7] The simplest encoding is to use a new relation for each constant, s.t. $p_c(X)$ holds iff $X = c$.

of $\mathcal{G}_{P^n}$ contains a negative edge, and a topological sort of the scc's yields the desired partition $P_1^n, \cdots, P_{k_n}^n$ of P^n. Finally, the local stratification is given by $P_1^0, \cdots, P_{k_0}^0, P_1^1, \cdots, P_{k_1}^1, \cdots$

($\Leftarrow$) Suppose P is not state-stratified. Then $\mathcal{LG}_P$ contains a local cycle $C := r_1 \to p_1 \to \cdots \to r_m \to p_m \to r_1$ such that at least one edge in C is negative. Consider the ground instances $r'_1, \ldots, r'_m$ of $r_1, \ldots, r_m$ in which all variables have been substituted by some fixed $c \in U_P$.[8] Since P is rectified, the rules $r'_1, \ldots, r'_m$ induce a negative cyclic dependency C' in $\mathcal{G}_{ground(P)}$. It follows that P is not locally stratified. ■

In order to prove that state-stratification implies local stratification, we have used the fact that Statelog rules are progressive. Indeed, for non-progressive rules, the absence of a negative local cycle is not sufficient to imply local stratification:

Example 6.3 (Non-Progressive Rules) Neither of the two programs

$$P_1: \quad \boxed{[S{+}1]\ \mathsf{p} \leftarrow [S]\ \neg\,\mathsf{p}\ .} \qquad \text{and} \qquad P_2: \quad \boxed{[S]\ \mathsf{p} \leftarrow [S{+}1]\ \neg\,\mathsf{p}\ .}$$

contains a negative local cycle. Note that in P_1 the truth-value of $[n]\,\mathsf{p}$ depends negatively on the *founded* sequence $[n{-}1], [n{-}2], \ldots, [0]$, whereas in P_2 it depends negatively on the *unfounded* sequence $[n{+}1]$, $[n{+}2]$, $\ldots$ Thus, P_1 is locally stratified and has a unique intended model $\{[0]\,\neg\,\mathsf{p}, [1]\,\mathsf{p}, [2]\,\neg\,\mathsf{p}, \ldots\}$ (which is also the well-founded and the unique stable model). In contrast, P_2 is not locally stratified and the intended semantics of P_2 is unclear. P_2 has two stable models $\{[0]\,\neg\,\mathsf{p}, [1]\,\mathsf{p}, [2]\,\neg\,\mathsf{p}, \ldots\}$ and $\{[0]\,\mathsf{p}, [1]\,\neg\,\mathsf{p}, [2]\,\mathsf{p}, \ldots\}$; under the well-founded semantics $[n]\,\mathsf{p}$ is *undefined* for all n. □

It is well known that every locally stratified program has a unique *perfect model* [Prz88]. Roughly speaking, the perfect model corresponds to the stratified model of the ground instantiated program. Since this model is universally accepted as the intended semantics of locally stratified programs, we use it as the canonical model for state-stratified programs. Together with Theorem 6.4 we immediately get:

Corollary 6.5 (Canonical Model $\mathcal{M}_{P\cup\mathcal{D}}$) *For every state-stratified Statelog program P and database $\mathcal{D}$, $P \cup \mathcal{D}$ has a unique perfect model $\mathcal{M}_{P\cup\mathcal{D}}$.*

The (local) stratification of a program not only avoids semantical problems due to negation, but also induces a partial order on rules which may be used in the bottom-up evaluation of the perfect model $\mathcal{M}_{P\cup\mathcal{D}}$: In order to compute $\mathcal{M}_{P\cup\mathcal{D}}[n]$, the rules defining the current state $[n]$ are applied according to the state-stratification, until $\mathcal{M}_{P\cup\mathcal{D}}[n]$ becomes stationary. Then the rules defining $\mathcal{M}_{P\cup\mathcal{D}}[n{+}1]$ have to be applied. This process is repeated until a fixpoint is reached.[9]

Example 6.4 (Integrity Enforcement, Cont'd) Consider the state-stratified program P from Example 6.1 and its labeled dependency graph $\mathcal{LG}_P$ in Figure 6.2. Careful inspection of $\mathcal{LG}_P$ shows that the rules defining state $[n]$ have to be applied in the order

$$\{r_2, r_3\} \prec \{r_1\} \prec \{r_{4a}, r_{4b}\}\ .$$

[8]As usual, we assume that U_P contains at least one constant.

[9]We postpone details of computing $\mathcal{M}_{P\cup\mathcal{D}}$ until Section 6.8, where we can make use of Statelog programs in normal form.

Note that the 1-progressive rules $\{r_2, r_3\}$ do not derive information for [0]. Assume the initial database state depicted in Figure 4.1 is given as a Statelog database, i.e.,

$$\mathcal{D} := \{\ [0]\,\mathsf{emp(ed1,7000,d1)},\ \ldots\ ,\mathsf{emp(md3,9000,d3)},\\ [0]\,\mathsf{dept(d1,md1)},\ [0]\,\mathsf{dept(d2,md2)},\ [0]\,\mathsf{dept(d3,md3)}\ \}.$$

By adding the fact $[0]\,\triangleright\mathsf{del\_emp(md1)}$ to $\mathcal{D}$, we model the occurrence of an external request to delete md1. This allows to derive del_emp(md1,10000,d1) by r_1. Then r_{4a} and r_{4b} compute *within* [0] all employees and departments which are directly or indirectly affected by the initial deletion of md1. Thus, the final state

$$\mathcal{M}_{P\cup\mathcal{D}}[\$]\ = \{\mathsf{emp(md3,9000,d3)}, \mathsf{dept(d3,md3)}\}$$

is already reached after the first transition (so \$=1). In contrast, if we replace r_4 by

$$\begin{array}{ll} r_4'\text{:} & [S{+}1]\ \mathsf{del\_dept(D,E),\ del\_emp(E1,S1,D)} \leftarrow \\ & \quad [S]\ \mathsf{del\_emp(E,\_,\_),\ dept(D,E)},\ [S{+}1]\ \mathsf{emp(E1,S1,D)} \end{array}$$

then new delete requests are processed one state after another, resulting in the following sequence of delta relations:

state	del_emp	del_dept
[0]	(md1, 10000, d1)	∅
[1]	(ed1,7000,d1), (md2,1000,d1)	(d1,md1)
[2]	(ed2,6000,d2), (md2,8000,d2)	(d2,md2)
[3]	∅	∅

In this case, the final state $\mathcal{M}_{P\cup\mathcal{D}}[\$]$ is reached after three transitions. □

$\mathcal{M}_{P\cup\mathcal{D}}$ does not always terminate (e.g., P_1 in Example 6.3 above), i.e., the sequence $(\mathcal{M}_{P\cup\mathcal{D}}[i])_{i\in\mathbb{N}_0}$ need not become stationary. Nevertheless, $\mathcal{M}_{P\cup\mathcal{D}}$ can be finitely represented due to its periodic structure:

Theorem 6.6 (Periodicity of $\mathcal{M}_{P\cup\mathcal{D}}$) *The perfect model $\mathcal{M}_{P\cup\mathcal{D}}$ of a state-stratified program P and database $\mathcal{D}$ is ultimately periodic.*

Proof Let w be the maximal leap occurring in a given Statelog program P. For every n, the snapshot $\mathcal{M}_{P\cup D}[n]$ is uniquely determined by the window of w predecessor states $\mathcal{M}_{P\cup\mathcal{D}}[n{-}w..n{-}1]$ (consider the subset of rules of *ground*$(P\cup\mathcal{D})$ which define $\mathcal{M}_{P\cup\mathcal{D}}[n]$, and assume $\mathcal{M}_{P\cup\mathcal{D}}[n{-}w..n{-}1]$ is given as a set of facts). Therefore, $\mathcal{M}_{P\cup\mathcal{D}}$ is a w-deterministic Statelog interpretation (cf. Definition 6.6). By Proposition 6.1 it follows that $\mathcal{M}_{P\cup\mathcal{D}}$ is ultimately periodic. ■

For nondeterministic Statelog extensions, periodicity may not be guaranteed, e.g., if an arbitrary stable model is chosen as the intended model:

Example 6.5 (Non-Stratified Rules) Consider the Statelog program

$$P\text{:}\quad \boxed{\begin{array}{ll} r_1\text{:} & [S]\ \mathsf{p} \leftarrow [S]\ \neg\,\mathsf{q}. \\ r_2\text{:} & [S]\ \mathsf{q} \leftarrow [S]\ \neg\,\mathsf{p}. \end{array}}$$

$\mathcal{LG}_P$ contains the negative local cycle $r_1 \overset{\neg}{\rightsquigarrow} r_2 \overset{\neg}{\rightsquigarrow} r_1$, so P is not state-stratified (and thus not locally stratified). The well-founded semantics assigns *undefined* to p and q in every state $[n]$, hence yields a periodic 3-valued model. In contrast, the nondeterministic stable semantics allows to choose between two stable snapshots $\{\mathsf{p}, \neg\,\mathsf{q}\}$ and $\{\mathsf{q}, \neg\,\mathsf{p}\}$ for every state $[n]$, independent from the choice at any other state $[m]$. Thus, P has stable models which are not periodic.[10] □

It should be clear that the set of Statelog atoms which are true in some or all stable models of P, respectively, define w-deterministic—and thus periodic—Statelog interpretations $\mathcal{I}_P^{\exists}$ and $\mathcal{I}_P^{\forall}$. For P in Example 6.5, for instance, we have $\mathcal{I}_P^{\exists}[n] = \{\mathsf{p}, \mathsf{q}\}$ and $\mathcal{I}_P^{\forall}[n] = \emptyset$, for all $n \in \mathbb{N}_0$. Observe that neither $\mathcal{I}_P^{\exists}$ nor $\mathcal{I}_P^{\forall}$ is a stable model of P ($\mathcal{I}_P^{\forall}$ is not even a model): as usual, union and intersection of stable models do *not* yield (stable) models.

6.5 Normal Forms and Expressive Power

In the remainder of this chapter we investigate fundamental properties of Statelog programs like expressive power, complexity, and termination behavior. A very useful tool will be normal forms of Statelog programs. This allows to characterize the expressive power and complexity of Statelog and to compare it with existing approaches like the production rule semantics for Datalog$^{\neg}$. Moreover, by considering Statelog programs in normal form, the canonical model $\mathcal{M}_{P \cup \mathcal{D}}$ can be computed using standard fixpoint techniques developed for the bottom-up evaluation of deductive and production rules.

Definition 6.10 (NF-Statelog, P-Statelog) By *NF-Statelog* we denote the class of Statelog program in *normal form*, i.e., where every rule is of the form

$$\begin{array}{lrcl} (P): & [S{+}1]\,\mathit{head} & \leftarrow & [S]\,\mathit{body}\ ,\ \text{or} \\ (L): & [S]\,\mathit{head} & \leftarrow & [S]\,\mathit{body}\ . \end{array}$$

The class *P-Statelog* of Statelog programs in *P-normal form* consists of all programs comprising only rules of type (P). □

Basically, rules of type (L) correspond to S-Datalog rules, whereas rule of type (P) correspond to production rules in the style of P-Datalog; see Section 6.6 for a detailed investigation.

6.5.1 Different Notions of Equivalence

In the sequel, we will show that Statelog programs can be equivalently rewritten into certain normal forms. Beforehand we identify several natural notions of equivalence which arise in the context of state-oriented languages like Statelog or XY-Datalog (cf. Section 6.10.2).

Recall the basic execution model from Figure 6.1: It allows to distinguish between transitions, transactions, and the database evolution. Intuitively, the *transition expressiveness* characterizes the class of db-transformations definable by a single transition from $[n]$ to $[n{+}1]$. Since this class is independent of the particular choice of n, w.l.o.g., we restrict to transitions from $[0]$ to $[1]$. The *transaction expressiveness* corresponds to the class of definable mappings from the initial state $[0]$ to the final state $[\$]$. Transactions are the Statelog analogue

[10] Encode the snapshots $\{\mathsf{p}, \neg\,\mathsf{q}\}$ and $\{\mathsf{q}, \neg\,\mathsf{p}\}$ by 0 and 1, respectively. Then the stable models of P are precisely the infinite strings (or ω-words, cf. [Tho90]) $\{0, 1\}^{\omega}$.

to "classical" db-transformations, since they describe the overall effect of a program P on a given database $\mathcal{D}$. Note that the transaction expressiveness characterizes the expressive power of the language as a whole. Therefore, if we compare the expressive power of Statelog with conventional "one-state" languages, we use transaction expressiveness. Finally, the *evolution expressiveness* is the class of database evolutions definable in the language:

Definition 6.11 (Statelog Database Mappings) Let $\mathbf{R}$ be a fixed schema. With every Statelog program P over the schema $\mathbf{P} \supseteq \mathbf{R}$ we associate the following mappings:

- $\Pi^1_P : \; inst(\mathbf{R}) \rightarrow inst(\mathbf{R}), \quad \mathcal{D}[0] \mapsto \mathcal{M}_{P\cup\mathcal{D}}[1]\,|_{\mathbf{R}}$ (*Statelog transition*)
- $\Pi^\$_P : \; inst(\mathbf{R}) \rightarrow inst(\mathbf{R}), \quad \mathcal{D}[0] \mapsto \mathcal{M}_{P\cup\mathcal{D}}[\$]\,|_{\mathbf{R}}$ (*Statelog transaction*)
- $\Pi^\infty_P : \; inst(\mathbf{R}) \rightarrow (\mathbb{N}_0 \rightarrow inst(\mathbf{R})), \quad \mathcal{D}[0] \mapsto (\mathcal{M}_{P\cup\mathcal{D}}[n]\,|_{\mathbf{R}})_{n\in\mathbb{N}_0}$ (*Statelog evolution*)

Two programs P_1 and P_2 over the schemas $\mathbf{P_1}, \mathbf{P_2} \supseteq \mathbf{R}$ are called *transition equivalent*, denoted $P_1 \equiv_1 P_2$, if $\Pi^1_{P_1} = \Pi^1_{P_2}$, *transaction equivalent*, denoted $P_1 \equiv_\$ P_2$, if $\Pi^\$_{P_1} = \Pi^\$_{P_2}$, and *evolution equivalent*, denoted $P_1 \equiv_\infty P_2$, if $\Pi^\infty_{P_1} = \Pi^\infty_{P_2}$.

The *transition*, *transaction*, and *evolution expressiveness* of a language is the class of transitions, transactions and evolutions definable in the language. □

Note that the relations in $\mathbf{P} \setminus \mathbf{R}$ are regarded as auxiliary relations which do not belong to the "answer" computed by P. Moreover, there is no distinction between *edb* and *idb* relations, so far. In fact every relation which does not occur in the head of a rule, will be empty in the final state [\$]. Hence, also relations which are usually considered as *edb* relations have to occur in rule heads and must be propagated across states using so-called *frame rules*. We will elaborate on the several types of relations in Section 7.2.

From the previous definitions it is clear that evolution equivalence implies transaction equivalence, i.e., $P_1 \equiv_\infty P_2 \Rightarrow P_1 \equiv_\$ P_2$. However, the converse generally does not hold:

Example 6.6 ($P_1 \equiv_\infty P_2 \not\Leftarrow P_1 \equiv_\$ P_2$) The programs

P_1:

```
[S] reachable(Y)  ←  [S] Y=a.                    % compute reachable
[S] reachable(Y)  ←  [S] reachable(X),e(X,Y).    % ... nodes locally
    [S] del_e(X,Y) ←  [S] reachable(X),e(X,Y).    % ... and delete
```

and

P_2:

```
[S+1] del_e(a,X)  ←  [S] e(a,X).                 % delete reachable nodes
[S+1] del_e(Y,Z)  ←  [S] del_e(X,Y),e(Y,Z).      % ... state by state
```

are transaction equivalent, but not evolution equivalent: In P_1, the view reachable is used to delete from e all edges which are reachable from node a. Thus, the final result is obtained after a single transition. In contrast, P_2 needs n transitions, where n is the maximal length of a finite path starting at a, in order to compute the final result via intermediate states.

Here, we have tacitly assumed that a *frame rule*

$$[S{+}1]\ \mathsf{e(X,Y)} \leftarrow [S]\ \mathsf{e(X,Y)}, \neg\,\mathsf{del\_e(X,Y)}$$

is added to both P_1 and P_2, which is used to propagate the edge-relation e—modulo deletions—through all states (cf. Section 7.4). □

6.5.2 From Statelog to NF-Statelog

In the following, we show that for every Statelog program P there is an evolution equivalent (and thus transaction equivalent) NF-Statelog program P' (here, equivalence is wrt. the schema $\mathbf{P}$ of the original program P). Therefore, NF-Statelog programs retain the full expressiveness of Statelog, even wrt. intermediate transitions. In contrast, as will be proven in Section 6.5.3, one can find for every Statelog program a *transaction* equivalent P-Statelog program, but in general no *evolution* equivalent P-Statelog program. Thus, although P-Statelog has the same overall expressiveness as Statelog, it is generally not powerful enough to preserve intermediate transitions. We need some preliminaries:

Definition 6.12 (k-Extension) Fix $k \in \mathbb{N}$. Given a Statelog program P over the schema $\mathbf{P}$, we define its *k-extension* P' as follows: The new schema $\mathbf{P}'$ contains all $p \in \mathbf{P}$, and additionally for each of these and every $i = 1, \ldots, k$ a new relation symbol p^{-i} of the same arity as p. The rules of P' are as follows:

$$\begin{array}{rlcl} P' := P \cup \{ & [S{+}1]\, p^{-1}(\bar{X}) & \leftarrow & [S]\, p(\bar{X}) \;, \\ & [S{+}1]\, p^{-2}(\bar{X}) & \leftarrow & [S]\, p^{-1}(\bar{X}) \;, \\ & & \vdots & \\ & [S{+}1]\, p^{-k}(\bar{X}) & \leftarrow & [S]\, p^{-k+1}(\bar{X}) \;\mid\; p \in \mathbf{P} \;\} \,. \end{array}$$ □

Observe that the new relation symbols hold "delayed versions" of p:

Lemma 6.7 (k-Extension) *Let P' be the k-extension of P for some $k \in \mathbb{N}$. Then for all relations $p \in \mathbf{P}$, databases $\mathcal{D}$, $n \in \mathbb{N}_0$, and $i = 1, \ldots, k$:*

$$\mathcal{M}_{P' \cup \mathcal{D}} \models [n{+}i]\, p^{-i}(\bar{x}) \;\Leftrightarrow\; \mathcal{M}_{P' \cup \mathcal{D}} \models [n]\, p(\bar{x}).$$

Proof The direction ($\Leftarrow$) is established using the new rules of P' which define p^{-i}. The direction ($\Rightarrow$) follows from the fact, that these are the only rules defining p^{-i}. ■

The next lemma shows how to enforce that Statelog rules are only "enabled" for certain states; this will be useful in subsequent proofs.

Lemma 6.8 ($\mathbf{eq}_m$, $\mathbf{geq}_m$) *Let $k \in \mathbb{N}$ be fixed. There is a Statelog program P with schema $\mathbf{P} = \{\mathsf{eq}_i, \mathsf{geq}_i \mid i = 0, \ldots, k\}$ such that for all $n \leq k$ and all databases $\mathcal{D}$:*

- $\mathcal{M}_{P \cup \mathcal{D}} \models [n]\, \mathsf{eq}_m$ *iff* $n = m$, *and*
- $\mathcal{M}_{P \cup \mathcal{D}} \models [n]\, \mathsf{geq}_m$ *iff* $n \geq m$.

Proof Consider the following program P:

$$[S]\, \mathsf{geq}_0 \leftarrow [S]\, \mathsf{eq}_0.$$
$$[S{+}1]\, \mathsf{geq}_i \leftarrow [S]\, \mathsf{geq}_i \parallel i \leq k.$$
$$[S{+}1]\, \mathsf{geq}_{i+1} \leftarrow [S]\, \mathsf{geq}_i \parallel i \leq k.$$
$$[S]\, \mathsf{eq}_0 \leftarrow [S]\, \neg\, \mathsf{geq}_1.$$
$$[S{+}1]\, \mathsf{eq}_{i+1} \leftarrow [S]\, \mathsf{eq}_i \parallel i \leq k.$$

Note that P is state-stratified but not stratified! ■

Notation. Above and in the sequel, for a rule r and a condition ϕ, the rule schema $r \parallel \phi$ denotes the set $\{r \mid \phi\}$ of all rules for which ϕ holds.

Now we are prepared to establish the first main theorem, which shows that it is sufficient to consider Statelog programs in normal form. As a consequence of Theorem 6.9, we often tacitly assume that a Statelog program P is in normal form.

Theorem 6.9 (Statelog $\equiv_\infty$ NF-Statelog)
For every Statelog program there is an evolution equivalent NF-Statelog program.

Proof Let P be a Statelog program, and $r \in P$ a rule of the form

$$[S{+}k_0]\,H \;\leftarrow\; [S{+}k_1]\,B_1,\;\ldots\;,[S{+}k_n]\,B_n.$$

Since we are only dealing with progressive programs, $k_0 \geq k_i$ for all $i = 1, \ldots, n$. We have to consider two cases:

Case I: $k_0 > k_i$, i.e., r is strictly progressive. Let $\delta_i := k_0 - 1 - k_i$ for $i = 1, \ldots, n$. The original rule may be equivalently rewritten as follows:

$$\begin{array}{rrcl}
 & [S{+}k_0]\,H & \leftarrow & [S{+}k_1]\,B_1,\;\ldots\;,[S{+}k_n]\,B_n \\
\stackrel{(Ia)}{=} & [S{+}k_0]\,H & \leftarrow & [S{+}k_0{-}1{-}\delta_1]\,B_1,\;\ldots\;,[S{+}k_0{-}1{-}\delta_n]\,B_n \\
\stackrel{(Ib)}{\Leftrightarrow} & [S{+}k_0]\,H & \leftarrow & [S{+}k_0{-}1]\,B_1^{-\delta_1},\;\ldots\;,[S{+}k_0{-}1]\,B_n^{-\delta_n} \\
\stackrel{(Ic)}{\Leftrightarrow} & [S{+}1]\,H & \leftarrow & [S]\,B_1^{-\delta_1},\;\ldots\;,[S]\,B_n^{-\delta_n},[S]\,\mathsf{geq}_{k_0-1}\;.
\end{array}$$

(Ia) is immediate, since $k_i = k_0 - 1 - \delta_i$. (Ib) is valid by Lemma 6.7. Finally, Lemma 6.8 and the fact that $k_0 \geq 1$ establish the correctness of step (Ic).

Case II: $k_0 \geq k_i$, and $k_0 = k_j$ for some j. Thus, r is progressive but not strictly progressive. Let $\delta_i := k_0 - k_i$ for $i = 1, \ldots, n$. Now the original rule may be equivalently rewritten as follows:

$$\begin{array}{rrcl}
 & [S{+}k_0]\,H & \leftarrow & [S{+}k_1]\,B_1,\;\ldots\;,[S{+}k_n]\,B_n \\
\stackrel{(IIa)}{=} & [S{+}k_0]\,H & \leftarrow & [S{+}k_0{-}\delta_1]\,B_1,\;\ldots\;,[S{+}k_0{-}\delta_n]\,B_n \\
\stackrel{(IIb)}{\Leftrightarrow} & [S{+}k_0]\,H & \leftarrow & [S{+}k_0]\,B_1^{-\delta_1},\;\ldots\;,[S{+}k_0]\,B_n^{-\delta_n} \\
\stackrel{(IIc)}{\Leftrightarrow} & [S]\,H & \leftarrow & [S]\,B_1^{-\delta_1},\;\ldots\;,[S]\,B_n^{-\delta_n},[S]\,\mathsf{geq}_{k_0}\;.
\end{array}$$

Again, (IIa) is immediate due to the definition of δ_i, while (IIb) is implied by Lemma 6.7. In (IIc), Lemma 6.8 is used. ∎

It is important to note that the above transformation steps used to establish Theorem 6.9 remain valid also for other declarative semantics like the well-founded or stable semantics.[11] Since the whole database evolution is preserved by the rewritten NF-Statelog program, it is justified to consider only Statelog programs in normal form.

6.5.3 From NF-Statelog to P-Statelog

In this section, we show that NF-Statelog programs can be further normalized by eliminating local rules. The basic idea is to simulate local recursion by *progressive recursion*, i.e., by iterating 1-progressive rules. The resulting P-Statelog programs have the same transaction expressiveness as Statelog, but are strictly less expressive wrt. (intermediate) transitions, as will be seen in Section 6.6.3.

[11] Indeed, it is hard to imagine a semantics where the presented transformations are *not* valid.

Theorem 6.10 (Statelog $\equiv_\$$ P-Statelog)
For every Statelog program there is a transaction equivalent P-Statelog program.

PROOF P-Statelog $\leq_\$$ Statelog holds trivially. To establish the converse direction, let P be a Statelog program. Due to Theorem 6.9 we may assume that P is in normal form, i.e., $P \in$ NF-Statelog. We construct a P-Statelog program P' which simulates the local rules of P: Since P is state-stratified, the set of local rules P_l can be partitioned into strata $P_1, \dots, P_n$. Let $P_{n+1} = P \setminus P_l$ be the set of 1-progressive rules. The evaluation of P can be conceived as two nested fixpoints: the inner fixpoint is computed locally in each state using the rules P_l, the outer fixpoint is computed using state transitions and the 1-progressive rules P_{n+1}. In order to simulate the inner fixpoint using 1-progressive rules, we have to "stretch" the local iteration across different states. As a consequence of this, transition equivalence is lost! The simulation is accomplished as follows (in Figure 6.3, each box, e.g., $\boxed{\mathsf{a}_i}$ denotes a state in the simulation P', whereas $[S]$, $[S{+}1]$, and $[S{+}2]$ correspond to the original state sequence computed by P):

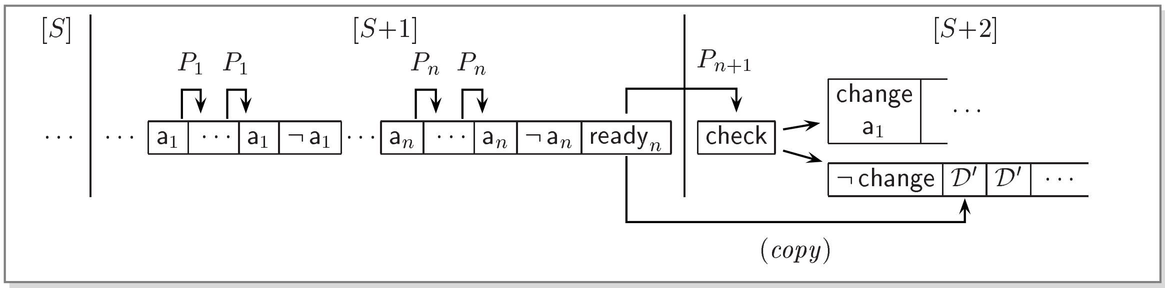


Figure 6.3: Simulation of local rules ($\mathsf{a}_i \mathrel{\widehat{=}} \mathsf{active}_i$)

For each stratum i we use an auxiliary relation active_i to specify that stratum i is currently evaluated or "active". Initially, active_1 holds. In order to allow for comparison of the current with the previous state, active_i has to be true for at least two successive states. While there are changes in the current stratum, active_i remains true. Because the computation within each stratum is monotone, it is sufficient to check whether some $p(\bar{x})$ is true now, and was false in the previous state. (Recall that $[s]\,p^{-1}$ iff $[s{-}1]\,p$ by Lemma 6.7). As soon as active_i changes from true to false, the next stratum $i{+}1$ is activated:

$$
\begin{array}{ll}
[S{+}1]\,\mathsf{active}_1 \leftarrow [S]\,\mathsf{eq}_0. & \\
[S{+}1]\,\mathsf{active}_i \leftarrow [S]\,\neg\,\mathsf{active}_i^{-1}, \mathsf{active}_i \parallel i = 1, \dots, n. & \textit{\% at least 2 active states} \\
[S{+}1]\,\mathsf{active}_i \leftarrow [S]\,\mathsf{active}_i, \neg\,p^{-1}(\bar{X}), p(\bar{X}) \parallel p \in \mathbf{P}_i, i = 1, \dots, n. & \textit{\% changes?} \\
[S{+}1]\,\mathsf{active}_{i+1} \leftarrow [S]\,\mathsf{active}_i^{-1}, \neg\,\mathsf{active}_i \parallel i = 1, \dots, n-1. & \textit{\% next stratum}
\end{array}
$$

The main evaluation step is the application of the local rules from stratum i as long as active_i is true:

$$[S{+}1]\,H \leftarrow [S]\,B, \mathsf{active}_i \parallel ([S]\,H \leftarrow [S]\,B) \in P_i, \text{ for } i = 1, \dots, n.$$

While the states are computed stratum by stratum, all intermediate results are propagated through the strata.[12] Here, p_{old} holds a copy of the original value of p. It will be used to check for termination after the n-th stratum has been completed:

[12]Note again, that stratified evaluation is monotone: once a fact has been derived, it remains true until all strata have been completed.

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \mathsf{ready}_n \parallel p \in \mathbf{P}. \qquad \textit{\% propagate current}$$
$$[S{+}1]\, p_{old}(\bar{X}) \leftarrow [S]\, p_{old}(\bar{X}), \neg\, \mathsf{ready}_n \parallel p \in \mathbf{P}. \qquad \textit{\% ... and old values}$$

If the n-th stratum is finished, ready_n is derived and an intermediate state check is created. The non-local rules in P_{n+1} are applied in the transition from ready_n to check:

$$[S{+}1]\, \mathsf{ready}_n \leftarrow [S]\, \mathsf{active}_n^{-1}, \neg\, \mathsf{active}_n.$$
$$[S{+}1]\, \mathsf{check} \leftarrow [S]\, \mathsf{ready}_n.$$
$$[S{+}1]\, H \leftarrow [S]\, B, \mathsf{ready}_n \parallel ([S{+}1]\, H \leftarrow [S]\, B) \in P_{n+1}.$$

If there have been changes in the values of p, this is noted in the 0-ary relation change, and the first stratum is activated again. Note that in general p may decrease wrt. the previous value p_{old} (since non-local rules may add new tuples to some relation q which is used negatively in a local rule):

$$[S{+}1]\, \mathsf{change}, \mathsf{active}_1 \leftarrow [S]\, \mathsf{check}, p(\bar{X}), \neg\, p_{old}(\bar{X}) \parallel p \in \mathbf{P}.$$
$$[S{+}1]\, \mathsf{change}, \mathsf{active}_1 \leftarrow [S]\, \mathsf{check}, p_{old}(\bar{X}), \neg\, p(\bar{X}) \parallel p \in \mathbf{P}.$$

In the transition from check to the successor state, p_{old} is set to the current value of p. Moreover, if there have been no changes, the final result is copied:

$$[S{+}1]\, p_{old}(\bar{X}) \leftarrow [S]\, p(\bar{X}), \mathsf{check} \parallel p \in \mathbf{P}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \mathsf{check}^{-1}, \neg\, \mathsf{change}, p^{-2}(\bar{X}) \parallel p \in \mathbf{P}.$$

Notice that the final database state $\mathcal{D}'$ obtained by the last rule is propagated to all subsequent states (see the corresponding rule above). ■

Theorems 6.9 and 6.10 provide the main steps in determining the expressive power of Statelog, since P-Statelog and P-Datalog rules are just syntactical variants (see the next section).

6.6 Production Rules and Deductive Rules in Statelog

In this section, we show how the production rule semantics P-Datalog and I-Datalog on the one hand, and the declarative semantics S-Datalog and WF-Datalog on the other hand, may be conceived as special cases of Statelog rules. To this end, we introduce Statelog representations P_{PD}, P_{ID}, P_{SD}, and P_{WD} of a Datalog$^{\neg}$ program P evaluated under the P-Datalog, I-Datalog, S-Datalog, and WF-Datalog semantics, respectively.

Apart from reconciling these seemingly discordant approaches within the Statelog framework, the (transaction) equivalence between Statelog and P-Datalog concludes the investigation of the expressive power of Statelog, and shows that Statelog expresses precisely the WHILE queries (Corollary 6.14).

6.6.1 Production Rules

P-Datalog

Consider a Datalog$^\neg$ rule

$$H \leftarrow B, \neg C.$$

Here and in the sequel, let $B = B_1, \ldots, B_k$ and $\neg C = \neg C_1, \ldots, \neg C_l$. Under the P-Datalog semantics (Section 4.2.2), only those tuples are in the answer, which are derived anew in the current stage of the iteration. Thus, the rule corresponds to the P-Statelog rule

$$[S{+}1]\, H \leftarrow [S]\, B, [S] \neg C.$$

More formally, the Statelog representation P_{PD} of a P-Datalog program P is defined as

$$P_{\text{PD}} := \{\ [S{+}1]\, H \leftarrow [S]\, B, [S] \neg C \mid (H \leftarrow B, \neg C) \in P\ \}\ .$$

Now the precise correspondence between P-Statelog and P-Datalog can be stated:[13]

Proposition 6.11 (P-Statelog vs. P-Datalog) *Let P be a P-Datalog program, P_{PD} its Statelog representation. For all databases $\mathcal{D}$ and all $n \in \mathbb{N}_0$:*

$$\mathcal{M}_{P_{\text{PD}} \cup \mathcal{D}}[n] = T_P^n(\mathcal{D}[0])\ ,$$

where $T_P^0(\mathcal{I}) := \mathcal{I}$ and $T_P^{n+1}(\mathcal{I}) := T_P(T_P^n(\mathcal{I}))$.

Proof For $n = 0$, note that all rules of P_{PD} are of the form $[S{+}1]\, H \leftarrow \ldots$, hence, do not derive tuples in $[0]$. It follows that $\mathcal{M}_{P_{\text{PD}} \cup \mathcal{D}}[0] = \mathcal{D}[0] = T_P^0(\mathcal{D}[0])$.

For the induction step, observe that $\mathcal{M}_{P_{\text{PD}} \cup \mathcal{D}}[n{+}1]$ is obtained by applying rules of the form $[n{+}1]\, \mathit{head} \leftarrow [n]\, \mathit{body}$ to $\mathcal{M}_{P_{\text{PD}} \cup \mathcal{D}}[n]$. This precisely corresponds to one application of T_P on $\mathcal{M}_{P_{\text{PD}} \cup \mathcal{D}}[n]$, which—by induction hypothesis—is equal to $T_P^n(\mathcal{D}[0])$. ■

Thus, a P-Statelog program can be viewed as a P-Datalog program and vice versa. However, there is a subtle difference between P-Statelog and P-Datalog: while state transitions are *explicitly* defined in the former (i.e., they are part of the logical language), the latter hides these transitions—instead they are implicitly defined within the semantics of programs. This is the reason why P-Datalog fails to satisfy many of the desirable properties of deductive rules (Theorem 5.1), whereas P-Statelog programs—being locally stratified—inherit all nice properties of the (locally) stratified semantics. In this sense, P-Statelog is more “declarative”, as it makes the forward chaining (or bottom-up) evaluation strategy for production rules explicit.

Example 6.7 (Unfolding in P-Statelog) Consider the Datalog$^\neg$ program P in the proof of Theorem 5.1. By adding the rule $\mathsf{q} \leftarrow \mathsf{q}$ we obtain a P-Datalog program which violates UNFOLD. The Statelog representation of this program is

P_{PD}:

$$\begin{array}{ll}
r_1: & [S{+}1]\ \mathsf{p} \leftarrow . \\
r_2: & [S{+}1]\ \mathsf{p'} \leftarrow [S]\ \mathsf{p}. \\
r_3: & [S{+}1]\ \mathsf{q} \leftarrow [S]\ \mathsf{p'},\ [S]\ \neg\, \mathsf{p'}. \\
r_4: & [S{+}1]\ \mathsf{q} \leftarrow [S]\ \mathsf{q}.
\end{array}$$

[13]Here, we use the P-Datalog semantics from Example 4.5, as it allows to express deletions.

In order to unfold $[S]$ p' in r_3 with its definition in r_2, with have to unify $[S]$ p' with the head of the (renamed) rule r_2. This eventually yields the rewritten rule

$$r_3'\colon\quad [S{+}2]\ \mathsf{q} \leftarrow [S]\ \mathsf{p},\ [S{+}1]\ \neg\,\mathsf{p'}.$$

It should be clear that—due to the explicit notion of state—the rewritten program is equivalent to P_{PD} (note that by r_2, p holds in $[n]$ iff p' holds in $[n{+}1]$) and therefore does not violate UNFOLD. □

By Proposition 6.11, if $\mathcal{M}_{P_{\text{PD}}\cup\mathcal{D}}$ becomes stationary, the final state $\mathcal{M}_{P_{\text{PD}}\cup\mathcal{D}}[\$]$ coincides with the limit of the sequence $(T_P^n(\mathcal{D}[0]))_{n\in\mathbb{N}_0}$. This allows to characterize the *transaction* expressiveness of P-Statelog:

Corollary 6.12 (P-Statelog $\equiv_\$$ P-Datalog)
For every P-Statelog transaction there is an equivalent P-Datalog program and vice versa.

For the characterization of the *transition* expressiveness, recall that the transition from $\mathcal{M}_{P\cup\mathcal{D}}[0]$ to $\mathcal{M}_{P\cup\mathcal{D}}[1]$ corresponds to one application of the immediate consequence operator T_P, and thus is equivalent to a $\exists$FO formula φ_P.[14] Conversely, from the disjunctive normal form of a $\exists$FO formula φ one directly obtains a transition equivalent P-Datalog program P_φ. This establishes

Proposition 6.13 (P-Statelog $\equiv_1$ $\exists$FO)
For every P-Statelog transition there is an equivalent $\exists$FO *formula and vice versa.*

From Corollary 6.12 and the normal form Theorems 6.9 and 6.10 we immediately get the following characterization of the overall Statelog expressiveness:[15]

Corollary 6.14 (Statelog $\equiv_\$$ P-Datalog)
For every Statelog transaction there is an equivalent P-Datalog program and vice versa.

In other words, Statelog transactions express the WHILE queries (cf. Figure 4.2). Thus on the upside, the language is rather powerful and expresses all DB-**PSPACE** queries on ordered databases. On the downside however, this expressiveness also leads to intractable worst-case computations on large databases. The complexity of Statelog is further investigated in Section 6.8.

I-Datalog

Under the I-Datalog semantics, previously derived tuples are never retracted. Thus, the I-Datalog rule $H \leftarrow B, \neg C$ corresponds to two rules

$$\begin{aligned}&[S{+}1]\, H \leftarrow [S]\, B, [S]\, \neg C, \text{ and}\\ &[S{+}1]\, H \leftarrow [S]\, H.\end{aligned}$$

The Statelog representation P_{ID} of an I-Datalog program P is given by

[14] A first-order formula is *existential* if (i) it contains no universal quantifiers, and (ii) negation only occurs in front of atomic formulas. $\exists$FO is the set of all existential FO formulas.

[15] Recall that we use the db-transformation given by the *transaction* of a Statelog program as a measure of the overall expressiveness, i.e., when comparing Statelog with other database languages (Section 6.5.1).

$$\begin{aligned} P_{\text{ID}} := \quad & \{\, [S{+}1]\, H \leftarrow\ [S]\, B, [S]\, \neg\, C \mid (H \leftarrow\ B, \neg\, C) \in P \,\} \\ \cup \quad & \{\, [S{+}1]\, p(\bar{X}) \leftarrow\ [S]\, p(\bar{X}) \mid p \in \mathbf{P}, \mathit{arity}(p) = \mathit{arity}(\bar{X}) \,\} \ . \end{aligned}$$

Let *I-Statelog* denote the class of all such Statelog programs P_{ID}. As in the case of P-Datalog, it is straightforward to verify the following correspondence between I-Datalog and I-Statelog:

Proposition 6.15 (I-Statelog vs. I-Datalog) *Let P be an I-Datalog program, P_{ID} its Statelog representation. For all databases $\mathcal{D}$ and all $n \in \mathbb{N}_0$:*

$$\mathcal{M}_{P_{\text{ID}} \cup \mathcal{D}}[n] = (T_P^+)^n(\mathcal{D}[0]) \ ,$$

where $T_P^+(\mathcal{I}) := \mathcal{I} \cup T_P(\mathcal{I})$.

From this, analogous to the P-Datalog case above, we immediately get:

Corollary 6.16 (I-Statelog $\equiv_\$$ I-Datalog)
For every I-Statelog transaction there is an equivalent I-Datalog program and vice versa.

Corollary 6.17 (I-Statelog $\equiv_1$ ∃FO)
For every I-Statelog transition there is an equivalent ∃FO *formula and vice versa.*

6.6.2 Deductive Rules

S-Datalog

Local Statelog rules correspond to deductive rules in the following sense: Suppose given an *L-Statelog* program P, that is, comprising only rules of type (L). Since all rules of P are local, $\mathcal{M}_{P \cup \mathcal{D}}[n{+}1]$ is independent of $\mathcal{M}_{P \cup \mathcal{D}}[n]$, for all $n \in \mathbb{N}_0$. Therefore L-Statelog rules alone do not allow to express non-trivial transitions between successor states. On the other hand, it is clear that local rules are useful for defining queries like the complement of transitive closure *within* a state, which is not possible using rules of type (P). In this case, local rules serve as deductive rules which increase the knowledge about the current state without defining a new one.

The close correspondence between local rules and S-Datalog is established using the following Statelog representation P_{SD} of a S-Datalog program P:

$$P_{\text{SD}} := \{\, [S]\, H \leftarrow\ [S]\, B, [S]\, \neg\, C \mid (H \leftarrow\ B, \neg\, C) \in P \,\} \ .$$

Proposition 6.18 (L-Statelog vs. S-Datalog)
Let P be an S-Datalog program, P_{SD} its Statelog representation. For all databases $\mathcal{D}$:

$$\mathcal{M}_{P_{\text{SD}} \cup \mathcal{D}}[0] = \mathcal{S}_{P \cup \mathcal{D}[0]} \ ,$$

where $\mathcal{S}_{P \cup \mathcal{D}[0]}$ is the unique stratified model of $P \cup \mathcal{D}[0]$.

WF-Datalog

For a WF-Datalog program P, the Statelog representation P_{WD} is given by:

$$\begin{aligned} P_{\text{WD}} := \quad & \{\, [S{+}1]\, H \leftarrow [S{+}1]\, B, [S]\, \neg C \mid (H \leftarrow B, \neg C) \in P \,\} \\ \cup \quad & \{\, [S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}) \mid p \in \mathit{edb}(\mathbf{P}), \mathit{arity}(p) = \mathit{arity}(\bar{X}) \,\} \ . \end{aligned}$$

The first type of rules mimic the alternating fixpoint computation of the well-founded model (Section 5.4), the second type is needed to propagate extensional relations from the current state to all subsequent states.

Proposition 6.19 (Statelog vs. WF-Datalog) *Let P be a WF-Datalog program, P_{WD} its Statelog representation. For all databases $\mathcal{D}$ and all $n \in \mathbb{N}_0$:*

$$\mathcal{M}_{P_{\text{WD}} \cup \mathcal{D}}[n] = \Gamma^n_{P \cup \mathcal{D}[0]} \ ,$$

where $\Gamma^0_{P \cup \mathcal{I}} := \mathcal{I}$ and $\Gamma^{n+1}_{P \cup \mathcal{I}} := \Gamma_{P \cup \mathcal{I}}(\Gamma^n_{P \cup \mathcal{I}})$.

Here, since we have to distinguish between program P and database $\mathcal{D}$, we use an initialization of Γ^0_P which is slightly different from the one in equation (5.1) in Section 5.4. It should be clear that this has no influence on the fixpoints $\mathrm{lfp}(\Gamma^2_{P \cup \mathcal{D}[0]})$ and $\mathrm{gfp}(\Gamma^2_{P \cup \mathcal{D}[0]})$.

Proof For $n = 0$, P_{WD} does not derive new information, so $\mathcal{M}_{P_{\text{WD}} \cup \mathcal{D}}[0] = \mathcal{D}0 = \Gamma^0_{P \cup \mathcal{D}[0]}$.

For the induction step, observe that the transition from $[n]$ to $[n{+}1]$ using the rules from P_{WD} corresponds to one application of the antimonotone $\Gamma_{P \cup \mathcal{D}[0]}$ operator (cf. Definition 5.2), i.e., the computation of the least fixpoint using the fixed values from the *previous* stage for negative literals. ∎

By Proposition 6.19, even and odd states of the Statelog representation correspond to underestimates of *true* and overestimates of *true* or *undefined* atoms, respectively. Thus, if the well-founded model contains undefined atoms, the set of underestimates is always strictly included in the set of overestimates, and the Statelog model becomes 2-periodic. On the other hand, if the well-founded model is total, then the Statelog representation is 1-periodic and terminates for $\mathcal{D}$. As will be shown in Theorem 6.30 one can "accelerate" the 2-periodic program P_{WD} to obtain an equivalent program which always terminates. Such a construction will also used in the proof of Theorem 6.22 below.

$$\boxed{H \leftarrow B, \neg C} \Leftrightarrow \left\{ \begin{array}{ll} \textit{P-Datalog}: & \boxed{[S{+}1]\, H \leftarrow [S]\, B, [S]\, \neg C} \\ \textit{I-Datalog}: & \boxed{\begin{array}{l}[S{+}1]\, H \leftarrow [S]\, B, [S]\, \neg C \\ [S{+}1]\, H \leftarrow [S]\, H\end{array}} \\ \hline \textit{S-Datalog}: & \boxed{[S]\, H \leftarrow [S]\, B, [S]\, \neg C} \\ \textit{WF-Datalog}: & \boxed{[S{+}1]\, H \leftarrow [S{+}1]\, B, [S]\, \neg C} \end{array} \right.$$

Figure 6.4: Encoding schema for different rule semantics

Figure 6.4 illustrates the Statelog representations of different rule semantics for a given Datalog$^{\neg}$ rule $H \leftarrow B, \neg C$. In the case of WF-Datalog, additional frame rules of the form $[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X})$ are needed for relations $p \in \mathit{edb}(\mathbf{P})$; see P_{WD} above.

6.6.3 NF-Statelog vs. P-Statelog Transitions

The normal form Theorems 6.9 and 6.10 show that NF-Statelog and P-Statelog have the same transaction expressiveness. It remains to compare their expressive power wrt. transitions.

Proposition 6.20 (NF-Statelog $\equiv_1$ S-Datalog)
For every NF-Statelog transition there is an equivalent S-Datalog program and vice versa.

Proof A NF-Statelog transition is given by evaluating a set of local rules, i.e., a S-Datalog program, followed by an application of the 1-progressive rules. The latter correspond to an $\exists$FO operation by Proposition 6.13. Since $\exists$FO is (strictly) contained in S-Datalog, we have NF-Statelog $\leq_1$ S-Datalog. The converse direction is obvious. ∎

By Theorem 6.9, Statelog and NF-Statelog are evolution equivalent, so the previous proposition also applies to arbitrary Statelog programs not necessarily in normal form.

Since P-Statelog transitions correspond to $\exists$FO queries, which are strictly included in the S-Datalog queries, it follows that P-Statelog is strictly less expressive than S-Datalog wrt. *transitions*. This comes as no surprise, since NF-Statelog allows local recursion whereas P-Statelog is non-recursive within a state:

Corollary 6.21 (P-Statelog $\lneq_1$ NF-Statelog)
P-Statelog is less expressive than NF-Statelog wrt. transition expressiveness.

6.7 Simulation of Well-Founded Statelog

In Section 6.4.4, we have defined the perfect model $\mathcal{M}_{P\cup\mathcal{D}}$ as the canonical model of a state-stratified program P and database $\mathcal{D}$. Since P can be identified with the reified program $P^\star$, we may also apply other logic programming semantics to Statelog, if P is *not* state-stratified, i.e., involves negative local cycles:

As demonstrated by Example 6.5, *stable models* of a Statelog program need not be periodic. The intersection or union of all stable models—although periodic—do not provide (stable) Statelog models (Example 6.5). A possible way out is to consider for every state $[n]$, the *set* of all stable models. However, this approach seems prohibitively expensive since computing a single stable snapshot for a state $[n]$ is already an **NP**-complete operation in general. Therefore it is necessary to restrict to tractable classes similar to the nondeterministic Datalog$^\neg$ programs with *choice* (see, e.g., [GGSZ97]). We do not pursue these issues here.

In the sequel, we consider the deterministic well-founded semantics for Statelog, which always yields a unique 3-valued model $\mathcal{W}_{P\cup\mathcal{D}}$. The same argument as in the proof of Theorem 6.6 shows that $\mathcal{W}_{P\cup\mathcal{D}}$ is ultimately periodic. Moreover, it is easy to see that a snapshot $\mathcal{W}_{P\cup\mathcal{D}}[n{+}1]$ can be computed in polynomial time, given the previous state $\mathcal{W}_{P\cup\mathcal{D}}[n]$ as input.

Let *WF-Statelog* denote the class of Statelog programs evaluated under the well-founded semantics. The question arises, whether the expressive power increases, if we consider WF-Statelog programs instead of "standard" Statelog programs with state-stratified semantics. Recall that Theorem 6.9 also applies to WF-Statelog, so we may assume w.l.o.g. that WF-Statelog programs are in normal form.

Transition and Evolution Expressiveness

It is easy to see that the transitions expressible in WF-Statelog are precisely the db-transformations expressible in the "one-state" language WF-Datalog, which in turn correspond to the class of FIXPOINT transformations (cf. Figure 4.2). Consequently, WF-Statelog is strictly more expressive wrt. *transitions* than Statelog, which only allows stratified local views. For example, the famous *win-move* game can be expressed with local rules in WF-Statelog, but not in Statelog.[16] It follows that Statelog $\lneq_\infty$ WF-Statelog.

However, since the transaction expressiveness of Statelog corresponds to WHILE, it is possible to simulate local well-founded views using progressive recursion, i.e., iteration over states. In fact the simulation of a WF-Datalog program P by a Statelog program P_{WD} is very simple and elegant, as has been shown in Section 6.6.2.

Transaction Expressiveness

Intuitively, one would conjecture that WF-Statelog does not increase the *transaction* expressiveness of Statelog, since local WF-Statelog rules can be simulated as a Statelog transaction, and 1-progressive WF-Statelog rules can be conceived as an outer noninflationary fixpoint which is applied to an inner WF-Datalog query. In the parlance of fixpoint logic this means that a PFP formula is applied to some LFP formulas, which still lies within PFP. In the sequel, we present a direct simulation of WF-Statelog in state-stratified Statelog, thereby formally proving this statement.

Theorem 6.22 (Statelog $\equiv_\$$ WF-Statelog)
For every WF-Statelog program there is a transaction equivalent Statelog program.

PROOF Statelog $\leq_\$$ WF-Statelog holds trivially, since every Statelog program is also a WF-Statelog program. For the converse direction, let P be a WF-Statelog program. Due to Theorem 6.9, we may assume that P is in normal form. We construct a transaction equivalent state-stratified program P' as follows (cf. Figure 6.5):

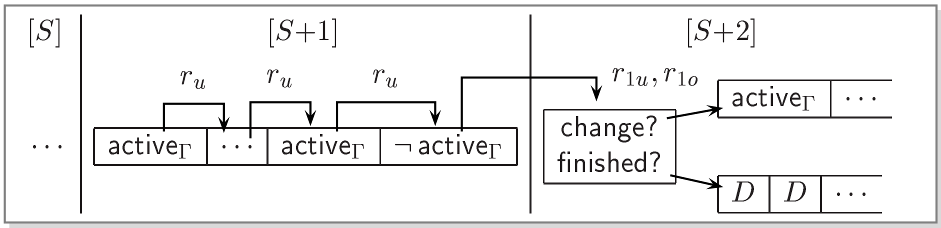


Figure 6.5: Simulation of a WF-Statelog transaction

Let $\mathbf{P}$ be the schema of P. Then $\mathbf{P}'$ contains for every original relation $p \in \mathbf{P}$ two distinct relations p_u and p_o, holding the underestimates of true, and the overestimates of true or undefined atoms of the AFP computation, respectively. Eventually, we will have

$$\begin{array}{lcl} \mathcal{M}_{P'\cup\mathcal{D}}[\$] \models p_u(\bar{x}) & \Leftrightarrow & p(\bar{x}) \in \mathcal{W}^{\boldsymbol{t}}_{P\cup\mathcal{D}}[\$] \\ \mathcal{M}_{P'\cup\mathcal{D}}[\$] \models p_o(\bar{x}), \neg\, p_u(\bar{x}) & \Leftrightarrow & p(\bar{x}) \in \mathcal{W}^{\boldsymbol{u}}_{P\cup\mathcal{D}}[\$] \\ \mathcal{M}_{P'\cup\mathcal{D}}[\$] \models \neg\, p_o(\bar{x}) & \Leftrightarrow & p(\bar{x}) \in \mathcal{W}^{\boldsymbol{f}}_{P\cup\mathcal{D}}[\$] \end{array}$$

[16] Every FIXPOINT query can be expressed as the win-move game from Example 5.2, see [FKL97].

for all $p \in \mathbf{P}$ and all databases $\mathcal{D}$ over $\mathbf{P}$. The rules of P' are obtained as follows: First, in [0], underestimates and overestimates are initialized by the given database relations:

$$[S]\, p_u(\bar{X}), p_o(\bar{X}) \leftarrow [S]\, p(\bar{X}), \mathsf{eq}_0 \parallel p \in \mathbf{P} \ .$$

Next, for every local rule

$$r: \quad [S]\, H \leftarrow [S]\, B, \neg\, C$$

of P, where $B = B_1, \ldots, B_n$ and $\neg\, C = \neg\, C_1, \ldots, \neg\, C_m$, there are two rules in P':

$$\begin{array}{ll} r_u: & [S{+}1]\, H_u \leftarrow [S{+}1]\, B_u, [S]\, \neg\, C_o, \mathsf{active}_\Gamma. \\ r_o: & [S]\, H_o \leftarrow [S]\, B_o, \neg\, C_u, \mathsf{active}_\Gamma. \end{array}$$

Here, H_v, B_v, and C_v are obtained from H, B, and C by replacing every relation symbol p by p_v, for $v \in \{u, o\}$. Note that these rules mimic two inner fixpoints $\Gamma^{2k}_{P \cup \mathcal{D}}$ and $\Gamma^{2k+1}_{P \cup \mathcal{D}}$ of AFP within a single state: The rules of type r_u compute in the next state $[S{+}1]$ a new underestimate based on the previous overestimate in $[S]$. Using this underestimate, the next overestimate is then computed by the r_o rules. The condition active_Γ guarantees that these rules are only applied while the inner Γ-loop (the computation of the fixpoint $\Gamma_{P \cup \mathcal{D}}$) is running, i.e., while the underestimates are increasing:

$$\begin{array}{l} [S]\, \mathsf{active}_\Gamma \leftarrow [S]\, \mathsf{eq}_0. \\ [S{+}1]\, \mathsf{active}_\Gamma \leftarrow [S]\, \mathsf{active}_\Gamma, \neg\, p_u(\bar{X}), [S{+}1]\, p_u(\bar{X}) \parallel p_u \in \mathbf{P}'. \end{array}$$

While the inner Γ-loop is active, the (initially empty) values from the previous round of the outer loop are propagated:

$$[S{+}1]\, p_{old,v}(\bar{X}) \leftarrow [S]\, p_{old,v}(\bar{X}), \mathsf{active}_\Gamma \parallel p \in \mathbf{P}, v \in \{u, o\}.$$

When the Γ-loop is completed, the 1-progressive rules of P have to be applied. Thus, for every 1-progressive rule

$$r_1: \quad [S{+}1]\, H \leftarrow [S]\, B, \neg\, C$$

of P, there are two rules in P', which are enabled if active_Γ just became false:[17]

$$\begin{array}{ll} r_{1u}: & [S{+}1]\, H_u \leftarrow [S]\, B_u, \neg\, C_o, \mathsf{active}^{-1}_\Gamma, \neg\, \mathsf{active}_\Gamma. \\ r_{1o}: & [S{+}1]\, H_o \leftarrow [S]\, B_o, \neg\, C_u, \mathsf{active}^{-1}_\Gamma, \neg\, \mathsf{active}_\Gamma. \end{array}$$

In the same state, it is checked whether there are changes in the outer loop:

$$\begin{array}{l} [S{+}1]\, \mathsf{change} \leftarrow [S]\, \mathsf{active}^{-1}_\Gamma, \neg\, \mathsf{active}_\Gamma, p_{old,v}(\bar{X}), [S{+}1]\, \neg\, p_v(\bar{X}) \parallel p \in \mathbf{P}, v \in \{u, o\}. \\ [S{+}1]\, \mathsf{change} \leftarrow [S]\, \mathsf{active}^{-1}_\Gamma, \neg\, \mathsf{active}_\Gamma, \neg\, p_{old,v}(\bar{X}), [S{+}1]\, p_v(\bar{X}) \parallel p \in \mathbf{P}, v \in \{u, o\}. \end{array}$$

If so, the results of the outer round are used as new start values (and also remembered in p_{old} for the next outer comparison), and the inner Γ-loop is activated again:

$$\begin{array}{l} [S{+}1]\, p_v(\bar{X}), p_{old,v}(\bar{X}) \leftarrow [S]\, \mathsf{change}, p_v(\bar{X}) \parallel p \in \mathbf{P}, v \in \{u, o\}. \\ [S{+}1]\, \mathsf{active}_\Gamma \leftarrow [S]\, \mathsf{change}. \end{array}$$

Otherwise, a fixpoint has been reached eventually and the final values are propagated through all states:

$$\begin{array}{l} [S{+}1]\, \mathsf{finished} \leftarrow [S]\, \mathsf{active}^{-1}_\Gamma, \neg\, \mathsf{active}_\Gamma, [S{+}1]\, \neg\, \mathsf{change}. \\ [S{+}1]\, \mathsf{finished} \leftarrow [S]\, \mathsf{finished}. \\ [S{+}1]\, p_v(\bar{X}) \leftarrow [S]\, p_v(\bar{X}), \mathsf{finished} \parallel p \in \mathbf{P}, v \in \{u, o\}. \end{array}$$ ■

[17]Recall that $\mathsf{active}^{-1}_\Gamma$ holds in a state iff active_Γ holds in the previous state (Lemma 6.7).

6.8 Evaluation and Complexity

Statelog programs are special logic programs. Therefore, they can be evaluated using standard evaluation techniques from deductive databases and logic programming. Due to Theorem 6.9 we may assume, w.l.o.g., that Statelog programs are in normal form.

6.8.1 Bottom-Up Evaluation

A straightforward bottom-up evaluation of the perfect model $\mathcal{M}_{P \cup \mathcal{D}}$ can be obtained as follows. Every $P \in$ NF-Statelog can be partitioned into a set P_L of local rules and P_P of 1-progressive rules. In a first preprocessing step, the (partial) evaluation order among local rules is determined: Consider the labeled dependency graph $\mathcal{LG}_{P_L}$ of all local rules. Since P is state-stratified, the strongly connected components of $\mathcal{LG}_{P_L}$ induce a stratification $P_L^1, \ldots, P_L^k$ of P_L. In contrast, the 1-progressive rules from P_P can be applied in any order or in parallel, since they do not depend on each other (observe that rules in P_P define the initial values of relations in $[n{+}1]$ given the old values from $[n]$ as input).

function *evaluate*
<u>*given:*</u> – $P = P_L \dot\cup P_P$ *in normal form*
– *a Statelog database* $\mathcal{D}$

$n := 0$;
$\mathcal{M}[0] := P_L^*(\mathcal{D}[0])$;
repeat
  $n := n{+}1$;
  $\mathcal{M}[n] := P_P(\mathcal{M}[n{-}1])$;
  $\mathcal{M}[n] := P_L^*(\mathcal{M}[n])$;
until $\mathcal{M}[n] = \mathcal{M}[n{-}1]$;
return $\mathcal{M}[0..\$]$ $(= \mathit{evaluate}(P, \mathcal{D}))$

function P_L^*
<u>*given:*</u> – $P_L = P_L^1 \dot\cup P_P \ldots \dot\cup P_L^k$
– *a database state* $\mathcal{I}$

$\mathcal{J} := \mathcal{I}$;
for $i := 1$ **to** k
  repeat
    $\mathcal{J}' := P_L^i(\mathcal{J})$;
    $\mathit{new} := \mathcal{J}' \setminus \mathcal{J}$;
    $\mathcal{J} := \mathcal{J} \cup \mathit{new}$;
  until $\mathit{new} = \emptyset$;
return $\mathcal{J}$ $(= P_L^*(\mathcal{I}))$

Figure 6.6: Bottom-up evaluation of the perfect model $\mathcal{M}$ $(= \mathcal{M}_{P \cup \mathcal{D}})$

After this preprocessing phase, the perfect model $\mathcal{M}_{P \cup \mathcal{D}}$ can be computed one state after another by the function *evaluate* in Figure 6.6: Initially, the local rules are iterated on the given input database $\mathcal{D}$ until the first inner fixpoint for $\mathcal{M}_{P \cup \mathcal{D}}[0]$ is reached. The function P_L^* corresponds to the evaluation of a S-Datalog program. In the main loop of *evaluate*, $\mathcal{M}[n]$ is computed: First, the rules of P_P are applied once (possibly in parallel) on the already computed previous state $\mathcal{M}[n{-}1]$. Next, the local rules P_L are iterated stratum by stratum in the function P_L^* until the computation of $\mathcal{M}[n]$ is completed. The whole process is repeated by *evaluate* until the outer fixpoint $\mathcal{M}_{P \cup \mathcal{D}}[\$]$ is reached.

The algorithm in Figure 6.6 can be easily implemented on top of a relational database system, since all rule applications ($P_P(\mathcal{I})$ and $P_L^i(\mathcal{I})$) correspond to standard relational algebra operations, provided the rule are range-restricted.

Note that this algorithm for NF-Statelog programs terminates iff $\mathcal{M}_{P \cup \mathcal{D}}$ is terminating, i.e., if a unique final state exists (cf. Definition 6.7). As will be seen in Section 6.9, it is undecidable whether P terminates for *all* databases $\mathcal{D}$, and **PSPACE**-complete provided that $\mathcal{D}$ is given.

6.8.2 Complexity

In this section, we study the complexity of evaluating Statelog programs. Recall that the data complexity of a db-transformation q on a database instance $\mathcal{D}$ is the complexity (wrt. the size $|\mathcal{D}|$) of deciding whether $t \in q(\mathcal{D})$ for a tuple (or ground atom) t. With every Statelog program P we can associate two kinds of db-transformations, that is, transitions and transactions. The complexity of transactions is more important here, since it describes the computational cost of computing the overall effect of P on a given database $\mathcal{D}$. For transitions, w.l.o.g., we restrict to the transition from [0] to [1] (cf. Section 6.5.1). Now the data complexity of transitions and transactions is given by the following recognition problems:

- TRANSITION: Given a Statelog program P, a database $\mathcal{D}$, and a ground atom $p(\bar{x})$, decide whether $\mathcal{M}_{P\cup\mathcal{D}}[1] \models p(\bar{x})$, and
- TRANSACTION: Given P, $\mathcal{D}$, and $p(\bar{x})$, decide whether $\mathcal{M}_{P\cup\mathcal{D}}[\$] \models p(\bar{x})$.

Recall that we have defined $\mathcal{M}_{P\cup\mathcal{D}}[\$] := \emptyset$ if $\mathcal{M}_{P\cup\mathcal{D}}$ does not terminate (Definition 6.7). The complexity of these problems follows immediately from the expressiveness results for Statelog presented in Section 6.5 and the following well-known results for S-Datalog and P-Datalog (cf. [AHV95]):

Theorem 6.23 (Expressiveness: S-Datalog, P-Datalog)
On the class of ordered *databases:*

- *S-Datalog expresses* DB-**PTIME**,
- *P-Datalog expresses* DB-**PSPACE**.

In other words, there are db-transformations q which are expressible in S-Datalog and P-Datalog, such that the recognition problem $t \in q(\mathcal{I})$ for ordered databases $\mathcal{I}$ is complete for **PTIME** and **PSPACE**, respectively. As shown in Section 6.5, the db-transformations expressible as Statelog *transactions* are precisely the P-Datalog queries (Corollary 6.14). On the other hand, by Theorem 6.9 and Proposition 6.20, for every Statelog *transition* there is an equivalent S-Datalog query and vice versa. From this we immediately get:

Theorem 6.24 (Statelog Data Complexity)
Given a Statelog program P, a database $\mathcal{D}$, and a ground atom $p(\bar{x})$,

- *deciding whether $\mathcal{M}_{P\cup\mathcal{D}}[1] \models p(\bar{x})$ is* **PTIME**-*complete,*
- *deciding whether $\mathcal{M}_{P\cup\mathcal{D}}[\$] \models p(\bar{x})$ is* **PSPACE**-*complete.*

To get an intuitive understanding of these results, consider the algorithm depicted in Figure 6.6. To see that TRANSACTION can be computed within polynomial space, observe that it is sufficient to store only the two most recent states $\mathcal{M}_{P\cup\mathcal{D}}[n\text{–}1]$ and $\mathcal{M}_{P\cup\mathcal{D}}[n]$, provided we are only interested in computing $\mathcal{M}_{P\cup\mathcal{D}}[\$]$ (instead of $\mathcal{M}_{P\cup\mathcal{D}}[0..\$]$). Moreover, there are at most

$$|\mathcal{B}_{P\cup\mathcal{D}}| = \sum_{p\in\mathbf{P}} |adom(\mathcal{D})|^{arity(p)}$$

many different tuples in any state $\mathcal{M}_{P\cup\mathcal{D}}[n]$, which is clearly polynomial in the size $|\mathcal{D}|$ of the given database $\mathcal{D}$. (Recall that the Herbrand base $\mathcal{B}_{P\cup\mathcal{D}}$ consists of all (Datalog) ground atoms which can be built from the data constants in P and $\mathcal{D}$; cf. Definition 6.4.) Therefore, if $\mathcal{M}_{P\cup\mathcal{D}}$ terminates, the final state $\mathcal{M}_{P\cup\mathcal{D}}[\$]$ can be computed in polynomial space. Conversely, if after $2^{|\mathcal{B}_{P\cup\mathcal{D}}|}$ iterations $\mathcal{M}_{P\cup\mathcal{D}}[n] \neq \mathcal{M}_{P\cup\mathcal{D}}[n{+}1]$, we know that $\mathcal{M}_{P\cup\mathcal{D}}[\$] = \emptyset$. Clearly, adding such a counter does not violate the **PSPACE** bound of the algorithm.

A direct proof for the **PTIME**-completeness of TRANSITION can be obtained as follows: By executing the body of the repeat-loop of function *evaluate* in Figure 6.6 once, we obtain $\mathcal{M}_{P\cup\mathcal{D}}[1]$. It is easy to see that the evaluation of $P_P(\ldots)$ and $P_L^*(\ldots)$ can be accomplished in polynomial time, so TRANSITION lies in **PTIME**. To show that the problem is hard for **PTIME**, one can reduce CIRCUIT-VALUE to TRANSITION: CIRCUIT-VALUE is the problem of deciding whether a distinguished output gate of a Boolean circuit evaluates to 1 given the values of the input gates.[18] It is well known that this problem is **PTIME**-complete [Pap94], even for circuits consisting only of NAND-gates, i.e., for which all inner nodes have in-degree two, out-degree one, and value $out = \neg(in_1 \wedge in_2)$ [BGG97]. The problem has an elegant representation in Datalog:[19]

Example 6.8 (CIRCUIT-VALUE in Datalog) A given Boolean circuit is encoded as a database $\mathcal{D}$ as follows: The *edb*-relations zero, one, e, and out hold the input gates with values 0 and 1, the edges between the gates, and an distinguished output node, respectively. The *idb*-relations is_0 and is_1 are used to compute the values of gates:

P_{cv}:

r_1:	is_0(X) ← zero(X).
r_2:	is_1(X) ← one(X).
r_3:	is_1(Y) ← is_0(X), e(X,Y).
r_4:	is_0(Z) ← is_1(X), is_1(Y), e(X,Z), e(Y,Z), X ≠ Y.
r_5:	yes ← out(X), is_1(X).

r_1 and r_2 determine the values of the input gates. An inner gate evaluates to 1 if at least one incoming edge is 0 (r_3), and to 0 if both incoming edges are 1 (r_4). Thus, yes holds in the minimal model of $P_{cv} \cup \mathcal{D}$ iff the output gate evaluates to 1. □

By attaching the state term "$[S]$" to all rules of P_{cv} and adding the rule

$$[S{+}1]\ \mathsf{yes} \leftarrow [S]\ \mathsf{yes}$$

we obtain a Statelog program which accomplishes the desired reduction from CIRCUIT-VALUE to TRANSITION.

[18] A Boolean circuit is a directed acyclic graph whose nodes (or *gates*) are marked with Boolean connectives (e.g., "¬", "∧", and "∨"), s.t. the in- and out-degrees of gates match the type of the connective. Nodes without incoming edge are *source gates*, those without outgoing edge are *output gates*.

[19] Recall that we allow the use of inequalities $X \neq Y$ in Datalog; see Definition 4.1.

Inflationary Statelog

As can be seen from Theorem 6.24, it may be infeasible to compute the final state of Statelog transaction. An ad-hoc "solution" is to consider an inflationary variant of Statelog i.e., programs P where a rule

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X})$$

is used for every relation symbol p of P, in order to make the semantics inflationary. The resulting language is a slight generalization of I-Statelog defined on page 61. The number of transitions needed to reach the final state is at most polynomial in $|\mathcal{D}|$, since in every intermediate transition at least one tuple is added. It follows that termination of inflationary Statelog is guaranteed, and computing $\mathcal{M}_{P\cup\mathcal{D}}$ is **PTIME**-complete. Unfortunately, inflationary Statelog is not really useful as a database language, because deletions cannot be expressed. In Section 7.5.2 several so-called Δ-*monotonic* Statelog variants are proposed, whose data complexity is within **PTIME** yet allow both insertion and deletion of tuples.

6.8.3 Number of Transitions

The canonical model $\mathcal{M}_{P\cup\mathcal{D}}$ is ultimately periodic, hence it can be represented by an initial window $\mathcal{M}_{P\cup\mathcal{D}}[0..n_0{-}1]$ and a T-periodic window $(\mathcal{M}_{P\cup\mathcal{D}}[n_0..n_0{+}T{-}1])^\omega$. Although we are only interested in the final state of terminating computations, we would like to estimate the maximal number of intermediate transitions needed to reach $\mathcal{M}_{P\cup\mathcal{D}}[\$]$. Furthermore, we are interested in the maximal length of the period T, when there is no unique final state.

As for the upper bound, note that in the worst case there are $2^{|\mathcal{B}_{P\cup\mathcal{D}}|}$ different database states before $\mathcal{M}_{P\cup\mathcal{D}}$ becomes periodic or stationary.[20] Indeed this exponential bound is attained for some programs:

Theorem 6.25 (Number of Transitions) *The lower bound for the length of the initial window and the length of the periodic window of $\mathcal{M}_{P\cup\mathcal{D}}$ is exponential in the size of $\mathcal{D}$.*

Analogous results have been given in [CI88, CI93, Cho95a] for the related language Datalog$_{1S}$(Section 6.10.1). The program P_{add} below, encoding a binary counter, is taken from [CI93].

Proof We use the following relations to encode adding 1 to an n-bit number: The value of a bit b_i is determined by unary relations zero and one. The binary relation next defines the order of significance of bits. Finally, relations chg and unchg describe whether or not a bit has been changed in the transition to the current state. The initial database encodes the n-bit representation of 0, and asserts that the least significant bit b_1 has been changed (this is required to start the computation):

$$\begin{array}{rcl}\mathcal{D} & := & \{\, [0]\, \mathsf{zero}(\mathsf{b}_i) \mid 1 \le i \le n \,\} \cup \{\, [0]\, \mathsf{next}(\mathsf{b}_i, \mathsf{b}_{i+1}) \mid 1 \le i \le n-1 \,\} \\ & \cup & \{\, [0]\, \mathsf{lsb}(\mathsf{b}_1),\ [0]\, \mathsf{chg}(\mathsf{b}_1) \,\}\ .\end{array}$$

The program P_{add} below successively adds 1 to the current value of the binary counter. The first two rules describe which bits are changed. Note that the least significant bit b_1 always changes. The next pair of rules define the unchanged bits. The rules for one and zero compute the new value of a bit, given the old value and change information. The last rule guarantees that the next relation is available in all states.

[20] We tacitly assume that P is in normal form.

P_{add}:

```
[S+1] chg(X) ← [S] lsb(X), chg(X).
[S+1] chg(Y) ← [S] next(X,Y), one(X), [S+1] chg(X).

[S+1] unchg(Y) ← [S] next(X,Y), zero(X), [S+1] one(X).
[S] unchg(Y) ← [S] next(X,Y), [S] unchg(X).

[S+1] one(X) ← [S+1] chg(X), [S] zero(X).
[S+1] one(X) ← [S+1] unchg(X), [S] one(X).

[S+1] zero(X) ← [S+1] chg(X), [S] one(X).
[S+1] zero(X) ← [S+1] unchg(X), [S] zero(X).

[S+1] next(X,Y) ← [S] next(X,Y).
```

It is clear that $\mathcal{M}_{P_{add} \cup \mathcal{D}}$ becomes periodic with period $T = 2^n$, which is exponential in $|\mathcal{D}|$. In order to obtain a terminating program with exponential initial window, we add to every rule body the goal "$[S]$ running". The latter is defined by

$$\begin{array}{rcl} [0]\,\mathsf{running} & \leftarrow & . \\ [S{+}1]\,\mathsf{running} & \leftarrow & [S]\,\mathsf{zero}(X). \end{array}$$

Thus, when eventually all bits have been set to one, running becomes false, and all rules are blocked. ■

6.9 Termination

Termination is arguably the most crucial property of rule execution. While for some classes like S-Datalog, I-Datalog, and I-Statelog, termination of the standard bottom-up algorithms is guaranteed,[21] this is no longer the case for languages with "noninflationary" semantics. In this section we study the theoretical limits in determining whether a Statelog program terminates. In Section 7.5.2 we propose a practical solution, i.e., a class of programs with noninflationary semantics which guarantees termination.

When analyzing the properties of a program P, there is a crucial difference between compile-time and run-time properties. A *compile-time property* is a property that a program P may have, *independent* of a given database. Thus, if $\mathcal{P}(\mathbf{R})$ is the set of all programs over a schema $\mathbf{R}$, a compile-time property defines a subset of $\mathcal{P}(\mathbf{R})$. In general, properties which depend on first-order conditions are undecidable (see below). In contrast, a *run-time property* is a property which a program P together with a given database $\mathcal{D}$ may have, i.e., a subset of $\mathcal{P}(\mathbf{R}) \times inst(\mathbf{R})$. Termination can be considered as a run-time and as compile-time property:

Definition 6.13 (Statelog Termination) A Statelog program P *terminates for* $\mathcal{D}$, if $\mathcal{M}_{P \cup \mathcal{D}}$ is ultimately periodic with period $T = 1$; otherwise P *diverges for* $\mathcal{D}$.

P *always terminates* (*diverges*), if P terminates (diverges) for *all* databases $\mathcal{D}$, and P *sometimes terminates* (*diverges*), if it terminates (diverges) for *some* $\mathcal{D}$. □

As will be seen in the sequel, for Statelog programs termination is undecidable at compile-time and **PSPACE**-complete at run-time.

[21] Recall that we are dealing with relational database instances, i.e., finite structures.

6.9.1 Compile-Time Termination

In order to establish the undecidability result, we make use of Trakhtenbrot's Theorem [Tra50] (see [EF95] for a recent reference).

Theorem 6.26 (Trakhtenbrot's Theorem) *The set of first-order sentences* satisfiable *in finite structures is recursively enumerable, but not decidable; the set of first-order sentences* valid *in all finite structures is not recursively enumerable.*

Using Trakhtenbrot's Theorem we are able to show:

Theorem 6.27 (Compile-Time Termination)
Given an arbitrary Statelog program P, it is undecidable whether P

- *always terminates,*
- *sometimes terminates.*

Proof It is well-known that every first-order query can be expressed as a S-Datalog program (cf. Figure 4.2). Thus, for every first-order sentence φ there is an equivalent S-Datalog program P_φ with distinguished 0-ary answer relation ans_φ such that for all databases $\mathcal{D}$: $\mathcal{D} \models \varphi$ iff $\mathcal{S}_{P_\varphi \cup \mathcal{D}} \models \mathsf{ans}_\varphi$, where $\mathcal{S}_{P_\varphi \cup \mathcal{D}}$ is the stratified model of $P_\varphi \cup \mathcal{D}$. By attaching state terms $[S]$ to the rules of P_φ we obtain a state-stratified set of local Statelog rules P'_φ. Consider the Statelog program

$$\begin{aligned} P''_\varphi := P'_\varphi \cup \{\, [S{+}1]\,\mathsf{q} \quad &\leftarrow [S]\,\neg\,\mathsf{q}, \neg\,\mathsf{ans}_\varphi, \\ [S{+}1]\,\mathsf{ans}_\varphi \quad &\leftarrow [S]\,\mathsf{ans}_\varphi\,\}\ , \end{aligned}$$

where q is a new relation symbol not occurring in P'_φ. It is easy to see, that P''_φ terminates for $\mathcal{D}_0 := \{[0]\,p(\bar{x}) \mid p(\bar{x}) \in \mathcal{D}\}$ if and only if $\mathcal{D} \models \varphi$: If ans_φ holds in $\mathcal{D}$, the rule for q does not "fire", so all states apart from [0] are empty. Otherwise, q oscillates between *true* and *false*. Therefore P''_φ always (sometimes) terminates iff φ is valid (satisfiable), both of which is undecidable by Theorem 6.26. ∎

A similar result for Datalog$^{\neg\neg}$ has been presented in [AS91] [22] using a reduction from the undecidable *FD-implication problem* (basically, decide whether for *all* databases $\mathcal{D}$, a functional dependency on some *edb*-relation implies a functional dependency on some *idb*-relation).

Note that also negation-free programs may diverge. A simple example is the program

$$\begin{aligned} [S{+}1]\ \ \mathsf{p} &\leftarrow [S]\ \ \mathsf{q}. \\ [S{+}1]\ \ \mathsf{q} &\leftarrow [S]\ \ \mathsf{p}. \end{aligned}$$

which oscillates between the states $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$ given the database $\mathcal{D} = \{[0]\,\mathsf{p}\}$. Interestingly, for negation-free programs the problem becomes decidable, which follows from a result in [Cho95a] on universal safety for Datalog$_{nS}$ [Cho97].

[22] In [AS91], the name Datalog$^{\neg *}$ is used for Datalog$^{\neg\neg}$.

Further Undecidability Results

By reducing the problem of deciding validity in the finite to compile-time properties, we obtain further undecidability results: A Statelog program P is called

- *inflationary*, if $\mathcal{M}_{P\cup\mathcal{D}}[n] \subseteq \mathcal{M}_{P\cup\mathcal{D}}[n{+}1]$,
- *deflationary*, if $\mathcal{M}_{P\cup\mathcal{D}}[n] \supseteq \mathcal{M}_{P\cup\mathcal{D}}[n{+}1]$, and
- *constant*, if $\mathcal{M}_{P\cup\mathcal{D}}[n] = \mathcal{M}_{P\cup\mathcal{D}}[n{+}1]$,

for all databases $\mathcal{D}$ and all n. All these properties are undecidable. Other natural questions are:

- Is the *period* of P *bounded*, i.e., is there some $k \in \mathbb{N}_0$ such that $\mathcal{M}_{P\cup\mathcal{D}}$ is at most k-periodic for all databases $\mathcal{D}$?
- Is the length of the *initial window bounded*, i.e., is there some $k \in \mathbb{N}_0$ such that the length of the initial window of $\mathcal{M}_{P\cup\mathcal{D}}$ is at most k for all databases $\mathcal{D}$?

Theorem 6.28 (Undecidability Results)
Given an arbitrary Statelog program P, it is undecidable whether

1. *P is inflationary/deflationary/constant,*
2. *the period of P is bounded,*
3. *the length of the initial window of P is bounded.*

Proof (1) Consider P''_φ in the proof of Theorem 6.27. By adding the rules

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}) \parallel p \in \mathbf{P}''_\varphi \setminus \{\mathsf{q}\},$$

a program is obtained which is constant (and thus both inflationary and deflationary) iff φ is valid. Note that the subclass I-Statelog *is* decidable (one only needs to check whether there is a frame rule $[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X})$ for *every* relation p of the program; see the definition of P_{ID} on page 61).

(2) Let P'_φ be as in the proof of Theorem 6.27, and

$$\begin{array}{rll} P''_\varphi := P'_\varphi \cup \{\, [S{+}1]\, \mathsf{del\_pos(X)}, & & \\ \mathsf{ins\_pos(Y)} & \leftarrow & [S]\, \mathsf{pos(X)},\ \mathsf{next(X,Y)}, \neg\, \mathsf{ans}_\varphi, \\ [S{+}1]\, \mathsf{pos(X)} & \leftarrow & [S]\, \mathsf{pos(X)},\ \neg\, \mathsf{del\_pos(X)}, \\ [S{+}1]\, \mathsf{pos(X)} & \leftarrow & [S]\, \mathsf{ins\_pos(X)}, \\ [S{+}1]\, \mathsf{next(X,Y)} & \leftarrow & [S]\, \mathsf{next(X,Y)}, \\ [S{+}1]\, \mathsf{ans}_\varphi & \leftarrow & [S]\, \mathsf{ans}_\varphi \,\}\ . \end{array}$$

The first rule moves an item from position X to the next position Y provided that ans_φ is *false*. We have to show: the period of P''_φ is bounded $\Leftrightarrow$ φ is valid.

($\Rightarrow$) Assume φ is not valid over finite structures, so $\mathcal{D} \models \neg\,\varphi$ for some database $\mathcal{D}$. Then, for every $k \geq 1$, $\mathcal{M}_{P''_\varphi \cup \mathcal{D}_k}$ is k-periodic, where

$$\begin{array}{rl} D_k := & \{\, [0]\, p(\bar{x}) \mid p(\bar{x}) \in \mathcal{D} \,\} \cup \{\, [0]\, \mathsf{pos}(\mathsf{a}_1) \,\} \\ \cup & \{\, \mathsf{next}(\mathsf{a}_i,\mathsf{a}_{i+1}) \mid i = 1, \ldots, k-1 \,\} \cup \{\, \mathsf{next}(\mathsf{a}_k,\mathsf{a}_1) \,\} \end{array}$$

encodes in $\mathsf{next(X,Y)}$ a cycle of length k. Hence, the period of P''_φ is not bounded.

($\Leftarrow$) Conversely, assume that φ is valid. Then $\mathcal{D} \models \varphi$ for all database $\mathcal{D}$, so the first rule never "fires" and P''_φ always terminates.

(3) We add to every rule of P''_φ the goal "$[S]$ $\mathsf{running}$", which we define by

$$\begin{array}{rl} [0]\, \mathsf{running} & \leftarrow \ . \\ [S{+}1]\, \mathsf{running} & \leftarrow \ [S]\, \mathsf{running}, [S{+}1]\, \neg\, \mathsf{pos}(\mathsf{a}_1). \end{array}$$

For the database $\mathcal{D}_k$ above, $\mathsf{running}$ will become (and remain) *false* when the initial position a_1 is revisited for the first time. Therefore the length of the initial window is bounded iff φ is valid. ■

6.9.2 Run-Time Termination

In this section, we study the problem of detecting termination at run-time. More precisely, we want to decide

- TERMINATION: Given a program P and a database $\mathcal{D}$, does P terminate for $\mathcal{D}$?

Not surprisingly, the problem is **PSPACE**-complete and closely related to TRANSACTION:

Theorem 6.29 (Run-Time Termination) *Given a Statelog program P and a database $\mathcal{D}$, deciding whether P terminates for $\mathcal{D}$ is* **PSPACE***-complete.*

PROOF After at most $n_0 := 2^{|\mathcal{B}_{P\cup\mathcal{D}}|}$ iterations of the repeat-loop in function *evaluate* (Figure 6.6), we can decide whether P terminates for $\mathcal{D}$, which is the case iff $\mathcal{M}_{P\cup\mathcal{D}}[n_0] = \mathcal{M}_{P\cup\mathcal{D}}[n_0{+}1]$ (as usual, we assume that P is in normal form). By storing only the two most recent states, all computations can be accomplished within polynomial space, hence TERMINATION lies in **PSPACE**.

To prove hardness for **PSPACE**, we reduce TRANSACTION to TERMINATION. For a given Statelog program P and ground atom $p(\bar{x})$, let

$$P' := P \cup \{[S{+}1]\, \mathsf{q} \leftarrow [S]\, \neg\, \mathsf{q}, \neg\, p(\bar{x})\} \ .$$

If P' terminates for $\mathcal{D}$, so does P and we have that $\mathcal{M}_{P\cup\mathcal{D}}[\$] \models p(\bar{x})$ (otherwise q would oscillate). If P' does not terminate, then either P also diverges (so $\mathcal{M}_{P\cup\mathcal{D}}[\$] = \emptyset$), or P terminates but $\mathcal{M}_{P\cup\mathcal{D}}[\$] \not\models p(\bar{x})$. Thus, P' terminates for $\mathcal{D}$ iff $\mathcal{M}_{P\cup\mathcal{D}}[\$] \models p(\bar{x})$. The latter is a **PSPACE**-complete problem by Theorem 6.24. ■

In Statelog, it is straightforward to check for the *occurrence* of a fixpoint, and thus to detect termination *when* it occurs: We only have to check whether the current state and the previous state differ—if they are the same, a fixpoint has been reached. For a Statelog program P in normal form over the schema $\mathbf{P}$, we can use the following rules:[23]

$$\begin{array}{l} [S]\, \mathsf{change} \leftarrow \ [S]\, p^{-1}(\bar{X}), \neg\, p(\bar{X}) \ \| \ p \in \mathbf{P}. \\ [S]\, \mathsf{change} \leftarrow \ [S]\, p(\bar{X}), \neg\, p^{-1}(\bar{X}) \ \| \ p \in \mathbf{P}. \\ [S]\, \mathsf{fixpoint} \leftarrow \ [S]\, \neg\, \mathsf{change}. \end{array} \qquad (FP)$$

[23]Here, p^{-1} is defined by the rule $[S{+}1]\, p^{-1}(\bar{X}) \leftarrow [S]\, p(\bar{X})$ (cf. Definition 6.12).

Consider the converse situation: Is it possible to detect *within the language* that P does not terminate for $\mathcal{D}$? Although run-time termination is decidable within **PSPACE** according to Theorem 6.29, it might be the case that this test cannot be encoded in Statelog. This does not contradict the fact that Statelog expresses all DB-**PSPACE** transformations on ordered databases (Corollary 6.14 and Theorem 6.23), since in general an order on the underlying domain may not be available. As will be proven below, it is possible to detect *within* Statelog that P does *not* terminate for $\mathcal{D}$, without resorting to an order on the underlying domain. More precisely, we show that for every Statelog program P there is a transaction equivalent program $P^{\downarrow}$, which always terminates. Notice that this does not contradict Theorem 6.27, since $P^{\downarrow}$ solves the problem only for a *given* database $\mathcal{D}$, i.e., at run-time.

Acceleration and Delay

We start with two preliminary constructions, k-*acceleration* and k-*delay* which are interesting in their own right.

k-acceleration allows to "speed up" the evolution of $\mathcal{M}_{P \cup \mathcal{D}}$ by a factor of k, by encoding the effect of k transitions into a single transition using a rewritten program P'. The price for this "acceleration" is that each state now contains k relations for each original relation.[24] Nevertheless, the technique can be useful if the period of a program is known in advance. Consider for example the standard encoding of WF-Datalog program P as a Statelog program P_{WD} (Section 6.6.2). Since P_{WD} is 2-periodic, accelerating the computation by the factor $k = 2$ yields a terminating program.

Theorem 6.30 (k-Acceleration) *Let P be a NF-Statelog program with schema $\mathbf{P}$. For every $k \geq 1$ there is a NF-Statelog program P' over the schema $\mathbf{P}' = \{p^i \mid p \in \mathbf{P}, 0 \leq i \leq k-1\}$[25] such that for all $p \in \mathbf{P}$, $n \in \mathbb{N}_0$, $i = 0, \ldots, k-1$, and all databases $\mathcal{D}$ over $\mathbf{P}$:*

1. $\mathcal{M}_{P \cup \mathcal{D}} \models [k \cdot n{+}i]\, p(\bar{x}) \Leftrightarrow \mathcal{M}_{P' \cup \mathcal{D}} \models [n]\, p^i(\bar{x})$.
2. *If P is k-periodic, then P' terminates.*
3. *If P terminates, then:* $\mathcal{M}_{P \cup \mathcal{D}}[\$] \models p(\bar{x}) \Leftrightarrow \mathcal{M}_{P' \cup \mathcal{D}}[\$] \models p^i(\bar{x})$.

Proof (1) For every local rule $[S]\, head \leftarrow [S]\, body$ of P, P' contains k local rules

$$[S]\, head^i \leftarrow [S]\, body^i \parallel i = 0, \ldots, k-1.$$

For every 1-progressive rule $[S{+}1]\, head \leftarrow [S]\, body$ of P, P' contains k rules

$$\begin{array}{l} [S]\, head^{i+1} \leftarrow [S]\, body^i \parallel i = 0, \ldots, k-2. \\ [S{+}1]\, head^0 \leftarrow [S]\, body^{k-1}. \end{array}$$

(2) Let P be k-periodic, i.e., for some s_0 and all $s \geq s_0$ we have that $\mathcal{M}_{P \cup \mathcal{D}}[s] = \mathcal{M}_{P \cup \mathcal{D}}[s{+}k]$. Choose n such that $k \cdot n \geq s_0$. Then, for all $i = 0, \ldots, k-1$ we have $\mathcal{M}_{P \cup \mathcal{D}}[k \cdot n{+}i] = \mathcal{M}_{P \cup \mathcal{D}}[k \cdot (n{+}1){+}i]$. Using (1) this implies

$$\mathcal{M}_{P' \cup \mathcal{D}}[n] \models p^i(\bar{x}) \text{ iff } \mathcal{M}_{P' \cup \mathcal{D}}[n{+}1] \models p^i(\bar{x})$$

[24] Cf. the linear speed up theorem for Turing machines, where k symbols are encoded in one [HU79].

[25] Here, we identify p^0 with p.

for all $p^i \in \mathbf{P}'$. Thus, $\mathcal{M}_{P'\cup\mathcal{D}}[n] = M_{P'\cup\mathcal{D}}[n{+}1]$ for all sufficiently large n.

(3) Again, choose n such that $k \cdot n \geq s_0$. We have $\mathcal{M}_{P\cup\mathcal{D}}[k \cdot n] = \mathcal{M}_{P\cup\mathcal{D}}[k \cdot n{+}i]$ and $\mathcal{M}_{P\cup\mathcal{D}} \models [k \cdot n{+}i]\, p(\bar{x}) \Leftrightarrow \mathcal{M}_{P'\cup\mathcal{D}} \models [n]\, p^i(\bar{x})$. Obviously, $\mathcal{M}_{P\cup\mathcal{D}}[k \cdot n]$ is the final state for P, while $\mathcal{M}_{P'\cup\mathcal{D}}[n]$ is the final state for P', from which the claim follows. ■

The following delay technique will be used to simulate a Statelog program P using a terminating program $P^{\downarrow}$ (Theorem 6.33).

Theorem 6.31 (*k*-Delay) *Let P be a NF-Statelog program with schema $\mathbf{P}$, $k \geq 1$. There is a NF-Statelog program P' over the schema $\mathbf{P} \cup \{\mathsf{step}_0, \ldots, \mathsf{step}_k\}$ such that*

$$\mathcal{M}_{P\cup\mathcal{D}} \models [n]\, p(\bar{x}) \Leftrightarrow \mathcal{M}_{P'\cup\mathcal{D}} \models [k \cdot n]\, p(\bar{x}) \ ,$$

for all $p \in \mathbf{P}$, $n \in \mathbb{N}_0$, and all databases $\mathcal{D}$ over $\mathbf{P}$.

Proof

First, we generate a cycle of k successive steps by including the following rules in P':

$$\begin{array}{lcl} [S]\,\mathsf{step}_0 & \leftarrow & [S]\,\mathsf{eq}_0. \\ [S{+}1]\,\mathsf{step}_{i+1} & \leftarrow & [S]\,\mathsf{step}_i \parallel i = 0, \ldots, k-1. \\ [S{+}1]\,\mathsf{step}_0 & \leftarrow & [S]\,\mathsf{step}_k. \end{array}$$

Additionally, for every local rule $[S]\,\mathit{head} \leftarrow [S]\,\mathit{body}$ of P, there are k rules

$$\begin{array}{rcl} [S]\,\mathit{head} & \leftarrow & [S]\,\mathit{body}, \mathsf{step}_0. \\ [S{+}1]\,\mathit{head} & \leftarrow & [S]\,\mathit{head}, \mathsf{step}_i \parallel i = 1, \ldots, k-1. \end{array}$$

in P'. Finally, for every 1-progressive rule $[S{+}1]\,\mathit{head} \leftarrow [S]\,\mathit{body}$ of P, the rule

$$[S{+}1]\,\mathit{head} \leftarrow [S]\,\mathit{body}, \mathsf{step}_k.$$

is in P'. By this construction it is clear, that each transition $[n] \rightsquigarrow [n{+}1]$ in P, corresponds to k transitions $[m] \rightsquigarrow [m{+}1] \rightsquigarrow \cdots \rightsquigarrow [m{+}k]$ in P'. ■

Terminating Simulation

For the construction of $P^{\downarrow}$ we need a lemma which is adapted from [EF95]:

Lemma 6.32 (Repetition) *For every Statelog program P and database $\mathcal{D}$, there is a state $n \geq 1$ such that*

$$\mathcal{M}_{P\cup\mathcal{D}}[n] = \mathcal{M}_{P\cup\mathcal{D}}[2n]\,.$$

Proof By Theorem 6.6, $\mathcal{M}_{P\cup\mathcal{D}}$ is ultimately periodic, i.e., $\mathcal{M}_{P\cup\mathcal{D}}[s] = M_{P\cup\mathcal{D}}[s{+}T]$ for all $s \geq s_o$ and some $T \geq 1$. Thus we can choose some k such that $n := k \cdot T \geq s_0$. For this k we have $\mathcal{M}_{P\cup\mathcal{D}}[k \cdot T] = \mathcal{M}_{P\cup\mathcal{D}}[k \cdot T + k \cdot T]\,.$ ■

The construction used in the proof of part (1) of the following theorem is based on the idea of simultaneously evaluating P and its 2-delayed version. This idea has been applied earlier in the context of Datalog$^{\neg\neg}$ under the name *loop-free simulation* [AS91], and for partial fixpoint logic FO(PFP) in [EF95].

Theorem 6.33 (Terminating Simulation) *For every Statelog program P with schema $\mathbf{P}$, there is a transaction equivalent program $P^{\downarrow}$ which always terminates. In particular,*

1. *one can define relations* $\mathsf{terminates}$ *and* $\mathsf{diverges}$ *in $P^{\downarrow}$ such that for all databases $\mathcal{D}$*

$$\begin{array}{lcl}\mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$] \models \mathsf{terminates} & \Leftrightarrow & \mathcal{M}_{P\cup\mathcal{D}} \textit{ terminates for } \mathcal{D},\\ \mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$] \models \mathsf{diverges} & \Leftrightarrow & \mathcal{M}_{P\cup\mathcal{D}} \textit{ does not terminate for } \mathcal{D}, \textit{and}\end{array}$$

2. *for all $p \in \mathbf{P}$ one can define relations $p_{\boldsymbol{t}}, p_{\boldsymbol{f}}, p_{\boldsymbol{u}}$ in P' such that*

$$\begin{array}{lcl}\mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$] \models p_{\boldsymbol{t}}(\bar{x}) & \Leftrightarrow & \exists n_0 \forall n \geq n_0 : \ \mathcal{M}_{P\cup\mathcal{D}}[n] \models p(\bar{x}),\\ \mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$] \models p_{\boldsymbol{f}}(\bar{x}) & \Leftrightarrow & \exists n_0 \forall n \geq n_0 : \ \mathcal{M}_{P\cup\mathcal{D}}[n] \models \neg\, p(\bar{x}),\\ \mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$] \models p_{\boldsymbol{u}}(\bar{x}) & \Leftrightarrow & \forall n_0 \exists n, m \geq n_0 : \ \mathcal{M}_{P\cup\mathcal{D}}[n] \models p(\bar{x}), \ \mathcal{M}_{P\cup\mathcal{D}}[m] \models \neg\, p(\bar{x}).\end{array}$$

Thus $P^{\downarrow}$ allows to "speak" about (non-)termination of $P \cup \mathcal{D}$. In the case of nontermination, the relations $p_{\boldsymbol{t}}$, $p_{\boldsymbol{u}}$, and $p_{\boldsymbol{f}}$ can be used to determine the atoms which are *eventually true* in *every*, in *some* (but not every), or in *no state*, respectively. As usual, we assume that P is in normal form.

Proof The basic idea of the construction is to execute the original rules of P and the 2-delayed version P' of Theorem 6.31 in parallel. This allows to detect the first state $\mathcal{M}_{P\cup\mathcal{D}}[2n]$ such that $\mathcal{M}_{P\cup\mathcal{D}}[2n] = \mathcal{M}_{P\cup\mathcal{D}}[n]$ (for $n \geq 1$), simply by comparing the results of P on $\mathcal{D}$ with those of P'. Due to the repetition of states, we know that state $[2n]$ is already past the initial window and within the periodic window. Therefore, the fixpoint of the original program P exists iff there is no change from the previous to the current state. $P^{\downarrow}$ is built from the given program P and its 2-delayed version P' as follows: First, in order to avoid interferences between the rules of P and P', we assume that the relations of P' which correspond to the original relations of P are primed in P'. Next, all bodies of rules of $P \cup P'$ are augmented with the goal "$\neg\,(\mathsf{diverges} \wedge \mathsf{ready}_2)$", thereby disabling the rules when the whole computation is finished and divergence has been detected. In addition, $P^{\downarrow}$ comprises the following rules:

The primed relations of P' are initialized at $[0]$; the $\mathsf{different}$ predicate checks whether the results of P and P' are different:

$$[S]\, p'(\bar{X}) \leftarrow\ [S]\, p(\bar{X}), \mathsf{eq}_0 \ \|\ p \in \mathbf{P}.$$

$$\begin{array}{l}[S]\,\mathsf{different} \leftarrow\ [S]\,\mathsf{eq}_0.\\ [S]\,\mathsf{different} \leftarrow\ [S]\, p'(\bar{X}), \neg\, p(\bar{X}) \ \|\ p \in \mathbf{P}.\\ [S]\,\mathsf{different} \leftarrow\ [S]\, p(\bar{X}), \neg\, p'(\bar{X}) \ \|\ p \in \mathbf{P}.\end{array}$$

The change predicate compares previous and current values of relations from P:

$$\begin{array}{l}[S]\,\mathsf{change}_p \leftarrow\ [S]\, p^{-1}(\bar{X}), [S]\,\neg\, p(\bar{X}) \ \|\ p \in \mathbf{P}.\\ [S]\,\mathsf{change}_p \leftarrow\ [S]\, p(\bar{X}), [S]\,\neg\, p^{-1}(\bar{X}) \ \|\ p \in \mathbf{P}.\end{array}$$

$$[S]\,\mathsf{change}_{\mathbf{P}} \leftarrow\ [S]\,\mathsf{change}_p \ \|\ p \in \mathbf{P}. \quad \%\ \textit{thus, } [0]\,\mathsf{change}_{\mathbf{P}} \textit{ iff } \mathcal{D} \neq \emptyset$$

As soon as $\mathsf{different}$ becomes *false*, it is checked whether there has been a change in the relations of P. If so, termination is detected $(*{\downarrow}*)$; otherwise we know that P diverges for $\mathcal{D}$ $(*{\uparrow}*)$. The first occurrence of ready_1 marks the point when (non)termination is detected:

$$
\begin{array}{ll}
[S]\,\mathsf{terminates} \leftarrow\ [S]\,\neg\,\mathsf{different}, \neg\,\mathsf{change}_{\mathbf{P}}, \neg\,\mathsf{diverges}. & \%\ (*{\downarrow}*)\\
[S{+}1]\,\mathsf{terminates} \leftarrow\ [S]\,\mathsf{terminates}. & \\
[S]\,\mathsf{diverges} \leftarrow\ [S]\,\neg\,\mathsf{different}, \mathsf{change}_{\mathbf{P}}. & \%\ (*{\uparrow}*)\\
[S{+}1]\,\mathsf{diverges} \leftarrow\ [S]\,\mathsf{diverges}. & \\
[S]\,\mathsf{ready}_1 \leftarrow\ [S]\,\mathsf{terminates}. & \\
[S]\,\mathsf{ready}_1 \leftarrow\ [S]\,\mathsf{diverges}. &
\end{array}
$$

The construction so far essentially establishes part (1). For part (2), additional rules are required to compute relations $p_{\boldsymbol{t}}$ and $p_{\boldsymbol{u}}$ in the phase between ready_1 and ready_2.
For all $p \in \mathbf{P}$:

$$
\begin{array}{ll}
[S]\,p_{old}(\bar{X}), p_{\boldsymbol{t}}(\bar{X}), p_{\boldsymbol{u}}(\bar{X}) \leftarrow\ [S]\,p(\bar{X}), \mathsf{ready}_1, \neg\,\mathsf{ready}_1^{-1}. & \%\ \textit{initialization}\\
[S{+}1]\,p_{old}(\bar{X}) \leftarrow\ [S]\,p_{old}(\bar{X}). & \\
[S{+}1]\,p_{\boldsymbol{t}}(\bar{X}) \leftarrow\ [S]\,p_{\boldsymbol{t}}(\bar{X}), p(\bar{X}), \mathsf{ready}_1, \neg\,\mathsf{ready}_2. & \%\ \textit{intersect with } p(\bar{X})\\
[S{+}1]\,p_{\boldsymbol{t}}(\bar{X}) \leftarrow\ [S]\,p_{\boldsymbol{t}}(\bar{X}), \mathsf{ready}_2. & \\
[S{+}1]\,p_{\boldsymbol{u}}(\bar{X}) \leftarrow\ [S]\,p_{\boldsymbol{u}}(\bar{X}), \mathsf{ready}_1, \neg\,\mathsf{ready}_2. & \%\ \textit{compute union}\\
[S{+}1]\,p_{\boldsymbol{u}}(\bar{X}) \leftarrow\ [S]\,p(\bar{X}), \mathsf{ready}_1, \neg\,\mathsf{ready}_2. & \%\ \ldots\ \textit{with } p(\bar{X})\\
[S{+}1]\,p_{\boldsymbol{u}}(\bar{X}) \leftarrow\ [S]\,p_{\boldsymbol{u}}(\bar{X}), \mathsf{ready}_2. & \\
[S]\,p_{\boldsymbol{f}}(\bar{X}) \leftarrow\ [S]\,\neg\,p_{\boldsymbol{u}}(\bar{X}). &
\end{array}
$$

The last (unsafe) rule computes the complement of $p_{\boldsymbol{u}}$ wrt. the active domain $adom(\mathcal{D})$. Since $adom(\mathcal{D})$ is easily defined by rules, the rule can be made safe by adding corresponding goals $\mathsf{adom}(X_i)$ for $i = 1, \ldots, n$ and $\bar{X} = X_1, \ldots, X_n$.

The intermediate phase after ready_1 is finished when the state remembered in p_{old} is encountered again, i.e., when there are no more changes:

$$
\begin{array}{l}
[S]\,\mathsf{more} \leftarrow\ [S]\,p_{old}(\bar{X}), \neg\,p(\bar{X}), \neg\,\mathsf{ready}_2 \ \|\ p \in \mathbf{P}.\\
[S]\,\mathsf{more} \leftarrow\ [S]\,p(\bar{X}), \neg\,p_{old}(\bar{X}), \neg\,\mathsf{ready}_2 \ \|\ p \in \mathbf{P}.\\
[S{+}1]\,\mathsf{ready}_2 \leftarrow\ [S]\,\mathsf{ready}_1, \neg\,\mathsf{more}.\\
[S{+}1]\,\mathsf{ready}_2 \leftarrow\ [S]\,\mathsf{ready}_2.
\end{array}
$$

It should be clear that the presented construction guarantees that $P^{\downarrow}$ always terminates, and derives "$\mathsf{terminates}$" ("$\mathsf{diverges}$") iff P terminates (diverges) for $\mathcal{D}$. Moreover, $\mathcal{M}_{P^{\downarrow}\cup\mathcal{D}}[\$]|_{\mathbf{P}} = \mathcal{M}_{P\cup\mathcal{D}}[\$]$: in case of nontermination of P, all relations $p \in \mathbf{P}$ are emptied by $P^{\downarrow}$ in accordance with Definition 6.7. ■

The proof of Theorem 6.33 provides a theoretical construction to decide within Statelog whether a program P terminates for $\mathcal{D}$. There are obviously simpler ways to detect run-time (non)termination in an implementation and to determine the relations $p_{\boldsymbol{t}}$, $p_{\boldsymbol{u}}$, and $p_{\boldsymbol{f}}$. Theorem 6.33 assures that adding such features does not increase the expressiveness of the language. An important practical application of the theorem is the possibility to program compensating reactions in response to nonterminating rules: The simplest strategy is to restore the old database state before the update. The information which tuples are oscillating between true and false (those in $p_{\boldsymbol{u}}$), or which are always true (those in $p_{\boldsymbol{t}}$) can be useful in implementing more elaborated strategies.

A very different approach to enforcing termination is presented in Section 7.5.2, where a tractable class of terminating Statelog programs is introduced.

6.10 Related Approaches

In this section, we compare Statelog to closely related approaches based on Datalog and some notion of state. In Section 7.9, we comment on further related work on formal frameworks for active rules and updates in deductive databases.

6.10.1 Datalog$_{1S}$

Datalog$_{nS}$ [CI88, CI93, Cho95a] is a class of logic programs where function symbols are either unary or limited n-ary and may only appear in one distinguished argument. The subset *Datalog$_{1S}$* allowing only a single unary function symbol "+1" is an important special case, and is used as a query language for temporal databases as defined in [BCW93b]:

Let $inst(\mathbf{R})$ be the set of all relational database instances over a given schema $\mathbf{R}$. A *temporal database* $\mathcal{TD}$ with time isomorphic to $\mathbb{N}_0$ is a mapping $\mathcal{TD} : \mathbb{N}_0 \to inst(\mathbf{R})$, i.e., an infinite sequence of relational databases. Thus, the Statelog evolution of a database (Definition 6.11) may be viewed as a temporal database and vice versa.

However, there is a major conceptual difference between these viewpoints: The Statelog evolution corresponds to the intermediate steps of a transaction, transforming a given relational database $\mathcal{D}$ to the updated database $\mathcal{D}'$. We require only for $\mathcal{D}$ and $\mathcal{D}'$ to represent meaningful snapshots of the modeled world, whereas intermediate states can be seen as holding "scratch paper" relations, not necessarily reflecting a consistent view of the real world. In contrast, a temporal database does not deal with updates per se, but statically describes temporal phenomena. For example, periodic models with period $\neq 1$ are undesirable in Statelog, but very natural for representing periodic phenomena in Datalog$_{1S}$. Consider the following example taken from [Cho95a]:

Example 6.9 (Temporal Database) We want to describe the schedule of meetings of a given professor with her students:

```
meets_first(0, emma).
next(emma,kathy).
next(kathy,emma).

meets(T+1, X) ← meets_first(T, X).
meets(T+1, Y) ← meets(T, X), next(X,Y).
```

The first arguments of meets_first and meets are temporal; T is the temporal variable and corresponds to the state variable S of Statelog. The minimal model of these rules induces a temporal database $\mathcal{TD}$ containing the infinite sequence of facts meets(0,emma), meets(1,kathy), meets(2,emma), meets(3,emma), etc. It is easy to see that $\mathcal{TD}$ is ultimately 2-periodic. □

Syntax. Although (reified) Statelog and Datalog$_{1S}$ are quite similar syntactically, there are some crucial differences: Statelog rules are required to be *progressive*, because they define transitions from past to future states. In contrast, there is no concept like transition or transaction in Datalog$_{1S}$; instead, a single infinite *temporal* database is modeled, possibly with *non-progressive* rules. For example, the rules

$$\boxed{\begin{array}{l}\mathsf{u}(T) \leftarrow \mathsf{p}(T).\\ \mathsf{u}(T) \leftarrow \mathsf{u}(T{+}1).\end{array}}$$

define that u holds iff $\Diamond$p holds, where "$\Diamond$" means *sometime in the future* [BCW93b]. Such a statement is not meaningful in Statelog, since updates happening in the future are unknown and can not be used to define the current state.

On the other hand, Statelog programs allow for state-stratified negation, whereas Datalog$_{1S}$ has only been investigated for positive and (predicate) stratified programs. Indeed, as illustrated in Example 6.5, the combination of non-progressive rules and negation easily leads to programs with unclear total semantics. Summarizing, compared to Datalog$_{1S}$, Statelog is more restrictive wrt. progressiveness of rules and less restrictive wrt. the use negation.

Expressiveness. Statelog and Datalog$_{1S}$ give rise to different notions of expressiveness. In Statelog, transition and—more importantly—transaction expressiveness characterize the updates definable in the language. In contrast, there are two other notions in Datalog$_{1S}$, i.e., data expressiveness and query expressiveness. The *data expressiveness* of a temporal database formalism is the set of temporal databases definable in this formalism [BCW93b]. For Datalog$_{1S}$, it has been shown that the data expressiveness are the ultimately periodic sets [CI88, Cho90, CI93]. Note that for relational databases, data expressiveness is not an issue, because all relational databases can be represented as finite conjunctions of ground atoms.

The second notion, *query expressiveness* is the class of boolean (yes/no) queries definable on the set of all temporal databases. This notion is the temporal analogue to the "classical" notion of expressiveness for relational query languages. In [Bau89, Bau95] the query expressiveness of *Templog*, a syntactical variant of Datalog$_{1S}$, has been investigated and compared with the temporal fixpoint calculus μTL [Var88]. It is shown that the query expressiveness of monadic[26] Datalog$_{1S}$ with (predicate) stratified negation corresponds to the class of all ω-regular languages, and to μTL [BCW93b].

Complexity. The data complexity of evaluating Datalog$_{nS}$ queries is the complexity of deciding whether a given ground temporal atom is contained in the minimal (or stratified) model of a Datalog$_{nS}$ program P with database $\mathcal{D}$ (cf. Section 2.2). This recognition problem is **PSPACE**-complete for Datalog$_{1S}$ and **EXPTIME**-complete for Datalog$_{nS}$ [CI88, Cho95a]. Algorithms for depth-bounded bottom-up evaluation working in **PSPACE** and **EXPTIME**, respectively, are presented in [Cho95a].

Summarizing, although Statelog and Datalog$_{1S}$ are closely related, the formal results for expressiveness and complexity of Datalog$_{1S}$ can not be directly applied to Statelog due to substantial differences in concepts and notions of expressiveness. For Statelog, we have presented several normal forms which allow to characterize the transition and transaction expressiveness of the language, as well as its complexity. Moreover, we have studied the problem of detecting

[26] I.e., all relations have temporal arity 1 and data arity 0.

termination of programs both at compile-time and at run-time, which is crucial in the context of Statelog updates, but not an issue in the context of $Datalog_{1S}$, because there is no distinguished final state.

6.10.2 XY-Datalog

In [Zan93, ZS94], *XY-Datalog* has been proposed as a unified framework for active and deductive rules. Like Statelog rules, XY-Datalog rules are syntactically very close to $Datalog_{1S}$: relation symbols have a distinguished first argument, the *stage argument*, which corresponds to the temporal argument of $Datalog_{1S}$ and to the state terms of Statelog. A stage argument may be either J, $J{+}1$, or a stage constant; the variable J may only appear in the stage argument. A recursive XY-Datalog rule r is an

- *X-rule*, if all predicates in r share the same stage argument J, and a
- *Y-rule*, if (i) some goal of r has stage argument J, (ii) the head of r has stage argument $J{+}1$, and (iii) all other stage arguments are either J or $J{+}1$.

A canonical semantics for negation is obtained by requiring that programs are XY-stratified:

The *primed version* P' of a XY-Datalog program P is obtained by priming all occurrences of recursive predicates in X-rules, and all predicates with stage argument $J{+}1$ in Y rules. P is called *XY-stratified* if (i) all recursive rules are either X-rules or Y-rules, (ii) the primed version P' is non-recursive, and (iii) all exit rules have as stage argument the same constant [Zan93].[27]

The conditions for XY-stratification guarantee that programs are locally stratified and, therefore, possess a unique perfect model. Moreover, it should be clear that XY-stratification implies state-stratification, X-rules correspond to local Statelog rules, and Y-rules to strictly progressive rules (cf. Definition 6.3). Before presenting an example adopted (with minor variations) from [Zan93], we introduce some commonly used syntactic sugar, which will also be used in subsequent examples.

Notation. Each occurrence of the *anonymous variable* "_" is considered as a distinct new variable, which is implicitly $\exists$-quantified at the *innermost* level. Hence,

$$\boxed{p(X) \leftarrow q(X), \neg\, r(X, \_, \_).} \quad \Leftrightarrow \quad \boxed{p(X) \leftarrow q(X), \neg\, \exists X_1, X_2\; r(X, X_1, X_2).}$$

which is expressible by two conventional rules:

$$\boxed{\begin{array}{l} p(X) \leftarrow q(X), \neg\, r'(X). \\ r'(X) \leftarrow r(X, X_1, X_2). \end{array}}$$

Example 6.10 (Minimal Distance) The program computes the nodes X of a graph g reachable from a given node a, together with the minimal distance from a:

$$P: \quad \boxed{\begin{array}{lll} r_0 : & \mathsf{new}(0,\mathsf{X},1) & \leftarrow \mathsf{g}(\mathsf{a},\mathsf{X}). \\ r_1 : & \mathsf{new}(J{+}1,\mathsf{Y},\mathsf{D1}) & \leftarrow \mathsf{new}(J,\mathsf{X},\mathsf{D}),\ \mathsf{g}(\mathsf{X},\mathsf{Y}), \\ & & \quad \neg\,\mathsf{all}(J,\mathsf{Y},\_),\ \mathsf{D1}{=}\mathsf{D}{+}1. \\ r_2 : & \mathsf{all}(J{+}1,\mathsf{Y},\mathsf{D}) & \leftarrow \mathsf{all}(J,\mathsf{Y},\mathsf{D}),\ \mathsf{new}(J{+}1,\_,\_). \\ r_3 : & \mathsf{all}(J,\mathsf{Y},\mathsf{D}) & \leftarrow \mathsf{new}(J,\mathsf{Y},\mathsf{D}). \end{array}}$$

[27] A non-recursive rule defining a recursive predicate is called *exit rule*.

All nodes directly reachable from a have distance 1 and are new at stage 0 (r_0). Additionally, a node Y is new at stage $J{+}1$, provided the node X has been derived anew in the previous stage J, g(X,Y) holds, and Y has not been derived previously (r_1). Finally, all stores all previously derived nodes together with there minimum distance from a (r_2, r_3). Note that r_2 ceases to fire if there are no more tuples in new. Therefore, only finitely many stages J are created.

Given the definitions above, it is easy to see that r_3 is an X-rule, r_1 and r_2 are Y-rules, and r_0 is an exit rule. The primed version of P is non-recursive, so P is XY-stratified. □

The Statelog notions of transition and transaction expressiveness carry over naturally to XY-Datalog. Using the results established for Statelog, we are immediately able to characterize the expressive power of XY-Datalog:

Corollary 6.34 (XY-Datalog $\equiv_\$$ P-Statelog) *For every XY-Datalog program there is a transaction equivalent P-Statelog program and vice versa.*

Proof ($\leq_\$$) holds, since every XY-Datalog program can be viewed as a Statelog program in the obvious way, and Statelog $\equiv_\$$ P-Statelog by Theorem 6.10.

($\geq_\$$) holds, since the 1-progressive rules of a P-Statelog program correspond to a set of XY-stratified Y-rules. ■

Like for Statelog, this implies that in general evaluation of XY-Datalog rules is **PSPACE**-complete, and termination is undecidable at compile-time. The transition expressiveness of XY-Datalog lies between Statelog and P-Statelog:

Theorem 6.35 (XY-Datalog $\equiv_1$ FO) *XY-Datalog is transition equivalent to* FO.

Proof ($\leq_1$) The transition from one stage to the next is given by the *non-recursive* X-rules, and *one* application of the Y-rules. Since there is no recursion, the new stage can be obtained by a first-order query from the current stage.

($\geq_1$) Let φ be a first-order formula. As is well-known, φ can be translated into a *non-recursive* S-Datalog program (cf. Figure 4.2) which in turn corresponds to a set of X-rules. ■

Recently, a slightly different version of XY-Datalog was introduced which also allows X-rules to be recursive [Zan97, ZCF+97]. This modification does not affect the overall (transaction) expressiveness, but increases the transition expressiveness from ∃FO to S-Datalog. [Zan97] also shows how XY-Datalog can be combined with the choice construct to obtain a nondeterministic semantics.

6.10.3 Further Approaches

The idea of using some kind of "state index" for relations has come up several times, apart from Statelog, Datalog$_{IS}$, and XY-Datalog. For example in [KLS92], a rule language integrating updates and deductive rules has been proposed. Since the underlying data model is object-oriented, instead of state terms, so-called *version identifiers* are used to allow for the specification of updated objects. Using version identifiers, a certain stratification is defined on rules, which allows to execute the rules stratum by stratum using an iterated fixpoint procedure.

In [KRS95], the class of *explicitly locally stratified* (ELS) programs was suggested as a means to compute the well-founded model of non-stratified programs using standard evaluation techniques. ELS-programs are very similar to XY-Datalog and Statelog programs. However, like Statelog rules and unlike X-rules, ELS-rules allow recursion within a state. Thus, the authors state in [KRS95] that *"ELS-programs are more general than XY-programs"*. This can be made more precise by our distinction between transitions and transactions: XY-Datalog is indeed less expressive wrt. *transitions*, since only FO transitions can be expressed (Theorem 6.35). On the other hand, both languages have the same expressive power wrt. *transactions*, since local recursion can be simulated by progressive recursion (cf. Theorem 6.10, Corollary 6.34).

6.11 Summary

Statelog is a state-oriented extension of Datalog integrating the declarative semantics of deductive rules and the capability to define updates in the style of production rules. In this chapter, we have focused on a thorough investigation of theoretical properties of the language, most notably state-stratified semantics, normal forms, expressive power, complexity, and termination:

Every state-stratified Statelog program P with database $\mathcal{D}$ is locally stratified (Theorem 6.4), hence has a unique intended model $\mathcal{M}_{P\cup\mathcal{D}}$ which is ultimately periodic (Theorem 6.6). W.l.o.g., P can be assumed to be in normal form (Theorem 6.9). Deductive rules and production rules in the style of P-Datalog turn out to be special cases of Statelog rules, or have natural representations in the language (cf. Figure 6.4). In particular, by relating Statelog rules with production rules in the style of P-Datalog, a precise characterization of the overall Statelog expressiveness is obtained (Corollary 6.12 and Theorem 6.10). Statelog programs can be evaluated using standard techniques developed for deductive databases. In general, single transitions can be evaluated in **PTIME**, whereas transactions may require **PSPACE**. Figure 6.7 summarizes the main expressiveness results; see Figure 4.2 for the corresponding complexities.

	Equivalence wrt.	
Language	Transitions ($\equiv_1$)	Transactions ($\equiv_\$$)
I-Statelog	∃FO	FIXPOINT
P-Statelog	∃FO	WHILE
XY-Datalog	FO	WHILE
NF-Statelog	S-DATALOG	WHILE
Statelog	S-DATALOG	WHILE
WF-Statelog	FIXPOINT	WHILE

Figure 6.7: Summary of expressiveness results

The termination behavior of Statelog has been analyzed: It is undecidable whether a Statelog program terminates for all databases, i.e., at compile-time (Theorem 6.27), while termination can be detected within **PSPACE** at run-time (Theorem 6.33). Our results also yield new insight into the properties of the closely related language XY-Datalog; in particular, they imply that evaluation of XY-Datalog rules is **PSPACE**-complete in general, and that termination can be checked at run-time *within* the rule language.

7 A Statelog Framework for Active Rules

7.1 Introduction

In the previous chapter, theoretical properties of "pure" Statelog have been investigated, and it was shown that certain types of deductive and production rules correspond to special Statelog rules. In the present chapter, a logical framework for active rules is developed, based on pure Statelog as the underlying core language. Whereas previously there was no intrinsic connection between relation names like "emp" and "del_emp", in the sequel, we will distinguish between different types of relations and fix their meaning using special system-defined rules. In this framework, the user is relieved from dealing with low-level procedural aspects like the specification of primitive updates, or of transaction control predicates like abort and commit. Moreover, since the meaning of primitive updates is completely specified by system-defined rules in the extended framework, the problem of oscillating update requests and nontermination can be solved by restricting to Δ-*monotonic* rules. Such rules have the additional benefit of being evaluable in polynomial time. Next, we show how to handle integrity constraints in Statelog, in particular *dynamic* constraints given as *past first-order temporal logic* (PTL) formulas. For these, a space efficient evaluation method was developed in [Cho95b], which directly translates to the Statelog framework. The handling of composite events in Statelog is discussed and, using Chomicki's notion of *bounded history encoding*, it is shown that composite events with chronicle context do not admit a bounded history encoding.

7.2 Signature

In order to model the specific features of active rules, we introduce several types of relations; their precise meaning will be specified by system-defined Statelog below. In the sequel, we assume that the schema $\mathbf{P}$ of a Statelog program P is composed of the following disjoint sets of relation symbols:

$$\mathbf{P} = edb(\mathbf{P}) \,\dot\cup\, idb(\mathbf{P}) \,\dot\cup\, \delta(\mathbf{P}) \,\dot\cup\, \triangleright ev(\mathbf{P}) \,\dot\cup\, \triangleleft act(\mathbf{P}) \,\dot\cup\, ctl(\mathbf{P}).$$

As explained below, some of these relations are definable by the user, while others are automatically maintained by the system.

7.2.1 Base and Derived Relations

The *extensional database* $edb(\mathbf{P})$ comprises all base relations stored explicitly in the database. In particular, the initial database $\mathcal{D}$ at [0] is given as a finite set of *edb*-relations. These can be updated by the user through so-called *delta relations* from $\delta(\mathbf{P})$. Sometimes it is useful to distinguish a set $cdb(\mathbf{P}) \subseteq edb(\mathbf{P})$ of *constant* (i.e., immutable) base relations, which cannot be changed by delta relations. In user-defined rules, *edb*-relations may only occur in rule bodies; they are propagated to subsequent states—modulo the changes induced by delta relations—by system-defined frame rules.

The *intensional database* $idb(\mathbf{P})$ consists of the set of derived relations, that is, which are used to define *views*. Thus, *idb*-relations may occur in heads and bodies of user-defined rules, but may not be changed directly through delta relations. Instead, the underlying *edb*-relations have to be updated. Typically, *idb*-relations are not materialized but computed on demand.

7.2.2 Delta Relations

For every base relation $p \in edb(\mathbf{P})$ there are *delta relations* $\mathsf{\textbf{del}}{:}p, \mathsf{\textbf{ins}}{:}p, \mathsf{\textbf{mod}}{:}p \in \delta(\mathbf{P})$. When used in the head of rules, delta relations (or just *deltas*) for a relation p correspond to *internal actions* (Section 3.2.3), i.e., requests to delete, insert, or modify the corresponding tuples. The current values of deltas can be queried by using them in rule bodies. An update request derived in state $[n]$ becomes effective in the transition to the next state $[n{+}1]$, which corresponds to an immediate coupling between conditions (evaluation of rule bodies) and actions (execution of requests defined in rule heads; cf. Section 3.3). The precise semantics of deltas is specified by system-defined frame rules.

The notation $\mathsf{\textbf{del}}{:}p$, $\mathsf{\textbf{ins}}{:}p$, and $\mathsf{\textbf{mod}}{:}p$ emphasizes the special role of delta relations and allows to distinguish between system-defined relations and ordinary user-defined relations.[1] To simplify the presentation, we view modification requests of the form $\mathsf{\textbf{mod}}{:}p(\bar{X}/\bar{X}')$ as an abbreviation for the *simultaneous* deletion of the old and insertion of the new value, i.e., for every $p \in edb(\mathbf{P})$:

$$\mathsf{\textbf{mod}}{:}p(\bar{X}/\bar{X}') :\Leftrightarrow \mathsf{\textbf{del}}{:}p(\bar{X}) \wedge \mathsf{\textbf{ins}}{:}p(\bar{X}').$$

7.2.3 External Events and Actions

The interface between a Statelog program an the environment is modeled by external events and actions: Relations from $\triangleright ev(\mathbf{P})$ represent *external events* of interest, which have to be monitored by the ADB. Consequently, external events can only occur in rule bodies. The occurrence of an external event e with parameters $\bar{x}$ is modeled by asserting the fact $\triangleright e(\bar{x})$ in the current (final) state of the database. Unlike *edb*-relations, external events are *not* propagated to subsequent states.

Relations from $\triangleleft act(\mathbf{P})$ define *external actions* and represent requests to execute certain actions outside the ADB system. It is assumed that these actions have no side effect on database relations. To distinguish external actions from other relation symbols, we denote them in the form $\triangleleft a(\bar{x})$.

7.2.4 Control Relations

The set of *control relations* $ctl(\mathbf{P})$ contains special auxiliary relations for transaction control (e.g., BOT, EOT, abort, running), for historical information and the detection of composite events (Sections 7.6.2 and 7.7), and for *protocol relations* $\mathsf{\textbf{insd}}{:}p$ and $\mathsf{\textbf{deld}}{:}p$ (for **ins**erte**d** and **del**ete**d**). Protocol relations store the accumulated *net effect* of a sequence of updates, and

[1] The notion del_p was used in Chapter 6, since there was no distinction between user- and system-defined relations; cf. Example 6.1.

maintain all necessary information for aborting a transaction. Similar relations can be used to implement deferred execution (Section 7.4.2), or to enforce termination (Section 7.5).

7.3 Refined Execution Model

The basic Statelog execution model in Figure 6.1 illustrates a single Statelog transaction. In the extended framework, we refine this model and consider a stream of incoming external events $\mathcal{E}_0, \mathcal{E}_1, \ldots$ which induces (i) a sequence of transactions and final states, and (ii) a stream of outgoing external actions $\mathcal{A}_0, \mathcal{A}_1, \ldots$ (Figure 7.1): Assume P is a Statelog program, and $[s_i]$ is the current (final) state of the database. The database remains in this state, as long as no new external events occur. Queries are executed against $[s_i]$ and may involve base relations from $edb(\mathbf{P})$, local views from $idb(\mathbf{P})$, or historical information using auxiliary relations from $ctl(\mathbf{P})$ (Section 7.6.2).

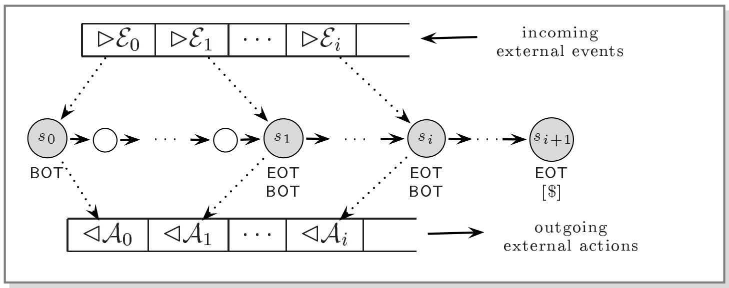
The (simultaneous) occurrence of a set of external events $\mathcal{E}_i$ is modeled by asserting a finite set of facts

$$\mathcal{E}_i = \{[s_i] \rhd e(\bar{x}) \mid \text{event } e(\bar{x}) \text{ has occurred}\}$$

at the i-th final state $[s_i]$. In general, using this new "seed" information, the rules of P define a new transaction to the next final state $[s_{i+1}]$. Subsequently occurring events $\mathcal{E}_{i+1}$ are mapped to this new final state. Let $\mathcal{E}_{\leq k}$ denote the set of events occurring up to state $[k]$, that is,

$$\mathcal{E}_{\leq k} := \bigcup_{s_i \leq k} \mathcal{E}_i .$$

It should be clear from Figure 7.1 that the Statelog model $\mathcal{M}_{P \cup \mathcal{D} \cup \mathcal{E}_{\leq k}}$ is "add-only", i.e., past states cannot be changed, and new events influence only the current and future states. System-defined predicates BOT and EOT can be used to distinguish between the different kinds of states.



Classification of states

state	BOT	EOT
initial	1	0
intermediate	0	0
past final	1	1
current final	0	1

Figure 7.1: Mapping of external events to final states and classification of states

7.4 User-Defined vs. System-Defined Rules

A benefit of the extended Statelog framework for active rules is that it relieves the programmer from specifying procedural details of transitions and transaction control, by providing a pre-defined programming "environment" which takes care of these low-level aspects. Nevertheless, the complete framework enjoys a precise formal semantics, since the meaning of user-defined and system-defined rules is given by the logical semantics of Statelog.

7.4.1 User-Defined Rules

In the extended framework, we can restrict user programs to local rules, since updates which cause the transition to the next state are definable via delta relations. Hence, we define a *Statelog user program* to be a finite set of local Statelog rules, where only relations from $idb(\mathbf{P})$, $\delta(\mathbf{P})$, $\lhd act(\mathbf{P})$, and certain distinguished relations from $ctl(\mathbf{P})$ are allowed to occur in rule heads. More precisely, we can distinguish the following types of user-defined rules:

View Definitions: $[S]\, p(\bar{X}) \leftarrow [S] \ldots$, where $p \in idb(\mathbf{P})$.

Such rules are used to define (possibly recursive) views, and correspond to the usual deductive rules.

Internal Actions: $[S]\, \textbf{ins}{:}p(\bar{X})/\textbf{del}{:}p(\bar{X})/\textbf{mod}{:}p(\bar{X}, \bar{X}') \leftarrow [S] \ldots$, where $p \in edb(\mathbf{P})$.

These rules allow to express requests to update base relations. As illustrated by Example 6.1, (re)active programming is possible by querying deltas in the body and defining new deltas in the head. The meaning of delta relations and the handling of conflicting requests is specified by system-defined frame rules.

External Actions: $[S] \lhd a(\bar{X}) \leftarrow [S] \ldots$, for $\lhd a \in \lhd act(\mathbf{P})$.

The effect of external actions is not specified within the Statelog framework. Logically, they can be seen as "write-only" built-in relations.

Integrity Maintenance: $[S]\, \mathsf{abort} \leftarrow [S] \ldots$

A simple mechanism to enforce integrity is to abort transactions which would otherwise violate the given integrity constraints. The semantics of abort is specified by system-defined rules. In Section 7.6 we show how the user can express static and dynamic constraints in Statelog.

To simplify notation, we may omit the state term "$[S]$" in user-defined rules, since these are local by definition:

Example 7.1 (Traffic Light Control) The following program models the state transitions of a simple traffic light. After each external tick[2] occurrence, the next phase is computed:

r_1 :	**ins**:on(yellow)	$\leftarrow \rhd$tick, on(red), $\neg$ on(yellow).
r_2 :	**del**:on(red), **del**:on(yellow), **ins**:on(green)	$\leftarrow \rhd$tick, on(red), on(yellow).
r_3 :	**del**:on(green), **ins**:on (yellow)	$\leftarrow \rhd$tick, on(green).
r_4 :	**del**:on(yellow), **ins**:on(red)	$\leftarrow \rhd$tick, on(yellow), $\neg$ on(red).
r_5 :	$\lhd$light_on(X)	$\leftarrow$ on(X).
r_6 :	$\lhd$light_off(X)	$\leftarrow \neg$ on(X).

Given the initial database $\mathcal{D} = \{[0]\,\mathsf{on(red)}\}$, the first external tick occurrence leads to the assertion of $[0] \rhd \mathsf{tick}$. Thus, by r_1, the new final state is $\{\mathsf{on(red)}, \mathsf{on(yellow)}\}$. After the next tick, the new final state is $\{\mathsf{on(green)}\}$, etc. The rules r_5 and r_6 model the output to some external devices. Note that the program provides a partial specification only. For example, if the initial state is $\{\mathsf{on(red)}, \mathsf{on(yellow)}, \mathsf{on(green)}\}$, then both r_2 and r_3 fire, leading to conflicting

[2]See http://www.m-w.com/mw/art/tick.htm for a completely different type of tick.

[n]			[$n{+}1$] $p(\bar{x})$				
$p(\bar{x})$	**ins**:$p(\bar{x})$	**del**:$p(\bar{x})$	INS	DEL	NOP	DOP	ABORT
false	*false*	*false*	*false*	*false*	*false*	*false*	*false*
false	*false*	*true*	*false*	*false*	*false*	*false*	*false*
false	*true*	*false*	*true*	*true*	*true*	*true*	*true*
false	***true***	***true***	***true***	***false***	***false***	***true***	$\perp$
true	*false*	*false*	*true*	*true*	*true*	*true*	*true*
true	*false*	*true*	*false*	*false*	*false*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*	*true*
true	***true***	***true***	***true***	***false***	***true***	***false***	$\perp$

Figure 7.2: Deterministic conflict resolution policies

actions **ins**:on(green) and **del**:on(green) (similar for yellow). The problem can be avoided either by extending the program such that it covers all situations, or by using one of the conflict resolution policies below. □

7.4.2 System-Defined Rules

System-defined rules are used to specify the basic functionality of active rules and transaction control.

Beginning of Transaction

If one or more external events occur, the beginning of the transaction has to be signaled. Thus, for every external event $\triangleright e$, there is a rule

$$[S]\,\mathsf{BOT} \leftarrow [S]\,\triangleright e(\bar{X}).$$

As explained in Section 7.3, all external events which are detected within a certain time unit are raised in the most recent final state; all events occurring subsequently are associated with the next final state as soon as the latter is completely computed.

Frame Rules and Conflict Resolution

The meaning of basic update requests is specified using *frame rules*[3], which define the new values of *edb*-relations at the next state [$n{+}1$], given the values of delta relations and *edb*-relations at the current state [n]. Basically, all tuples not requested for deletion are propagated to the next state. Thus, from a logic point of view, no destructive update occurs, but a (possibly smaller) copy of the current state is used at the next state. Additionally, insertions have to be processed. A conflict arises if the same tuple is requested for both insertion and deletion. In Figure 7.2 all possible deterministic conflict resolution policies are depicted: On the left, all combinations of truth-values of p, **ins**:$p(\bar{x})$, and **del**:$p(\bar{x})$ at [n] are listed. Out of the eight cases only two (the framed ones) lead to a conflict. The new truth-value of $p(\bar{x})$ at [$n{+}1$] is shown on the right-hand side for each of the five deterministic conflict resolution policies.[4]

[3]Frame rules can be seen as a very restricted form of *frame axioms* [MH69].

[4]These policies—apart from DOP—have also been considered for Datalog$^{\neg\neg}$; see, e.g., [Via97b].

INS: This policy assigns priority to insertions, and is implemented for relations $p \in edb(\mathbf{P})$ as follows:

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$

DEL: Analogously, policy DEL gives priority to deletions:

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$

NOP: With this policy, a conflict is regarded as a "*no-op*", and simply leaves the old value of p unchanged:

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$

DOP: In contrast, the bustling "*do-op*" policy always complements the old value of p in case of a conflict:

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$

ABORT: The final possibility is to abort the transaction in case of a conflict:

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, p(\bar{X}), \neg\, \textsf{\textbf{del}}{:}p(\bar{X}), \neg\, \textsf{abort}.$$
$$[S]\, \textsf{abort} \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \textsf{\textbf{del}}{:}p(\bar{X}).$$

In this case, the value at $[n{+}1]$ is not determined from the values of $p(\bar{x})$, $\textsf{\textbf{ins}}{:}p(\bar{x})$, and $\textsf{\textbf{del}}{:}p(\bar{x})$ at $[n]$, but from the value of $p(\bar{x})$ at the previous final state. This value can be restored using the current values of protocol relations (see below).

By inspecting all boolean combinations of truth-values in Figure 7.2, one easily verifies that the above rules implement correctly the desired conflict resolution policies. Of these, ABORT is obviously the safest and most skeptical policy. However, depending on the application, other policies may be more appropriate.

Protocol Relations

Protocol relations are auxiliary relations from $ctl(\mathbf{P})$ that can be used for different purposes like implementing deferred coupling (see below), enforcing termination (Section 7.5), or querying past temporal constraints (Section 7.6.2). Here, we show how they are used to maintain the accumulated *net effect* of changes since the beginning of transaction. In case of transaction abort, this allows to restore the old values of relations. Although in principle, one could simply store in an auxiliary relation p_{old} the original value of p, it is more efficient to only maintain p and the actual deltas $\textsf{\textbf{insd}}{:}p$, $\textsf{\textbf{deld}}{:}p$, and compute p_{old} from the latter.

Given a *fixed* conflict resolution policy from above, one can compute the values of $\textsf{\textbf{insd}}{:}p$ and $\textsf{\textbf{deld}}{:}p$ at $[n{+}1]$ from the values of p, $\textsf{\textbf{insd}}{:}p$, and $\textsf{\textbf{deld}}{:}p$ at $[n]$ (this has been done in

[LLM98b], where an ABORT policy is presupposed). Here, we use a more generic rule set which works for *all* of the above policies. This is possible because we maintain **insd**:p and **deld**:p at $[n{+}1]$ using (i) their values at $[n]$, and (ii) the values of p at $[n]$ and $[n{+}1]$.

The following rules maintain for each *edb*-relation p the actual insertions (**insd**:p) and deletions (**deld**:p) since the initial state $[0]$:

$$\begin{array}{l} [S{+}1]\,\textbf{insd}{:}p(\bar{X}) \leftarrow [S{+}1]\,p(\bar{X}), [S]\,\neg\,\textbf{deld}{:}p(\bar{X}), [S]\,\neg\,p(\bar{X}). \\ [S{+}1]\,\textbf{insd}{:}p(\bar{X}) \leftarrow [S{+}1]\,p(\bar{X}), [S]\,\neg\,\textbf{deld}{:}p(\bar{X}), [S]\,\textbf{insd}{:}p(\bar{X}). \end{array} \qquad (PR^+)$$

$$\begin{array}{l} [S{+}1]\,\textbf{deld}{:}p(\bar{X}) \leftarrow [S{+}1]\,\neg\,p(\bar{X}), [S]\,\textbf{deld}{:}p(\bar{X}). \\ [S{+}1]\,\textbf{deld}{:}p(\bar{X}) \leftarrow [S{+}1]\,\neg\,p(\bar{X}), [S]\,p(\bar{X}), [S]\,\neg\,\textbf{insd}{:}p(\bar{X}). \end{array} \qquad (PR^-)$$

Protocol relations yield a simple way of restoring the value from p at $[0]$ using the current values at $[n]$ of p, **insd**:p, and **deld**:p, see the rules ($UNDO$) below. The correctness of (PR^+) and (PR^-) wrt. the intuitive semantics is established next:

Proposition 7.1 (Protocol Relations)
Let P be a Statelog program which includes the rules (PR^+) and (PR^-) for every base relation p. Let $\mathcal{D}$ be a Statelog database. Then for all $\bar{x}$, and all n:

$$\mathcal{M}_{P\cup\mathcal{D}} \models [0]\,p(\bar{x}) \;\Leftrightarrow\; \mathcal{M}_{P\cup\mathcal{D}} \models (\,[n]\,\textbf{deld}{:}p(\bar{x}) \vee ([n]\,p(\bar{x}) \wedge [n]\,\neg\,\textbf{insd}{:}p(\bar{x}))\,).$$

Proof By induction on n, we show that

1. $[n]\,\textbf{insd}{:}p(\bar{x}) \;\Leftrightarrow\; [0]\,\neg\,p(\bar{x}) \wedge [n]\,p(\bar{x})$,
2. $[n]\,\textbf{deld}{:}p(\bar{x}) \;\Leftrightarrow\; [0]\,p(\bar{x}) \wedge [n]\,\neg\,p(\bar{x})$, and
3. $[0]\,p(\bar{x}) \;\Leftrightarrow\; [n]\,\textbf{deld}{:}p(\bar{x}) \vee [n]\,(p(\bar{x}) \wedge \neg\,\textbf{insd}{:}p(\bar{x}))$,

from which the claim follows.

(1–3) are clear for $n = 0$ because **insd**:p and **deld**:p are initially empty. Using the induction hypothesis (IH) for (3), we show that (1) and (2) also hold for $n{+}1$. Then (3) follows from (1) and (2).
For (1) we have:

$$\begin{array}{rcl} [n{+}1]\,\textbf{insd}{:}p(\bar{x}) & \stackrel{(PR^+)}{\Longleftrightarrow} & [n{+}1]\,p(\bar{x}) \wedge [n]\,\neg\,\textbf{deld}{:}p(\bar{x}) \wedge [n]\,(\neg\,p(\bar{x}) \vee \textbf{insd}{:}p(\bar{x})) \\ & \Longleftrightarrow & [n{+}1]\,p(\bar{x}) \wedge \neg\,(\,[n]\,\textbf{deld}{:}p(\bar{x}) \vee [n]\,(p(\bar{x}) \wedge \neg\,\textbf{insd}{:}p(\bar{x}))\,) \\ & \stackrel{(IH)}{\Longleftrightarrow} & [n{+}1]\,p(\bar{x}) \wedge \neg\,[0]\,p(\bar{x}). \end{array}$$

Similarly, (2) is shown:

$$\begin{array}{rcl} [n{+}1]\,\textbf{deld}{:}p(\bar{x}) & \stackrel{(PR^-)}{\Longleftrightarrow} & [n{+}1]\,\neg\,p(\bar{x}) \wedge [n]\,(\textbf{deld}{:}p(\bar{x}) \vee p(\bar{x}) \wedge \neg\,\textbf{insd}{:}p(\bar{x})) \\ & \stackrel{(IH)}{\Longleftrightarrow} & [n{+}1]\,\neg\,p(\bar{x}) \wedge [0]\,p(\bar{x}). \end{array}$$

∎

Initialization of Protocol Relations. In the refined execution model, the initial state of a transaction is some $[s_i]$ instead of $[0]$ (Figure 7.1). Thus, all protocol relations have to be cleared in the transition to the next final state $[s_{i+1}]$, which is the initial state of the next transaction. This is accomplished by adding the goal

$$[S]\,\mathsf{running}$$

to all rule bodies in (PR^+) and (PR^-). Additionally, we have to ensure that "running" becomes false just before the end of transaction:

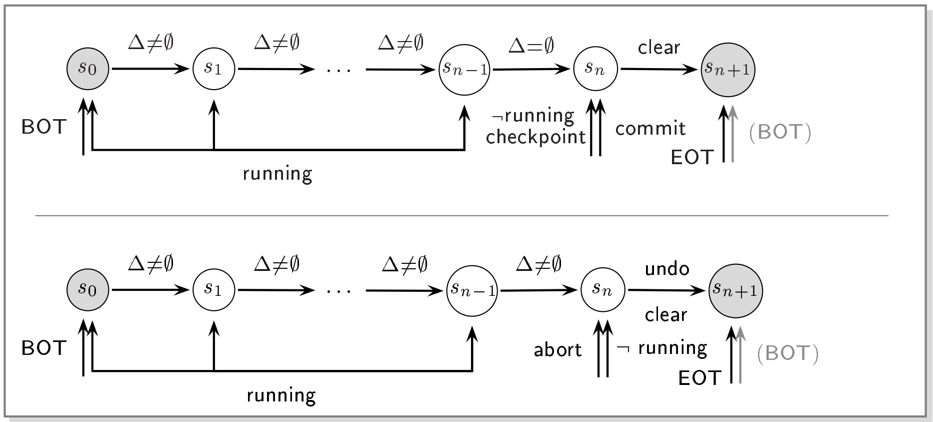


Figure 7.3: Committing and aborting transactions

Transaction Control

Whether the current transaction is still running is determined by checking for pending change requests:[5]

$$[S]\,\mathsf{running} \leftarrow [S]\,\mathbf{ins}{:}p(\bar{X}), \neg\, p(\bar{X}), \neg\,\mathsf{abort}.$$
$$[S]\,\mathsf{running} \leftarrow [S]\,\mathbf{del}{:}p(\bar{X}), p(\bar{X}), \neg\,\mathsf{abort}.$$

Finally, if there are no more unprocessed update requests, a "checkpoint" is reached. At this state, integrity constraints can be checked (see Example 7.6). If the transaction is not aborted, it can commit:

$$[S{+}1]\,\mathsf{checkpoint} \leftarrow [S]\,\mathsf{running}, [S{+}1]\,\neg\,\mathsf{running}.$$
$$[S]\,\mathsf{commit} \leftarrow [S]\,\mathsf{BOT}, \neg\,\mathsf{running}, \neg\,\mathsf{abort}.$$
$$[S]\,\mathsf{commit} \leftarrow [S]\,\mathsf{checkpoint}, \neg\,\mathsf{abort}.$$
$$[S{+}1]\,\mathsf{EOT} \leftarrow [S]\,\mathsf{commit}.$$

If the transaction is aborted however, the previous final state is restored by undoing the net effect of the transaction:

$$\begin{aligned} &[S{+}1]\,p(\bar{X}) \leftarrow [S]\,\mathsf{abort}, \mathbf{deld}{:}p(\bar{X}).\\ &[S{+}1]\,p(\bar{X}) \leftarrow [S]\,\mathsf{abort}, p(\bar{X}), \neg\,\mathbf{insd}{:}p(\bar{X}). \qquad (\mathit{UNDO})\\ &[S{+}1]\,\mathsf{EOT} \leftarrow [S]\,\mathsf{abort}. \end{aligned}$$

The execution schema of committing and aborting transactions is shown in Figure 7.3. Notice that, due to the goal "$[S]$ running", protocol relations are cleared in the transition after running has become false, so they are indeed empty at the beginning of the next transaction.

Immediate vs. Deferred Coupling

The system-defined rules above execute update requests **ins**:p and **del**:p as soon as they are derived. Consider, for example, the following user-defined rules:

[5] These rules are similar to the rules defining change in Section 6.9.2 (*FP*), but potentially more efficient since they involve joins with delta relations **ins**:p, **del**:p (which are usually much smaller than p).

$$\begin{array}{llcl} r_1: & [S]\,\textsf{ins}{:}p(\bar{X}) & \leftarrow & [S]\,\langle condition\rangle \\ r_2: & [S]\,\ldots & \leftarrow & [S]\,\ldots,\ p(\bar{X}),\ \ldots \end{array}$$

If r_1 derives $\textsf{ins}{:}p(\bar{x})$ at $[n]$, then r_2 "sees" the effect on p at $[n{+}1]$, i.e., immediately after the update on p (provided there is no conflict or transaction abort). Therefore, the rules presented above implement an *immediate coupling* between condition evaluation (body of r_1) and action execution (head of r_1).

In contrast, under *deferred coupling*, update requests to base relations are not executed immediately, but deferred to the end of transaction. In this case, r_2 would "see" the new value of p only after the current transaction has committed. Under deferred coupling, the transaction is running, as long as there are new update requests. Frame rules simply copy the original values of p—without considering insertions or deletions—while running holds:

$$[S{+}1]\,p(\bar{X}) \leftarrow [S]\,p(\bar{X}), \textsf{running}.$$

When running eventually becomes false, all accumulated updates are executed together. At this point, one can add another transition in order to check whether the new database state is consistent, and abort the transaction otherwise. It is clear that this kind of deferred coupling is easily specified by a different set of system-defined Statelog rules.

In principle, also *decoupled* execution of rules can be modeled, but this requires an extension of the Statelog framework in order to simulate nested transactions (see [LML96] and Section 8.2).

7.5 Enforcing Termination

In Section 6.9, we have investigated the problem of nontermination in the context of pure Statelog: at compile-time termination is undecidable, at run-time deciding termination is **PSPACE**-complete (and can be checked within the language). In the sequel, using the extended framework for active rules, we present an alternative approach for enforcing termination, based on the class of Δ-*monotonic* Statelog programs. This class not only guarantees termination, but has the additional benefit of yielding a tractable subset of Statelog, i.e., which is evaluable in **PTIME**. Another, less expressive class of terminating programs are *guarded rules* which are considered first.

7.5.1 Guarded Rules

In Statelog two kinds of recursion can be distinguished:

- *local recursion*, (i.e., involving locally recursive rules[6]) describes the recursion *within* a state $[n]$, and captures the static "deductive" aspect of the language. It does not lead to nontermination in the case of Statelog, because the language is restricted to finite structures. In contrast,

[6] cf. Definition 6.8

- *progressive recursion* (i.e., involving progressively recursive rules) is the effect of recursive rule triggering *across* different states. Even for finite structures, progressive recursion may lead to nonterminating execution due to *oscillating* update requests.

For example, the two rules of the user program

P_{osc}: $\boxed{\textbf{ins}{:}\mathsf{p} \leftarrow \neg\, \mathsf{p}. \qquad \textbf{del}{:}\mathsf{p} \leftarrow \mathsf{p}.}$

trigger each other indefinitely and lead to an oscillation of p. Therefore, the basic idea to enforce rule termination is to restrict progressive recursion such that oscillation is avoided. In the Statelog framework for active rules, the user may issue update requests only via local rules and internal actions **ins**:p and **del**:p. Thus, one possibility of enforcing termination is to "guard" user-defined update rules by a positive goal of the form $\triangleright e(\bar{X})$:[7]

Definition 7.1 (Guarded Rules)
A Statelog *rule* is called *guarded*, if some external event $\triangleright e(\bar{X})$ occurs *positively* in the body. A Statelog *user program* is *guarded*, if every update rule (i.e., with **ins**:p, **del**:p, or **mod**:p in the head) is guarded. □

According to the refined execution model described above (Figure 7.1), external events only occur once at the beginning of transaction, so guarded rules are only applicable in the first transition of a transaction and can be neglected in the termination analysis. In particular, if a user program P is guarded, this implies that progressive recursion is bounded and therefore termination of P is guaranteed.

Example 7.2 (Guarded Rules) Many basic updates can be expressed in a concise and intuitive way using guarded rules. For example, updates in the style of SQL like the insertion of individual tuples, the deletion of all tuples satisfying a certain condition φ, and the unrestricted deletion of all tuples of a base relation p can be expressed as follows:

$$\begin{aligned}
\textbf{ins}{:}p(\bar{X}) &\leftarrow \triangleright\mathsf{insert_into_}p(\bar{X}).\\
\textbf{del}{:}p(\bar{X}) &\leftarrow \triangleright\mathsf{delete_from_}p(\bar{X}), p(\bar{X}), \varphi(\bar{X}).\\
\textbf{del}{:}p(\bar{X}) &\leftarrow \triangleright\mathsf{discard_}p, p(\bar{X}).
\end{aligned}$$

A typical modification is the increase of all salaries of employees by a factor F:

$$\textbf{mod}{:}\mathsf{emp(E,Sal/Sal')} \leftarrow \triangleright\mathsf{raise(F)}, \mathsf{emp(E,\ Sal)}, \mathsf{Sal'} := \mathsf{Sal{*}F}.$$

If modifications are modeled by simultaneous deletion of old and insertion of new values, the NOP policy can be applied to handle conflicting update requests for F=1. Note that the use of arithmetics or other built-ins may increase the complexity of the language and can lead to unsafe rules if not limited in a certain way (see Section 8.1).

The following rules swap the contents of two relations p and q of the same arity, whenever $\triangleright\mathsf{swap\_}p\mathsf{\_}q$ occurs:

$$\begin{aligned}
\textbf{del}{:}p(\bar{X}), \textbf{ins}{:}q(\bar{X}) &\leftarrow \triangleright\mathsf{swap_}p\mathsf{_}q, p(\bar{X}).\\
\textbf{del}{:}q(\bar{X}), \textbf{ins}{:}p(\bar{X}) &\leftarrow \triangleright\mathsf{swap_}p\mathsf{_}q, q(\bar{X}).
\end{aligned}$$

Again, NOP can be used to handle correctly conflicting updates, which occur for all $\bar{x}$ where both $p(\bar{x})$ and $q(\bar{x})$ hold. □

[7] The idea of guarded Statelog rules was presented earlier in [LL94].

Example 7.3 (Unguarded Rules) The program in Example 7.1 always terminates since all update rules are guarded. In contrast, if the guard $\triangleright$tick is removed, the program no longer terminates and becomes 4-periodic:

$$\{\mathsf{on(red)}\};\ \{\mathsf{on(red),\ on(yellow)}\};\ \{\mathsf{on(green)}\};\ \{\mathsf{on(yellow)}\};\ \{\mathsf{on(red)}\};\ \ldots$$

This is due to the unbounded progressive recursion of the first four rules of Example 7.1: if r_i is triggered in $[n]$, this triggers r_j in $[n{+}1]$, where $(i, j) \in \{(1, 2), (2, 3), (3, 4), (4, 1)\}$. $\square$

A theoretical drawback of guarded rules is their limited expressive power. Indeed, after the first transition, relations become constant since there are no more update requests. Thus, the transaction expressiveness of guarded Statelog corresponds to the expressiveness of a single transition, i.e., to S-DATALOG.

7.5.2 Δ-Monotonic Rules

A common idea to guarantee the existence of a fixpoint and at the same time to obtain a tractable language is to consider an inflationary semantics. However, such an inflationary Statelog variant is not useful as an update language since deletions cannot be expressed. The underlying idea of Δ-monotonic Statelog is to allow both insertions and deletions (hence, the database evolution is not inflationary in general), but to ensure that the *deltas* change monotonically, thereby preventing oscillation.[8] In this case, rules can trigger each other recursively across different states, as long as Δ-monotonicity is obeyed. Thus, progressive recursion is not bounded like for guarded rules, resulting in a more expressive class of programs. Δ-monotonicity can be achieved at several levels of granularity:

Compile-Time Δ-Monotonicity

The coarsest granularity is to ensure by the following compile-time check that every relation updated by a program is either increasing or decreasing:

Definition 7.2 (Compile-Time Δ-Monotonic)
A Statelog user program P is *compile-time Δ-monotonic* if for every $p \in edb(\mathbf{P})$:

- $(\textbf{ins}{:}p(\ldots) \leftarrow \ldots) \in P \;\Rightarrow\; (\textbf{del}{:}p(\ldots) \leftarrow \ldots) \notin P.$ $\square$

Therefore, for every *edb*-relation p of P, **ins**:p and **del**:p may not both be defined in P, so it is determined at compile-time that every *edb*-relation is either monotonically non-decreasing or non-increasing.

Example 7.4 (Integrity Enforcement, Cont'd)
In the extended framework, the user-defined rules of Example 6.1 are:

del:emp(E,Sal,D) ← ▷**del**:emp(E), emp(E,Sal,D).

del:dept(D,E), **del**:emp(E1,S1,D) ←
  del:emp(E,_,_), dept(D,E), emp(E1,S1,D).

[8] A preliminary definition of essentially the same idea can be found in [LHL95].

In contrast to Example 6.1, rules r_2 and r_3 are omitted (since frame rules are system-defined), as well as all state terms (since user-defined rules are local). Clearly, the program is compile-time Δ-monotonic and therefore terminating. □

Notice that we have tacitly assumed that frame rules (as presented in Section 7.4.2) are used to propagate unchanged tuples through the transaction in the extended framework. Indeed, this requirement is crucial as can be seen from the user program

$$P: \quad \boxed{\textsf{\textbf{ins}:p} \leftarrow \neg\, \textsf{p}.}$$

The program P is Δ-monotonic and terminates since the inserted fact p is propagated by frame rules. However, if a system-defined frame rule like

$$[S{+}1]\,\textsf{p} \leftarrow [S]\,\textsf{p}, \neg\, \textsf{\textbf{del}:p}$$

were *not* given, then p would oscillate between *true* and *false*: if ¬ p holds in $[n]$, then **ins**:p is derived, so $[n{+}1]$ p holds. Thus, ¬ p is false in $[n{+}1]$, preventing the derivation of $[n{+}1]$ **ins**:p. Therefore, p becomes false again at $[n{+}2]$ (since there is no frame rule for p), etc.

Run-Time Δ-Monotonicity

Instead of restricting to programs which define either insertions or deletions, one can obtain a more flexible class by allowing both types of updates and checking for Δ-monotonicity at run-time. The price to pay is that it cannot be guaranteed in advance that rule execution is Δ-monotonic. Instead, if Δ-monotonicity is violated, the current transaction has to be aborted at run-time. Thus, although it is unknown at compile-time whether the program is Δ-monotonic, at least the program is guaranteed to terminate—if necessary via a transaction abort. At run-time, Δ-monotonicity can be checked at the level of relations, or at the level of individual tuples:

We say that a Statelog user program P is *run-time Δ-monotonic* at the *relation level*, if for every $p \in edb(\mathbf{P})$ it contains the rules:

$$
\begin{aligned}
&[S{+}1]\,\textbf{inc}{:}p \leftarrow [S]\,\textbf{ins}{:}p(\bar{X}), \neg\, p(\bar{X}).\\
&[S{+}1]\,\textbf{dec}{:}p \leftarrow [S]\,\textbf{del}{:}p(\bar{X}), p(\bar{X}).\\
&[S{+}1]\,\textbf{inc}{:}p \leftarrow [S]\,\textbf{inc}{:}p, \neg\, \textsf{EOT}.\\
&[S{+}1]\,\textbf{dec}{:}p \leftarrow [S]\,\textbf{dec}{:}p, \neg\, \textsf{EOT}.\\
&[S]\,\textsf{abort} \leftarrow [S]\,\textbf{inc}{:}p, \textbf{dec}{:}p.
\end{aligned}
$$

The first pair of rules check whether p is increasing or decreasing. This information is propagated by the second pair of rules until the end of transaction. If at some point it is detected that both, requests to insert into and to delete from p have occurred, the last rule initiates a transaction abort.

In order to check Δ-monotonicity at the tuple level, we can make use of the protocol relations **insd**:p and **deld**:p defined above. This yields a more precise approximation for enforcing termination by preventing oscillation of tuples. A Statelog user program P is called *run-time Δ-monotonic* at the *tuple level*, if it contains the rules

$$
\begin{aligned}
&[S]\,\textsf{abort} \leftarrow [S]\,\textbf{insd}{:}p(\bar{X}), \textbf{del}{:}p(\bar{X}).\\
&[S]\,\textsf{abort} \leftarrow [S]\,\textbf{deld}{:}p(\bar{X}), \textbf{ins}{:}p(\bar{X}).
\end{aligned}
$$

for every $p \in edb(\mathbf{P})$. The first rule checks whether a previously inserted tuple $p(\bar{x})$ is now requested for deletion; the second rule checks the dual case. Observe that these rules allow simultaneous insertion and deletion on the same relation p within a transaction, as long as the sets of complementary update requests are disjoint.

Example 7.5 (View Materialization) Consider P_{tc} in Example 4.2. Viewed as a Statelog user program, it defines a recursive view for tc, i.e., the transitive closure of a graph given by the *edb*-relation e. A more efficient access to the view can be accomplished by *materializing* tc in some *edb*-relation, say mtc, thereby avoiding the need to compute tc on demand:

P_{mtc}:

r_1:	**del**:mtc(X,Y) $\leftarrow$ $\triangleright$discard_mtc, mtc(X,Y).
r_2:	**ins**:mtc(X,Y) $\leftarrow$ $\triangleright$materialize_mtc, e(X,Y).
r_3:	**ins**:mtc(X,Y) $\leftarrow$ e(X,Z), mtc(Z,Y), $\neg$ mtc(X,Y).

r_1 empties the materialized view upon the occurrence of an external event $\triangleright$discard_mtc, whereas r_2 and r_3 materialize the view when $\triangleright$materialize_mtc occurs. If $\triangleright$materialize_mtc occurs in a separate transaction after $\triangleright$discard_mtc has occurred, run-time Δ-monotonicity is satisfied: the first transaction only deletes from the materialized view, and the second only inserts into it. In contrast, run-time Δ-monotonicity is violated, if both events occur simultaneously (although—in this example—the rules would terminate). □

Complexity and Expressiveness

The common idea of the different Δ-monotonic Statelog variants is to prevent oscillation of tuples, from which termination follows. As a "side-effect" Δ-monotonic programs can also be evaluated more efficiently than general Statelog programs:

Recall from Section 6.8.2 that the maximal number of tuples in any state $[n]$ is polynomial in the size of the given database $\mathcal{D}$. For a Δ-monotonic program P with database $\mathcal{D}$ this implies that the length of the initial window is also polynomial, because (i) at least one tuple is changed in each transition, and (ii) no changes are revoked (provided the transaction commits). Moreover, each of the polynomially many transitions corresponds to a S-Datalog computation (Proposition 6.20), hence Δ-monotonic Statelog programs are evaluable in **PTIME**.

It is easy to see that Δ-monotonic Statelog is strictly more expressive than guarded Statelog: First, recall that I-Statelog expresses the FIXPOINT transformations (Figure 4.2 and Corollary 6.16). Now consider the I-Statelog representation[9] of the I-Datalog rule $H \leftarrow B, \neg C$:

$$[S{+}1]\, H \leftarrow [S]\, B, [S] \neg C.$$
$$[S{+}1]\, H \leftarrow [S]\, H.$$

Clearly, in the extended framework, the same effect can be expressed by the following compile-time Δ-monotonic rules:

ins:$H \leftarrow B, \neg C$.
ins:$H \leftarrow H$.

It follows that the more restrictive variant of compile-time Δ-monotonic programs already expresses FIXPOINT. Summarizing, we have the following results for guarded and Δ-monotonic Statelog:

[9] see P_{ID}, p. 61

Theorem 7.2 (Guarded and Δ-Monotonic Statelog)

- *Guarded and Δ-monotonic programs always terminate within* **PTIME**.
- *Guarded Statelog expresses all* S-DATALOG *transformations;*

  Δ-monotonic Statelog expresses all FIXPOINT *transformations.*

7.6 Integrity Constraints

Integrity constraints are used to ensure that the database state remains consistent wrt. the modeled application and play an important role in practice, especially in relational databases. Integrity constraints may be roughly classified as either static or dynamic. *Static* integrity constraints speak only about the current database state. For example, the inclusion dependency "*every manager is also an employee*" (denoted $\mathsf{dept[MgrNo]} \subseteq \mathsf{emp[EmpNo]}$ in Example 3.1) is a typical static constraint. In contrast, *dynamic* (or *temporal*) integrity constraints may refer to several database states. An example for a dynamic constraint is "*salaries of employees should never decrease*". In the sequel, we show how such static and dynamic integrity constraints can be handled in Statelog.

7.6.1 Static Constraints

Classical constraints in relational databases are static, the most important being functional dependencies, join dependencies, and inclusion dependencies. It is well known that practically all static integrity constraints used in relational databases can be expressed as first-order sentences in the implication form

$$\forall \bar{X}\ (\varphi(\bar{X}) \rightarrow \exists \bar{Z}\, \psi(\bar{Y})). \qquad (*)$$

Here, $\bar{Y} \setminus \bar{X} \subseteq \bar{Z}$, i.e., all variables in $\bar{Y}$ not occurring in $\bar{X}$ are among $\bar{Z}$, φ is a (possibly empty) conjunction of atoms, and ψ a nonempty conjunction [AHV95]. The implication $(*)$ essentially says that the presence of some tuples in the database should imply the presence of certain other tuples (or the coincidence of certain arguments of tuples).

Denials

In deductive databases, it is common to express an integrity constraint φ in the form of a *denial*

$$\mathit{false} \leftarrow \neg\, \varphi.$$

As long as $\mathcal{D} \models \varphi$ for the current database $\mathcal{D}$, the special predicate "*false*" cannot be derived. Conversely, if the database violates φ then "*false*" is derivable, thereby indicating an inconsistency. Since every first-order sentence can be expressed as a S-Datalog query (Figure 4.2), it follows that every denial can be expressed as a set of local Statelog rules. A transformation from FO to local Statelog rules is a special case of the transformation used in the proof of Theorem 7.3.

Example 7.6 (Inclusion Dependency) The active rules in Example 3.1 are used to enforce the inclusion dependencies $\mathsf{dept[MgrNo]} \subseteq \mathsf{emp[EmpNo]}$ and $\mathsf{emp[DeptNo]} \subseteq \mathsf{dept[DeptNo]}$. The latter, for example, can be expressed by the first-order sentence

$$\forall E, Sal, D\ (\mathsf{emp}(E, Sal, D) \rightarrow \exists M\ \mathsf{dept}(D, M)). \qquad (\varphi)$$

The corresponding denial is

$$\mathit{false} \leftarrow \exists E, Sal, D\ (\mathsf{emp}(E, Sal, D) \wedge \neg \exists M\ \mathsf{dept}(D, M)),$$

which can be equivalently expressed in Statelog by the following local rules:

P_φ:

```
false ← emp(E,Sal,D), ¬ ex_depts(D).
ex_depts(D) ← dept(D,M).
```

Here, ex_depts is an auxiliary relation storing all existing departments. □

In general, a transaction may temporarily violate integrity, as long as the final outcome satisfies all integrity constraints. Therefore, denials should not be evaluated in every intermediate state, but only immediately before the end of transaction, i.e., when checkpoint is true.[10] This can be accomplished by simply adding the goal "checkpoint" to all denials.

Enforcing Integrity

Denials are deductive rules and can be used to *check* for violations of integrity constraints. A language with active rules like Statelog additionally allows to *enforce* integrity. The simplest way is to abort a transaction whenever a violation is detected. For example, if we replace "false" by "abort" in P_φ above, we obtain a simple mechanism guaranteeing that every transaction which would violate the above inclusion dependency is aborted.

However, one of the strengths of active rules is the possibility to specify more elaborated integrity maintenance strategies. For instance, the rules of Example 7.4 maintain the given inclusion dependencies by removing all tuples which are affected by an external delete request to emp.

7.6.2 Dynamic Constraints and PTL

Dynamic integrity constraints are used to ensure that the *evolution* of the database satisfies certain properties. In the sequel, we show how dynamic constraints in the form of *past first-order temporal logic* (PTL for short) can be handled in Statelog.

Definition 7.3 (PTL Syntax)
PTL formulas are constructed according to the following formation rules:[11]

$$\frac{}{\varphi_{At}}\ ; \quad \frac{\varphi}{\neg\varphi}\ ; \quad \frac{\varphi}{\exists X\ \varphi}\ ; \quad \frac{\varphi, \psi}{\varphi \wedge \psi}\ ; \quad \frac{\varphi}{\bullet\varphi}\ ; \quad \frac{\varphi}{\blacksquare\varphi}\ ; \quad \frac{\varphi}{\blacklozenge\varphi}\ ; \quad \frac{\varphi, \psi}{\varphi\, \mathbf{S}\, \psi}\ .$$

Here, φ_{At} is an atomic formula, φ and ψ are PTL formulas. The past temporal operators are "$\bullet$" (*previously*), "$\blacksquare$" (*always in the past*), "$\blacklozenge$" (*sometimes in the past*), and "$\mathbf{S}$" (*since*). □

[10] See the corresponding rules for transaction control in Section 7.4.2.

[11] For simplicity, we view $(\forall X\varphi)$, $(\varphi \vee \psi)$, $(\varphi \leftarrow \psi)$ as abbreviations.

As shown in Section 6.4.3, a Statelog structure $\mathcal{I} = (\mathcal{I}[0], \mathcal{I}[1], \ldots)$ corresponds to a first-order temporal structure with time isomorphic to $\mathbb{N}_0$. Thus, PTL formulas can be evaluated wrt. Statelog structures $\mathcal{I}$, and we may use the terms *Statelog structure* and *PTL structure* interchangeably; likewise for interpretations. Whereas in Section 6.4.3, the progressive or "forward" interpretation of Statelog rules was given using future operators of first-order temporal logic, we are now interested in specifying properties of the database history, i.e., about the past.

Definition 7.4 (PTL Semantics)
Let $\mathcal{I}$ be a Statelog interpretation over some given domain U and $v : \mathbb{V} \to U$ a valuation (v extends to terms as usual). The satisfaction of PTL formulas is defined by:

- $\mathcal{I}[n], v \models p(t_1, \ldots, t_n)$, if $(v(t_1), \ldots, v(t_n)) \in p^{\mathcal{I}[n]}$, (*for terms t_i*)
- $\mathcal{I}[n], v \models \neg\varphi$, if not $\mathcal{I}[n], v \models \varphi$,
- $\mathcal{I}[n], v \models \varphi \wedge \psi$, if $\mathcal{I}[n], v \models \varphi$ and $\mathcal{I}[n], v \models \psi$, (*similar for $\vee, \leftarrow$*)
- $\mathcal{I}[n], v \models \exists X\varphi$, if there is some $a \in U$ s.t. $\mathcal{I}[n], v_{X/a} \models \varphi$, (*similar for $\forall$*)
- $\mathcal{I}[n], v \models \bullet\varphi$, if $\mathcal{I}[n-1], v \models \varphi$,
- $\mathcal{I}[n], v \models \blacksquare\varphi$, if for all $k < n$: $\mathcal{I}[k], v \models \varphi$,
- $\mathcal{I}[n], v \models \blacklozenge\varphi$, if for some $k < n$: $\mathcal{I}[k], v \models \varphi$,
- $\mathcal{I}[n], v \models \varphi\, \mathbf{S}\, \psi$, if (i) for some $k \in \{0, \ldots, n-1\}$: $\mathcal{I}[k], v \models \psi$, and (ii) for all $l \in \{k+1, \ldots, n\}$: $\mathcal{I}[k], v \models \varphi$. □

It is easy to see the operators "$\blacksquare$" and "$\blacklozenge$" can be defined in terms of "$\mathbf{S}$" as follows:

$$\blacklozenge\varphi \Leftrightarrow \mathit{true}\, \mathbf{S}\, \varphi, \qquad \blacksquare\varphi \Leftrightarrow \neg\blacklozenge\neg\varphi.$$

Like static constraints, dynamic constraints given as PTL formulas can be expressed in the form of denials.

Example 7.7 (Dynamic Constraint) The dynamic integrity constraint "*salaries must not decrease*" for the database schema of Example 3.1 can be expressed by the PTL formula

$$\neg(\exists E, Sal, Sal', D, D'\ (\blacklozenge\mathsf{emp}(E, Sal, D) \wedge \mathsf{emp}(E, Sal', D') \wedge Sal' < Sal)),$$

or, equivalently, as a denial:

$$\mathit{false} \leftarrow\ \blacklozenge\mathsf{emp}(E, Sal, \_) \wedge \mathsf{emp}(E, Sal', \_) \wedge Sal' < Sal.$$

Here, we assume that unary operators (like "$\blacklozenge$") have higher precedence than binary ones (like "$\wedge$"). □

The application of PTL formulas is not limited to dynamic constraints: For example, if the head of the previous denial is replaced by an *idb*-relation say "$\mathsf{poor\_emp}(E)$", then the rule defines a *temporal* (or *historical*) *query*, namely the set of all employees whose salary *has* decreased (see also Example 7.10). Another application is the specification of composite events in PTL (Section 7.7). As shown next, PTL formulas have a natural encoding in Statelog, thereby allowing to handle dynamic constraints, historical queries, and composite events in the Statelog framework.

From PTL to Statelog

Assume $\mathcal{I}[k]$ is the current database state. A naive approach to evaluate PTL formulas is to store the entire database history $\mathcal{I}[0..k]$. Clearly, the space requirements of this method depend on the length of the history, as well as on the number of domain elements. In [Cho95b], a method is developed which avoids storing the complete history using a so-called *bounded history encoding* (Section 7.8). The basic idea is that relevant information needed to query the history $\mathcal{I}[0..k]$ of a database over the schema $\mathbf{R}$ is encoded in an auxiliary database $\widehat{\mathcal{I}}_k$ over an extended database schema $\widehat{\mathbf{R}} \supseteq \mathbf{R}$. Every time the database is updated, the new encoding is computed using the previous encoding and the new database state. In this way, past temporal constraints can be efficiently evaluated based on the extended state $\widehat{\mathcal{I}}_k$, instead of considering the complete history $\mathcal{I}[0..k]$. Moreover, the size of $\widehat{\mathcal{I}}_k$ depends only on the number of elements in the active domain of $\mathcal{I}[0..k]$, but not on k itself.

In [Cho95b, CT95], the translation from PTL to inductive first-order auxiliary relations is presented. Such inductive definitions can be directly and elegantly expressed in Statelog:

Theorem 7.3 (PTL to Statelog) *Let P be a Statelog program and $\varphi(\bar{X})$ a PTL formula with free variables $\bar{X}$. There is a Statelog program P_φ and an new relation r_φ over the extended schema $\mathbf{P} \cup \mathbf{P}_\varphi$ such that for all databases $\mathcal{D}$ over $\mathbf{P}$ and all $n \in \mathbb{N}_0$:*

$$\mathcal{M}_{P\cup\mathcal{D}}[n] \models \varphi(\bar{x}) \quad \Leftrightarrow \quad \mathcal{M}_{P\cup P_\varphi\cup\mathcal{D}}[n] \models r_\varphi(\bar{x}) \ .$$

Proof The idea of the construction of P_φ is to define a new auxiliary relation r_α for every subformula α of φ. Below, R_α denotes an atomic formula of the form $r_\alpha(\bar{X})$, where $\bar{X}$ are the free variables of α. The transformation Ψ mapping $\varphi \in$ PTL to the desired Statelog program $P_\varphi := \Psi(\varphi)$ is defined for temporal connectives as follows:

$$\begin{array}{rcl}
\Psi(\bullet\varphi) & := & \{[S{+}1]\, R_{\bullet\varphi} \leftarrow [S]\, R_\varphi\} \cup \Psi(\varphi) \\
\Psi(\blacklozenge\varphi) & := & \{[S{+}1]\, R_{\blacklozenge\varphi} \leftarrow [S]\, R_\varphi, \\
 & & \ [S{+}1]\, R_{\blacklozenge\varphi} \leftarrow [S]\, R_{\blacklozenge\varphi}\} \cup \Psi(\varphi) \\
\Psi(\blacksquare\varphi) & := & \{[S]\, R_{\blacksquare\varphi} \leftarrow [S]\, \mathsf{eq}_0, \\
 & & \ [S{+}1]\, R_{\blacksquare\varphi} \leftarrow [S]\, R_{\blacksquare\varphi}, R_\varphi\} \cup \Psi(\varphi) \\
\Psi(\varphi\, \mathbf{S}\, \psi) & := & \{[S{+}1]\, R_{\varphi\mathbf{S}\psi} \leftarrow [S]\, R_\psi, [S{+}1]\, R_\varphi, \\
 & & \ [S{+}1]\, R_{\varphi\mathbf{S}\psi} \leftarrow [S]\, R_{\varphi\mathbf{S}\psi}, [S{+}1]\, R_\varphi\} \cup \Psi(\varphi) \cup \Psi(\psi) \ .
\end{array}$$

Standard first-order connectives translate to local Statelog rules as follows:

$$\begin{array}{rcl}
\Psi(\varphi_{At}) & := & \{[S]\, R_{\varphi_{At}} \leftarrow [S]\, \varphi_{At}\}, \quad \textit{if } \varphi_{At} \textit{ is an atom} \\
\Psi(\neg\,\varphi) & := & \{[S]\, R_{\neg\varphi} \leftarrow [S]\, \neg\, R_\varphi\} \cup \Psi(\varphi) \\
\Psi(\exists X\ \varphi(X,\bar{Y})) & := & \{[S]\, R_{\exists X\varphi}(\bar{Y}) \leftarrow [S]\, R_\varphi(X,\bar{Y})\} \cup \Psi(\varphi) \\
\Psi(\forall X\ \varphi) & := & \ \Psi(\neg\,\exists X \neg\,\varphi) \\
\Psi(\varphi \wedge \psi) & := & \{[S]\, R_{\varphi\wedge\psi} \leftarrow [S]\, R_\varphi, R_\psi\} \cup \Psi(\varphi) \cup \Psi(\psi) \\
\Psi(\varphi \vee \psi) & := & \{[S]\, R_{\varphi\vee\psi} \leftarrow [S]\, R_\varphi, \\
 & & \ [S]\, R_{\varphi\vee\psi} \leftarrow [S]\, R_\psi\} \cup \Psi(\varphi) \cup \Psi(\psi) \ .
\end{array}$$

Recall that the notation $\varphi(\bar{Z})$ emphasizes that all free variables of φ are among $\bar{Z}$; if we only write φ, nothing is said about the free variables of φ. Thus, in the $\exists$-step, the existentially quantified variable occurs in the body, but not in the head.

Notice further that $R_{\bullet\varphi}$, $R_{\blacklozenge\varphi}$, and $R_{\varphi\mathbf{S}\psi}$ are always false at $[0]$, whereas $R_{\blacksquare\varphi}$ is always true. From the definition of Ψ it is clear that the generated rules are stratified (hence state-stratified): The only rules involving recursion are those for $R_{\blacklozenge\varphi}$, $R_{\blacksquare\varphi}$, and $R_{\varphi\mathbf{S}\psi}$, but this recursion is positive and progressive (from $[S]$ to $[S{+}1]$). ■

Example 7.8 (PTL to Statelog)
Consider the dynamic constraint "*no employee is hired after she left*" [CT95]. In PTL this can be expressed by the formula $\varphi =$

$$({}_1\neg\,({}_2\exists X\,({}_3\,({}_4\mathsf{emp}(X)\,{}_4)\wedge({}_5\blacklozenge({}_6\,({}_7\neg\,({}_8\mathsf{emp}(X)\,{}_8)\,{}_7)\wedge({}_9\bullet({}_{10}\mathsf{emp}(X)\,{}_{10})\,{}_9)\,{}_6)\,{}_5)\,{}_3)\,{}_2)\,{}_1).$$

Using the above transformation Ψ, the following program P_φ is obtained:

P_φ :

$[S]\,r_1$	$\leftarrow [S]\,\neg\,r_2.$	$[S]\,r_6(X)$	$\leftarrow [S]\,r_7(X), r_9(X).$
$[S]\,r_2$	$\leftarrow [S]\,r_3(X).$	$[S]\,r_7(X)$	$\leftarrow [S]\,\neg\,r_8(X).$
$[S]\,r_3(X)$	$\leftarrow [S]\,r_4(X), r_5(X).$	$[S]\,r_8(X)$	$\leftarrow [S]\,\mathsf{emp}(X).$
$[S]\,r_4(X)$	$\leftarrow [S]\,\mathsf{emp}(X).$	$[S{+}1]\,r_9(X)$	$\leftarrow [S]\,r_{10}(X).$
$[S{+}1]\,r_5(X)$	$\leftarrow [S]\,r_6(X).$	$[S]\,r_{10}(X)$	$\leftarrow [S]\,\mathsf{emp}(X).$
$[S{+}1]\,r_5(X)$	$\leftarrow [S]\,r_5(X).$		

Observe how the inductive temporal operators $\blacklozenge$, $\blacksquare$, and $\mathbf{S}$ translate to Statelog rules with progressive recursion, as exemplified by the subformula $({}_5\blacklozenge(\ldots)\,{}_5)$ and the corresponding Statelog rules for r_5.

The translation Ψ generates highly redundant rules, and leaves a lot of room for simplifications: For example, r_4, r_8, and r_{10} are "copies" of emp, so their occurrences in rule bodies can be replaced by emp. Moreover, by unfolding[12] positive occurrences of defined relations, the rules can be further simplified, eventually yielding the program:[13]

P'_φ:

$[S]\,r_1$	$\leftarrow [S]\,\neg\,r_2.$
$[S]\,r_2$	$\leftarrow [S]\,\mathsf{emp}(X), r_5(X).$
$[S{+}1]\,r_5(X)$	$\leftarrow [S]\,\neg\,\mathsf{emp}(X), r_9(X).$
$[S{+}1]\,r_5(X)$	$\leftarrow [S]\,r_5(X).$
$[S{+}1]\,r_9(X)$	$\leftarrow [S]\,\mathsf{emp}(X).$

As above, we can enforce the constraint by adding the corresponding denial "$\mathsf{abort} \leftarrow r_2$" to the user program ($r_1$ is superfluous, since $\neg\,\varphi \Leftrightarrow r_2$). □

Domain Independence

The problem of unsafe or domain dependent first-order formulas (Section 2.2) recurs in the context of PTL. Let $\varphi(\bar{X})$ be a PTL formula. Similar to the FO case, we say that

$$q_{\varphi,n}: \qquad \mathcal{I} \mapsto \{\mathsf{ans}(\bar{x}) \mid \mathcal{I}[n] \models \varphi(\bar{x})\}$$

is the *answer* of φ in $\mathcal{I}$ at $[n]$. A PTL formula is *domain independent*, if for all $\mathcal{I}$ and all n, the answer $q_{\varphi,n}(\mathcal{I})$ does not depend on the underlying domain. For example, $\blacklozenge p(X)$ is domain

[12] see UNFOLD, p. 38

[13] If we allow Statelog rules *not* in normal form, we can drop r_9 and replace the first rule for r_5 by $[S{+}2]\,r_5(X) \leftarrow [S{+}1]\,\neg\,\mathsf{emp}(X), [S]\,\mathsf{emp}(X)$.

independent, because only the tuples which have been previously in p have to be considered. In contrast, $\blacksquare p(X)$ is domain dependent since the answer at [0] is the entire domain. Even if a PTL formula is domain independent, the rules for auxiliary relations generated by the translation Ψ may be unsafe:

Example 7.9 (Unsafe Rule) The rule "$r_7(X) \leftarrow \neg\, r_8(X)$" from Example 7.8 is not range-restricted and domain dependent. On the other hand, the original PTL formula φ is domain independent, as can be seen from the simplified program P_φ, which is equivalent to φ and range-restricted. □

Since domain independence for PTL—being a superset of FO—is undecidable, one has to restrict to decidable classes which guarantee this property. One possibility is to first compute $\Psi(\varphi)$, obtaining a (possibly unsafe) program P_φ, which is then rewritten into a simplified program P'_φ as above. If P'_φ is range-restricted (cf. Section 6.4), then φ can be safely evaluated, otherwise φ has to be rejected.

Alternatively, we can use the general solution for translating evaluable PTL formulas to *relational algebra normal form* (RANF) presented in [CT95] (see also [Cho95b]):

First, observe that the transitions induced by $P_\varphi := \Psi(\varphi)$ are first-order queries, i.e., the new values of auxiliary relations at $[n{+}1]$ can be computed from the values at $[n]$ by means of first-order operations. For example, let

$$\gamma(\bar{Z}) \;:=\; \varphi(\bar{X})\,\mathbf{S}\,\psi(\bar{Y}),$$

where $\bar{Z}$ comprises the (not necessarily disjoint) free variables $\bar{X}$ and $\bar{Y}$. Then $\Psi(\gamma)$ corresponds to the inductive definition

$$[n{+}1]\, r_\gamma(\bar{Z}) \;:\Leftrightarrow\; ([n]\, r_\psi(\bar{Y}) \wedge [n{+}1]\, r_\varphi(\bar{X})) \vee ([n]\, r_\gamma(\bar{Z}) \wedge [n{+}1]\, r_\varphi(\bar{X})),$$

where all variable are implicitly $\forall$-quantified. In this way, all auxiliary relations for temporal connectives can be defined inductively using first-order formulas. Now a PTL formula φ is called *evaluable*, if all first-order formulas used in the inductive definition of φ are evaluable [Cho95b, CT95].[14] If this is not the case, then φ is rejected. Otherwise, the definitions of auxiliary relations can be translated into RANF as described in [CT95], and thus into range-restricted Statelog rules.

Relativized PTL Operators

When evaluating PTL formulas in a Statelog structure, all states of the database history are considered. However, one may wish to restrict or *relativize* the evaluation to certain distinguished states, most notably in order to distinguish between intermediate and final states.

Example 7.10 (Not Again) Consider the temporal query "*who married the same person again?*". It may be expressed in PTL as follows:

$$\varphi_{again}(X) := \quad \exists Y \;\blacklozenge(m(X,Y) \wedge \blacklozenge(\neg\, m(X,Y) \wedge \bullet m(X,Y))).$$

If a tuple $m(x,y)$ is transiently deleted during a transaction which eventually fails (e.g., in the course of a hypothetical update), this "pseudo"-divorce should not yield an answer for φ_{again}. Instead, the PTL formula should only "see" final states. □

[14] See [VGT91] for the definition of *evaluable* in the case of FO.

There is a simple way to relativize the scope of PTL operators to specific states using some sentence $\varrho \in$ PTL to identify the relevant states:

$$\begin{array}{rcl} \blacksquare_\varrho\, \varphi & := & \blacksquare(\varrho \to \varphi) \\ \blacklozenge_\varrho\, \varphi & := & \blacklozenge(\varrho \wedge \varphi) \\ \bullet_\varrho\, \varphi & := & (\neg\, \varrho)\, \mathbf{S}\, (\varrho \wedge \varphi) \\ \varphi\, \mathbf{S}_\varrho\, \psi & := & (\varrho \to \varphi)\, \mathbf{S}\, (\varrho \wedge \psi). \end{array}$$

For example, if $\varrho = \mathsf{BOT} \wedge \mathsf{EOT}$ then precisely the past final states are specified (Figure 7.1). Therefore, we may presuppose such a relativization for the evaluation of PTL integrity constraints and for the detection of composite events, which is considered next.

7.7 Composite Events

So far, we have only considered primitive events, i.e., external events $\rhd e(\bar{X})$ and internal events given in the rule body (e.g., **del**:$p(\bar{X})$). However, Statelog and the related approaches XY-Datalog and Datalog$_{1S}$ also allow to express composite events:

In [MZ95], it is shown how composite events expressed in an event pattern language *EPL* can be precisely specified by translating them into a Datalog$_{1S}$-like language.

Alternatively, for composite events evaluated with unrestricted parameter context, one can obtain a very intuitive and declarative specification by expressing them as PTL formulas. Since PTL formulas have a natural representation in Statelog, this yields an elegant method for expressing composite events. Consider the event algebra from Section 3.2.1. The corresponding operators are definable in PTL as follows:

- $(E_1 \mid E_2) \;:\Leftrightarrow\; E_1 \vee E_2,$ (*disjunction*)
- $(E_1\,; E_2) \;:\Leftrightarrow\; (\blacklozenge E_1) \wedge E_2,$ (*sequence*)
- $(E_1\,, E_2) \;:\Leftrightarrow\; (\blacklozenge E_1) \wedge E_2 \;\vee\; (\blacklozenge E_2) \wedge E_1,$ (*conjunction*)
- $(E_1 \& E_2) \;:\Leftrightarrow\; E_1 \wedge E_2,$ (*simultaneous conjunction*)
- $E_2 \in [E_1; E_3] \;:\Leftrightarrow\; E_2 \wedge (\neg E_3\, \mathbf{S}\, E_1),$ (*within event*)
- **not** $E_2 \in [E_1; E_3] \;:\Leftrightarrow\; \neg E_2 \wedge (\neg E_3\, \mathbf{S}\, E_1).$ (*negation*)

Apart from these events, many others can be defined in PTL. For example, an "immediate" version of the sequence $(E_1\,; E_2)$ is $(\bullet E_1) \wedge E_2$. Note that E_1, E_2, and E_3 are event *atoms* possibly containing parameters.

Example 7.11 (Composite Event Detection)
The parameterized composite event $e(X,Y) := (\rhd f(X)\,; \rhd g(Y))$ occurs whenever the external event $\rhd g(Y)$ occurs, provided $\rhd f(X)$ occurred sometime before. The corresponding PTL formula

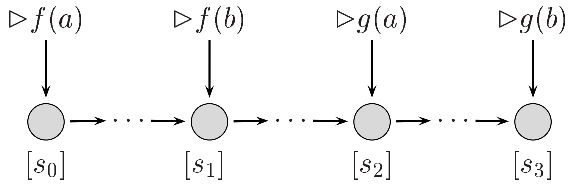
$$\varphi_e(X,Y) := \blacklozenge(\rhd f(X)) \;\wedge\; \rhd g(Y)$$

is expressed by the following Statelog program P_e (obtained from Ψ):

P_e:

$$\begin{array}{l} [S{+}1]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \triangleright\mathsf{f}(\mathsf{X}). \\ [S{+}1]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}). \\ [S]\ \mathbf{detd}_e(\mathsf{X},\mathsf{Y}) \leftarrow [S]\ \triangleright\mathsf{g}(\mathsf{Y}),\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}). \end{array}$$

Here, the notation $\mathbf{detd}_{ev}$ emphasizes that event "ev" has been **det**ecte**d**. For example, given the external event occurrences



(EXT)

we obtain at $[s_2]$: $\mathbf{detd}_e(a,a), \mathbf{detd}_e(b,a)$, and at $[s_3]$: $\mathbf{detd}_e(a,b), \mathbf{detd}_e(b,b)$. □

In the extended Statelog framework, rule sets like P_e for detecting composite events—and similar ones for checking static and dynamic constraints—are system-defined. Thus, composite event detection is included in a Statelog user program simply by adding the corresponding goal (e.g., $\mathbf{detd}_e(\mathsf{X},\mathsf{Y})$ in Example 7.11) to the rule body.

Parameter Contexts

Parameter contexts other than the unrestricted context have a procedural flavor, since they define which of the constituent events "take part" in the composite event, and how they are "consumed" by the composite event. In Section 3.2.1, several such parameter contexts (or *event consumption modes*) have been considered. As shown in [MZ95], such contexts can also be expressed in Datalog$_{IS}$. We illustrate the ideas of the encodings in [MZ95] using Statelog syntax and the composite event from Example 7.11:

Unrestricted Context: In this context, events are not consumed at all, but may be reused arbitrarily many times. The above specification of composite events in PTL corresponds to event detection with unrestricted context. The program P_e of Example 7.11 illustrates the encoding in Statelog.

Recent Context: In this context, only the most recent occurrences of constituent events are used; all previous occurrences are lost:

$$\begin{array}{l} [S{+}1]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \triangleright\mathsf{f}(\mathsf{X}). \\ [S{+}1]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}),\ \neg\triangleright\mathsf{f}(\_). \\ [S]\ \mathbf{detd}_e(\mathsf{X},\mathsf{Y}) \leftarrow [S]\ \triangleright\mathsf{g}(\mathsf{Y}),\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}). \end{array}$$

The second rule propagates a previously detected f-event only if there is no occurrence which is more recent.[15] For the external events (EXT) from Example 7.11, $[s_2]\,\mathbf{detd}_e(b,a)$ and $[s_2]\,\mathbf{detd}_e(b,b)$ are reported.

Alternatively, one may define that the occurrence of the composite event consumes all constituent events (Section 3.2.1, p. 14), so only $[s_2]\,\mathbf{detd}_e(b,a)$ is reported. This semantics is obtained by adding the goal "$\neg\,\mathbf{detd}_e(\_,\_)$" to the second rule.

[15]Recall that "$\neg\, p(\_)$" is shorthand for "$\neg\exists X\, p(X)$".

Continuous Context: In this context, a constituent event can be shared by several simultaneous occurrences of the composite event, and will be consumed only after the composite event is detected:

$$
\begin{array}{l}
[S{+}1]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \ \triangleright\mathsf{f}(\mathsf{X}). \\
[S{+}1]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}) \leftarrow [S]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}),\ \neg\,\mathbf{detd}_{e}(\_,\_). \\
[S]\ \ \mathbf{detd}_{e}(\mathsf{X},\mathsf{Y}) \leftarrow [S]\ \ \triangleright\mathsf{g}(\mathsf{Y}),\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}).
\end{array}
$$

The second rule discards all previous f-events, when the composite event is detected. Thus, only $[s_2]\,\mathbf{detd}_e(a,a)$ and $[s_2]\,\mathbf{detd}_e(b,a)$ are detected.

Chronicle Context: In this context, events are consumed in their chronological order. Thus, a g-event has to be paired with the oldest unconsumed f-event:

$$
\begin{array}{l}
[S{+}1]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}_T) \leftarrow [S]\ \ \triangleright\mathsf{f}(\mathsf{X}),\ T = S. \\
[S{+}1]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}_T) \leftarrow [S]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}_T),\ \neg\,\mathbf{detd}_{e}(\mathsf{X}_T,\_). \\
[S]\ \ \mathsf{not\_oldest}_f(\mathsf{X}_{T'}) \leftarrow [S]\ \ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}_T),\ \mathbf{detd}_{\blacklozenge f}\ (\mathsf{X}_{T'}),\ T < T'. \\
[S]\ \ \mathbf{detd}_{e}(\mathsf{X}_T,\mathsf{Y}) \leftarrow [S]\ \ \triangleright\mathsf{g}(\mathsf{Y}),\ \mathbf{detd}_{\blacklozenge f}(\mathsf{X}_T),\ \neg\,\mathsf{not\_oldest}_f(\mathsf{X}_T).
\end{array}
$$

The first rule "timestamps" X with the occurrence time T. The second rule discards a previous f-event only if it has been used in the composite e-event. The third rule defines when an f-event is not the oldest unconsumed event. The last rule detects the composite event by pairing the current g-event with the oldest unconsumed f-event. As expected, the rules report $[s_2]\,\mathbf{detd}_e(a,a)$ and $[s_3]\,\mathbf{detd}_e(b,b)$. Notice that the program is state-stratified but not stratified.

Strictly speaking, the rules for implementing event detection with *chronicle* context are not pure Statelog (and not Datalog_{1S} either). First, note that we can dispense with the subscript T by adding an extra argument for all involved relations. Hence, the expression X_T is *not* critical here. However, in the first rule, the state variable S is equated with the data variable T, thereby mixing the temporal and data domain. If such an extension is not used in a restricted way, many desirable properties of Statelog no longer hold. For example, the rule "$[S]\,p(X) \leftarrow\ [S]\,X = S$" induces a non-periodic infinite model.

In [MZ95], the expression $\mathsf{X.Time}$ was used instead of X_T, which avoids the "illegal" equation $T = S$. It should be clear, however, that both notations are essentially equivalent and extend the original Datalog_{1S} (respectively Statelog) framework. Indeed, as shown below, it is impossible to encode in pure Statelog (and therefore also in PTL) composite event detection with chronicle context. The reason is that all properties expressible in Statelog and Datalog_{1S} admit a so-called *bounded history encoding* [Cho95b], whereas composite event detection with chronicle context does not admit such an encoding.

7.8 Bounded History Encoding

The Statelog rules used to encode PTL constraints (Theorem 7.3) are essentially the inductive definitions of auxiliary relations given in [Cho95b]. There, Chomicki shows that the space required by his method only depends on the number of elements in the active domain but not on the length of the database history. This property is called *bounded history encoding*. The basic idea of the encoding is that relevant information needed to query the history $\mathcal{I}[0..k]$ of a database over the schema $\mathbf{R}$ is encoded in an auxiliary database $\widehat{\mathcal{I}}_k$ over an extended database schema $\widehat{\mathbf{R}} \supseteq \mathbf{R}$. If the database is updated leading to a new state $\mathcal{I}[k+1]$, then the history encoding $\widehat{\mathcal{I}}_{k+1}$ of $\mathcal{I}[0..k+1]$ is computed based only on $\mathcal{I}[k+1]$ and $\widehat{\mathcal{I}}_k$.

Intuitively, such an encoding is called *bounded*, if the size $|\widehat{\mathcal{I}}_k|$ depends solely on the number of domain elements occurring in $\mathcal{I}[0..k]$, but not on the length k of the history.

Using the notion of bounded history encoding, methods that require space depending on the length of the history can be distinguished from those that require space only on the cardinality of the active domain.

Definition 7.5 (History Encoding) Let $\mathbf{R}$, $\widehat{\mathbf{R}}$ be database schemas such that $\widehat{\mathbf{R}} \supseteq \mathbf{R}$. A *history encoding* is a function

$$\begin{array}{rrcl} enc: & inst(\mathbf{R}) \times inst(\widehat{\mathbf{R}}) & \to & inst(\widehat{\mathbf{R}}) \\ & (\,\mathcal{I}[k],\, \widehat{\mathcal{I}}_{k-1}\,) & \mapsto & \widehat{\mathcal{I}}_k \end{array}$$

mapping the current state $\mathcal{I}[k]$ and an encoding $\widehat{\mathcal{I}}_{k-1}$ of $\mathcal{I}[0..k-1]$ to an encoding $\widehat{\mathcal{I}}_k$ of $\mathcal{I}[0..k]$ over the extended schema $\widehat{\mathbf{R}}$. A history encoding is called *bounded* if there is a function $b: \mathbb{N} \to \mathbb{N}$ such that

$$|\widehat{\mathcal{I}}_k| \leq b(|adom(\mathcal{I}[0..k])|),$$

for all database histories $\mathcal{I}[0..k]$. A history encoding is called *polynomially bounded* if additionally there is some $c \in \mathbb{N}$ such that for all n: $b(n) < n^c$. □

Thus, a history encoding is bounded if the size of $\widehat{\mathcal{I}}_k$ is bounded from above by a function depending only on the size of the active domain.

Statelog History Encoding

Definition 7.5 can be easily applied to the Statelog framework: The "input" relations of a Statelog schema $\mathbf{P}$ are the base relations $edb(\mathbf{P})$ and the relations $\triangleright ev(\mathbf{P})$ for occurrences of external events. All other relations (e.g., views, delta relations, and control relations) may be conceived as derived or auxiliary relations. This allows to associate with every Statelog program P (over the schema $\mathbf{P}$) a history encoding enc_P as follows: Let

$$\mathcal{I}[k] \;:=\; \mathcal{M}_{P \cup \mathcal{D} \cup \mathcal{E}_{\leq k}}[k]|_{edb(\mathbf{P}) \cup \triangleright ev(\mathbf{P})}\,, \text{ and} \tag{7.1}$$

$$\widehat{\mathcal{I}}_{k-1} \;:=\; \mathcal{M}_{P \cup \mathcal{D} \cup \mathcal{E}_{\leq k-1}}[k-1]. \tag{7.2}$$

Thus, we view only relations from $edb(\mathbf{P})$ and $\triangleright ev(\mathbf{P})$ as the database relations of $\mathcal{I}$, whereas all other relations belong to the extended schema needed to encode $\widehat{\mathcal{I}}$.[16] Given (7.1) and (7.2), and $P \in$ NF-Statelog, it is clear that

$$enc_P: \quad (\mathcal{I}[k], \widehat{\mathcal{I}}_{k-1}) \mapsto \widehat{\mathcal{I}}_k$$

is a history encoding: The new (full) state $\widehat{\mathcal{I}}_k$ is completely determined from the previous (full) state $[k-1]$ and the values of external events at $[k]$. Similar to the method in [Cho95b], we can show for enc_P:

[16] $\mathcal{E}_{\leq k}$ is the set of external events that have occurred no later than state $[k]$; cf. Section 7.3.

Theorem 7.4 (Statelog History Encoding)
The history encoding induced by a Statelog program is polynomially bounded.

PROOF Let $\mathbf{P}$ be the schema of P, and $\widehat{\mathcal{I}}_k$ as in (7.2). The domain elements of the Statelog model $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}$ are given by the constants in $P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}$ (recall that Statelog rules cannot invent new values, since they are evaluated wrt. the active domain). It follows that there are at most $n_k := |adom(\mathcal{I}[0..k])|$ elements in the active domain. Thus, the size of the snapshot at $[k]$ is bounded from above by

$$|\widehat{\mathcal{I}}_k| \leq \sum_{p\in\mathbf{P}} n_k^{arity(p)},$$

which is polynomial in n_k. ■

The above timestamping technique for encoding composite event detection with chronicle context implements a kind of event queue. Indeed, it seems that any encoding of composite event detection with chronicle context essentially requires the use of a queue, and thus is not representable using a bounded history encoding. The affirmative answer to this intuition is provided by the following negative result:

Theorem 7.5 (Inexpressibility of Chronicle Context)
Composite events with chronicle context do not admit a bounded history encoding.

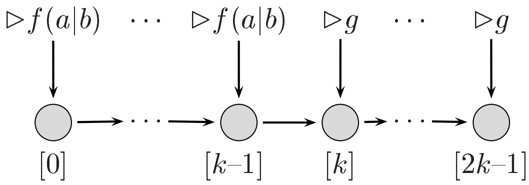
PROOF Assume the contrary. Then it is possible to detect the composite event

$$e(X) := (f(X)\,;g)$$

with chronicle context over the schema $\mathbf{R} := \{f, g\}$ and some fixed $\widehat{\mathbf{R}} \supseteq \mathbf{R}$. We restrict the domain of f to $\{a, b\}$, so no more than $n_k := 2$ elements occur in $\mathcal{I}[0..k]$. Therefore, a bounded history encoding must satisfy

$$|\widehat{\mathcal{I}}_k| \leq b(2),$$

for all k. Thus, we can assume that the size of $\widehat{\mathcal{I}}_k$ is at most n_0 bits. Consider a sequence of $k := n_0{+}1$ consecutive f-events over $\{a, b\}$, followed by a sequence of k g-events:



Observe that for $x \in \{a, b\}$ and all $n = k, \ldots, 2k-1$, $e(x)$ occurs at $[n]$ if and only if $f(x)$ occurs at $[n{-}k]$. Therefore, a bounded history encoding would have to repeat at $[k], \ldots, [2k{-}1]$ an arbitrary f-sequence $(a|b)^k$, based solely on $\widehat{\mathcal{I}}_{k-1}$ and the constant window $\mathcal{I}[k..2k{-}1] = \{g\}^k$ (see Definition 7.5). However, there are 2^{n_0+1} f-sequences of length $n_0{+}1$, so not all of them can be distinguished using only $|\widehat{\mathcal{I}}_{k-1}| \leq n_0$ bits. ■

Although Theorem 7.5 implies that composite events with chronicle context cannot be handled in pure Statelog (and Datalog$_{1S}$), in practice, this limitation is not as severe as it may seem. First, by bounding the length of the event queue to some fixed n_0, an encoding in the pure framework becomes possible (e.g., one can simulate a ring-buffer in Statelog for that purpose). Alternatively, if we consider timestamps of occurred events as part of the active domain, composite events with chronicle context *are* expressible. However, note that by admitting timestamps as elements of the active domain, the advantage of bounded history encoding is partially undermined, since the active domain increases continually even if only a single event, say $\triangleright f(a)$, keeps recurring. Moreover, the burden is on the programmer to ensure that timestamps are used in a disciplined way. A sufficient condition for the safe use of built-ins, which also applies to timestamping, is presented in Section 8.1.3.

7.9 Related Work

The Statelog framework for active rules combines formal foundations of deductive databases with dynamic features of active rules. From this perspective, related work can be roughly centered around the notions of *formal frameworks for active rules* and *updates and deductive rules*. There is a vast literature in both areas, hence the following provides a—necessarily incomplete—overview only. Some further approaches are discussed in [LLM98b, LLM98a].

7.9.1 Formal Frameworks for Active Rules

Heraclitus. The *Heraclitus* approach [GHJ96] is based on the idea of elevating deltas to "first-class citizens" of the language. A number of operations on delta relations are presented which can be used for database programming. Moreover, it is shown how execution models for active rules can be implemented in Heraclitus. In a sense, the possibility to refer to different database states and to delta relations is equivalent, and the presented Statelog framework for active rules can be regarded as a logical pendant to Heraclitus.

Situation Calculus Based Approaches. In [BL96b, BLT97] a language $\mathcal{L}_{active}$ for active rules is developed, which allows to formalize and reason about the behavior of active rules. The language borrows from $\mathcal{L}_1$ [BGP97], an extension of the action description language $\mathcal{A}$ [GL93] used for modeling actual and hypothetical actions and situations, which in turn is based on the situation calculus. The main constructs of $\mathcal{L}_{active}$ are *causal laws* describing which fluents are added or deleted by an action, *executability conditions* stipulating when actions can be executed, and *active rules* defining a triggering event, an evaluation mode, a conjunctive precondition, and a sequence of actions. The automaton-based semantics of $\mathcal{L}_{active}$ uses transition diagrams with states (labeled by sets of fluents) and transitions (labeled by actions) to specify the meaning of an active database description in $\mathcal{L}_{active}$. A translation of $\mathcal{L}_{active}$ into logic programs is presented using a situation calculus notation. The generated rules are non-stratified, and the choice operator of [SZ90] (which is based on stable models; see Section 8.3) is used for nondeterministically selecting one rule among all rules that may be fired in a situation. Like the situation calculus, $\mathcal{L}_{active}$ focuses more on reasoning about the effect of actions than on providing a language—like Statelog—for defining and executing actions.

Logic-Based Formalization of Operational Semantics. In [FWP97], a framework for the integration of the different *operational semantics* of active and deductive rules is developed. The meaning of ECA rules is specified using distinct specification languages for events, conditions, and actions, respectively. The operational semantics for these ECA sublanguages and their interplay is formalized by means of deductive rules. More precisely, the database history (i.e., the ordered set of database states) is modeled using timestamped atoms, and the meaning of events, conditions, and actions is defined based on the *event calculus* in [Kow92]. The approach has been used for the formalization of the active rule component which is added to the deductive object-oriented database system ROCK & ROLL [BFP+95]. In contrast to Statelog and XY-Datalog, which provide a single *unified language* for active and deductive rules, [FWP97] *integrate* the different *operational semantics* of active and deductive rules using a common (deductive) specification formalism.

Another approach to logic-based formalization of active rule semantics is presented in [FT95], where so-called *Extended ECA rules* are used to encode the operational semantics of an ADB. Using these user-readable EECA rules, existing ADBs can be compared and classified. The precise meaning of EECA rules is obtained by translating them into a logical core language which specifies procedural details like event consumption, coupling modes, etc. Unlike in Statelog, this logical language is *not* meant to be handled by the rule programmer, but is considered as an internal representation which is used by the execution model, and thus is on a lower-level than the EECA rules.

Expressive Power and Complexity of Active Rules. [PV97] develop a generic formal framework for the specification of active databases: A *trigger program* consists of rules of the form *condition* $\rightarrow$ *action*, where *condition* is a first-order sentence and *action* is an external program. Each rule is assigned a coupling mode (either immediate or deferred) and a set of database events (insertion, deletion) on which it reacts. It is assumed that a priority is assigned to rules, and that the semantics is deterministic. Existing active database prototypes can be obtained by specializing certain parameters of the framework which allows to compare their relative expressiveness. Moreover, the impact of active database features on expressive power and complexity is studied. In the framework of [PV97], the complexity of immediate triggering is essentially **EXPTIME**, even without delta relations and **PSPACE** if there is a bound on the nesting of immediate queues. Deferred triggering is more expressive and captures **PSPACE**, **EXPSPACE**, or all computations on ordered databases, depending on the allowed operations for queue management.

The Integrity-Rule-System IRS. In [Rei96b], an active rule system IRS (*Integrity-Rule-System*) for integrity maintenance in relational databases is developed. To obtain an unambiguous semantics, a notion of *logical simultaneousness* of operations is used (basically, conflicting operations have to be prevented in order to execute them simultaneously). The proposed ECAC rules ensure integrity using an additional test (hence the second "C") at the end of rule processing. In contrast to many other approaches, the presented language deliberately leans on the syntax an execution model of SQL.

7.9.2 Updates and Deductive Rules

The research on updates in deductive databases, or more generally on updates and change in logic frameworks is immense. Here, we restrict to the quite specific aspect of languages for *defining* (or *programming*) updates in deductive databases and refer the reader to [Win90] and [KM91] for a broader perspective.

[TU95] present a framework for classifying update problems in deductive databases: *Upward problems* deal with the propagation of changes from base relations to derived relations, whereas *downward problems* aim at finding base updates required to accomplish certain updates to derived relations. According to this classification, Statelog is a language for specifying (and implementing) rule-based strategies for solving upward and downward problems. For example, the DRed algorithm illustrated in Appendix A.1 solves an upward problem. Similarly, downward problems like view updates can be *programmed* in Statelog provided a strategy for translating derived updates to base updates is given (say by the user, or some procedure like abductive reasoning [KM90, Bry90]). Not all problems can be classified as either upward or downward, however. For example, the application described in Section A.2 deals with finding the intended set of induced base updates given a user-defined set of updates to base relations.

A common classification for rule-based database languages is according to the syntactic position of update requests:

Updates in the Body

Languages which express updates in the body are usually evaluated in a top-down fashion. For example, the semantics of `assert` and `retract` in Prolog is tied to the procedural top-down evaluation strategy. Well-known approaches in the deductive database community subsumable under the *updates in the body* paradigm are the early works on DLP [MW88] (which is based on dynamic logic), LDL updates [NK88, NT89], and the recent work on $\mathcal{T}_\mathcal{R}$ (Transaction Logic) [BK94, BK96]. $\mathcal{T}_\mathcal{R}$ deals, on a high level of abstraction, with the phenomenon of state changes in logic databases and employs a powerful model theory and proof theory. In contrast to Statelog, $\mathcal{T}_\mathcal{R}$ focuses on the composition of complex updates, whereas primitive updates (so-called *elementary transitions*) are not part of $\mathcal{T}_\mathcal{R}$ but regarded as parameters which are supplied by a transition oracle.

Approaches based on top-down evaluation are often tuple-oriented, so the specification of set-oriented updates (bulk-updates) becomes an issue. [WF97] present an update language based on so-called *deferred updates*: In a first top-down evaluation phase, a set of potential update requests is computed. In a second phase, one of the computed update sets is nondeterministically chosen and applied to the database. One of the benefits of the language is the integration of bulk-updates in a top-down framework.

Updates in the Head

Languages expressing update requests in the head of rules lend themselves naturally to a bottom-up evaluation. In this setting, bulk-updates are usually not an issue, since the standard bottom-up evaluation techniques are based on set-oriented algebra operations. Typical representatives are production rules in general, and the Datalog extensions for updates in particular [AV88, AV91], [Via97b]. As shown in the present work, these Datalog extensions are special cases of Statelog rules (cf. Section 6.6). Indeed Statelog can be seen in this tradition of data-driven update languages due to its natural bottom-up reading of rules. However, by the introduction of an explicit notion of state and the specification of primitive updates by frame rules, a declarative semantics of updates is attained in Statelog. In principle, a (reified) Statelog program can also be evaluated with top-down techniques enhanced by some form of tabling to ensure termination [CW93]. These tabling techniques are part of the XSB-Prolog system, which has been used for the prototypical implementation of Statelog described in [Ham95].

The *PARK semantics* of active rules [GMS92] can be conceived as an inflationary fixpoint semantics extended by a mechanism to handle update literals $+L$ and $-L$, denoting insertion and deletion of L, respectively. Similar as in Statelog, update literals correspond to events if they occur in the body, and to actions if they occur in the head. A main benefit of the approach is the simple and precise semantics with its flexible conflict resolution policy, the latter being a parameter to the PARK semantics. The Statelog approach also allows flexible conflict resolution policies, however, they are not treated as a black box as in [GMS92], but can be programmed *within* the logical rule language (Section 7.4.2). [BM97] apply the well-founded semantics to resolve conflicting update requests; [AKLS95] present another update approach which uses the well-founded semantics and explicitly stores inserted and deleted facts of an extended Datalog$^{\neg}$ program.

7.10 Summary

In this chapter, a Statelog framework for active rules has been developed. An important concept for modeling active rules are *delta relations* (deltas). Deltas can be queried by the user in the body of rules; new updates are requested by delta relations in the head of user-defined rules. Since user-defined rules are local, their semantics corresponds to the well-known declarative semantics of deductive rules. Therefore, their meaning and behavior is usually much easier to comprehend and predict than the rather involved procedural semantics prevailing in active databases. The meaning of deltas during the transition to the next state is specified by system-defined frame rules. Apart from the obvious effects for insertions and deletions, frame rules also specify the behavior in case of conflicting updates. Further system-defined rules are used to handle some basic transaction control like maintaining the net effect in protocol relations, and committing or aborting a transaction. By restricting user programs to guarded or Δ-monotonic rules, terminating classes of programs evaluable in **PTIME** are obtained. Integrity constraints, specifically dynamic constraints expressed in past first-order temporal logic (PTL), can be easily incorporated into the framework: The method for evaluating PTL formulas proposed in [Cho95b, CT95] uses a bounded history encoding and can be directly implemented in Statelog. The notion of bounded history encoding has proven fruitful, as it allows to distinguish composite event detection modes expressible in Statelog (unrestricted, recent, and continuous parameter contexts) from those which are not expressible in the pure language (chronicle context). The applicability of the presented framework for advanced rule-based programming tasks is illustrated in Appendix A by two typical examples, i.e., incremental view maintenance and maintenance of referential integrity.

8 Extensions

The Statelog framework for active rules can be extended in several ways. First, we propose conditions for the safe use of built-ins. These safeness conditions are not only applicable for the usual built-ins like arithmetics, but also provide means for the safe value invention of timestamps, as required for detecting composite events with chronicle context. The main ideas of the Statelog extension with procedures and nested transactions developed in [LML96] are presented next. Finally, nondeterministic extensions and their simulation using an ordered domain are discussed.

8.1 Built-in Relations

The use of built-in functions and relations (or *built-ins* for short) is essential in many practical applications. For instance, the ubiquitous update example for increasing the salaries of all employees (see Example 7.2) uses built-in arithmetics to compute the new salary values. Other common relations which may be conceived as built-ins are comparison operators like "$<$", "$\leq$", and "$\neq$". In principle, built-ins can be regarded as (often infinite) predefined relations. For example, the goal Z := X*Y corresponds to a relation `mult(X,Y,Z)` with the obvious fixed extension (cf. [CGT90, Ch. 11]). For simplicity of presentation, we assume that built-ins are totally defined, so "ill-typed" goals like `mult(3,john,X)` evaluate to *false* for all values of X. Built-ins increase the practical applicability of a language, but often also its complexity if not used in a disciplined way. Specifically, the nice properties of Statelog may be lost: For example, the Statelog user rule

$$\textbf{mod}{:}\mathsf{emp(E,Sal/Sal')} \leftarrow \mathsf{emp(E,Sal),\ Sal'{:=}Sal{+}42}$$

creates an infinite *non-periodic* sequence of states. Indeed, already by incorporating integer counters into the language *While*, a language While_N is obtained which expresses all queries on ordered databases, so this language has no upper complexity bound (cf. Figure 4.2 and [AHV95]). In order to add built-ins to Statelog and at the same time preserve the established expressiveness and complexity results, we present sufficient conditions for the safe use of built-in relations. The conditions guarantee that the active domain of $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}$ is finite for all $\mathcal{D}$ and all $\mathcal{E}_{\leq k}$, hence one remains in the realm of the finite. In particular, the presented use of built-ins is readily implementable using state-of-the-art database technology.[1]

In the sequel, let $\mathbf{R}$ be a given database schema (i.e., not involving built-ins), and $\mathbf{B}$ a set of built-in relation symbols, the *built-in schema*.

[1]Another very general framework for dealing with certain decidable classes of infinite relations are *constraint databases* [KKR95], which require special algorithms for handling constraints.

8.1.1 Limited Variables

To avoid the creation of infinitely many values in the active domain, we first use a notion of *limited variables*, which allows to handle arbitrary safe built-ins and generalizes similar notions from [Ull88] (where "=" is the only built-in) and [NT89] (where *covered variables* in the presence of "=" and "≠" are considered).

In the following, we assume that for all built-ins $\beta \in \mathbf{B}$ a function *ltd* is given, mapping a built-in atom $\beta(\bar{X})$ with free variables among $\bar{X}$, and a set of (already limited) variables $\bar{Y}$, to a set $\bar{Z}(= \mathit{ltd}(\beta(\bar{X}), \bar{Y}))$ of new limited variables. We further assume that the function *ltd* is defined in such a way, that for any finite set of instances $\bar{y}$ of $\bar{Y}$ for which β is defined, the evaluation of $\beta(\bar{X})\{\bar{Y}/\bar{y}\}$ always yields a finite answer.[2] For the built-in relation $\beta = \mathtt{mult}$ for example, the definition of *ltd* includes

- $\mathit{ltd}(\mathtt{mult}(X, Y, Z), \{X, Y\}) := \{Z\}$, and
- $\mathit{ltd}(\mathtt{mult}(X, Y, Z), \{Y\}) := \emptyset$.

The first equation states that for any finite set of values for X and Y (say over `integer`), a finite number of values for Z is created. In the second case, even if Y is limited, infinitely many values for X and Z can be created, so *ltd* does not yield new limited variables.

Definition 8.1 (Limited Variables)
Let $\mathbf{R}$ be a database schema, $\mathbf{B}$ a built-in schema, and r a rule involving only relations from $\mathbf{R}$ and $\mathbf{B}$. We denote by $\mathit{pos}(r)$ the positive literals occurring in the body of r. The *limited variables* $\mathit{lvars}(r)$ of a rule r are defined as the least set such that

1. if $p(\bar{X}) \in \mathit{pos}(r)$, $p \in \mathbf{R}$, and X occurs in $\bar{X}$, then $X \in \mathit{lvars}(r)$,
2. if $\beta(\bar{X}) \in \mathit{pos}(r)$, $\beta \in \mathbf{B}$, $X \in \mathit{ltd}(\mathit{lvars}(r), \beta(\bar{X}))$, then $X \in \mathit{lvars}(r)$. □

Observe that the usual definitions for limited variables [Ull88] and covered variables [NT89] are subsumed by this definition. For instance, when considering "=" and "≠" as built-ins, the definition of *ltd* includes

- $\mathit{ltd}(=(X, Y), \{X\}) := \{Y\}$,
- $\mathit{ltd}(=(X, Y), \{Y\}) := \{X\}$, and
- $\mathit{ltd}(\neq(X, Y), \_) := \emptyset$.

8.1.2 Safe Programs

Statelog database relations are usually state dependent: base relations from $\mathit{edb}(\mathbf{R})$ are directly updateable by user programs, whereas derived relations from $\mathit{idb}(\mathbf{R})$ are changed indirectly updates to base relations. In contrast, built-in relations $\beta \in \mathbf{B}$ are typically *state independent*, i.e., for all $n \in \mathbb{N}_0$:

$$[n]\,\beta(\bar{X}) \;\Leftrightarrow\; [n{+}1]\,\beta(\bar{X}).$$

However, some built-in relations are also subject to change, e.g., a built-in like `date(D)` which returns the current date. It turns out that the distinction between state independent and state

[2] $\beta(\bar{X})\{\bar{Y}/\bar{y}\}$ denotes the atom obtained by substituting the values $\bar{y}$ for the variables $\bar{Y}$ in $\beta(\bar{X})$.

dependent built-ins is crucial for their safe use. Intuitively, the reason is that the repeated application of safe state independent built-ins (like `mult`)—under certain conditions—creates only a bounded number of new values, whereas in the case of state dependent built-ins (such as `date`), a continually increasing number of new values may be created.

Definition 8.2 (Safe Programs)
A Statelog *rule* r is called *built-in safe*, or just *safe*, if the set of all variables of r coincides with $lvars(r)$. A Statelog *program* P is called *(built-in) safe*, if every rule of P is safe, and additionally, for every rule r involving built-ins:

1. r is guarded, *or*

2. r is non-recursive, and all built-ins of r are state independent. □

Note that if r contains no built-ins, then r is safe iff r is range-restricted, since the latter holds if every variable of r occurs positively in the body (cf. (1) in Definition 8.1 and Section 6.4, p. 45).

Given the above definitions, it is clear that for every finite database $\mathcal{I}$, a *single* evaluation of a safe rule r creates at most finitely many values, i.e., for $P = \{r\}$, the set $T_P(\mathcal{I})$ of immediate consequences is finite. The requirements (1) and (2) of Definition 8.2 are used to ensure that also a *set* of rules can be evaluated safely:

Proposition 8.1 (Safe Programs) *If P is safe, then for all databases $\mathcal{D}$, all k, and all $\mathcal{E}_{\leq k}$, the active domain of $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}$ is finite.*

Proof Fix P, $\mathcal{D}$, and k. New values in the active domain can only be invented via built-ins. The active domain of $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}$ can become infinite only if (i) already in some state $[n]$, infinitely many values are created, or (ii) if the active domain of $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}[0..n]$ is continually increasing with n. Clearly, the safeness conditions guarantee that (i) cannot occur. Therefore, the active domain of $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}[0..k]$ is finite. Consider all rules r with built-ins and their evaluation at $[k{+}1], [k{+}2], \ldots$ For rules r which are guarded, no values are invented because external events have only occurred before state $[k{+}1]$. Thus, the only rules with built-ins which eventually contribute to $\mathcal{M}_{P\cup\mathcal{D}\cup\mathcal{E}_{\leq k}}$ are non-recursive and state independent, which implies that the number of possible new values in $[k{+}1], [k{+}2], \ldots$ is finite. ■

Update rules like **mod**:$p(\bar{X}/\bar{X}') \leftarrow p(\bar{X}), \ldots$ are progressively recursive since they define $[n{+}1]\,p(\ldots)$ based on $[n]\,p(\ldots)$, and possibly some other conditions in the rule body. Thus, such rules are excluded by condition (2) of Definition 8.2. Indeed, in the presence of built-ins the use of guards is essential for such rules to be safe. Similarly, for non-recursive rules with built-ins, state independence is crucial:

Example 8.1 (Safe and Unsafe Rules) Consider the following Statelog user rules:

r_1:	**mod**:emp(E,Sal/Sal') ← ▷raise(F), emp(E,Sal), `mult`(Sal,F,Sal').
r_2:	net_salary(E, Sal') ← emp(E,Sal), taxation(E,F), `mult`(Sal,F,Sal').

r_1 is progressively recursive, since it defines the value of emp at the next state using the value of emp at the current state. Since r_1 is guarded, it is applicable only once within a transaction. If the guard $\triangleright$raise(F) is removed, the rule becomes unsafe and induces an infinite active domain; in this case, the Statelog model is not terminating and not periodic.

r_2, in contrast to r_1, may be applied in every state. However, since it defines a non-recursive view based solely on state independent built-ins, it is safe and invents only finitely many new values. If the database relation `taxation` is replaced by a state dependent built-in relation `arb_tax` (assigning some arbitrary, state dependent tax factor to employees), then r_2 may create infinitely many new values, e.g., if the taxation factor F at $[n]$ happens to coincide with n. □

8.1.3 Safe Timestamping

Apart from the usual arithmetic operations, we can employ built-ins to implement a safe use of timestamping, especially for detecting composite events with chronicle context. Note that this does *not* contradict Theorem 7.5, because there, the conceptually superfluous values of timestamps are not part of the active domain. In contrast, by using a built-in relation for timestamping, the invented values become part of the active domain. With this enlarged active domain a bounded history encoding of chronicle context is possible. Clearly, if timestamped external events occur regularly, the active domain and thus the required space increases continually, so the advantage of bounded history encoding is partially undermined.

Example 8.2 (Safe Timestamping) Assume a built-in function $\texttt{time\_}e(\bar{X}, T)$ exists which returns the system time of the most recent occurrence of the external event $\triangleright e(\bar{X})$. Thus, we can assume that $ltd(\texttt{time\_}e(\bar{X}, T), \bar{X}) = \{T\}$. For the composite event of Example 7.11, event detection with chronicle context can be expressed as follows:

$[S{+}1]$ $\mathbf{detd}_{\blacklozenge f}$(X,T) $\leftarrow [S]$ $\triangleright$f(X), time_f(X,T).
$[S{+}1]$ $\mathbf{detd}_{\blacklozenge f}$(X,T) $\leftarrow [S]$ $\mathbf{detd}_{\blacklozenge f}$(X,T), $\neg\,\mathbf{detd}_e$(X,T,_).
$[S]$ not_oldest$_f$(X,T') $\leftarrow [S]$ $\mathbf{detd}_{\blacklozenge f}$(X,T), $\mathbf{detd}_{\blacklozenge f}$(X,T'), T < T'.
$[S]$ $\mathbf{detd}_e$(X,T,Y) $\leftarrow [S]$ $\triangleright$g(Y), $\mathbf{detd}_{\blacklozenge f}$(X,T), $\neg$ not_oldest$_f$(X,T).

Note that for a fixed value of X, `time_f` may return different values for T, depending on the state $[n]$ in which it is invoked. Hence, `time_f` is state dependent. □

8.2 Nested Transactions

A Statelog program defines for a given database and a given set of external events, a single transaction from the current state to the new final state. Subsequently occurring external events initiate the next transaction. Thus, the Statelog execution model depicted in Figure 7.1 corresponds to a flat transaction model. To distinguish this model from the extension described below, we sometimes call the former *flat Statelog*. In [LML96], it is shown how Statelog can be extended to model *nested transactions*. In the following, we confine ourselves to a presentation of the main ideas of this extension.

The following example, adapted from [MW88, Che95], motivates why structuring capabilities and a more nested transaction model may be useful:

Example 8.3 (To Hire or Not to Hire) Consider the database schema from Example 6.1. We want to hire an employee only if the average salary *after* the update does not exceed a certain limit. Such a "post-conditional" update may be expressed in flat Statelog as follows:[3]

$[S]$ **ins**:empsal(E,Sal,D), $[S{+}1]$ checksal(D) $\leftarrow [S]$ $\triangleright$hire_if_possible(E,Sal,D).
$[S]$ check_ok $\leftarrow [S]$ checksal(D), avg(D, AvgSal), AvgSal $<$ 50000.
$[S{+}1]$ **del**:empsal(E,Sal,D) $\leftarrow [S]$ $\triangleright$hire_if_possible(E,Sal,D), $[S{+}1]$ $\neg$ check_ok.

On occurrence of the external event $\triangleright$hire_if_possible(E,Sal,D), the new employee is preliminarily inserted and the new average salary is checked in $[S{+}1]$. If it exceeds the admissible amount, the effect of the insert is undone by the last rule. □

Although this program specifies the desired transaction, there are some problems with this "flat" approach:

- Undoing the effect of changes (here: the compensation of insertions by corresponding deletions) has to be programmed by the rule designer. However, it is often desirable to *automatically* propagate the failure of a subtransaction like checksal.
- There is no structure which allows grouping of semantically closely related rules.
- The effects of *ephemeral changes* [Zan95], i.e., changes whose effect is undone later within the same transaction, and *hypothetical changes* are visible to other rules, since there is no encapsulation of effects of semantically related rules.

8.2.1 Procedures and Nested Transactions

In order to solve the above problems, [LML96] propose the concept of Statelog *procedures.* A Statelog procedure π is a named and possibly parameterized set of Statelog user rules. In this sense, the flat Statelog programs considered so far can be seen as parameterless anonymous Statelog procedures. When π is called at run-time, it defines a transaction T_π by issuing primitive updates (through delta relations) and/or calling other procedures which in turn may define subtransactions, etc. T_π either terminates successfully (indicated by a special predicate **committed**:π), or aborts. When π calls another procedure μ, a subtransaction T_μ is started whose results are either incorporated into T_π, if T_μ commits, or discarded otherwise. From the point of view of the calling transaction T_π, the subtransaction T_μ is atomic, therefore requests derived directly within T_π and those submitted by T_μ should be indistinguishable. This is achieved by certain system-defined rules.

The behavior of μ is *encapsulated*, since deltas defined by T_μ are only visible within T_μ, but not in other (concurrent) transactions. Transactions execute in isolation and in an *all-or-nothing* manner, i.e., no results of T_μ will be visible in T_π if T_μ aborts. Note that this does *not* mean that T_π also aborts—on the contrary, π can detect the failure of T_μ (via **aborted**:μ) and issue alternative or compensating actions, or retry the execution of μ later.

In order to model the isolated execution of a Statelog procedure π as a (sub)transaction T_π, a unique name space for each (parameterized) invocation of $\pi(\bar{x})$ has to be introduced. This is accomplished by extended state terms and frame terms. The latter provide the *transaction frame* in which π executes.

[3]For simplicity, we view the predicate holding the average salary avg(D,AvgSal) of a department D as a built-in. Moreover, we also allow non-local rules here (although they could be avoided).

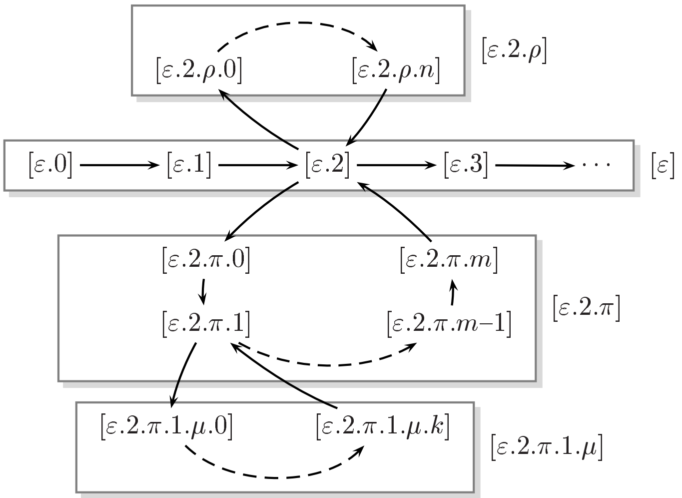


Figure 8.1: States and frames

Hierarchical State Space

The execution of Statelog procedures as nested transactions induces a hierarchical structure of the state space instead of the linear structure considered before (cf. Figure 8.1). Every state term has to encode the complete transaction hierarchy from the top-level transaction down to the current transaction. States on the same level are grouped into (transaction) frames.

Given the fixed set Π of procedure names of a Statelog program, the set of *(extended) state identifiers* $\mathbb{S}_\Pi$ and *frame identifiers* $\mathbb{F}_\Pi$ are defined as the least sets such that

- $[\varepsilon] \in \mathbb{F}_\Pi$,
- $[f.n] \in \mathbb{S}_\Pi$, if $[f] \in \mathbb{F}_\Pi$ and $n \in \mathbb{N}_0$,
- $[s.\pi(\bar{x})] \in \mathbb{F}_\Pi$, if $[s] \in \mathbb{S}_\Pi, \pi \in \Pi$, and $\bar{x} \in U$.

Here, the arity of π matches that of $\bar{x}$; U is the underlying domain. *(Extended) state terms* and *frame terms* are defined similarly but may also involve variables. State and frame identifiers induce a hierarchically structured state space: The *initial frame* $[\varepsilon]$ denotes the *top-level transaction*, its initial state is $[\varepsilon.0]$. Let $[s]$ denote the current state. Then for a procedure call $\pi(\bar{x})$ the frame of the subtransaction induced by the execution of $\pi(\bar{x})$ is $[f] = [s.\pi(\bar{x})]$, the first state of the transaction is $[f.0] = [s.\pi(\bar{x}).0]$. The successor state of $[f.n]$ (on the same level) is $[(f.n)+1] := [f.(n+1)]$. The grouping of states into a frame $[f] \in \mathbb{F}_\Pi$ is defined as

$$[f] := \{ [f.n] \mid n \in \mathbb{N}_0 \} ,$$

which implies that every state $[f.n]$ belongs to exactly one frame $[f]$. Using this representation, the frames $[s.\pi(\bar{x})]$ and $[s.\mu(\bar{y})]$ induced by *different* parallel procedure calls of π and μ in the *same state* $[s]$ can be uniquely identified (if the name of the procedure is the same, at least the parameters are different). Similarly, frames of transactions induced by the *same procedure call* from *different states*, $[s_1.\pi(\bar{x})]$ and $[s_2.\pi(\bar{x})]$, can also be distinguished.

8.2.2 User-Defined Rules

A Statelog *program* is a finite collection of Statelog procedures. A Statelog *procedure* π is an expression of the form

$$\textsf{\textbf{proc}}\ \pi(X_1, \ldots, X_n)\ \{H_1 \leftarrow B_1\,;\ \ldots\,;\ H_k \leftarrow B_k\,\}$$

where $X_1, \ldots, X_n$ are the *parameters* of π which may occur in the Statelog user rules $H_i \leftarrow B_i$. Since we assume that user rules are local, state terms can be omitted.

Let $P(\pi)$ denote the rules of π. As in the case of flat Statelog, the meaning of rules is given by the declarative semantics of their representation as a logic program. Especially, if rules are state-stratified, a unique perfect model exists.[4]

Translation of User-Defined Rules

For every rule $(H \leftarrow B) \in P(\pi)$ of a procedure π, the rule

$$[S_1.\pi(\bar{X}).S_2]\, H \leftarrow [S_1.\pi(\bar{X}).S_2]\, B$$

is generated. The top-level transaction is defined by a special procedure called main. Every $(H \leftarrow B) \in P(\mathsf{main})$ translates to a rule

$$[\varepsilon.S]\, H \leftarrow [\varepsilon.S]\, B.$$

8.2.3 System-Defined Rules

Like in flat Statelog, system-defined rules specify the meaning of primitive updates, conflict resolution, protocol relations, etc. The rules presented in Section 7.4.2 can be taken over literally to the extended framework. Therefore, we do not repeat them here, but only note that in a frame rule like

$$[S{+}1]\, p(\bar{X}) \leftarrow [S]\, \textsf{\textbf{ins}}{:}p(\bar{X}), \neg\, \mathsf{abort}$$

the state variable S may now be bound to an *extended* state identifier from $\mathbb{S}_\Pi$.

Procedure Rules

Apart from the rules handling presented in Section 7.4.2, additional rules are necessary to specify the handling of procedure calls.

A procedure call creates the initial state of a new frame, signals BOT and initializes all *edb*-relations p:

$$[S.\pi(\bar{X}).0]\, \mathsf{BOT} \leftarrow [S]\, \pi(\bar{X}).$$
$$[S.\pi(\bar{X}).0]\, p(\bar{Y}) \leftarrow [S]\, p(\bar{Y}), \pi(\bar{X}).$$

At the end of a subtransaction, results collected in protocol relations are returned to the calling transaction:

$$[S_1]\, \textsf{\textbf{ins}}{:}p(\bar{Y}) \leftarrow [S_1]\, \pi(\bar{X}), [S_1.\pi(\bar{X}).S_2]\, \textsf{\textbf{insd}}{:}p(\bar{Y}), \mathsf{EOT}.$$
$$[S_1]\, \textsf{\textbf{del}}{:}p(\bar{Y}) \leftarrow [S_1]\, \pi(\bar{X}), [S_1.\pi(\bar{X}).S_2]\, \textsf{\textbf{deld}}{:}p(\bar{Y}), \mathsf{EOT}.$$

[4] A model-theoretic semantics for Statelog with nested transaction based on Kripke structures is presented in [LML96].

Parent transactions perform some bookkeeping about committed and aborted subtransactions:

$$[S_1]\,\textbf{committed}{:}\pi(\bar{X}) \leftarrow [S_1]\,\pi(\bar{X}), [S_1.\pi(\bar{X}).S_2]\,\mathsf{commit}.$$
$$[S_1]\,\textbf{aborted}{:}\pi(\bar{X}) \leftarrow [S_1]\,\pi(\bar{X}), [S_1.\pi(\bar{X}).S_2]\,\mathsf{abort}.$$

8.2.4 Serial Conjunction

It is sometimes convenient, especially in the context of procedures and nested transactions, to provide the user with a special connective "$\otimes$" denoting a serial version of conjunction.[5] To this end, we define

$$A_0 \otimes \cdots \otimes A_n \leftarrow B_0 \otimes \cdots \otimes B_m$$

as a shorthand notation for the Statelog rule

$$[S{+}m]\,A_0, \ldots, [S{+}m{+}n]\,A_n \leftarrow [S]\,B_0, \ldots, [S{+}m]\,B_m.$$

For example, a rule of the form $A_0 \otimes A_1 \leftarrow B$, where A_0 and A_1 are primitive actions (like **ins**:p, **del**:p) or procedure calls, informally means: "*if B is true, then do A_0 followed by A_1*". Conversely, the rule $A \leftarrow B_0 \otimes B_1$ intuitively says: "*if B_0 was previously true, and B_1 holds now, then do A*".

Example 8.4 (Hypothetical Update)
In order to implement a hypothetical deletion of an employee, we can use the rule

$$\textbf{del}{:}\mathsf{emp(E,Sal,D)} \otimes \textbf{ins}{:}\mathsf{emp(E,Sal,D)} \leftarrow \triangleright\mathsf{hyp\_del\_emp(E)},\ \mathsf{emp(E,Sal,D)}.$$

When $\triangleright$hyp_del_emp(E) occurs, employee E is removed from the database and immediately inserted afterwards. If we want to determine if E is an "indispensable" employee, i.e., one whose deletion would result in an unpopulated department, we can use the rule:

$$\textbf{ins}{:}\mathsf{indispensable(E)} \leftarrow (\triangleright\mathsf{hyp\_del\_emp(E)},\ \mathsf{emp(E,Sal,D)}) \otimes \neg\,\mathsf{emp(\_,\_,D)}.$$ □

Finally, we are sufficiently equipped to revisit Example 8.3 in the extended framework. Observe that the problems of the flat approach mentioned at the beginning of Section 8.2 are resolved here:

Example 8.5 (To Hire or Not to Hire, Cont'd)
We specify the hire and checksal transactions by procedures:

```
proc main {
    hire(E,Sal,D) ← ▷hire_if_possible(E,Sal,D) }
proc hire(E,Sal,D) {
    ins:emp(E,Sal,D) ⊗ checksal(D) ← BOT;
    abort← aborted:checksal(D) }
proc checksal(D) {
    abort← avg(D,AvgSal), AvgSal>50000 }
```

[5] The symbol "$\otimes$" is borrowed from [BK94], where it denotes a similar connective.

Whenever ▷hire_if_possible occurs, a subtransaction hire is initiated. At the beginning of transaction (and only then), hire adds the new employee followed by a call to checksal. If checksal aborts then hire also aborts, resulting in an unchanged database. Otherwise hire commits and the insertion is realized. □

The precise expressive power and complexity of Statelog with nested transactions is open. If one restricts to programs whose procedure call graph is acyclic, i.e., where procedures may not call themselves recursively, one can show that rules can be simulated in flat Statelog: In this case, the maximal depth of procedure calls is known at compile-time, so the prefixes occurring in state terms of rules are bounded and can be encoded in the relation name. Moreover, variables occurring in state terms can be added as extra arguments to relations. For example, the goal $[\varepsilon.S_1.\pi(\bar{X}).S_2{+}1.\mu(\bar{Y})]\, p(\bar{Z})$ is encoded as $p\_\pi\_\mu(S_1, S_2{+}1, \bar{X}, \bar{Y}, \bar{Z})$. Since each single transaction corresponds to a computation in flat Statelog, the overall computation can be conceived as a nested fixpoint akin to the nested fixpoints encountered in FO(PFP). As is well-known[6] such nested fixpoint computations can be simulated using a single fixpoint operator only, so similar constructions can be used to flatten Statelog with (non-recursive) procedures.

Related Approaches

The concept of nested transactions in Statelog is related to the transaction model of HiPAC [DBC96]. Using the system-defined Statelog predicates BOT, EOT, and running, rules can be executed at specific points within a transaction.

Early approaches in the database community for grouping rules and adding more procedural control to rule execution are the rule algebras of [IN88] and [KdMS90]. The idea to structure rule sets using procedures or modules has also been introduced in the area of logic programming. For example, [BMPT94] develop a modular design for logic programs including union, intersection, and encapsulation. However, they do not deal with active rules, state change, and transactions. In particular, negation and serial conjunction are not considered.

8.3 Nondeterminism

Nondeterministic extensions for deductive databases are interesting both from a practical and a theoretical perspective: First, many practical queries are naturally expressed as nondeterministic queries. Additionally, nondeterministic extensions can have a significant impact on the expressive power of the language.

The basic operation common to most approaches is the nondeterministic selection of certain tuples: Assume a relation $p(X, Y)$ is given. In order to choose for each X some arbitrary Y_X, we have to obey the functional dependency $X \to Y$. The following example from [GGSZ97] illustrates a simple nondeterministic query:

Example 8.6 (Choosing Advisors) Suppose given two relations student(S,M) holding for every student S her major M, and professor(P,M) storing the majors which a professor teaches. We want to assign every student *some* advisor. This nondeterministic query can be expressed in Datalog using a so-called *choice operator*:

$$\boxed{\mathsf{advisor(S,P)} \leftarrow \mathsf{student(S,M),\ professor(P,M),\ choice((S),(P)).}}$$

[6]... and somewhat tedious to prove; see, e.g., [EF95] for the result in the case of FO(LFP) and FO(PFP).

The goal choice((S),(P))[7] enforces the functional dependency S $\rightarrow$ P, so every student S has only one advisor P. □

There are numerous proposals for extending database languages—particularly Datalog—with nondeterministic constructs. For instance, given a first-order formula $\varphi(\bar{X}, \bar{Y})$, the *witness operator* $W_{\bar{Y}}(\varphi(\bar{X}, \bar{Y}))$ returns for each combination of values $\bar{x}$ for $\bar{X}$ such that $\exists \bar{Y} \varphi(\bar{x}, \bar{Y})$ holds, some "witness" $\bar{y}$ [AV91, AHV95]. Moreover, a whole family of choice operators (*static*, *lazy dynamic*, *eager dynamic*, and *well-founded*) have been proposed [NT89], [SZ90], [CGP93], and [CZ96], respectively. For non-recursive rules like the one in Example 8.6, a static choice operator is sufficient. An elegant declarative semantics for choice goals in recursive rules, i.e., dynamic choice, is obtained by translating a choice program P into a standard logic program P_{sv}, its *stable version*, and then applying the stable model semantics to P_{sv}. The basic idea of the construction of P_{sv} can be seen from the following example:

Example 8.7 (Ordering the Domain)
The following choice program P defines an order on the given domain [GGSZ97]:

P:
```
less(c,c) ← .
less(X,Y) ← less(_,X), adom(Y),
            choice((X),(Y)), choice((Y),(X)).
```

Here, c is a distinguished new constant. The choice goals guarantee that once less(x,y) is derived, no other values x' and y' are derived such that less(x,y') or less(x',y) holds. This declarative semantics of P is obtained by its stable version P_{sv}:

P_{sv}:
```
less(c,c)          ← .
less(X,Y)          ← less(_,X), adom(Y), chosen(X,Y).
chosen(X,Y)        ← less(_,X), adom(Y), ¬ diffchoice(X,Y) .
diffchoice(X,Y)    ← chosen(X,Y'), Y' ≠ Y.
diffchoice(X,Y)    ← chosen(X',Y), X' ≠ X.
```

Note that P_{sv} is not stratified. On the other hand, it is easy to see that P_{sv} has stable models, and that in every stable model the chosen relation satisfies the functional dependency of the corresponding choice goals (these properties hold generally for the stable versions of choice programs [SZ90]). □

Example 8.7 also illustrates the possible impact of nondeterministic extensions on expressive power: since dynamic choice allows to define an order on the underlying domain, S-Datalog with dynamic choice expresses all queries in DB-**PTIME** (and all nondeterministic queries, i.e., NDB-**PTIME**) [CGP93]. Similar results for I-Datalog and P-Datalog extended with a witness operator are presented in [AV91].

It should be clear that also Statelog can be extended with choice operators. Indeed, for the related language XY-Datalog such an extension has been proposed in [GGSZ97] and [BSZ97]. However, nondeterministic Statelog extensions have to be defined "cautiously" in order to preserve the properties of the language: Consider the program

[7] The extra parentheses, customary in the literature, indicate that sets of variables can be used.

P:

```
[S] q(Y) ← [S] p(X,Y), choice((X),(Y)).
```

which may choose in each state $[S]$ a value for Y given the value of X, *independent* of the choices at other states. Thus, the choice models of P are not necessarily periodic. This can also be seen from the stable version

P_{sv}:

```
[S] q(Y)               ← [S] p(X,Y), chosen(X,Y).
[S] chosen(X,Y)        ← [S] p(X,Y), ¬ diffchoice(X,Y).
[S] diffchoice(X,Y)    ← [S] chosen(X,Y'), Y' ≠Y.
```

whose stable models exhibit basically the same behavior as the program from Example 6.5: the stable snapshots at $[n]$ are independent of those at any other state $[m]$. As an alternative to nondeterminism, one can use an order on the underlying domain, which has a similar impact on expressiveness, yet preserves the results established for Statelog.

8.3.1 An Alternative to Nondeterminism: Order

As argued in [AHV95],[8] the impact of an order on the domain and nondeterminism are related: Example 8.7 shows how a nondeterministic extension can be used to compute an order on the domain. Conversely, if a database is ordered but the order is hidden from the user at the conceptual level, then queries which access the order appear as nondeterministic:

Example 8.8 (Choosing Advisors, Cont'd)
The effect of the choice operator in Example 8.6 can be simulated by an order:

```
advisor(S,P) ← student(S,M), professor(P,M), ¬ not_least(S,P).
not_least(S,P) ← student(S,M), professor(P',M), P'<P.
```

The first rule assigns to every student the "least" professor (wrt. some order, say lexicographic) who teaches the student's major. Clearly, if the order "<" is invisible to the user, the answer to the query advisor(S,P) appears nondeterministic. □

Note that an order allows access to individual elements of the domain, so the data independence principle[9] is undermined to some extent, and the advantage of genericity is partially lost. On the other hand, nondeterministic extensions, in principle, allow the same "non-generic" access to individual elements via the introduction of an order as presented in Example 8.7.

Simulation of ST-Datalog

We conclude the discussion of nondeterministic extensions with a sketch of the simulation of ST-Datalog (Datalog$^\neg$ with stable semantics) in Statelog, using an order on the domain. Recall that under possibility semantics (ST$_\exists$-Datalog) and certainty semantics (ST$_\forall$-Datalog), the language expresses DB-**NP** and DB-**coNP**, respectively (Figure 5.2). A nondeterministic semantics *ST$_!$-Datalog* is obtained by returning *some* stable model if it exists (see [Sac97] for

[8] See Chapter 17.4, p. 453: *An Alternative to Order: Nondeterminism.*

[9] The separation of the logical from the physical levels of a database system is usually called the *data independence principle* [AHV95].

details). Since Statelog expressed DB-**PSPACE** on ordered databases, all three ST-Datalog semantics can be simulated in Statelog, provided that an order is available:

Let P be a Datalog$^{\neg}$ program and $\mathcal{D}$ an ordered database over the schema $\mathbf{P}$. An order on the domain allows to enumerate—via intermediate transitions—all possible interpretations over $\mathbf{P}$ and the active domain $adom(P \cup \mathcal{D})$. The Statelog rules needed in this enumeration employ essentially the same technique as the rules of P_{add} in the proof of Theorem 6.25. For each interpretation $\mathcal{I}$ over $\mathbf{P}$ of the enumeration, we have to check whether $\mathcal{I}$ is a stable model of P. Using the operator Γ_P (Sections 5.4 and 5.5), this is the case iff $\Gamma_P(\mathcal{I}) = \mathcal{I}$. It is clear that this test can be expressed in Statelog, since Γ_P corresponds to a simple fixpoint computation (cf. the rules simulating Γ_P in the proof of Theorem 6.22). The ST$_!$-Datalog semantics is obtained by terminating the enumeration as soon as the first stable model is found. Again, this deterministic computation appears nondeterministic if the order is invisible to the user. The ST$_\exists$-Datalog and ST$_\forall$-Datalog semantics are obtained by computing "on the fly" the union and intersection, respectively, of all found stable models. The rules for these operations resemble those in the proof of Theorem 6.33, where the eventually undefined and true atoms are computed in $p_{\boldsymbol{u}}$ and $p_{\boldsymbol{t}}$, respectively.

9 Summary and Conclusions

Paraphrasing Kowalski's famous equation *algorithm = logic + control*, Abiteboul argued in [Abi88] that

$$update = logic + control,$$

i.e., some control features have to be added to a logic-based database language in order to express updates. Alternatively, as shown in the present work, a precise and intuitive logical semantics can be given to the procedural aspects of updates by incorporating an explicit notion of state into the language. The idea of using state terms to refer to different database states allows to exert a limited form of control in a logical way and has come up several times, for example, in [KLS92, Zan93, LL94, KRS95]. In the present work, the impact of adding states to deductive rules is studied in-depth using *Statelog*, a state-oriented extension of Datalog. In a nutshell, we can summarize the results by adapting the above equation to the rule paradigm and state that

$$active\ rules = deductive\ rules + states.$$

Active rules allow to define updates using a state-oriented extension of deductive rules. Due to the possibility of specifying change, a wider range of applications can be handled than by stateless deductive rules. Conversely, active rules can be given a logical semantics by recasting them as extended deductive rules.

The theoretical properties of Statelog are investigated in the first part of the thesis: A Statelog program P with database $\mathcal{D}$ corresponds to a locally stratified logic program and, therefore, has a unique perfect model $\mathcal{M}_{P\cup\mathcal{D}}$, provided that the rules of P are *state-stratified* (i.e., there are no negative cyclic dependencies within a state) and *progressive* (i.e., rules cannot define the current state in terms of future states). The database evolution is modeled by the linear structure of $\mathcal{M}_{P\cup\mathcal{D}} = (\mathcal{M}_{P\cup\mathcal{D}}[0], \mathcal{M}_{P\cup\mathcal{D}}[1], \ldots)$. Since the language is deterministic and involves only a finite domain, this sequence becomes ultimately periodic. In particular, if $\mathcal{M}_{P\cup\mathcal{D}}$ is 1-periodic then P is said to *terminate for* $\mathcal{D}$, and a unique final state $\mathcal{M}_{P\cup\mathcal{D}}[\$]$ is defined.

Using a novel distinction between transition and transaction expressiveness, the expressive power of the language is characterized: Statelog is equivalent to S-DATALOG wrt. transitions, and to WHILE wrt. transactions. This is shown using the normal forms *NF-Statelog*, where each rule is either 1-progressive or local and *P-Statelog*, where only 1-progressive rules are allowed. The expressiveness results for various Statelog normal forms including the related language XY-Datalog [Zan93] are summarized in Figure 9.1. These results imply that generally the evaluation of transitions and transactions is **PTIME**-complete and **PSPACE**-complete, respectively. Moreover, it is undecidable whether a program P terminates for all (or some) $\mathcal{D}$, whereas the problem becomes **PSPACE**-complete if the database $\mathcal{D}$ is given. Interestingly, the test whether P terminates for $\mathcal{D}$ can be expressed within Statelog using similar techniques as

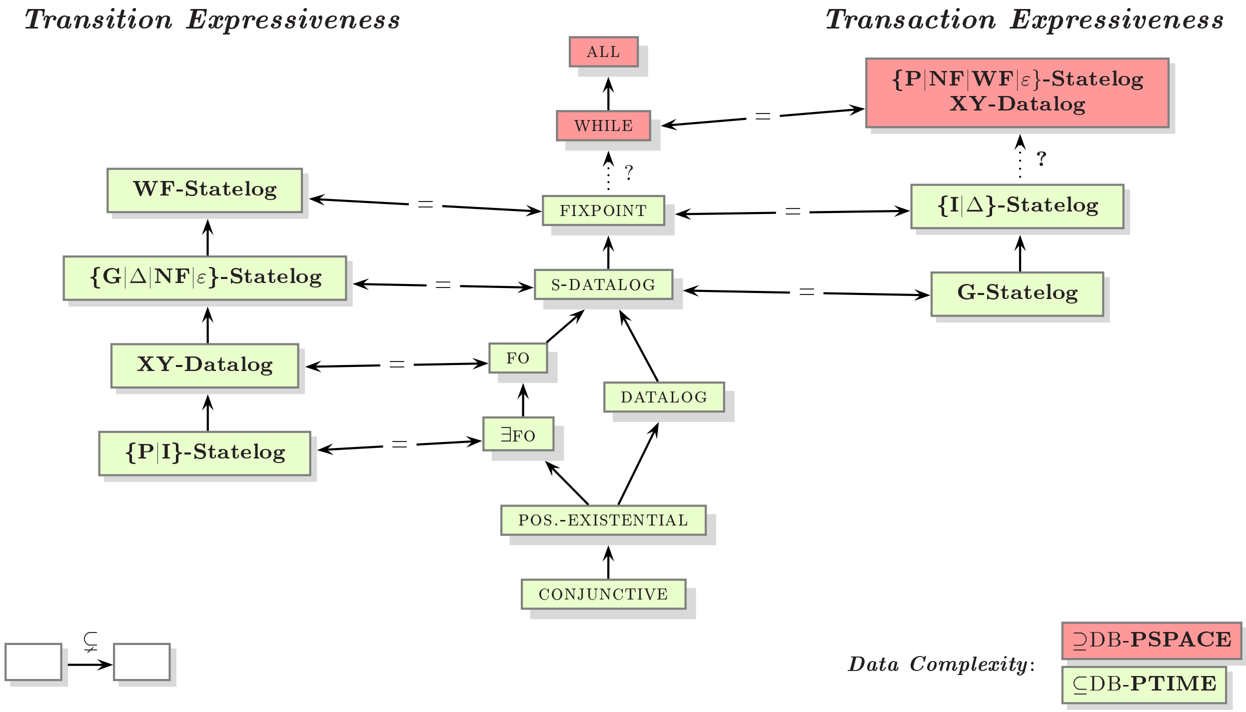


Figure 9.1: Summary: transition expressiveness and transaction expressiveness
({G|Δ|ε}-Statelog = guarded Statelog/Δ-monotonic Statelog/Statelog)

those developed for Datalog [AS91]: For every P, there is a terminating program $P^{\downarrow}$ which decides the termination of $P \cup \mathcal{D}$ for any given database $\mathcal{D}$.

In the second part, a Statelog framework for active rules is developed. With this framework, Statelog not only encompasses deductive and production rules, but also the essential features of active rules: Transitions are specified by user-defined rules using delta relations. Since new update requests can be derived in the head using queries against the current values of deltas in the body, a data-driven reactive programming style is possible. System-defined rules are used to specify the semantics of low-level procedural aspects like the meaning of primitive updates (especially in case of conflicts) and the handling of transaction commit and transaction abort. By restricting user programs to *guarded* or Δ-*monotonic* rules, terminating classes of programs evaluable in **PTIME** are obtained. Integrity constraints, specifically dynamic constraints expressed in past first-order temporal logic (PTL), can be easily incorporated into the framework. The notion of *bounded history encoding*, proposed in [Cho95b], has proven fruitful for distinguishing composite event detection modes which are expressible in Statelog (*unrestricted*, *recent*, and *continuous* parameter contexts) from those which are not expressible in Statelog (*chronicle* context). On the other hand, composite events with chronicle context are expressible if built-ins are used to increase the active domain.

A possible extension of the Statelog framework for modeling procedures and nested transactions is proposed. Such an extension is necessary in order to model more involved coupling modes like detached coupling. However, if too many low-level procedural features are encoded in the language, the main advantages of a logic-based framework—simplicity, elegance, and a declarative programming style—are lost. Hence, there is a trade-off between rich control features and declarativity. Consequently, depending on the intended application area, a suitable balance has to be found. As demonstrated by several examples including those in Appendix

A, Statelog provides a rather high-level declarative programming style, yet enough control to handle practical problems like maintenance of views or referential integrity.

Finally, it should be noted that the basic ideas of Statelog are orthogonal to the underlying data model and, thus, are not restricted to relational databases. Indeed, Statelog ideas have been used in [MSL97, MLL97] to extend the deductive object-oriented database language F-logic [KLW95] by states.

Appendix A

Application Examples

To illustrate database programming in the extended Statelog framework, we present two prototypical application examples for active rules. The first example shows how a rule-based algorithm for incremental view maintenance can be implemented using Statelog. The second example demonstrates the use of Statelog for specifying and executing referential actions.

A.1 Incremental View Maintenance

In [GMS93], an algorithm DRed (*Delete-Rederive*) is proposed, which allows to maintain materialized views involving recursion and stratified negation in a incremental way. We illustrate the basic ideas of the algorithm and its Statelog implementation with a simple example.

Consider the rules P_{tc} in Example 4.2 defining, as the recursive view tc, the transitive closure of a base relation e. Queries against the view can be executed very efficiently if tc is *materialized*. In this case, instead of evaluating the rules of P_{tc} on demand, they are evaluated before the actual query by the user and stored in a base relation, say mtc. The price for the decrease in processing time is an increase in storage. When the base relations on which the view depends are updated, one can either compute the view from scratch using the new values of base relations, or propagate the changes upwards from base relations to view relations. The former is straightforward to implement as shown in Example 7.5. The latter technique is called *incremental view maintenance* and is illustrated using DRed below.

For simplicity of presentation, we restrict to deletions from e only; the handling of insertions by DRed is straightforward (see [GMS93]). Thus, for the program P_{tc}, the input to DRed consists of a set of external delete requests to e and the old value of the materialized view mtc.

As usual, external events initiate updates to base relations:

$$\boxed{[S]\ \textbf{del}{:}\mathsf{e}(\mathsf{X,Y}),\ [S{+}1]\ \mathsf{e}^{-}(\mathsf{X,Y}) \leftarrow [S]\ \triangleright\mathsf{del\_e}(\mathsf{X,Y}).}$$

Here, an auxiliary view $\mathsf{e}^{-}(\mathsf{X,Y})$ is used to remember in state $[n{+}1]$ the values of delete requests issued at $[n]$ (note that the delete requests **del**:e are immediately executed in the transition to $[n{+}1]$ but not propagated to $[n{+}1]$).[1]

DRed consists of two phases. In the *delete phase*, an overestimate of deletions is computed in an auxiliary view $\mathsf{tc}^{-}(\mathsf{X,Y})$. The overestimate, consisting of all tuples having at least one

[1] One can avoid the introduction of state terms in user rules by adding a special connective "⊗" for serial conjunction; see Section 8.2.4.

derivation involving a deleted edge, is computed in a kind of semi-naive evaluation starting from e^- and mtc (cf. [GMS93] and [AHV95]):

$$\begin{array}{l}\mathsf{tc}^-(\mathsf{X,Y}) \leftarrow \mathsf{e}^-(\mathsf{X,Y}).\\ \mathsf{tc}^-(\mathsf{X,Y}) \leftarrow \mathsf{e}^-(\mathsf{X,Z}),\ \mathsf{mtc}(\mathsf{Z,Y}).\\ \mathsf{tc}^-(\mathsf{X,Y}) \leftarrow \mathsf{e}(\mathsf{X,Z}),\ \mathsf{tc}^-(\mathsf{Z,Y}).\\ \mathsf{tc1}(\mathsf{X,Y}) \leftarrow \mathsf{mtc}(\mathsf{X,Y}),\ \neg\,\mathsf{tc}^-(\mathsf{X,Y}).\end{array}$$

At the end of the first phase, using the overestimate of deletions, an underestimate of atoms remaining in the view is computed in tc1.

In the *rederive phase*, tuples for which an alternative derivation exists are rederived in another temporary view $\mathsf{tc}^+(\mathsf{X,Y})$:

$$\begin{array}{l}\mathsf{tc}^+(\mathsf{X,Y}) \leftarrow \mathsf{tc}^-(\mathsf{X,Y}),\ \mathsf{e}(\mathsf{X,Y}).\\ \mathsf{tc}^+(\mathsf{X,Y}) \leftarrow \mathsf{tc}^-(\mathsf{X,Y}),\ \mathsf{e}(\mathsf{X,Z}),\ \mathsf{tc1}(\mathsf{Z,Y}).\\ \mathsf{tc}^+(\mathsf{X,Y}) \leftarrow \mathsf{tc}^-(\mathsf{X,Y}),\ \mathsf{e}(\mathsf{X,Z}),\ \mathsf{tc}^+(\mathsf{Z,Y}).\end{array}$$

Observe how the goal $\mathsf{tc}^-(\mathsf{X,Y})$ acts as a filter and limits the possible rederivations. Finally, the materialized view mtc has to be updated by removing all tuples in $\mathsf{tc}^-(\mathsf{X,Y})$ and then adding those in $\mathsf{tc}^+(\mathsf{X,Y})$. This can be accomplished in a single transition, if we assume that insertions have higher priority, i.e., policy INS is used:

$$\begin{array}{l}\textbf{del}{:}\mathsf{mtc}(\mathsf{X,Y}) \leftarrow \mathsf{tc}^-(\mathsf{X,Y}).\\ \textbf{ins}{:}\mathsf{mtc}(\mathsf{X,Y}) \leftarrow \mathsf{tc}^+(\mathsf{X,Y}).\end{array}$$

A.2 Referential Integrity Maintenance

The basic idea behind *referential integrity constraints* (*ric*'s) is to ensure the existence of referenced tuples. In order to automatically maintain referential integrity in the presence of updates, specialized triggers, so-called *referential actions* (*rac*'s) have been proposed. Due to their practical importance, *rac*'s have been included in the SQL2 standard and SQL3 proposal [ISO92, ISO97]. However, the standards describe the meaning of *rac*'s in a lengthy and procedural way, making it difficult to understand or predict their global behavior. The problem of ambiguous global semantics is "solved" by fixing a rather ad-hoc run-time execution model. In contrast, in [LMR96, LML97, LM98] it is shown how the meaning of *rac*'s can be precisely specified using deductive rules. Below, we describe the main ideas of handling *rac*'s with Statelog. A detailed treatment including formal results can be found in [LML97, LML97].

Notation. In the sequel, R denotes a relation name; the first-order atom $R(\bar{X})$ identifies a tuple $\bar{X}$ of R, provided the variables $\bar{X}$ are appropriately bound. By $\vec{A} = (i_1, \ldots, i_k)$ we denote a vector of k distinct attributes (usually $\vec{A}$ will be some key). The *projection* of tuples $\bar{X}$ to an attribute vector $\vec{A}$ is denoted by $\bar{X}\langle\vec{A}\rangle$. For example, if $\bar{x} = (a, b, c)$, $\vec{A} = (1, 3)$, then $\bar{x}\langle\vec{A}\rangle = (a, c)$.

Referential Integrity Constraints. A *referential integrity constraint* (*ric*) is an expression of the form

$$R_C.\vec{F} \to R_P.\vec{K} \ ,$$

where $\vec{F}$ is a *foreign key* of the *child relation* R_C, referencing a candidate key $\vec{K}$ of the *parent relation* R_P. A *ric* $R_C.\vec{F} \to R_P.\vec{K}$ is *satisfied* by a given database $\mathcal{D}$, if for every child tuple $R_C(\bar{x})$ with foreign key values $\bar{x}\langle\vec{F}\rangle$, there exists a tuple $R_P(\bar{y})$ with matching key value, i.e., $\bar{x}\langle\vec{F}\rangle = \bar{y}\langle\vec{K}\rangle$. Thus, for a database instance $\mathcal{D}$, a *ric* is satisfied if $\mathcal{D} \models \varphi_{ric}$, where φ_{ric} is the first-order sentence

$$\forall \bar{X} \ (\ R_C(\bar{X}) \to \exists \bar{Y} \ (R_P(\bar{Y}) \wedge \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle) \ ) \ . \qquad (\varphi_{ric})$$

Note that φ_{ric} is a static constraint in implication form (see Section 7.6.1).

Referential Actions. The basic idea of maintaining *ric*'s by *referential actions* (*rac*'s) is to issue some "local repair" in case of updates to R_P or R_C. The following local strategies can be used for *rac*'s:

- `CASCADE`: propagate the update from the parent to the child.
- `RESTRICT`: (i) reject an update on the *parent* if there exists a child referencing it in the *current* database state, or (ii) reject an update on the *child* if there is no tuple with the respective parent key in the *current* database state.
- `NO ACTION`: similar to `RESTRICT`, but look at the database state *after* (hypothetically) applying all updates.

In SQL, referential actions for a *ric* $R_C.\vec{F} \to R_P.\vec{K}$ are specified with the definition of the child relation:

```
{CREATE|ALTER} TABLE R_C
    ...
    FOREIGN KEY F REFERENCES R_P K
    [ON UPDATE {CASCADE|RESTRICT|SET NULL|SET DEFAULT|NO ACTION}]
    [ON DELETE {CASCADE|RESTRICT|SET NULL|SET DEFAULT|NO ACTION}]
    ...
```

In the sequel, we associate with every *ric* $R_C.\vec{F} \to R_P.\vec{K}$ a corresponding *rac* of the form $R_C.\vec{F} \to R_P.\vec{K}$ `ON DELETE {CASCADE|RESTRICT|NO ACTION}`. Insertions can be handled in a straightforward way by rejecting updates which aim to insert a child tuple whose corresponding parent does not exist (this is also the SQL strategy) [LMR96]. On the other hand, we deliberately chose to exclude modifications of the form `ON UPDATE ...` since these can lead to intractable computations [LML97].

The Problem of Ambiguity

Since *rac*'s only specify some local behavior, there may be ambiguities wrt. the global semantics, leading to different final states. Given a database $\mathcal{D}$ and a set of user delete requests $U_\triangleright$, a set of *rac*'s *RA* is called *ambiguous wrt. $\mathcal{D}$ and $U_\triangleright$*, if there are different final states

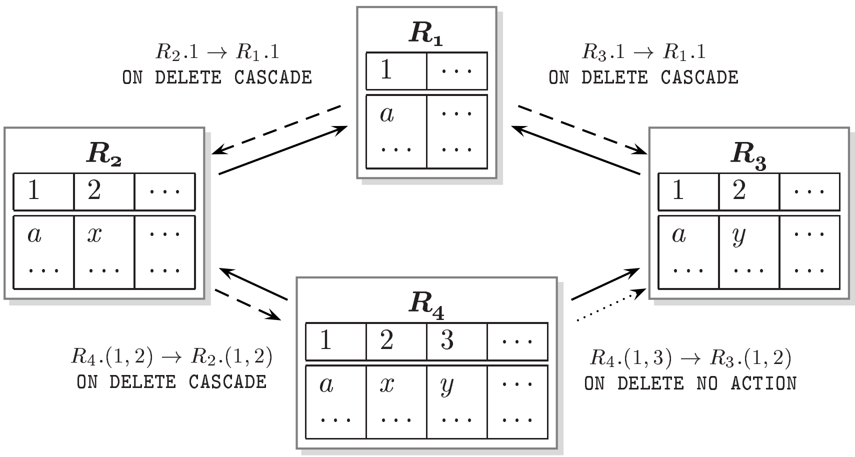


Figure A.1: Database with referential actions

$\mathcal{D}'$ (depending on the execution order of referential actions). A relational database *schema* $\mathbf{R}$ with *rac*'s RA is ambiguous, if RA is ambiguous wrt. *some* database $\mathcal{D}$ over $\mathbf{R}$ and *some* $U_\triangleright$. As shown in [Rei96a] it is in general undecidable, whether a schema with *rac*'s is ambiguous (if $\mathcal{D}$ and $U_\triangleright$ are given, the problem becomes decidable). The following example from [Rei96a] illustrates the problem of ambiguity:

Example A.1 (Ambiguity) Consider the database with *rac*'s depicted in Figure A.1. Solid arcs represent *ric*'s and point from R_C to R_P, *rac*'s are denoted by dashed (`CASCADE`) or dotted (`NO ACTION`) arcs. Let $U_\triangleright = \{\triangleright\textbf{del}{:}R_1(a)\}$ be a user request to delete the tuple $R_1(a)$. Depending on the order of execution of *rac*'s, one of two different final states may be reached:

(1) If execution follows the path R_1–R_3–R_4, the tuple $R_3(a, y)$ cannot be deleted: Since $R_4(a, x, y)$ references $R_3(a, y)$, the *rac* for R_4 restricts the deletion of $R_3(a, y)$. This in turn blocks the deletion of $R_1(a)$. Thus, the user request $\triangleright\textbf{del}{:}R_1(a)$ is rejected, and the database remains unchanged, i.e., $\mathcal{D}' = \mathcal{D}$.

(2) If execution follows the path R_1–R_2–R_4, the tuples $R_2(a, x)$ and $R_4(a, x, y)$ are requested for deletion. Hence, the *rac* for $R_4.(1, 3) \to R_3.(1, 2)$ can assume that $R_4(a, x, y)$ is deleted, thus no referencing tuple exists in R_4. Therefore, all deletions can be executed, resulting in a new database state $\mathcal{D}' \neq \mathcal{D}$. □

In general, (2) is preferable to (1), since it accomplishes the desired user request $U_\triangleright$ without violating referential integrity. Note that an ambiguity also occurs if the *rac* `ON DELETE NO ACTION` is replaced by `ON DELETE RESTRICT`. In this case, it is straightforward to obtain a unique semantics by requiring that `RESTRICT` *rac*'s are always evaluated wrt. the original database state. Hence, the only admissible result for the above example is (1).

In contrast, *rac*'s of type `NO ACTION` examine the database state as it would be *after* execution of all updates. Since the final state depends on the updates to be executed, and these may in turn depend on the final state (via `NO ACTION` triggers), there is a cyclic dependency. Such (possibly negative) cyclic dependencies can be handled by specifying a set RA of *rac*'s as a logic program P_{RA} and then applying the declarative semantics of P_{RA}:

Referential Actions as Logical Rules

Initially, a user delete request $\triangleright\textsf{\textbf{del}}{:}R(\bar{x}) \in U_\triangleright$ creates an internal delete request $\textsf{\textbf{req\_del}}{:}R(\bar{x})$, provided there is no blocking $\textsf{\textbf{blk\_del}}{:}R(\bar{x})$:

$$\textsf{\textbf{req\_del}}{:}R(\bar{X}) \leftarrow \ \triangleright\textsf{\textbf{del}}{:}R(\bar{X}), \neg\, \textsf{\textbf{blk\_del}}{:}R(\bar{X}). \qquad (I)$$

Referential actions of RA are formalized as follows:

- $R_C.\vec{F} \to R_P.\vec{K}$ `ON DELETE CASCADE` is specified by two rules: the first one propagates internal delete requests downwards from the parent to the child:

  $$\textsf{\textbf{req\_del}}{:}R_C(\bar{X}) \leftarrow \ \textsf{\textbf{req\_del}}{:}R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle. \qquad (DC_1)$$

  Additionally, blockings are propagated upwards, i.e., when the deletion of a child is blocked, the deletion of the referenced parent is also blocked:

  $$\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow \ R_P(\bar{Y}),\ \textsf{\textbf{blk\_del}}{:}R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle. \qquad (DC_2)$$

- $R_C.\vec{F} \to R_P.\vec{K}$ `ON DELETE RESTRICT` blocks the deletion of a parent tuple if there is a corresponding child tuple in the original state:

  $$\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow \ R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle. \qquad (DR)$$

- $R_C.\vec{F} \to R_P.\vec{K}$ `ON DELETE NO ACTION` blocks the deletion of a parent tuple if there is a corresponding child tuple which is not requested for deletion:

  $$\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow \ R_P(\bar{Y}),\ R_C(\bar{X}),\ \neg\,\textsf{\textbf{req\_del}}{:}R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle\ . \qquad (DN)$$

Note that P_{RA} is generally not stratified due to the negative cyclic dependency

$$\textsf{req\_del} \overset{\neg}{\rightsquigarrow} \textsf{blk\_del} \overset{\neg}{\rightsquigarrow} \textsf{req\_del}. \qquad (\neg)$$

It can be shown that the well-founded model of P_{RA} provides a skeptical semantics of *rac*'s and assigns the truth-value *undefined* to all controversial update requests [LMR96], whereas certain partial stable models characterize all maximal admissible solutions [LML97]. Alternatively, if modifications are not allowed, a unique semantics can be obtained by recasting P_{RA} in Statelog and then exploiting state-stratification [LM98]:

Formalization in Statelog

The deductive rules of P_{RA} can be conceived as (local) Statelog user rules, i.e., where the state term "$[S]$" has been omitted. However, due to the negative dependency $(\neg)$ above, P_{RA} is not state-stratified. This negative cycle can be "cut" by delaying the derivation of newly computed blockings $\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y})$ to the next state. The resulting program P^S_{RA} is state-stratified and, as shown in [LM98], accomplishes the desired maximal set of updates:

$$[S]\,\textsf{\textbf{req\_del}}{:}R(\bar{X}) \leftarrow\ [S]\,\triangleright\textsf{\textbf{del}}{:}R(\bar{X}),\ \neg\,\textsf{\textbf{blk\_del}}{:}R(\bar{X}). \qquad (I^S)$$

$$[S]\,\textsf{\textbf{req\_del}}{:}R_C(\bar{X}) \leftarrow\ [S]\,R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle,\ \textsf{\textbf{req\_del}}{:}R_P(\bar{Y}). \qquad (DC_1^S)$$

$$[S]\,\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow\ [S]\,R_P(\bar{Y}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle,\ \textsf{\textbf{blk\_del}}{:}R_C(\bar{X}). \qquad (DC_2^S)$$

$$[S]\,\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow\ [S]\,R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle. \qquad (DR^S)$$

$$[S{+}1]\,\textsf{\textbf{blk\_del}}{:}R_P(\bar{Y}) \leftarrow\ [S]\,R_P(\bar{Y}),\ R_C(\bar{X}),\ \bar{X}\langle\vec{F}\rangle = \bar{Y}\langle\vec{K}\rangle,\ \neg\,\textsf{\textbf{req\_del}}{:}R_C(\bar{X}). \qquad (DN^S)$$

$$[S{+}1]\,\textsf{running} \leftarrow\ [S{+}1]\,\textsf{\textbf{blk\_del}}{:}R(\bar{X}),\ [S]\,\neg\,\textsf{\textbf{blk\_del}}{:}R(\bar{X}). \qquad (R^S)$$

Here, we tacitly assume that running is true initially, and that delete requests $\triangleright$**del**:$R(\bar{X})$ and base relations $R_C(\bar{X})$, $R_P(\bar{Y})$ are propagated to subsequent states while running holds.

In [LM98], it is shown how a procedural implementation can be obtained from the state-stratification of this Statelog formalization. The steps of this algorithm correspond to the rules as follows: Initially, it is assumed that there are only those blockings which result from `ON DELETE RESTRICT` triggers (DR^S). Then, blockings are propagated upwards the `ON DELETE CASCADE` chains to parent tuples (DC_2^S), finally blocking the user requests (I^S). For the remaining unblocked user requests, the cascaded requests are propagated downwards (DC_1^S). Thus, some more tuples will remain in the database, which now could block other requests. In the *next* state, all blockings are computed which are caused by `ON DELETE NO ACTION` triggers from tuples which are not reachable via cascaded deletions (DN^S). The whole transaction keeps running until there are no more new blockings (R^S). When termination is detected, the final set of delete requests is executed via delta relations:

$$[S{+}1]\,\textsf{\textbf{del}}{:}R(\bar{X}) \leftarrow\ [S]\,\textsf{running},\ [S{+}1]\,\neg\,\textsf{running},\ \textsf{\textbf{req\_del}}{:}R(\bar{X}).$$

Bibliography

[AB94] K. R. Apt and R. N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[Abi88] S. Abiteboul. Updates, a New Frontier. In *Intl. Conference on Database Theory (ICDT)*, number 326 in LNCS, pp. 1–18. Springer, 1988.

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, 1988.

[ACL91] R. Agrawal, R. Cochrane, and B. G. Lindsay. On Maintaining Priorities in a Production Rule System. In Lohman et al. [LSC91], pp. 479–487.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[AK97] F. Afrati and P. Kolaitis, editors. *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, Delphi, Greece, 1997. Springer.

[AKLS95] M. H. F. Alves, S. Kekkonen, D. Laurent, and N. Spyratos. Alternating Fixpoint and Extended Datalog$^\neg$ Databases. Technical Report 1004, Université d'Orléans, LIFO, October 1995.

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1993.

[AS91] S. Abiteboul and E. Simon. Fundamental Properties of Deterministic and Nondeterministic Extensions of Datalog. *Theoretical Computer Science*, 78(1):137–158, 1991.

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *ACM Symposium on Principles of Database Systems (PODS)*, 1988.

[AV91] S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[AV95] S. Abiteboul and V. Vianu. Computing with First-Order Logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In M. Stonebraker, editor, *ACM Intl. Conference on Management of Data (SIGMOD)*, San Diego, California, 1992.

[AWH95] A. Aiken, J. Widom, and J. M. Hellerstein. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Transactions on Database Systems (TODS)*, 20(1):3–41, March 1995.

[Bau89] M. Baudinet. Temporal Logic Programming is Complete and Expressive. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 267–280, Austin, Texas, 1989.

[Bau95] M. Baudinet. On the Expressiveness of Temporal Logic Programming. *Information and Computation*, 117(2):157–180, 1995.

[Bay93] P. Bayer. State-of-the-Art Report on Reactive Processing in Databases and Artificial Intelligence. *The Knowledge Engineering Review*, 8(2):145–171, 1993.

[BBKZ93] H. Branding, A. P. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In Paton and Williams [PW93], pp. 111–126.

[BCP95a] E. Baralis, S. Ceri, and S. Paraboschi. Improving Rule Analysis by Means of Triggering and Activation Graphs. In Sellis [Sel95], pp. 165–181.

[BCP95b] E. Baralis, S. Ceri, and S. Paraboschi. Run-Time Detection of Non-Terminating Active Rule Systems. In Ling et al. [LMV95].

[BCW93a] E. Baralis, S. Ceri, and J. Widom. Better Termination Analysis for Active Databases. In Paton and Williams [PW93], pp. 163–179.

[BCW93b] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [TCG+93], pp. 294–320.

[BD97] S. Brass and J. Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.

[BF91] N. Bidoit and C. Froidevaux. General Logical Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation*, 91(1):15–54, 1991.

[BFKM85] L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[BFP+95] M. L. Barja, A. A. A. Fernandes, N. W. Paton, M. H. Williams, A. Dinn, and A. I. Abdelmoty. Design and implementation of ROCK & ROLL: a deductive object-oriented database system. *Information Systems*, 20(3):185–211, 1995.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.

[BGP97] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypotheses. *Journal of Logic Programming*, 31(1–3):201–243, 1997.

[BH95] M. Berndtsson and J. Hansson, editors. *1st Intl. Workshop on Active and Real-Time Database Systems (ARTDB)*, Workshops in Computing, Skövde, 1995. Springer.

[Bid91] N. Bidoit. Negation in Rule-Based Database Languages: a Survey. *Theoretical Computer Science*, 78, 1991.

[BK94] A. J. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[BK96] A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Joint International Conference and Symposium on Logic Programming (JICSLP)*, pp. 142–156, Bonn, Germany, 1996. MIT Press.

[BL96a] C. Baral and J. Lobo. Characterizing Production Systems using Logic Programming and Situation Calculus. http://cs.utep.edu/chitta/papers/char-prod-systems.ps, 1996.

[BL96b] C. Baral and J. Lobo. Formal Characterization of Active Databases. In Pedreschi and Zaniolo [PZ96], pp. 175–195.

[BLT97] C. Baral, J. Lobo, and G. Trajcevski. Formal Characterization of Active Databases: Part II. In Bry et al. [BRR97], pp. 247–264.

[BM97] N. Bidoit and S. Maabout. A Model Theoretic Approach to Update Rule Programs. In Afrati and Kolaitis [AK97], pp. 173–187.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, July 1994.

[BRR97] F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors. *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, Montreux, Switzerland, 1997. Springer.

[Bry90] F. Bry. Intensional Updates : Abduction via Deduction. In *Intl. Conference on Logic Programming (ICLP)*, 1990.

[BSZ97] A. Brogi, V. Subrahmanian, and C. Zaniolo. The logic of totally and partially ordered plans: a deductive database approach. *Annals of Mathematics and Artificial Intelligence*, 19(I-II):27–58, 1997.

[Buc94] A. Buchmann. Active Object Systems. In A. Dogac, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, NATO ASI Series. Springer, 1994.

[BW94] E. Baralis and J. Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 475–486, Santiago, Chile, 1994.

[CB94] P. Cholak and H. A. Blair. The Complexity of Local Stratification. *Fundamenta Informaticae*, 21(4), 1994.

[CCS94] C. Collet, T. Coupaye, and T. Svensen. NAOS – Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 132–143, Santiago, Chile, 1994.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.

[CFPT96] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active Rule Management in Chimera. In Widom and Ceri [WC96a], chapter 6, pp. 151–176.

[CGP93] L. Corciulo, F. Giannotti, and D. Pedreschi. Datalog with Non-Deterministic Choice Computes NDB-PTIME. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 760 in LNCS, pp. 49–66. Springer, 1993.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

[CH85] A. Chandra and D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, 1:1–15, 1985.

[Cha92] S. Chakravarthy, editor. *Bulletin of the Technical Committee on Data Engineering: Special Issue on Active Databases*, volume 15(1–4). IEEE Computer Society, 1992.

[Che95] W. Chen. Programming with Logical Queries, Bulk Updates and Hypothetical Reasoning. In B. Thalheim, editor, *Workshop Semantics in Databases*, Prague, January 1995. Technische Universität Cottbus.

[Cho90] J. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols*. PhD thesis, Rutgers University, 1990.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. In H. J. Ohlbach and D. M. Gabbay, editors, *Proc. 1st Intl. Conference on Temporal Logic*, number 827 in LNAI, pp. 506–534. Springer, 1994.

[Cho95a] J. Chomicki. Depth-Bounded Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 25(1):1–31, October 1995.

[Cho95b] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[Cho97] J. Chomicki. personal communication. 1997.

[CI88] J. Chomicki and T. Imieliński. Temporal Deductive Databases and Infinite Objects. In *ACM Symposium on Principles of Database Systems (PODS)*, 1988.

[CI93] J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems (TODS)*, 18(2), June 1993.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 606–617, Santiago de Chile, 1994.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14:1–26, 1994.

[Cod70] E. Codd. A Relational Model For Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Coh89] D. Cohen. Compiling Complex Database Transition Triggers. In J. Clifford, B. Lindsay, and D. Maier, editors, *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 225–234, Portland, 1989.

[CT95] J. Chomicki and D. Toman. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):566–582, 1995.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 566–577, 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In Lohman et al. [LSC91], pp. 577–589.

[CW93] W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *ACM Symposium on Principles of Database Systems (PODS)*, 1993.

[CW96] S. Ceri and J. Widom. Standards and Commercial Systems. In Widom and Ceri [WC96a], chapter 9.

[CZ96] W. Chen and J. Zeng. Nondeterminism through Well-Founded Choice. *Journal of Logic Programming*, 26(3):285–309, 1996.

[Day95] U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? In Berndtsson and Hansson [BH95], pp. 3–22.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining Active Databases and Timing Constraints. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 51–70, March 1988.

[DBC96] U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC Project. In Widom and Ceri [WC96a], chapter 7, pp. 177–206.

[DGG95] K. R. Dittrich, S. Gatziu, and A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In Sellis [Sel95], pp. 3–20.

[DHW95] U. Dayal, E. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 21, pp. 434–456. ACM Press, 1995.

[Dix95] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. In A. Fuhrmann and H. Rott, editors, *Logic, Action and Information.* de Gruyter, 1995.

[DJP93] O. Díaz, A. Jaime, and N. W. Paton. DEAR: a DEbugger for Active Rules in an object-oriented context. In Paton and Williams [PW93], pp. 180–193.

[dMS88a] C. de Maindreville and E. Simon. Modelling a Production Rule Language for Deductive Databases. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1988.

[dMS88b] C. de Maindreville and E. Simon. A Production Rule Based Approach to Deductive Databases. In *Intl. Conference on Data Engineering (ICDE)*, 1988.

[Don88] G. Dong. On the Composition and Decomposition of Datalog Program Mappings. In M. Gyssens, J. Paredaens, and D. V. Gucht, editors, *2nd Intl. Conference on Database Theory (ICDT)*, number 470 in LNCS, pp. 87–101, Bruges, Belgium, 1988. Springer.

[DP69] R. A. Di Paola. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *Journal of the ACM*, 16(2):324–327, 1969.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer, 1995.

[ELS96] T. Eiter, N. Leone, and D. Saccà. The Expressive Power of Partial Models for Disjunctive Deductive Databases. In Pedreschi and Zaniolo [PZ96].

[Esw76] K. P. Eswaran. Specification, Implementation and Interactions of a Trigger Subsystem in a Integrated Database System. IBM Research Report RJ-1820, IBM Almaden Research Center, IBM San Jose Research Laboratory, California, 1976.

[FKL97] J. Flum, M. Kubierschky, and B. Ludäscher. Total and Partial Well-Founded Datalog Coincide. In Afrati and Kolaitis [AK97], pp. 113–124.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems (TODS)*, 20(4):414–471, 1995.

[FWP97] A. A. A. Fernandes, M. H. Williams, and N. W. Paton. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205–244, 1997.

[GB97] A. Geppert and M. Berndtsson, editors. *3rd Intl. Workshop on Rules in Database Systems (RIDS)*, number 1312 in LNCS, Skövde, Sweden, 1997.

[GD93] S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In Paton and Williams [PW93], pp. 23–39.

[GGSZ97] F. Giannotti, S. Greco, D. Saccà, and C. Zaniolo. Programming with Non-Determinism in Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(I–II):97–125, 1997.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, J. Castillo, M. Escobar-Molano, S.-H. Lu, J. Luo, C. Tsang, and G. Zhou. On Implementing a Language for Specifying Active Database Execution Models. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating Deltas to be First-Class Citizens in a Database Programming Language. *ACM Transactions on Database Systems (TODS)*, 21(3):370–426, 1996.

[GJS92] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 81–90, 1992.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Intl. Conference on Logic Programming (ICLP)*, pp. 1070–1080, 1988.

[GL93] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[GMS92] G. Gottlob, G. Moerkotte, and V. S. Subrahmanian. The PARK Semantics for Active Rules. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Intl. Conference on Extending Database Technology*, number 1057 in LNCS, Avignon, France, 1992. Springer.

[GMS93] A. Gupta, I. S. Mumick, and V. Subrahmanian. Maintaining Views Incrementally. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 157–166, Washington, D.C., 1993.

[Ham95] U. Hamann. Ein System zur Beschreibung und Ausführung von Änderungsoperationen in einer zustandsorientierten Erweiterung von Datalog. Master's thesis, Institut für Informatik, Universität Freiburg, 1995.

[Han92] E. N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1992.

[Han96] E. N. Hanson. The Ariel Project. In Widom and Ceri [WC96a], chapter 3.

[HLM88] M. Hsu, R. Ladin, and D. R. McCarthy. An Execution Model for Active Data Base Management Systems. In C. Beeri, J. W. Schmidt, and U. Dayal, editors, *3rd Intl. Conference on Data and Knowledge Bases*, pp. 171–179, Jerusalem, Israel, 1988.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[HW93] E. N. Hanson and J. Widom. An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121–143, 1993.

[IN88] T. Imielinski and S. Naqvi. Explicit Control of Logic Programs Through Rule Algebra. In *ACM Symposium on Principles of Database Systems (PODS)*, March 1988.

[ISO92] ISO/IEC JTC1/SC21. Information Technology – Database Languages – SQL2, July 1992. ANSI, 1430 Broadway, New York, NY 10018.

[ISO97] ISO-ANSI Working draft. SQL3, April 1997. ISO/IEC JTC 1/SC 21/WG 3.

[Kan90] P. C. Kanellakis. Elements of Relational Database Theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[KdMS90] G. Kiernan, C. de Maindreville, and E. Simon. Making Deductive Database a Practical Technology: a step forward. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 237–246, 1990.

[KKR95] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1):26–52, 1995.

[KLS92] M. Kramer, G. Lausen, and G. Saake. Updates in a Rule-Based Language for Objects. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 251–262, Vancouver, 1992.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[KM90] A. Kakas and P. Mancarella. Database Updates Through Abduction. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1990.

[KM91] H. Katsuno and A. Mendelzon. On the Difference between Updating a Knowledge Base and Revising It. In *Proc. 2nd Intl. Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 387–394, Cambridge, MA, USA, 1991. Morgan Kaufmann.

[Kol91] P. G. Kolaitis. The Expressive Power of Stratified Programs. *Information and Computation*, 90(1):50–66, 1991.

[Kow92] R. A. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12(1&2):121–146, 1992.

[KP88] P. Kolaitis and C. Papadimitriou. Why not Negation by Fixpoint? In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 231–239, 1988.

[KRS95] D. B. Kemp, K. Ramamohanarao, and P. J. Stuckey. ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS. In Ling et al. [LMV95], pp. 91–108.

[KU96] A. P. Karadimce and S. D. Urban. Refined Triggering Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-oriented Database. In *12th International Conference on Data Engineering (ICDE)*, pp. 384–391, 1996.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A Logical Framework for Active Rules. In *Proc. 7th Intl. Conference on Management of Data (COMAD)*, pp. 221–238, Pune, India, 1995. Tata McGraw-Hill.

[LL94] G. Lausen and B. Ludäscher. Updates by Reasoning about States. In J. Eder and L. Kalinichenko, editors, *2nd Intl. East-West Database Workshop*, Workshops in Computing, pp. 17–30, Klagenfurt, Austria, 1994. Springer.

[LL98] B. Ludäscher and G. Lausen. Handling Termination in a Logical Language for Active Rules. *Informatica*, 1998. Lithuanian Academy of Sciences. Special issue on the *7th Advanced Course on Artificial Intelligence (ACAI'97)*, Vilnius. to appear.

[LLM98a] G. Lausen, B. Ludäscher, and W. May. On Active Deductive Databases: The Statelog Approach. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, number 1472 in LNCS. Springer, 1998. to appear.

[LLM98b] G. Lausen, B. Ludäscher, and W. May. On Logical Foundations of Active Databases. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 12, pp. 389–422. Kluwer Academic Publishers, 1998.

[LM98] B. Ludäscher and W. May. Referential Actions: From Logical Semantics to Implementation. In *6th Intl. Conference on Extending Database Technology (EDBT)*, number 1377 in LNCS, pp. 404–418, Valencia, Spain, 1998. Springer.

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested Transactions in a Logical Language for Active Rules. In Pedreschi and Zaniolo [PZ96], pp. 196–222.

[LML97] B. Ludäscher, W. May, and G. Lausen. Referential Actions as Logical Rules. In *Proc. 16th ACM Symposium on Principles of Database Systems (PODS)*, pp. 217–227, Tucson, Arizona, 1997.

[LMR96] B. Ludäscher, W. May, and J. Reinert. Towards a Logical Semantics for Referential Actions in SQL. In *Proc. 6th Intl. Workshop on Foundations of Models and Languages for Data and Objects: Integrity in Databases*, pp. 57–72, Dagstuhl, Germany, 1996. Universität Magdeburg, Fakultät für Informatik, Preprint Nr. 4.

[LMV95] T. W. Ling, A. O. Mendelzon, and L. Vieille, editors. *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1013 in LNCS, Singapore, 1995. Springer.

[LSC91] G. M. Lohman, A. Sernadas, and R. Camps, editors. *Intl. Conference on Very Large Data Bases (VLDB)*, Barcelona, 1991.

[McD82] J. McDermott. A Rule-Based Configurer of Computer Systems. *AI*, 19(1):39–88, 1982.

[MH69] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pp. 463–502. Edinburgh University Press, 1969.

[Min96] J. Minker. Logic and Databases: a 20 Year Retrospective. In Pedreschi and Zaniolo [PZ96], pp. 3–57.

[MLL97] W. May, B. Ludäscher, and G. Lausen. Well-Founded Semantics for Deductive Object-Oriented Database Languages. In Bry et al. [BRR97], pp. 320–336.

[MSL97] W. May, C. Schlepphorst, and G. Lausen. Integrating Dynamic Aspects into Deductive Object-Oriented Databases. In Geppert and Berndtsson [GB97].

[MT91] V. W. Marek and M. Truszczynski. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[MW88] S. Manchanda and D. S. Warren. A Logic-Based Language for Database Updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[MZ95] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In Ling et al. [LMV95], pp. 19–37.

[NK88] S. Naqvi and R. Krishnamurthy. Database Updates in Logic Programming. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 251–262, 1988.

[NT89] S. Naqvi and S. Tsur. *A logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[Pap94] C. M. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PCFW95] N. W. Paton, J. Campin, A. A. A. Fernandes, and M. H. Williams. Formal Specification of Active Database Functionality: A Survey. In Sellis [Sel95], pp. 21–37.

[PDW+93] N. W. Paton, O. Díaz, M. H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of Active Behaviour. In Paton and Williams [PW93], pp. 40–57.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann, 1988.

[Prz89] T. C. Przymusinski. Every Logic Program has a natural Stratification and an Iterated Least Fixed Point Model. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 11–21, 1989.

[Prz90] T. C. Przymusinski. The Well-Founded Semantics Coincides With The Three-Valued Stable Semantics. *Fundamenta Informaticae*, 13(4):445–464, 1990. special issue on Non-Monotonic Reasoning.

[PV97] P. Picouet and V. Vianu. Expressiveness and Complexity of Active Databases. In Afrati and Kolaitis [AK97], pp. 155–172.

[PW93] N. W. Paton and M. H. Williams, editors. *1st Intl. Workshop on Rules in Database Systems (RIDS)*, Workshops in Computing, Edinburgh, Scotland, 1993. Springer.

[PZ96] D. Pedreschi and C. Zaniolo, editors. *Intl. Workshop on Logic in Databases (LID)*, number 1154 in LNCS, San Miniato, Italy, 1996. Springer.

[Rei96a] J. Reinert. Ambiguity for Referential Integrity is Undecidable. In G. Kuper and M. Wallace, editors, *Constraint Databases and Applications*, number 1034 in LNCS, pp. 132–147. Springer, 1996.

[Rei96b] J. Reinert. *Ein Regelsystem zur Integritätssicherung in aktiven relationalen Datenbanksystemen*. PhD thesis, Universität Kaiserslautern, 1996. DISDBIS 11, infix-Verlag, St. Augustin.

[Sac97] D. Saccà. The Expressive Powers of Stable Models for Bound and Unbound DATALOG Queries. *Journal of Computer and System Sciences*, 54(3):441–464, 1997.

[Sch95] J. S. Schlipf. Complexity and Undecidability Results in Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 15(III-IV), 1995.

[Sel95] T. K. Sellis, editor. *2nd Intl. Workshop on Rules in Database Systems (RIDS)*, number 985 in LNCS, Athens, Greece, 1995. Springer.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres Next Generation Database Management System. *Communications of the ACM*, 34(10):78–93, October 1991.

[SK96] E. Simon and J. Kiernan. The A-RDL System. In Widom and Ceri [WC96a], chapter 5, pp. 111–149.

[SZ90] S. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. of the 9th ACM Symposium on Principles of Database Systems*, pp. 205–217, 1990.

[SZ91] D. Saccà and C. Zaniolo. Partial Models and Three-Valued Models in Logic Programs with Negation. In A. Nerode, V. W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-monotonic Reasoning, 1st Intl. Workshop (LNMR)*. MIT Press, 1991.

[TCG+93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases*. Benjamin/Cummings, 1993.

[Tho90] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.

[Tra50] B. A. Trakhtenbrot. Impossibility of an Algorithm for the Decision Problem in Finite Classes. *Doklady Akad. Nauk. SSSR*, 70:569–572, 1950.

[TU95] E. Teniente and T. Urpí. A Common Framework for Classifying and Specifying Deductive Database Updating Problems. In *Intl. Conference on Data Engineering (ICDE)*, pp. 173–182, Taipei, Taiwan, 1995.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1988.

[Var82] M. Y. Vardi. The Complexity of Relational Query Languages. In *Proc. 14th ACM Symposium on Theory of Computing*, pp. 137–146, 1982.

[Var88] M. Vardi. Decidability and Undecidability Results for Boundedness of Linear Recursive Queries. In *Proc. of the 7th ACM Symposium on Principles of Database Systems*, pp. 341–351, 1988.

[VG88] A. Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 149–176. Morgan Kaufmann, 1988.

[VG89] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–10, 1989.

[VG93] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VGD97] A. Vaduva, S. Gatziu, and K. R. Dittrich. Investigating Termination in Active Database Systems with Expressive Rule Languages. In Geppert and Berndtsson [GB97].

[VGRS88] A. Van Gelder, K. Ross, and J. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 221–230, 1988.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620 – 650, July 1991.

[VGT91] A. Van Gelder and R. W. Topor. Safety and Translation of Relational Calculus Queries. *TODS*, 16(2):235–278, 1991.

[Via97a] V. Vianu. personal communication. 1997.

[Via97b] V. Vianu. Rule-Based Languages. *Annals of Mathematics and Artificial Intelligence*, 19(I–II):215–259, 1997.

[WC94] J. Widom and S. Chakravarthy, editors. *4th Intl. Workshop on Research Issues in Data Engineering (RIDE)*. IEEE Computer Society Press, 1994.

[WC96a] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[WC96b] J. Widom and S. Ceri. Introduction to Active Database Systems. In *Active Database Systems: Triggers and Rules for Advanced Database Processing* [WC96a], chapter 1, pp. 1–41.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented Production Rules in Relational Database Systems. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 259–270, 1990.

[WF97] C.-A. Wichert and B. Freitag. Capturing Database Dynamics by Deferred Updates. In *Intl. Conference on Logic Programming (ICLP)*, Leuven, Belgium, 1997. MIT Press.

[Wid92] J. Widom. A Denotational Semantics for the Starburst Production Rule Language. *SIGMOD Record*, 21(3):4–9, September 1992.

[Wid93] J. Widom. Deductive and Active Databases: Two Paradigms or Ends of a Spectrum. In Paton and Williams [PW93].

[Wid94] J. Widom. Active Databases. In M. Stonebraker, editor, *Readings in Database Systems, 2nd edition*, pp. 345–350. Morgan Kaufmann, 1994. Introduction to chapter 4.

[Wid96] J. Widom. The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):583–595, 1996.

[Win90] M. Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

[Zan93] C. Zaniolo. A Unified Semantics for Active and Deductive Databases. In Paton and Williams [PW93], pp. 271–287.

[Zan95] C. Zaniolo. Active Database Rules with Transaction Conscious Stable Model Semantics. In Ling et al. [LMV95], pp. 55–72.

[Zan97] C. Zaniolo. The Nonmonotonic Semantics of Active Rules in Deductive Databases. In Bry et al. [BRR97].

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.

[ZS94] C. Zaniolo and R. Sadri. A Simple Model for Active Rules and their Behavior in Deductive Databases. In *Proc. of the 2nd ICLP Workshop on Deductive Databases and Logic Programming*, Santa Margherita Ligure, Italy, June 1994.

Index