Logics for Databases and Information Systems

Edited by
Jan Chomicki, Gunter Saake
Monmouth University / University of Magdeburg

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

12
On Logical Foundations of Active Databases 389
Georg Lausen and Bertram Ludäscher and Wolfgang May
12.1 Introduction 390
12.2 Basics of Active Rules 390
12.2.1 Terminology 390
12.2.2 Fundamental Properties 393
12.3 Research on Foundations of Active Rules 393
12.3.1 Production Rules 394
12.3.2 Declarative Rules 395
12.3.3 Extending Declarative Rules by States 396
12.3.4 Further Work 396
12.3.5 Bibliographic Notes 398
12.4 A Deductive State-Oriented Core Language 399
12.4.1 Basic Execution Model 400
12.4.2 Syntax 401
12.4.3 Semantics 402
12.4.4 Transitions, Termination and Transactions 404
12.4.5 Compile-Time vs. Run-Time Properties 405
12.5 A Framework for Active Rules 407
12.5.1 Signature 407
12.5.2 User-Defined vs. System-Defined Rules 408
12.5.3 Enforcing Termination 411
12.5.4 Expressive Power and Normal Forms 412
12.6 Conclusion 414

References 416

12 ON LOGICAL FOUNDATIONS OF ACTIVE DATABASES

Georg Lausen
and Bertram Ludäscher
and Wolfgang May

Abstract: In this chapter, we present work on logical foundations of active databases. After introducing the basic notions and terminology, we give a short overview of research on foundations of active rules. Subsequently, we present a specific state-oriented logical approach to active rules which aims at combining the declarative semantics of deductive rules with the possibility to define updates in the style of production rules. The resulting language *Statelog* models (flat) transactions as a sequence of intermediate transitions, where each transition is defined using deductive rules. Since Statelog programs correspond to a specific class of locally stratified logic programs, they have a unique intended model. Finally, after studying further fundamental properties like expressive power and termination behavior, a Statelog framework for active rules is presented. Although the framework is surprisingly simple, it allows to model many essential features of active rules, including immediate and deferred rule execution, and composite events. Different alternatives for enforcing termination are proposed leading to tractable subclasses of the language. Finally, we show that certain classes of Statelog programs correspond to Datalog programs with production rule semantics (i.e., with inflationary or noninflationary fixpoint semantics).

12.1 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 databases* (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 desirable to move functionality from application programs to the ADB. Thus, more code can be shared by different application programs, thereby facilitating software development and maintenance. Moreover, an ADB can monitor events more efficiently than a passive system which requires frequent polling 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.

Although the area of active databases has been quite active indeed, and produced a huge amount of literature as well as a number of prototypes, theoretical work on foundations is still rare and there is no generally accepted framework as is the case for deductive databases.

In Section 12.2 we present the basic notions of active rules, in Section 12.3 we briefly review the state of the art with a focus on theoretical aspects and foundations of active rules. Sections 12.4 and 12.5 elaborate on a logical framework based on a state-oriented extension of Datalog. In this framework, active rules are viewed from a more abstract and logical point of view, which allows to study fundamental properties of active rules independent from specific implementation aspects.

12.2 BASICS OF ACTIVE RULES

12.2.1 Terminology

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

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

Whenever the specified *event* occurs, the rule is triggered and the corresponding *action* is executed if the *condition* is satisfied in the current database

state. Rules without the event part are sometimes called *production rules*, rules without the condition part are sometimes referred to as *triggers*.

Events. Events can be classified as *internal* or *external*. Internal events are caused by database operations like retrieval or update (insertion, deletion, modification) of tuples, or transactional events like commit or abort. In object-oriented systems such internal events may take place through method invocations. External events occurring outside the database system may also be declared and have to be monitored by the ADB.

Starting from primitive (external or internal) events, more complex *composite events* can be specified using an *event algebra* (based on operators like disjunction, sequence, closure, etc., cf. [CKAK94; Sin95]), or some logical formalism like past temporal logic (see e.g., [LS87; Cho95b]). Several event detection algorithms have been developed which allow to detect composite events without storing the complete database history, for instance by applying temporal reduction rules [LS87; Cho95b] or residuation [Sin95].

Event consumption modes (also called *parameter contexts*) specify the correspondence between the occurrence of a composite event and the occurrences of its constituent primitive events. For example, the composite event may consume (be associated with) the most recent occurrence of its primitive events (*recent context*), or the first occurrence (*chronicle context*), or all occurrences (*unrestricted context*).

Conditions. If the triggering event of an active rule has been detected, the rule becomes eligible, and the condition part is checked. The condition can be a conventional SQL-like query on the current state of the database, or it may include *transition conditions*, i.e., conditions over changes in the database state. The possibility to refer to *different states* or *delta relations* is essential in order to allow for active state-changing rules.

Actions. If the condition of the triggered rule is satisfied, the action is executed. *Internal actions* are database updates (insert, delete, modify) and transactional commands (commit, abort), *external actions* are executed by procedure calls to application programs and can cause application-specific actions outside the database system (e.g., send-mail, turn-on-sensor). Usually, it is necessary to pass parameters between the different parts of ECA-rules, i.e., from the triggering event to the condition, and to the action part. In logic-based approaches this can be modeled very naturally using logical variables, while this issue may be more involved under the intricacies of certain execution models.

Execution Models. The basic execution model of active rules is similar to the *recognize-act cycle* of production rule languages like OPS5 [BFKM85]: 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 – for example, when no more rules can be triggered, or a fixpoint is reached. Clearly, there are a lot of possible choices and details which have to be elaborated in order to precisely specify the semantics of rule execution.

One issue is the *granularity* of rule processing, which specifies when rules are executed. This may range from execution at any time during the ADB's operation (finest granularity), over execution only at statement boundaries, to transaction boundary execution (coarsest granularity). Another important aspect is whether rules are executed in a *tuple-oriented* or *set-oriented* way. 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 another degree of nondeterminism to the language, since the outcome may now depend on the order in which individual rule instances are fired.

Finally, several *coupling modes* have been proposed, which describe the relationship between rule processing and database transactions. Under *immediate* and *deferred* coupling, the triggering event, as well as condition evaluation and action execution occur within the same transaction. In the former case, the action is executed immediately after the condition has become true, while in the latter case, action execution is deferred to the end of the current transaction. Under *decoupled* (sometimes called *detached* or *concurrent*) execution mode, a separate transaction is spawned for condition evaluation and action execution. Decoupled execution may be further divided into dependent or independent decoupled: in the former case, the separate transaction is spawned only after the original transaction commits, while in the latter case the new transaction is started independently. In the most sophisticated models, one may even have distinct coupling modes for event-condition coupling and for condition-action coupling.

The preceding paragraphs show that the possibility to refer to *different states* or *delta relations* is essential for event detection and condition checking in order to allow for active state-changing rules. As we will see in Sections 12.4 and 12.5 by incorporating states and deltas into the language, a logical framework for active rules can be developed, which allows to investigate fundamental properties like termination and expressiveness of active rules.

12.2.2 Fundamental Properties

Although there is a 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: Termination is arguably the most crucial property of rule execution. However, since rules may trigger each other recursively, nontermination of active rules is a permanent threat.

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 desirable feature, since the behavior of rules is easier to grasp if there is a unique final result. On the other hand, nondeterminism may be acceptable in some applications or may even be used as a "feature", e.g., to choose some arbitrary element from a set; cf. [GGSZ97].

Expressive Power: This is an abstract property of the underlying rule language: The expressive power of a database language is the class of database transformations expressible in the language, i.e., the class of mappings $m : inst(\mathbf{R}) \rightarrow inst(\mathbf{R})$ between database instances over a given schema $\mathbf{R}$. 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 sublanguages defining events, conditions, and actions, respectively.

Complexity: The complexity of a set of rules or a rule language measures the computational cost involved in determining the final result. Since the size of a database usually dominates by far the size of the program, it is common to consider *data complexity*, where the size of the program is fixed, and the size of the input databases varies. Not surprisingly, there is a trade-off between expressive power and complexity.

12.3 RESEARCH ON FOUNDATIONS OF ACTIVE RULES

There are many important aspects related to the efficient implementation of active rules, like efficient event detection (which becomes more and more involved in the presence of rich event algebras), condition evaluation, and execution of actions.[1] Since our aim is to investigate fundamental logical issues, we do not consider implementation aspects here. Instead, we discuss research on formal foundations for active rules.

12.3.1 Production Rules

Production rules can be viewed as ECA-rules without the event part. However, production rules have been around long before the ECA paradigm has been established. For example, the production rule language OPS5 [BFKM85] has been used in the AI community since the 1970's. From a more abstract point of view, one can regard general ECA-rules also as production rules since the event detection part can be encoded in the condition.[2] This abstraction is very useful as it allows to apply techniques and results developed for production rules to active rules.

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 (*candidate set*) one rule is selected using some conflict resolution strategy and the corresponding actions are executed. This process is repeated until there are no more triggered rules.

In the database community, such a forward chaining or *fixpoint* semantics has been studied for a number of Datalog variants (see e.g., [AV91]) thereby providing a logic-oriented formalization of production rules:

Let *Datalog*$^\neg$ denote the class of Datalog programs which allow negated atoms in rule bodies. The *inflationary* Datalog$^\neg$ semantics turns the well-known immediate consequence operator T_P [vEK76] developed for (definite) logic programs into an inflationary operator T'_P by keeping all tuples which have been derived before, i.e., $T'_P(\boldsymbol{I}) := \boldsymbol{I} \cup T_P(\boldsymbol{I})$ where $\boldsymbol{I}$ is the set of ground atoms derived in the previous round. Starting with a set of facts $\boldsymbol{I}$ (the inital state), T'_P is iterated until a fixpoint (the final state) is reached. Since the computation is inflationary, deletions cannot be expressed directly. In contrast, Datalog$^{\neg\neg}$ has a *noninflationary* semantics by allowing negative literals to occur also in the head of rules and interpreting them as deletions: if a negative literal $\neg A$ is derived, a previously inferred atom A is removed from $\boldsymbol{I}$. If both A and $\neg A$ are inferred in the same round, several options exists: priority may be given either to insertion or to deletion, or a "no-op" may be executed, using the truth value of A from the previous state, or the whole computation may be aborted [Via97]. While for inflationary Datalog$^\neg$ termination is guaranteed, this is no longer the case of Datalog$^{\neg\neg}$. In fact, it is undecidable whether a Datalog$^{\neg\neg}$ program reaches a fixpoint for all databases. Moreover, confluence is no longer guaranteed if instead of the presented semantics, a nondeterministic semantics is used [AS91]. On the other hand, nondeterminism can be a powerful programming paradigm which increases the (theoretical and practical) expressiveness of a language [AV91; GGSZ97].

A problem with these "procedural" Datalog semantics is that the handling of negation can lead to quite unintuitive results:

Example 1 Under the inflationary semantics, the program

```
tc(X,Y) ← e(X,Y).
tc(X,Y) ← e(X,Z), tc(Z,Y).
non-tc(X,Y) ← ¬ tc(X,Y).
```

does *not* compute in non-tc the complement of the transitive closure of a given edge-relation e. The reason is that the last rule is applied "too early", i.e., before the computation of the fixpoint for tc is completed. Thus, despite the fact that the derivation of non-tc(x,y) may be invalidated by a subsequent derivation of tc(x,y), this unjustified tuple remains in non-tc. □

Although the given program may be rewritten using a (somewhat intricate) technique for delaying rules, a better solution is to use one of the declarative semantics developed for logic programs whenever the use of negation is important, see Section 12.3.2.

RDL1 [KdMS90] is a deductive database language with production rule semantics; a rule algebra is used as an additional control mechanism. A-RDL [SK96] extends RDL1 by active database concepts, in particular delta relations and a module concept.

12.3.2 Declarative Rules

The logic programming and deductive databases communities have studied in-depth the problem of assigning an appropriate semantics to logic programs with negation like the one above. The *stratified*, *well-founded*, and *stable semantics* [ABW88; VG89; GL88] are now generally accepted as intended and intuitive semantics of logic programs with negation (see Chapter ??). For stratified programs, like the one in Example 1, all three semantics coincide.[3] For non-stratified programs, the well-founded semantics yields a unique three-valued model, whereas the stable semantics consists of a (possibly empty) set of two-valued stable models, each of them extending the well-founded model.

For relational databases, i.e., finite structures, termination and confluence of declarative rules can be guaranteed: For example, under the stratified semantics, rules are partitioned into strata according to the dependencies between defined relations. Thus, the strata induce a partial order on rules which is used to evaluate programs. Within each stratum the rules are fired simultaneously in a set-oriented way. Since the computation within strata is monotonic, the rules may also be evaluated in arbitrary order and/or tuple-oriented within a stratum without sacrificing confluence. Termination is guaranteed since it is not possible to add and remove the same fact repeatedly as is the case for Datalog$^{\neg\neg}$ and noninflationary Datalog$^{\neg}$.

In principle, although Datalog is primarily a query language, it could be used as a relational update language, for example by interpreting relations like

old_R and **new_**R as the old and new values of a relation R, respectively, or by assuming that R', R'', ..., refer to different states of R. However, such an approach has several drawbacks: First, part of the semantics is encoded into relation names and thus outside of the logical framework. More importantly, the language does not incorporate the notion of state which is central to updates and active rules. In particular, only a fixed number of state transitions can be modeled by "priming" relation names as described above.

A number of deductive database prototypes with declarative semantics exist including Aditi, LDL, FLORID, Glue-Nail, Coral, LOLA, and XSB-Prolog (cf., [RH94; Min96; SP97]).

12.3.3 Extending Declarative Rules by States

By incorporating states into a logical framework, (active) production rules and declarative rules can be handled in a unified way combining the strengths of both paradigms. This approach is followed by [Zan93; Zan95; MZ97] and [KLS92; LHL95; LML96] and will be discussed further in Sections 12.4 and 12.5. These approaches can be regarded as variants of Datalog$_{1S}$ [Cho95a] which is used as a query language for temporal databases. As an aside, state-oriented Datalog extensions have also been used as a technique for computing well-founded models [KRS95].

12.3.4 Further Work

Static Rule Analysis. [AWH95] develop static analysis techniques for active rules which guarantee termination, confluence, and observable determinism (i.e., whether each program produces a unique stream of observable actions) under the Starburst execution model. Rule analysis is based on a *triggering graph* which contains an edge between rules r_i and r_j if the former may trigger the latter. Termination is guaranteed if the triggering graph is acyclic, confluence is guaranteed if all unrelated rules commute pairwise. Related work on static rule analysis using triggering and dependency graphs or techniques from term rewriting include [ZH90; BW94; BCP95; KC95; KU96].

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 state-oriented logic framework presented in Section 12.5 can be regarded as the declarative pendant to Heraclitus.

Trigger Programs. [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 presented framework, the complexity of immediate triggering is essentially EXPTIME, even without delta relations and goes down to 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.

Logic-Based Integration of Operational Semantics. In [FWP97], an abstract formalization for active rules is given, using specialized sublanguages for event detection, condition evaluation, and execution of actions referring to explicit time points. The focus is on integrating the different *operational semantics* of active and deductive rules within a history-centered logic-based architecture. Currently, the approach is incorporated into the object-oriented deductive database language ROCK & ROLL [BFP+95].

Referential Actions as Logical Rules. As shown in [LML97], logic programming semantics can be used to disambiguate the meaning of *referential actions* in relational databases. Referential actions (called *declarative constraints* in [CPM96]) are specialized active rules to enforce referential integrity between the referencing child relation and the referenced parent relation. While the idea of using simple referential actions to guarantee referential integrity is very appealing and is included in the current SQL3 proposal [ISO95], the global effect of a set of referential actions is hard to predict or even define. [LML97] show how the intended meaning of a set of referential actions RA can be captured by certain stable models of a logic program P_{RA}.

Transaction Logic. *Transaction Logic* $\mathcal{T}_{\mathcal{R}}$ [BK94] deals, on a high level of abstraction, with the phenomenon of state change in logic databases and employs a powerful model theory and proof theory. $\mathcal{T}_{\mathcal{R}}$ is discussed in detail in Chapter **??** of this volume.

Active Rules and Action Languages. In [BL96] 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 to model actual and hypothetical actions and situations. 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) is used to nondeterministically select 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 the computationally easier task of executing them.

12.3.5 Bibliographic Notes

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 1970's, 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 [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 [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, in the HiPAC project, concepts for time-constrained rules and a number of sophisticated coupling modes were developed. Follow-on projects are Sentinel [CKAK94] (with its powerful event specification language SNOOP) and REACH [BBKZ93]. Other object-oriented systems are ADAM/EXACT [DPG91; DJP93], Chimera [CFPT96], ODE [GJS92], NAOS [CCS94], and SAMOS [GD93].

Commercial systems are also beginning to include ideas and concepts from active databases. The current SQL3 proposal offers so-called *declarative con-*

straints used specifically for maintaining referential integrity, and general purpose *triggers*. Triggers may be specified to execute *before*, *instead of*, or *after* the triggering database event (insert, delete, update), and can be executed individually for each row (tuple-oriented), or at the statement level (set-oriented) [CPM96; ISO95].

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] also has major sections on active rules.

12.4 A DEDUCTIVE STATE-ORIENTED CORE LANGUAGE

Although there has been lot of work in active databases, no single generally accepted framework for active rules has evolved (the ECA paradigm – though widely used and accepted – only gives a very rough idea of rule execution and leaves most issues unresolved). One major theoretical problem is that rule semantics is often defined only in an informal and procedural way, making it very difficult to understand and predict the behavior of rules. 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]. [AWH95] state that *"the unstructured, unpredictable, and often nondeterministic behavior of rule processing can become a nightmare for the database rule programmer"*. Not surprisingly, it is required in the active database manifesto [DGG95] that *"... rule execution must have a clear semantics, i.e., must define when, how, and on what database state conditions are evaluated and actions executed"*.

In the sequel, we develop a logical framework for active rules which precisely and unambiguously defines the meaning of active rules. Moreover, it allows to study fundamental properties of active rules like termination, confluence and expressive power. The framework does not account for all facets of active rules

which may be useful in practice (like SQL3's *before* and *instead of* triggers, or tuple-level execution), but covers many essential features including immediate and deferred execution and composite events.

12.4.1 Basic Execution Model

A distinctive feature of active rule languages is the ability to refer to different database states and delta relations. In contrast to query processing where a "one-state logic" is sufficient, data manipulating operations require access to more than one state. Therefore, we extend the declarative language Datalog with the ability to refer to different database states. The resulting language *Statelog* [LL94; LML96] is closely related to the temporal query languages *Templog* and $Datalog_{1S}$ [BCW93] and to *XY-Datalog* [Zan93].

A central question is which operations of a database should define a new state. Clearly, if the user *queries* the database using ad-hoc statements or recursive views, the computations of the system which are necessary to answer such queries should *not* give rise to a new state: Conceptually, the database should not be modified at all since the state of the modeled world is not affected by the query.

In contrast, if the modeled world *changes*, the database has to be updated, i.e., a new state should be created. 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. However, a finer granularity is needed to model active rule processing *within* the logical language. 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 be queried by the user, while the former are not (directly) accessible to the user. Another difference is that integrity constraints have to be maintained for final states but not necessarily for intermediate states.

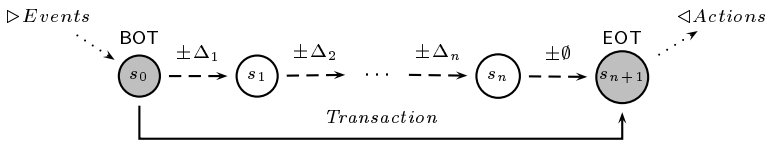

Figure 12.1 Database evolution: a transaction and intermediate transitions

Thus, the basic execution model of Statelog is as shown in Figure 12.1: $[s_0]$ is the *initial state*. Triggered by the occurrence of one or more external events, the *transaction* begins with the *transition* to the intermediate state $[s_1]$. Additional rules can be triggered until the *final state* $[s_{n+1}]$ is reached

which marks the end of transaction. Note that only gray states, i.e., the initial and final state, are materialized and directly accessible to the user.

In this model, the state space (or *temporal domain*) over which the database evolves is isomorphic to the natural numbers $\mathbb{N}_0$, i.e., a linear time model is used. Other, more general models are possible, for example branching time or a hierarchical state space [LML96].

12.4.2 Syntax

In Statelog, access to different database states is accomplished via *state terms* of the form $[S{+}k]$, where $S{+}k$ denotes the k-fold application of the unary function symbol "+1" to the *state variable* S. As mentioned before, the database evolves over a linear state space, so S may only be bound to some $n \in \mathbb{N}_0$.

Let $k \in \mathbb{N}_0$. A *Statelog database* $\boldsymbol{D}[k]$ at state k is a finite set of facts of the form $[k]\, p(x_1, \ldots, x_n)$ where p is an n-ary relation symbol and x_i are constants from the underlying domain. If $k = 0$, or is understood from the context, we simply write $\boldsymbol{D}$.

A *Statelog rule* r is an expression of the form

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

where the head H is a Datalog atom, B_i are Datalog literals (i.e., either atoms A or negated atoms $\neg A$), and the offsets $k_i \in \mathbb{N}_0$ are natural numbers. The *leap* $l_i := k_i - k_0$ of B_i is the distance between the state referred to in the head and the state for B_i in the body. If in a sequence of literals all of them refer to the same state, all occurrences of state terms but the first can be omitted. E.g., the body $[S]\, B_1, [S{+}1]\, B_2, [S{+}1]\, B_3$ may be abbreviated as $[S]\, B_1, [S{+}1]\, B_2, B_3$. A *Statelog program* is a finite set of Statelog rules.

Progressiveness. The basic idea of Statelog 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 notion of time, like *valid time* or *transaction time* [TCG+93]. The only requirement is that the (partial) order of the underlying time model has to correspond with the order on states in the following sense: For all state identifiers (= ground state terms) $[s_1]$, $[s_2]$:

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

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 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. This leads to the following progressiveness requirement:

The Statelog rule r in $(*)$ above is called *progressive*, if $k_0 \geq k_i$ for all $i = 1, \ldots, n$. If $k_0 = k_i$ for all $i = 1, \ldots, n$, r is called *local*. Finally, if $k_0 = 1$ and $k_i = 0$ for all $i \geq 1$, r is called *1-progressive*. In the sequel, all rules are assumed to be progressive.

It should be noted that Statelog programs can be regarded as variants of Datalog_{1S} programs [BCW93; Cho95a]. One major difference is in the interpretation of states: Whereas Datalog_{1S} is a query language for *temporal databases*, Statelog is a database programming language where subsequent states represent the *evolution* of a (non-temporal) *relational database* resulting from execution of active rules. This also explains why Statelog rules are required to be progressive while Datalog_{1S} rules are not. Moreover, in contrast to Datalog_{1S}, Statelog uses state-stratified negation (see below) to allow for updates by frame rules.

12.4.3 Semantics

Let $\mathbf{R}$ be a fixed relational schema over some finite domain U.[4] By $\mathcal{B}$ we denote the *Herbrand base* over $\mathbf{R}$ and U, i.e., the set of ground atoms which can be built using relation symbols from $\mathbf{R}$ and constants from U. Thus, a database (instance) over $\mathbf{R}$ is a subset of $\mathcal{B}$; $\mathit{inst}(\mathbf{R}) = 2^{\mathcal{B}}$ denotes the set of all database instances over $\mathbf{R}$.

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

- The *snapshot view* defines $\mathcal{I}$ as a mapping $\mathcal{I} : \mathbb{N}_0 \to 2^{\mathcal{B}}$ assigning to each $n \in \mathbb{N}_0$ the *snapshot (database)* $\mathcal{I}[n]$, i.e., the set of tuples which are true in state $[n]$. In the snapshot view, $\mathcal{I}$ is represented as the sequence $\mathcal{I} = (\mathcal{I}[n])_{n \in \mathbb{N}_0}$ of *snapshots*.

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

- The *reified* or *model-theoretic view* regards $\mathcal{I}$ as a subset of $\mathbb{N}_0 \times \mathcal{B}$. Using this representation, an atom with state term $[S{+}k]\, p(X_1, \ldots, X_n)$

is interpreted as $p(S{+}k, X_1, \ldots, X_n)$ where an additional state argument is added to every atom.

The reified view allows to identify a Statelog program P with a specific logic program $P^\star$, in which a single unary function symbol "+1" occurs in some restricted way. Thus, the well-known semantics for logic programs can be applied to the Statelog framework, since Herbrand interpretations of $P^\star$ correspond in a natural way to Statelog interpretations of P. As usual, an interpretation $\mathcal{I}$ is a *model* of a program P and database $\boldsymbol{D}$, if $\mathcal{I}$ satisfies every formula in $P \cup \boldsymbol{D}$. Wrt. the snapshot view, this is defined as

$$\begin{aligned}\mathcal{I} \models \quad & [S_0]\, H \leftarrow [S_1]\, B_1, \ldots, [S_n]\, B_n \;\; :\Leftrightarrow \\ & \mathcal{I}[s_0] \models h \text{ if } \mathcal{I}[s_1] \models b_1, \;\ldots,\; \mathcal{I}[s_n] \models b_n, \text{ for every ground instance} \\ & [s_0]\, h \leftarrow [s_1]\, b_1, \ldots, [s_n]\, b_n \text{ of } [S_0]\, H \leftarrow [S_1]\, B_1, \ldots, [S_n]\, B_n.\end{aligned}$$

Here, we adopt the *state-stratified* semantics as the canonical model $\mathcal{M}_{P \cup \boldsymbol{D}}$ of a Statelog program P with database $\boldsymbol{D}$. P is called state-stratified, if there are no negative cyclic rule dependencies *within a single state* [LHL95]. More precisely, state-stratification is based on the extended dependency graph $\mathcal{G}_P$ of P. Its nodes are the rules of P. Given two rule r_1, r_2 there is an edge $(r_1 \stackrel{l,(\neg)}{\longrightarrow} r_2) \in \mathcal{G}_P$ if the relation symbol in the head of r_1 occurs positively (negatively) in the body of r_2. Here, l is the leap of the corresponding literal in r_2. P is *state-stratified* if $\mathcal{G}_P$ contains no *local cycle* C (i.e., where $\sum_{(r_1 \stackrel{l,(\neg)}{\rightarrow} r_2) \in C} l = 0$) involving a negative edge. This notion is closely related to *XY-stratification* [Zan93] and *ELS-stratification* [KRS95]. Together with the requirement of progressiveness, state-stratification implies local stratification:

Theorem 1 *Every progressive state-stratified program P is locally stratified. Therefore, it has a unique perfect model $\mathcal{M}_{P \cup \boldsymbol{D}}$ for every database $\boldsymbol{D}$.*

Example 2 Consider the following Statelog program, which deletes all employees E from a department D which is deleted:

$$\begin{aligned} r_1: \quad & [S]\ \mathsf{del{:}emp(E,Sal,D)} \leftarrow [S]\ \mathsf{del{:}dept(D,_)},\ \mathsf{emp(E,Sal,D)}\ . \\ r_2: \quad & [S{+}1]\ \mathsf{emp(E,Sal,D)} \leftarrow [S]\ \mathsf{emp(E,Sal,D)},\ \neg\, \mathsf{del{:}emp(E,Sal,D)}\ . \end{aligned}$$

On occurrence of a delete event to dept, r_1 checks whether there is an employee E working at department D, and if so, this employee is put in a special *delta relation* del:emp (cf. Section 12.5). r_2 is a *frame rule* specifying that only those tuples are copied to the instance of emp at the next state which are not in del:emp. Thus, logically, no real deletion occurs but a smaller "copy" of the old database state is created. Note that although the program is not stratified, it is state-stratified since the negative cycle $r_2 \stackrel{0}{\rightarrow} r_1 \stackrel{1,\neg}{\rightarrow} r_2$ does not occur within a single state. □

12.4.4 Transitions, Termination and Transactions

Recall the basic Statelog execution model depicted in Figure 12.1. Given the model $\mathcal{M}_{P\cup\boldsymbol{D}}$ of a program P with database $\boldsymbol{D}$, the snapshot $\mathcal{M}_{P\cup\boldsymbol{D}}[n]$ is the database instance reached after n transitions.

We say that P *terminates for* $\boldsymbol{D}$, if for some $n_0 \in \mathbb{N}_0$ and all $k \geq 1$ we have that $\mathcal{M}_{P\cup\boldsymbol{D}}[n_0] = \mathcal{M}_{P\cup\boldsymbol{D}}[n_0{+}k]$; the least such n_0 is called the *final state* of the *transaction* starting at $[0]$ and ending at $[n_0]$. Note that for programs in normal form (cf. Section 12.5.4), it is sufficient to check for the least n_0 s.t. two subsequent states are the same. As a generic notation, we write $\mathcal{M}_{P\cup\boldsymbol{D}}[\$]$ to denote the final database state if P terminates for $\boldsymbol{D}$; otherwise we agree to set $\mathcal{M}_{P\cup\boldsymbol{D}}[\$] := \emptyset$.

Thus, for a given relational schema $\mathbf{R}$, we can associate the following mappings with every Statelog program:

- Γ^1_P : $\quad inst(\mathbf{R}) \rightarrow inst(\mathbf{R})$
  $\qquad\qquad \boldsymbol{D}[0] \mapsto \mathcal{M}_{P\cup\boldsymbol{D}[0]}[1]\,|_{\mathbf{R}}$
- $\Gamma^\$_P$: $\quad inst(\mathbf{R}) \rightarrow inst(\mathbf{R})$
  $\qquad\qquad \boldsymbol{D}[0] \mapsto \mathcal{M}_{P\cup\boldsymbol{D}[0]}[\$]\,|_{\mathbf{R}}$
- Γ^∞_P : $\quad inst(\mathbf{R}) \rightarrow (\mathbb{N}_0 \rightarrow inst(\mathbf{R}))$
  $\qquad\qquad \boldsymbol{D}[0] \mapsto \left(\mathcal{M}_{P\cup\boldsymbol{D}[0]}[n]\,|_{\mathbf{R}}\right)_{n\in\mathbb{N}_0}$

Γ^1_P maps a given initial database $\boldsymbol{D}[0]$ over $\mathbf{R}$ to the new (intermediate) database state $\boldsymbol{D}[1]$ and thus describes a single *transition*. In contrast, $\Gamma^\$_P$ maps $\boldsymbol{D}[0]$ to the final database state $\boldsymbol{D}[\$] := \mathcal{M}_{P\cup\boldsymbol{D}[0]}[\$]\,|_{\mathbf{R}}$ and describes the whole *transaction*. Finally, Γ^∞_P maps the input database $\boldsymbol{D}[0]$ to an infinite sequence of states, the *evolution* of $\boldsymbol{D}[0]$ under P.

Two Statelog programs P_1 and P_2 are called *evolution equivalent* (wrt. $\mathbf{R}$) denoted $P_1 \equiv_\infty P_2$, if $\Gamma^\infty_{P_1} = \Gamma^\infty_{P_2}$, *transaction equivalent* or simply *equivalent* (wrt. $\mathbf{R}$), denoted $P_1 \equiv_\$ P_2$, if $\Gamma^\$_{P_1} = \Gamma^\$_{P_2}$, and *transition equivalent* $P_1 \equiv_1 P_2$, if $\Gamma^1_{P_1} = \Gamma^1_{P_2}$. Clearly, evolution equivalence implies transaction and transition equivalence, i.e., if $P_1 \equiv_\infty P_2$ then $P_1 \equiv_\$ P_2$ and $P_1 \equiv_1 P_2$. However, the converse generally does not hold:

Example 3 **($P_1 \equiv_\infty P_2 \not\Leftarrow P_1 \equiv_\$ P_2$)** The programs

```
P1 :  [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 finite length of a path beginning at a) in order to compute the final result via intermediate states. Here, we assume that a *frame rule* [S+1] e(X,Y) ← [S] e(X,Y),¬ del:e(X,Y) is added to P_1 and P_2 which propagates the edge-relation e – modulo deletions – through the states, cf. Section 12.5.2. □

The presented notions will be used in Section 12.5.4 to characterize the expressive power of several Statelog variants.

12.4.5 Compile-Time vs. Run-Time Properties

A *compile-time property* is a property a program P may have, *independent* of a given database; a *run-time property* is a property which a program P together with a given database may have.

Run-Time Termination. Termination is arguably the most important property of rules. Since we assume a finite underlying domain, there are only finitely many possible database states and termination becomes decidable at run-time: As will be seen in Section 12.5.4, the transaction expressiveness of Statelog corresponds to the class of While queries (see e.g., [AHV95]). Furthermore, using an order on the underlying domain, PSPACE-complete problems can be encoded in While and thus in Statelog. In particular, let I be an instance of a PSPACE-complete problem encoded as a Statelog program P with database $\boldsymbol{D}_I$ s.t. $\mathcal{M}_{P\cup\boldsymbol{D}_I}[\$] \models \mathit{yes}$ iff I is a solution. Then, $P \cup \{\, [S{+}1]\, q \leftarrow [S] \neg q, \neg\, \mathit{yes} \,\}$ terminates for $\boldsymbol{D}_I$ iff I is a solution: Due to the additional rule, q oscillates between true and false if $\neg\, \mathit{yes}$ holds (i.e., if I is no solution). It follows that deciding termination at run-time is PSPACE-hard. Together with Theorem 4 below this implies the following result:

Theorem 2 *Given a Statelog program P and a database $\boldsymbol{D}$, deciding whether P terminates for $\boldsymbol{D}$ is* PSPACE*-complete.*

Compile-Time Termination. We say that a program P *sometimes terminates* if P terminates for some database $\boldsymbol{D}$, and *always terminates* if it terminates for all $\boldsymbol{D}$. Clearly, these are compile-time properties. Unfortunately, termination is in general not decidable at compile-time. This can be shown

using Trakhtenbrot's Theorem which says that satisfiability and validity of first-order sentences over finite structures is undecidable: Every first-order sentence φ is equivalent to a stratified Datalog$^\neg$ program P_φ. By attaching state terms $[S]$ to all literals in P_φ and adding the rules $[S{+}1]\, q \leftarrow [S] \neg q, \neg\, ans_\varphi$ and $[S{+}1]\, ans_\varphi \leftarrow [S]\, ans_\varphi$, we obtain a Statelog program P'_φ which sometimes (always) terminates iff φ is satisfiable (valid) in finite structures. Therefore:

Theorem 3 *Given an arbitrary Statelog program P, it is undecidable whether P sometimes (always) terminates.*

A simple negation-free program which does not terminate is $\{[S{+}1]\, p \leftarrow [S]\, q,\ [S{+}1]\, q \leftarrow [S]\, p\}$ (e.g., if $\boldsymbol{D}[0] = \{p\}$, then $\mathcal{M}_{P \cup \boldsymbol{D}}$ oscillates between $\{p\}$ and $\{q\}$). Interestingly, for negation-free programs the problem becomes decidable, which follows from a result on universal safety for $Datalog_{nS}$ [Cho95a].

Handling Nontermination. There are several ways to deal with termination problems. The first "solution" is to live with the danger and put the burden of ensuring termination on the programmer. Another approach is to restrict the class of allowed programs s.t. only terminating programs are acceptable. One obvious way is to make the evolution *inflationary* by adding a rule $[S{+}1]\, p(\bar{X}) \leftarrow\ [S]\, p(\bar{X})$ for every relation p occuring in the program. It is easy to see that all transitions in the database evolution are monotonically nondecreasing, so termination is guaranteed for finite domains. Note however, that this language is not well-suited for expressing updates, since it is not possible to express deletions. In Section 12.5.3 we will consider Δ-*monotonic* Statelog programs which allow deletions, yet guarantee termination at compile-time or run-time.

As mentioned above (Theorem 2), termination can be decided at run-time. Indeed, Statelog allows to detect that a fixpoint has been reached by simply comparing the current state $[S{+}1]$ with the previous state $[S]$. Interestingly, the converse, i.e., detecting nontermination is also possible *within* Statelog using a technique similar to loop-free simulation in [AS91]. In particular, the following can be shown:

Theorem 4 *For every Statelog program P over the schema $\mathbf{R}$ there is a transaction equivalent program $P^\downarrow$ which always terminates. Moreover, one can define relations "*terminates*" and "*diverges*" in $P^\downarrow$, such that for all databases $\boldsymbol{D}$:*

$$\mathcal{M}_{P^\downarrow \cup \boldsymbol{D}}[\$] \models \mathsf{terminates} \quad \Leftrightarrow \quad \mathcal{M}_{P \cup \boldsymbol{D}} \text{ terminates for } \boldsymbol{D}.$$
$$\mathcal{M}_{P^\downarrow \cup \boldsymbol{D}}[\$] \models \mathsf{diverges} \quad \Leftrightarrow \quad \mathcal{M}_{P \cup \boldsymbol{D}} \text{ does not terminate for } \boldsymbol{D}.$$

The crux of the construction is that, $P^{\downarrow}$ simultaneously evaluates the original program P and a "delayed" version P^d thereby allowing to detect nontermination at run-time (recall that $\boldsymbol{D}$ is finite). If termination is detected, $P^{\downarrow}$ derives "terminates" and the final state $\mathcal{M}_{P^{\downarrow}\cup\boldsymbol{D}}[\$]\big|_{\mathbf{R}} = \mathcal{M}_{P\cup\boldsymbol{D}}[\$]$. Otherwise, $P^{\downarrow}$ derives "diverges" and empties all relations of $\mathbf{R}$ (as required by the definition of $\mathcal{M}_{P\cup\boldsymbol{D}}[\$]$ in Sec. 12.4.4). Note that Theorem 4 does not contradict Theorem 3 since $P^{\downarrow}$ solves the problem only for a *given* $\boldsymbol{D}$, i.e., at run-time. It is undecidable whether $P^{\downarrow}$ derives "terminates" for all (resp. some) databases $\boldsymbol{D}$.

Confluence. The presented state-stratified semantics $\mathcal{M}_{P\cup\boldsymbol{D}}$ guarantees confluence of rule execution independent of the given database $\boldsymbol{D}$. This is due to the fact that the stratified semantics is deterministic, since it is defined by simultaneously firing all rule instances whose bodies are satisfied, i.e., in a set-oriented manner. Nevertheless, rules may also be fired one after another and in a tuple-oriented way without sacrificing confluence, provided the (state-) stratification is obeyed. Thus, the declarative semantics not only guarantees well-behavedness wrt. negation but also implies confluence since a partial order ("priority") on rules is implied by the stratification.

12.5 A FRAMEWORK FOR ACTIVE RULES

In this section, we develop a framework for active rules using Statelog as the logical core language.

12.5.1 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 rules, see below): The set of relation symbols of a given schema $\mathbf{R}$ is given as the disjoint union of the following sets:

$$\mathbf{R} = \mathit{edb}(\mathbf{R}) \dot\cup \mathit{idb}(\mathbf{R}) \dot\cup \rhd(\mathbf{R}) \dot\cup \lhd(\mathbf{R}) \dot\cup \delta(\mathbf{R}) \dot\cup \mathit{ctl}(\mathbf{R})$$

Base and Derived Relations. The *extensional database* $\mathit{edb}(\mathbf{R})$ comprises the base relations which are stored in the database. *edb*-relations may only occur in rule bodies and can be updated directly using *delta relations* from $\delta(\mathbf{R})$. In contrast, derived relations belong to the *intensional database* $\mathit{idb}(\mathbf{R})$ and define *views*. Thus, *idb*-relations may occur in rule heads and bodies but may not be changed directly. Typically, *idb*-relations are not materialized but computed on demand.

External Events and Actions. Relations from $\triangleright(\mathbf{R})$ represent *external events* of interest which are monitored by the ADB. Consequently, external events can only occur in rule bodies. *External actions* are defined by the relations from $\triangleleft(\mathbf{R})$ and represent requests to execute certain actions outside the ADB system. Relation symbols denoting external events and actions are prefixed with the symbols "$\triangleright$" and "$\triangleleft$", respectively.

Delta Relations. For every base relation $R \in edb(\mathbf{R})$ there are *delta relations* $\mathsf{del}{:}R, \mathsf{ins}{:}R \in \delta(\mathbf{R})$. Delta relations (or just *deltas*) denote update requests to delete resp. insert the corresponding tuples into R.

Control Relations. Finally, $ctl(\mathbf{R})$ contains special *control relations* like BOT, EOT, and abort for transaction control, auxiliary relations for the detection of composite events, and *protocol relations* $\mathsf{insd}{:}R$ and $\mathsf{deld}{:}R$ (for inserted and deleted, resp.) which store the accumulated *net effect* of a sequence of updates. The latter can be used to enforce termination (Section 12.5.3), or as an auxiliary store holding all necessary information to undo the effect of an aborting transaction.

12.5.2 User-Defined vs. System-Defined Rules

In the Statelog core language there is no distinction between user-defined and system-defined rules. However, an ADB system should provide the user with a predefined intuitive programming "environment" which takes care of low-level aspects of the execution model like transactional control. In particular, one may hide the explicit handling of states from the user by forcing her to use only local rules. If the user really needs to refer to different states, syntactic sugaring in the form of predefined operators can be used.

User-Defined Rules. A *user program* consists of a set of local Statelog rules. Only relations from $idb(\mathbf{R})$, $\triangleleft(\mathbf{R})$, $\delta(\mathbf{R})$, and certain distinguished relations from $ctl(\mathbf{R})$ are allowed to occur in rule heads. For example, the usual integrity constraints from databases like functional, join, and inclusion dependencies can be encoded in the form of *denials*, i.e., as a set of local rules s.t. abort is derived by these rules if an integrity violation is detected. Since all user-defined rules are local, state terms may be omitted.

Example 4 The rules from Example 1 constitute a Statelog user program with *idb*-relations tc and non-tc, and an *edb*-relation edge. Note that the intended model is computed since we use a declarative (state-stratified) semantics.

The first rule of Example 2 also is a valid user rule, while the second rule is not allowed since an *edb*-relation occurs in the head. □

System-Defined Rules. If one or more external events $\triangleright E(\bar{x})$ occur, the beginning of a transaction is signaled:

$$[S]\,\mathsf{BOT} \leftarrow\ [S]\,\triangleright E(\bar{X}).$$

We assume that all external events which are detected within a certain time interval are raised only in the initial state of a transaction (which coincides with the final state of the previous transaction). All events occurring subsequently are associated with the next initial state.

Frame rules are used to specify the handling of update requests for every *edb*-relation R:

$$\begin{array}{l}[S{+}1]\,R(\bar{X}) \leftarrow\ [S]\,\mathsf{ins}{:}R(\bar{X}), \neg\,\mathsf{abort}.\\ [S{+}1]\,R(\bar{X}) \leftarrow\ [S]\,R(\bar{X}), \neg\,\mathsf{del}{:}R(\bar{X}), \neg\,\mathsf{abort}.\end{array}$$

Thus, updates to base relations are executed *immediately* in the transition to the successor state. If a tuple is inserted and deleted at the same time, priority is given to insertions since the first rule allows to derive $R(\bar{X})$. In order to give priority to deletions, it suffices to add the literal $\neg\,\mathsf{del}{:}R(\bar{X})$ to the body of the first rule. Note that frame rules precisely define the effect of delta relations; in particular there is no "hidden semantics" of $\mathsf{del}{:}R$ and $\mathsf{ins}{:}R$. Apart from assigning priorities, other policies can also be easily specified. For example the rule

$$[S]\,\mathsf{abort} \leftarrow\ [S]\,\mathsf{ins}{:}R(\bar{X}), \mathsf{del}{:}R(\bar{X}).$$

aborts the current transaction if there is a request to insert and delete a tuple at the same time.

Protocol relations $\mathsf{insd}{:}R, \mathsf{deld}{:}R \in \mathit{ctl}(\mathbf{R})$ store the accumulated *net effect* of a sequence of a transaction as long as running holds (otherwise, they are cleared in the transition to $[S{+}1]$, cf. Figure 12.2):

$$\begin{array}{lll}[S{+}1]\,\mathsf{insd}{:}R(\bar{X}) \leftarrow\ [S]\,\mathsf{ins}{:}R(\bar{X}), \neg\,R(\bar{X}), \mathsf{running}. & & \%\ \textit{store net effect...}\\ [S{+}1]\,\mathsf{insd}{:}R(\bar{X}) \leftarrow\ [S]\,\mathsf{insd}{:}R(\bar{X}), \neg\,\mathsf{del}{:}R(\bar{X}), \mathsf{running}. & & \%\ \ldots\ \textit{of insertions}\\ [S{+}1]\,\mathsf{deld}{:}R(\bar{X}) \leftarrow\ [S]\,\mathsf{del}{:}R(\bar{X}), R(\bar{X}), \mathsf{running}. & & \%\ \textit{store net effect...}\\ [S{+}1]\,\mathsf{deld}{:}R(\bar{X}) \leftarrow\ [S]\,\mathsf{deld}{:}R(\bar{X}), \neg\,\mathsf{ins}{:}R(\bar{X}), \mathsf{running}. & & \%\ \ldots\ \textit{of deletions}\end{array}$$

Whether the current transaction is still running is determined by checking for pending change requests:

$$\begin{array}{l}[S]\,\mathsf{running} \leftarrow\ [S]\,\mathsf{ins}{:}R(\bar{X}), \neg\,R(\bar{X}), \neg\,\mathsf{abort}.\\ [S]\,\mathsf{running} \leftarrow\ [S]\,\mathsf{del}{:}R(\bar{X}), R(\bar{X}), \neg\,\mathsf{abort}.\end{array}$$

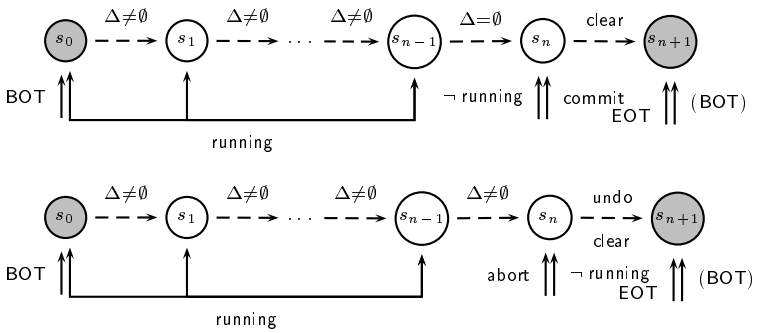

Figure 12.2 Committing and aborting transactions

Finally, if there are no more unprocessed update requests and the transaction is not aborted, commit is derived:

$$[S]\,\mathsf{commit} \leftarrow [S]\,\mathsf{BOT}, \neg\,\mathsf{running}, \neg\,\mathsf{abort}.$$
$$[S{+}1]\,\mathsf{commit} \leftarrow [S]\,\mathsf{running}, [S{+}1]\,\neg\,\mathsf{running}, \neg\,\mathsf{abort}.$$
$$[S{+}1]\,\mathsf{EOT} \leftarrow [S]\,\mathsf{commit}. \quad \%\ \textit{signal end of transaction}$$

If the transaction is aborted, the previous final state is restored by undoing the net effect of the transaction:

$$[S{+}1]\,R(\bar{X}) \leftarrow [S]\,\mathsf{abort}, \mathsf{deld}{:}R(\bar{X}). \qquad \%\ \textit{undo deletions}$$
$$[S{+}1]\,R(\bar{X}) \leftarrow [S]\,\mathsf{abort}, R(\bar{X}), \neg\,\mathsf{insd}{:}R(\bar{X}). \quad \%\ \textit{undo insertions}$$
$$[S{+}1]\,\mathsf{EOT} \leftarrow [S]\,\mathsf{abort}. \qquad \%\ \textit{signal end of transaction}$$

The execution schema of a committing resp. aborting transaction is shown in Figure 12.2. Note that intermediate states can be distinguished from user-visible final states by the control predicate EOT which holds in $[s]$ iff $[s]$ is a final state.

Different Coupling Modes. It should be clear, that instead of the above immediate coupling between condition evaluation and action execution, one may easily specify *deferred coupling* using a different set of frame rules: instead of applying deltas in every intermediate transition, requests are collected until there are no more new requests. Then all accumulated requests are executed once in a single final transition before the transaction commits. Note however, that in order to handle *decoupled* execution, a more involved framework which allows to model nested transactions (like the Statelog extension in [LML96]) is required.

Composite Events. Various kinds of composite events and consumption modes (e.g., of SNOOP [CKAK94]) can be expressed in Statelog, as shown in [MZ95] using the closely related language Datalog_{1S}. Assume for instance that we want to detect the composite event $E(X,Y) := (F(X)\,;\,G(Y))$, i.e., $F(X)$ followed by $G(Y)$ for some (external or internal) events F and G. Under an *unrestricted context*, this can be expressed by *temporal reduction rules* (similar to [LS87; Cho95b]):

$$[S]\,\mathsf{detd}{:}F(X) \leftarrow [S]\,F(X).$$
$$[S{+}1]\,\mathsf{detd}{:}F(X) \leftarrow [S]\,\mathsf{detd}{:}F(X).$$
$$[S{+}1]\,\mathsf{detd}{:}E(X,Y) \leftarrow [S]\,\mathsf{detd}{:}F(X), [S{+}1]\,G(Y).$$

Auxiliary relations $\mathsf{detd}{:}R$ store detected events. If one adds the goal $\neg\, F(_)$ to the second rule, only the most recent occurrences of F are used, thereby modeling event consumption with *recent context*. On the other hand, under the *chronicle context*, events are processed in a FIFO-like manner, and thus make use of a queue in an essential way. Therefore, composite events with chronicle contexts are *not* expressible in pure Statelog and require appropriate extensions, e.g., timestamping as in [MZ95].

12.5.3 Enforcing Termination

Apart from the handling of nontermination described in Section 12.4.5, a more elaborated treatment is possible in the extended framework. In addition to guaranteeing termination, we also obtain tractable fragments of Statelog: in contrast to general Statelog which has PSPACE data complexity, the following subclasses can be evaluated in PTIME.

Guarded Rules. A user-defined rule is called *guarded* if an external event $\triangleright E$ occurs positively in the body [LL94]. Guarded rules can fire only once within a transaction, since we assume that all external events are raised in the initial state of a transaction. Therefore, guarded rules cannot cause nontermination and can be excluded from further termination analysis.

Δ-Monotonic Rules. A drawback of guarded rules is their limited expressive power. Indeed, transactions consist of precisely one transition if all user rules are guarded. The reason for nontermination in our framework lies in an oscillating sequence of insertions and deletions of the same tuples. The idea of *Δ-monotonic rules* is to enforce termination by preventing such oscillation. This can be achieved at several levels of granularity:

Compile-Time Δ-Monotonicity: A user program P is called *compile-time Δ-monotonic* if for every $R \in \mathit{edb}(\mathbf{R})$ the following holds: if $\mathsf{ins}{:}R$ occurs

in the head of a rule of P, then $\mathsf{del}{:}R$ does *not* occur in any rule head. Thus, it is determined at compile-time that every *edb*-relation is either monotonically non-decreasing or non-increasing.

Run-Time Δ-Monotonicity: This is more flexible than the previous class since a program may contain both, rules which insert into and delete from the same relation. 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. This is accomplished by the following rules for $R \in \mathit{edb}(\mathbf{R})$:

$$
\begin{array}{l}
[S{+}1]\,\mathsf{inc}{:}R \leftarrow\ [S]\,\mathsf{ins}{:}R(\bar{X}), \neg\, R(\bar{X}).\\
[S{+}1]\,\mathsf{dec}{:}R \leftarrow\ [S]\,\mathsf{del}{:}R(\bar{X}), R(\bar{X}).\\
[S{+}1]\,\mathsf{inc}{:}R \leftarrow\ [S]\,\mathsf{inc}{:}R, \neg\, \mathsf{EOT}.\\
[S{+}1]\,\mathsf{dec}{:}R \leftarrow\ [S]\,\mathsf{dec}{:}R, \neg\, \mathsf{EOT}.\\
[S]\,\mathsf{abort} \leftarrow\ [S]\,\mathsf{inc}{:}R, \mathsf{dec}{:}R.
\end{array}
$$

Tuple Δ-Monotonicity: This property is also checked at run-time, but an even finer granularity at the level of individual tuples is used. This is the most precise approximation for enforcing termination by preventing oscillation of tuples. The protocol relations $\mathsf{insd}{:}R, \mathsf{deld}{:}R \in \mathit{ctl}(\mathbf{R})$ defined above can be used to abort the current transaction if tuple Δ-monotonicity is violated:

$$
\begin{array}{l}
[S]\,\mathsf{abort} \leftarrow\ [S]\,\mathsf{insd}{:}R(\bar{X}), \mathsf{del}{:}R(\bar{X}).\\
[S]\,\mathsf{abort} \leftarrow\ [S]\,\mathsf{deld}{:}R(\bar{X}), \mathsf{ins}{:}R(\bar{X}).
\end{array}
$$

12.5.4 Expressive Power and Normal Forms

Recall from Section 12.4.4 that two programs are called *transaction equivalent* if they produce the same final state for all databases $\boldsymbol{D}$, *evolution equivalent* if additionally the sequence of intermediate states is the same, and *transition equivalent* if the class of database mappings definable by a single transition from state $[n]$ to $[n{+}1]$ are the same (w.l.o.g., we may assume $n = 0$).

NF-Statelog, P-Statelog. *NF-Statelog* is the class of Statelog programs in *normal form*, i.e., every rule is of the form

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

Rules of type (P) are 1-progressive and correspond to a special variant of noninflationary Datalog$^\neg$ with production rule semantics, rules of type (L) are local rules.

P-Statelog is the subclass of NF-Statelog programs comprising only rules of type (P). E.g., P_1 from Example 3 – including the frame rule for e – is a NF-Statelog program; P_2 additionally is a P-Statelog program.

One can show that for every Statelog program there is an evolution equivalent (and thus transaction equivalent) NF-Statelog program.[5]

Furthermore, one can also find a transaction equivalent P-Statelog program, but in general no evolution equivalent P-Statelog program. This is due to the fact that P-Statelog is less expressive wrt. transitions since it is impossible to define recursion *within* a state:

Theorem 5 (Statelog $\equiv_\infty$ NF-Statelog $\equiv_\$$ P-Statelog)
For every Statelog program there is an evolution equivalent NF-Statelog program and a transaction equivalent P-Statelog program.

Since the transformation from Statelog programs into normal form can be done in such a way that the database evolution remains the same, we may assume w.l.o.g. that Statelog programs are in normal form. In contrast, P-Statelog is less expressive wrt. single transitions (cf. Table 12.3), so intermediate steps cannot always be preserved. In particular, the program P_1 from Example 3 computes the nodes reachable from a given node within a single state which cannot be achieved by P-Statelog rules.

Production Rules as Statelog Rules. It is well-known that Datalog$^\neg$ with noninflationary semantics expresses the class of While queries [AV91; AHV95], and inflationary Datalog$^\neg$ expresses the so-called Fixpoint queries.[6] There are several variants of noninflationary Datalog$^\neg$. The simplest is to compute the fixpoint of a Datalog$^\neg$ program by iterating the immediate consequence operator. This semantics is noninflationary, since tuples derived in round i are not automatically transferred to round $i+1$. As a consequence, the semantics is *partial* because a fixpoint does not always exist. In [EF95] this language is called *P-Datalog*. The evaluation of a P-Datalog rule $head \leftarrow body$ precisely corresponds to the evaluation of the P-Statelog rule $[S+1]\, head \leftarrow [S]\, body$. It follows that P-Statelog is transaction equivalent to P-Datalog and thus expresses the While queries.

Datalog$^{\neg\neg}$ is a noninflationary Datalog variant where negation in the head is interpreted as deletion. This is a special case of the Statelog framework described above. In particular, the Datalog$^{\neg\neg}$ rule $\neg H \leftarrow body$ corresponds to the user-defined rule $\mathsf{del}{:}H \leftarrow body$.

Similarly, the inflationary semantics of Datalog$^\neg$, called *I-Datalog* in [EF95], can be obtained in Statelog as follows: For every Datalog$^\neg$ rule $head \leftarrow body$ there is a Statelog rule $[S+1]\, head \leftarrow [S]\, body$. Additionally, for every relation $R(\bar{X})$ a frame rule $[S+1]\, R(\bar{X}) \leftarrow [S]\, R(\bar{X})$ is used which makes the semantics inflationary. Clearly, the resulting language *I-Statelog* is equivalent to I-Datalog and thus expresses the Fixpoint queries.

Although each of the production rule semantics for Datalog just described corresponds to a certain subclass of Statelog programs, there is a subtle difference between them: In Statelog programs, state transitions are part of the logical language since the reference to different states is made explicit. In contrast, in Datalog with production rule semantics, the reference to different states is *implicit* and hidden in the procedural fixpoint semantics. In this sense, Statelog programs are more "declarative" since states are incorporated into the language.

XY-Datalog. In [Zan93], XY-Datalog has been proposed as a unified framework for active and deductive rules. XY-Datalog rules can be regarded as a certain class of Statelog rules. Roughly speaking, *X-rules* correspond to nonrecursive local rules of Statelog, while *Y-rules* correspond to 1-progressive rules.

Summary of Expressiveness Results. Figure 12.3 summarizes the expressiveness results for several Statelog variants: *G-Statelog* is the class of guarded Statelog programs, Δ-*Statelog* are the Δ-monotonic programs described in Section 12.5.3. Note that although all three presented variants have the same expressive power, some programs may be much easier to express using runtime Δ-monotonic rules than by using compile-time Δ-monotonic rules, similar for tuple Δ-monotonic rules. S-Datalog is the class of queries expressible by *stratified Datalog*$^\neg$ programs, and is strictly contained in Fixpoint. $\exists$FO is the existential fragment of first-order logic FO. Both P-Statelog and XY-Datalog do not allow recursion within a state, so their limited transition expressiveness does not allow to express inductive concepts like transitive closure in a single transition. WF-Statelog is defined like Statelog with the exception that the well-founded semantics is used and rules need not be state-stratified.

12.6 CONCLUSION

Active rules extend the traditional passive database technology and are a powerful programming paradigm with a large number of application areas. While an increasing number of systems becomes available and active rule program-

Language	Equivalence wrt. Transitions ($\equiv_1$)	Transactions ($\equiv_\$$)
G-Statelog	S-Datalog	S-Datalog
Δ-Statelog	S-Datalog	Fixpoint
I-Statelog	$\exists$FO	Fixpoint
P-Statelog	$\exists$FO	While
XY-Datalog	FO	While
NF-Statelog	S-Datalog	While
Statelog	S-Datalog	While
WF-Statelog	Fixpoint	While

Figure 12.3 Summary of expressiveness results

ming is carried out in real world applications, theoretical foundations of active rules are still rare.

In the first part of this chapter, after briefly reviewing the terminology and basic aspects of active rules, we have discussed several state-of-the-art approaches related to theoretical aspects and foundations of active rules. In the second part, we have elaborated on a state-oriented logical framework which integrates active and deductive rules. The underlying core language *Statelog* precisely specifies the meaning of a set of active rules and allows to investigate fundamental properties like termination and expressive power. Although the presented execution model is relatively straightforward and corresponds to flat transactions dealing only with immediate and deferred coupling on the statement-level, it captures many essential features of active rules including composite events. However, some features like chronicle contexts of composite events cannot be expressed directly, but require certain extensions like event queues or timestamping, as presented in [MZ95]. Also, low-level procedural constructs like *before* and *instead of* triggers, which do not lend themselves to a logical semantics, are not covered. While the proposed framework enjoys the desirable feature of a deterministic semantics, it is sometimes useful to consider nondeterministic extensions, in particular to model existing nondeterministic systems. A possible extension is to use the choice construct of [SZ90] which can be integrated seamlessly with a state-oriented language like Statelog or XY-Datalog (see e.g., [GGSZ97]). Another limitation is that the presented framework does not support complex nested transactions, which is necessary to model decoupled rule execution and concurrent transactions. However, by using an extended Statelog framework as the one described in [LML96], which

is based on a hierarchical state space instead of a linear one, these concepts can also be modeled in a state-oriented logical way.

Notes

1. In fact [SD95] report that weak performance is one of the main reasons why users are reluctant to use active rules in large scale applications.

2. For efficiency reasons however, the distinction between events and conditions may be crucial in practice. A conceptual difference is that the event specifies *when* to check a rule, while the condition specifies *what* to check.

3. Intuitively, a program is *stratified* if no relation definition negatively depends on itself; thus, there is "no recursion through negation".

4. As usual, database instances are finite structures, hence we may assume that U is the finite active domain of relevant elements. U may also include finitely many new elements to model insertions of new constants into the database.

5. The transformed program will contain additional auxiliary relations which are ignored in the definition of evolution equivalence (cf. Section 12.4.4).

6. Called PFP and IFP in finite model theory, respectively.

References

[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.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[AS91] S. Abiteboul and E. Simon. Fundamental Properties of Deterministic and Nondeterministic Extensions of Datalog. *Theoretical Computer Science*, 78(1):137–158, 1991.

[AV91] S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[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*, 20(1):3–41, 1995.

[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.

[BCP95] E. Baralis, S. Ceri, and S. Paraboschi. Improving Rule Analysis by Means of Triggering and Activation Graphs. In Sellis [Sel95], pp. 165–181.

[BCW93] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [TCG+93], pp. 294–320.

[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.

[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.

[BK94] A. J. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[BL96] C. Baral and J. Lobo. Formal Characterization of Active Databases. In Pedreschi and Zaniolo [PZ96], pp. 175–195.

[BW94] E. Baralis and J. Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Intl. Conference on Very Large Data Bases*, pp. 475–486, Santiago, Chile, 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*, pp. 132–143, Santiago, Chile, 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.

[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.

[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*, 20(2):149–186, 1995.

[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*, pp. 606–617, Santiago de Chile, 1994.

[Coh89] D. Cohen. Compiling Complex Database Transition Triggers. In J. Clifford, B. Lindsay, and D. Maier, editors, *ACM SIGMOD Intl. Conference on Management of Data*, pp. 225–234, Portland, 1989.

[CPM96] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating Triggers and Declarative Constraints in SQL Database Sytems. In *Intl. Conference on Very Large Data Bases*, pp. 567–578, Mumbai (Bombay), India, 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 SIGMOD Intl. Conference on Management of Data*, 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.

[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.

[DPG91] O. Díaz, N. W. Paton, and P. M. D. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. In *Intl. Conference on Very Large Data Bases*, pp. 317–326, Barcelona, Spain, 1991.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer, 1995.

[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.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions on Database Systems*, 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.

[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.

[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*, 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 SIGMOD Intl. Conference on Management of Data*, 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*, 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.

[Han96] E. N. Hanson. The Design and Implementation of the Ariel Active Database Rules System. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):157–172, February 1996.

[ISO95] ISO/ANSI Working draft. *Database Languages – SQL3*, October 1995.

[KC95] S.-K. Kim and S. Chakravarthy. A Confluent Rule Execution Model for Active Databases. Technical Report UF-CIS-TR-95-032, University of Florida, 1995. http://www.cis.ufl.edu/~sharma.

[KdMS90] G. Kiernan, C. de Maindreville, and E. Simon. Making Deductive Databases a Practical Technology: A Step Forward. In *ACM SIGMOD Intl. Conference on Management of Data*, pp. 237–246, 1990.

[KLS92] M. Kramer, G. Lausen, and G. Saake. Updates in a Rule-Based Language for Objects. In *Intl. Conference on Very Large Data Bases*, pp. 251–262, Vancouver, 1992.

[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. Conf. 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.

[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 *16th ACM Symposium on Principles of Database Systems (PODS)*, pp. 217–227, Tucson, Arizona, 1997.

[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.

[LS87] U. W. Lipeck and G. Saake. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, pp. 255–269, 1987.

[Min96] J. Minker. Logic and Databases: a 20 Year Retrospective. In Pedreschi and Zaniolo [PZ96], pp. 3–57.

[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.

[MZ97] I. Motakis and C. Zaniolo. Temporal Aggregation in Active Database Rules. In *ACM SIGMOD Intl. Conference on Management of Data*, pp. 440–451, Tucson, Arizona, 1997.

[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.

[PV97] P. Picouet and V. Vianu. Expressiveness and Complexity of Active Databases. In F. Afrati and P. Kolaitis, editors, *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 155–172, Delphi, Greece, 1997. Springer.

[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.

[RH94] K. Ramamohanarao and J. Harland. An Introduction to Deductive Database Languages and Systems. *The VLDB Journal*, 3(2):107–122, April 1994.

[SD95] E. Simon and A. K. Dittrich. Promises and Realities of Active Database Systems. In *21st Intl. Conference on Very Large Data Bases*, pp. 642–653, Zurich, Switzerland, 1995.

[Sel95] T. K. Sellis, editor. *2nd Intl. Workshop on Rules in Database Systems (RIDS)*, number 985 in LNCS, Athens, Greece, 1995. Springer.

[Sin95] M. P. Singh. Semantical Considerations on Workflows: An Algebra for Intertask Dependencies. In *Intl. Workshop on Database Programming Languages*, electronic Workshops in Computing, Gubbio, Italy, 1995. Springer.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres Next Generation Database Management System. *Communications of 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.

[SP97] P. Sampaio and N. Paton. Deductive Object-Oriented Database Systems: A Survey. In A. Geppert and M. Berndtsson, editors, *Proc. of the 3nd Intl. Workshop on Rules in Database Systems (RIDS)*, number 1312 in LNCS, pp. 1–19, Skövde, Sweden, 1997.

[SZ90] S. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *9th ACM Symposium on Principles of Database Systems*, pp. 205–217, 1990.

[TCG+93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases.* Benjamin/Cummings, 1993.

[vEK76] M. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[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.

[Via97] 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 SIGMOD Intl. Conference on Management of Data*, pp. 259–270, 1990.

[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.

[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.

[ZH90] Y. Zhou and M. Hsu. A Theory for Rule Triggering Systems. In *Intl. Conf. on Extending Database Technology*, pp. 407–421, 1990.

action, 390
    external, 408
    internal, 391
active database, 390
active rule, 390, 407

complexity, 393, 397, 411
composite event, 391, 408, 411
condition, 390, 391
confluence, 393, 395, 407
coupling mode, 392, 398, 410
cycle
    local, 403
    negative, 403

database evolution, 400–402, 404
Datalog, 394
    Datalog$^{\neg}$, 394
    Datalog$^{\neg\neg}$, 394, 395, 413
    Datalog$_{1S}$, 396, 400, 402, 411
    inflationary Datalog$^{\neg}$, 394, 413
    noninflationary Datalog$^{\neg}$, 395, 413
    stratified Datalog$^{\neg}$, 406, 414
    XY-Datalog, 400, 414, 415
delta relation, 391, 396, 397, 403, 407, 408

ECA rule, 390
equivalence
    evolution, 404, 412
    transaction, 404, 412
    transition, 404, 412
event, 390
    algebra, 391
    composite, 391, 408, 411
    external, 390, 391, 407–409
    internal, 390, 391
event consumption mode, 391
expressive power, 393, 412

frame rule, 402, 403, 405, 409, 414

Herbrand base, 402

leap, 401, 403
local stratification, 403

net effect, 408, 409

parameter context, 391, 411
production rule, 391, 392, 394, 413
progressiveness, 401, 412

relation
    control, 408

delta, 391, 396, 397, 403, 407, 408
protocol, 408, 409
rule
Δ-monotonic, 411, 414
1-progressive, 402, 413, 414
guarded, 411, 414
local, 402, 413, 414
progressive, 402

semantics
fixpoint, 394
stable, 395
state-stratified, 403
stratified, 395
well-founded, 395
set-oriented, 392, 395, 407
snapshot, 402, 404
view, 402
state
final, 400, 404
initial, 400
intermediate, 400
term, 401
variable, 401
Statelog, 401
Δ-Statelog, 414
G-Statelog, 414
I-Statelog, 414
interpretation, 402
NF-Statelog, 412
normal form, 412
P-Statelog, 413

temporal domain, 401
termination, 393, 404–406, 411
timestamp view, 402
transaction, 400, 404
transition, 400, 404
trigger, 391, 399
tuple-oriented, 392, 395, 407