

Handling Termination in a Logical Language for Active Rules

Bertram LUDÄSCHER Georg LAUSEN

E-mail: {ludaesch,lausen}@informatik.uni-freiburg.de

Abstract. *Statelog* is a Datalog extension integrating the declarative semantics of deductive rules with the possibility to define updates in the style of active and production rules. The language is surprisingly simple, yet captures many essential features of active rules. After reviewing the basics of active rules, production rules, and deductive rules, we elaborate on the problem of handling rule termination in the context of *Statelog*: It is undecidable whether a *Statelog* program terminates for *all* databases, and PSPACE-complete for a *given* database. The latter can be accomplished *within* the logical language: for every *Statelog* program P , there is a terminating program P^\downarrow which decides for any given database \mathcal{D} , whether $P \cup \mathcal{D}$ terminates.

Key words: active databases, deductive databases, production rules, termination

1. Introduction. 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. [28]). 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:

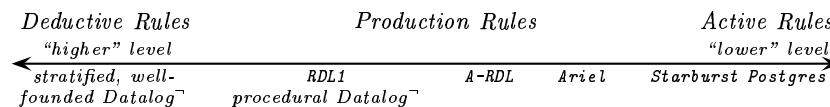


Fig. 1. Spectrum of database rule languages (adapted from [38])

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 events. Conceptually, most rule languages for active database systems (ADBs) 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

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.

1.1. Properties of Rules. Although there is a great variety of rule semantics, the following fundamental properties come up repeatedly and are of practical and theoretical importance.

- *Termination* is arguably the most crucial property of rule execution. 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.

- The *expressive power* of a database language is the class of database transformations expressible in the language, i.e., the class of mappings $inst(\mathbf{R}) \rightarrow inst(\mathbf{R})$ 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

¹ “The unstructured, unpredictable, and often nondeterministic behavior of rule processing can become a nightmare for the database rule programmer” [4].

is still interesting to investigate the impact of certain language constructs on expressive power.

- 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.

In the sequel, we briefly review the basics of active rules, production rules, and deductive rules, respectively. We then introduce *Statelog* (= *States*+*Datalog*), an extension of *Datalog* which allows to reconcile the seemingly discordant paradigms of active and deductive rules. The language captures many essential features of active rules, yet is surprisingly simple; the key idea is to incorporate an explicit notion of state into deductive rules. Finally, we investigate the problem of termination in the context of *Statelog*.

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

on $\langle event \rangle$ **if** $\langle condition \rangle$ **then** $\langle action \rangle$.

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*.

2.1. 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*. The following constructors for composite events are frequently used in active databases (cf. [7]):

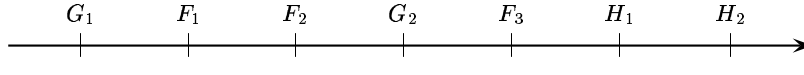
- The *disjunction* ($E_1 \mid E_2$) occurs when E_1 or E_2 occurs.
- The *sequence* ($E_1 ; E_2$) occurs when E_2 occurs provided E_1 has already occurred.

- 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*² $(E_2 \in [E_1; E_3])$ 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* (**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 `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.

2.1.1. 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 [7, 8].

Example 1 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, 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

²Called *aperiodic event* $A(E_1, E_2, E_3)$ in [7] (to contrast it with periodic events).

way. Apart from the unrestricted context, the following parameter contexts have been proposed [7]:

- *Recent*: In this context, only the most recent occurrences of constituent events are used; all previous occurrences are lost. In Example 1, E will be raised twice: for the constituent events $\{G_2, F_3, H_1\}$ and for $\{G_2, F_3, H_2\}$.

- *Chronicle*: In this context, events are consumed in their chronological order. In a sense, this corresponds to a first-in-first-out strategy. In Example 1, 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 $\{G_1, F_1, F_2, G_2, F_3, H_1\}$.

2.2. 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.

2.3. 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.

2.4. Execution Models. The basic execution model of active rules is similar to the *recognize-act cycle* of production rule languages like OPS5:

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.

There are now a number of ADB prototypes which are based on the relational model, e.g., A-RDL, Ariel, Starburst, or an object-oriented model, e.g., HiPAC, Chimera, ODE (cf. [12, 20]).

3. 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 [6] has been used in the AI community since the seventies. 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.³ 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., [3]) 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 developed for (definite) logic programs into an inflationary operator T_P^+ by keeping all tuples which have been derived before, i.e., $T_P^+(\mathcal{I}) := \mathcal{I} \cup T_P(\mathcal{I})$ where \mathcal{I} is the set of ground atoms derived in the previous round. Starting with a set of facts \mathcal{I} (the initial 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 \mathcal{I} . If both A and $\neg A$ are inferred in the same round, several options exist: 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 [37]. 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 [2]. On the other hand, nondeterminism can be a powerful programming paradigm which increases the (theoretical and practical) expressiveness of a language [3, 15].

³For efficiency reasons however, the distinction between events and conditions may be crucial in practice.

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

Example 2 Under the inflationary semantics, the program

$$\begin{aligned} \text{tc}(X,Y) &\leftarrow e(X,Y). \\ \text{tc}(X,Y) &\leftarrow e(X,Z), \text{tc}(Z,Y). \\ \text{non-tc}(X,Y) &\leftarrow \neg \text{tc}(X,Y). \end{aligned}$$

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 4.

RDL1 [17] is a deductive database language with production rule semantics; a rule algebra is used as an additional control mechanism. A-RDL [34] extends RDL1 by active database concepts, in particular delta relations and a module concept.

4. Deductive 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* [5, 36, 13] are now generally accepted as intended and intuitive semantics of logic programs with negation. For stratified programs like the one in Example 2, all three semantics coincide.⁴ 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,

⁴A program is *stratified* if no relation definition negatively depends on itself; thus, there is “no recursion through negation”.

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⁺⁺ and noninflationary Datalog⁺.

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''*, etc. 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. [32, 28, 33]).

5. Statelog is a Datalog extension which integrates the declarative semantics of deductive rules with the possibility to define updates in the style of production rules and active rules [25, 20]. Using Statelog as a formal framework, fundamental properties of active rules like termination, confluence and expressive power can be studied [23]. The framework does not account for all facets of active rules which may be useful in practice (like, e.g., SQL3’s *before* and *instead of* triggers, or tuple-level execution), but covers many essential features including immediate and deferred execution and composite events.

The basic execution model of Statelog is illustrated in Fig. 2: s_0 denotes the *initial state* of the database and may be queried using local Statelog rules. 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

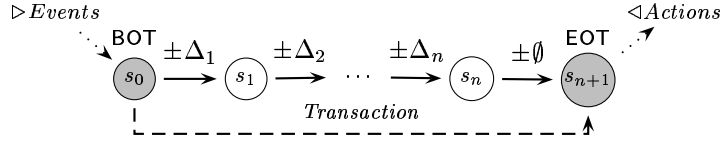


Fig. 2. Database evolution: transaction and transitions

is used. Other, more general models are possible, for example branching time or a hierarchical state space [25].

5.1. 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 . Since the database evolves over a linear state space, S may only be bound to some $n \in \mathbb{N}_0$.

A *Statalog database* $\mathcal{D}[k]$ at state $k \in \mathbb{N}_0$ is a finite set of facts of the form $[k]p(x_1, \dots, x_n)$ 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 \mathcal{D} .

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

$$[S+k_0]H \leftarrow [S+k_1]B_1, \dots, [S+k_n]B_n$$

where the head H is a Datalog atom, B_i are Datalog literals (atoms A or negated atoms $\neg A$), and $k_i \in \mathbb{N}_0$. If several literals share the same state term $[S+k]$, then $[S+k]$ can be “factored out”: e.g., the body $[S]B_1, [S+1]B_2, [S+1]B_3$ may be abbreviated as $[S]B_1, [S+1]B_2, B_3$.

We require that Statelog rules are *progressive*, since the current state cannot be defined in terms of future states, nor should it be possible to change past states: A rule r is called *progressive*, if $k_0 \geq k_i$ for all $i = 1, \dots, n$. If $k_0 = k_i$ for all $i = 1, \dots, n$, then r is called *local* and corresponds to the usual query rules. On the other hand, if $k_0 = 1$ and $k_i = 0$ for all $i \geq 1$, r is called *1-progressive* and denotes a *transition rule*. A *Statalog program* is a finite set of progressive Statelog rules.

5.2. Semantics. Every Statelog program P can be viewed as a logic program P^* , by defining $([S+k]p(t_1, \dots, t_n))^* := p(S+k, t_1, \dots, t_n)$ and extending $()^*$ to literals and rules in the obvious way. Thus, the declarative semantics developed for deductive rules can be applied directly to Statelog.

Here, we adopt the *state-stratified* semantics as the canonical model $\mathcal{M}_{P \cup \mathcal{D}}$ of a Statelog program P with database \mathcal{D} . P is called state-

stratified, if there are no negative cyclic rule dependencies *within a single state* [24]. 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 $(r_1 \xrightarrow{l, (\neg)} r_2) \in \mathcal{G}_P$ 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 $\sum_{(r_1 \xrightarrow{l, (\neg)} r_2) \in C} l = 0$) involving a negative edge. This notion is closely related to *XY-stratification* [39] and *ELS-stratification* [16]. Together with the requirement of progressiveness, state-stratification implies local stratification [23]:

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

5.3. User-Defined vs. System-Defined Rules. In the Statelog core language presented above there is no distinction between user-defined and system-defined rules. However, an active database system should provide the user with a predefined intuitive programming “environment” which takes care of low-level aspects of the execution model like transaction control and semantics of primitive update requests. This is achieved by partitioning the relation schema \mathbf{R} into different sets and providing a set of system-defined rules for certain relations:

$$\mathbf{R} = edb(\mathbf{R}) \dot{\cup} idb(\mathbf{R}) \dot{\cup} \triangleright ev(\mathbf{R}) \dot{\cup} \triangleleft act(\mathbf{R}) \dot{\cup} \delta(\mathbf{R}) \dot{\cup} ctl(\mathbf{R}).$$

Here, $edb(\mathbf{R})$ and $idb(\mathbf{R})$ denote the usual *extensionally* and *intensionally* defined relations, respectively. Relations from $\triangleright ev(\mathbf{R})$ represent *external events* of interest which are monitored by the system. Consequently, external events can only occur in rule bodies. *External actions* are defined by the relations from $\triangleleft act(\mathbf{R})$ and represent requests to execute certain actions outside the database system. Relations denoting external events and actions can be viewed as special input and output relations, and are prefixed with the symbols “ \triangleright ” and “ \triangleleft ”, respectively.

For every base relation $p \in edb(\mathbf{R})$ there are *delta relations* (short: *deltas*) $del:p, ins:p \in \delta(\mathbf{R})$ denoting update requests to delete resp. insert the corresponding tuples into p . Finally, $ctl(\mathbf{R})$ contains special *control relations* like BOT, EOT, and abort for transaction control, and *protocol relations* $insd:p$ and $deld:p$ (for **inserted** and **deleted**) which store the accumulated *net effect* of a sequence of updates. The latter can be used to

enforce termination (Section 7), or as an auxiliary store holding all necessary information to undo the effect of an aborting transaction.

5.3.1. User-Defined Rules. To relieve the programmer from handling states explicitly, we require that user-defined rules are local, so state terms may be omitted. Depending on the relation symbol p in the head of a user-defined rule, different rule types can be distinguished, in particular *query rules* ($p \in \text{idb}(\mathbf{R})$), *update rules* ($p \in \delta(\mathbf{R})$), and *control rules* ($p \in \text{ctl}(\mathbf{R})$). Restricting user programs to local rules results in no loss of expressive power since, by using delta relations, non-local rules can be simulated by local ones. Particularly, all transactions expressible in Statelog can be expressed using local and 1-progressive rules only [23]. If a program only comprises query rules, termination is guaranteed since the databases never changes. On the other hand, although user-defined update rules are local—together with the above-mentioned frame rules—they can cause progressive recursion and thus oscillating update requests (see Example 3 below).

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

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

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

$$\begin{aligned} [S+1]p(\bar{X}) &\leftarrow [S]\mathbf{ins}:p(\bar{X}), \neg \text{abort}. \\ [S+1]p(\bar{X}) &\leftarrow [S]p(\bar{X}), \neg \mathbf{del}:p(\bar{X}), \neg \text{abort}. \end{aligned}$$

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 $p(\bar{X})$. Using a different set of frame rules, also *deferred* execution can be modeled; similarly, other conflict resolution policies (e.g., priority to deletions) can be easily specified.

Protocol relations $\mathbf{insd}:p, \mathbf{deld}:p \in \text{ctl}(\mathbf{R})$ store the accumulated *net effect* of a sequence of a transaction as long as running holds:

$$\begin{aligned} [S+1]\mathbf{insd}:p(\bar{X}) &\leftarrow [S]\mathbf{ins}:p(\bar{X}), \neg p(\bar{X}), \text{running}. && \% \text{ store net effect...} \\ [S+1]\mathbf{insd}:p(\bar{X}) &\leftarrow [S]\mathbf{insd}:p(\bar{X}), \neg \mathbf{del}:p(\bar{X}), \text{running}. && \% \dots \text{ of insertions} \\ [S+1]\mathbf{deld}:p(\bar{X}) &\leftarrow [S]\mathbf{del}:p(\bar{X}), p(\bar{X}), \text{running}. && \% \text{ store net effect...} \\ [S+1]\mathbf{deld}:p(\bar{X}) &\leftarrow [S]\mathbf{deld}:p(\bar{X}), \neg \mathbf{ins}:p(\bar{X}), \text{running}. && \% \dots \text{ of deletions} \end{aligned}$$

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

$$\begin{aligned}
 [S] \text{running} &\leftarrow [S] \text{ins}:p(\bar{X}), \neg p(\bar{X}), \neg \text{abort}. \\
 [S] \text{running} &\leftarrow [S] \text{del}:p(\bar{X}), p(\bar{X}), \neg \text{abort}.
 \end{aligned}$$

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

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

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

$$\begin{aligned}
 [S+1] p(\bar{X}) &\leftarrow [S] \text{abort}, \text{deld}:p(\bar{X}). \quad \% \text{ undo deletions} \\
 [S+1] p(\bar{X}) &\leftarrow [S] \text{abort}, p(\bar{X}), \neg \text{insd}:p(\bar{X}). \quad \% \text{ undo insertions} \\
 [S+1] \text{EOT} &\leftarrow [S] \text{abort}. \quad \% \text{ signal end of transaction}
 \end{aligned}$$

5.4. Composite Events. Various kinds of composite events and consumption modes can be expressed in Statelog, as shown in [29] using the closely related language Datalog_{IS}. 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 [22, 10]):

$$\begin{aligned}
 [S] \text{detd}_F(X) &\leftarrow [S] F(X). \\
 [S+1] \text{detd}_F(X) &\leftarrow [S] \text{detd}_F(X). \\
 [S+1] \text{detd}_E(X, Y) &\leftarrow [S] \text{detd}_F(X), [S+1] G(Y).
 \end{aligned}$$

Auxiliary relations 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 first-in-first-out 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 [29]; see [23].

6. Deciding Termination. Recall the basic Statelog execution model depicted in Fig. 2. Given the model $\mathcal{M}_{P \cup \mathcal{D}}$ of a program P with database \mathcal{D} , the *snapshot* $\mathcal{M}_{P \cup \mathcal{D}}[n]$ is the database instance reached after n transitions.

Definition 1 A Statelog program P *terminates for* \mathcal{D} , if for some $n_0 \in \mathbb{N}_0$ and all $k \geq 1$ we have that $\mathcal{M}_{P \cup \mathcal{D}}[n_0] = \mathcal{M}_{P \cup \mathcal{D}}[n_0 + k]$; the least such n_0 is called the *final state*. 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} .

Thus, whether a program terminates always or sometimes is a *compile-time property* of the program, independent of the given database. Conversely, for given a database \mathcal{D} , the question whether P terminates for \mathcal{D} is a *run-time property* of $P \cup \mathcal{D}$.

Example 3 Consider the Statelog user program

$$P : \quad \text{ins:}q \leftarrow \neg q, p. \quad \text{del:}q \leftarrow q, p.$$

where the state term $[S]$ has been omitted. We assume that frame rules for specifying the meaning of deltas have been added to P . Clearly, P diverges for $\mathcal{D}[0] = \{[0] p\}$, since q is repeatedly inserted and deleted. If $\mathcal{D}[0] = \emptyset$, then P terminates for \mathcal{D} , since the rules with p in the body are not applicable.

6.1. Compile-Time Termination. It is well-known that satisfiability and validity of first-order sentences over *finite structures* (i.e. relational databases) is undecidable (*Trakhtenbrot's Theorem* [35]⁵). This implies that most non-trivial compile-time properties become undecidable in languages which allow to encode first-order sentences like, e.g., stratified Datalog. As a consequence, if a language allows to define possibly nonterminating updates which depend on first-order conditions, termination at compile-time becomes undecidable. Thus, the following theorem not only holds for Statelog but for all such languages, in particular XY-Datalog and (noninflationary) Datalog^{¬¬}:

Theorem 2 *Given a Statelog program P , it is undecidable whether P sometimes (always) terminates.*

PROOF (Sketch) Every first-order sentence φ can be translated into an equivalent stratified Datalog program P_φ with a distinguished answer relation ans_φ , s.t. in the stratified model $\mathcal{M}_{P_\varphi \cup \mathcal{D}} \models \text{ans}_\varphi$ iff $\mathcal{D} \models \varphi$. Consider the Statelog program $P'_\varphi := P_\varphi \cup \{ [S+1] q \leftarrow [S] \neg q, \neg \text{ans}_\varphi \}$. Clearly, P'_φ terminates for \mathcal{D} iff $\mathcal{M}_{P_\varphi \cup \mathcal{D}} \models \text{ans}_\varphi$, which in turn holds iff $\mathcal{D} \models \varphi$. Thus, P'_φ sometimes (always) terminates iff φ is satisfiable (valid) in the finite, which is undecidable by Trakhtenbrot's Theorem. ■

⁵For a recent reference see, e.g., [11].

A similar result for Datalog^{¬¬} has been presented in [2]⁶ 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

$$[S+1] p \leftarrow [S] q. \quad [S+1] q \leftarrow [S] p.$$

which oscillates between the states $\{p\}$ and $\{q\}$ given the database $\mathcal{D} = \{[0] p\}$. Interestingly, for negation-free programs the problem becomes decidable, which follows from a result in [9] on universal safety for Datalog_{ns}.

6.2. Run-Time Termination. In Statelog it is easy 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 (provided rules are in *normal form*, i.e., either local or 1-progressive). For a Statelog program P over the schema \mathbf{P} , we can use the following rules for all $p \in \mathbf{P}$:

$$\begin{aligned} [S+1] p^{-1}(\bar{X}) &\leftarrow [S] p(\bar{X}). \\ [S] \text{change} &\leftarrow [S] p^{-1}(\bar{X}), \neg p(\bar{X}). \\ [S] \text{change} &\leftarrow [S] p(\bar{X}), \neg p^{-1}(\bar{X}). \\ [S] \text{fixpoint} &\leftarrow [S] \neg \text{change}. \end{aligned}$$

Consider the converse situation: Is it possible to detect *within the language* that P does not terminate for \mathcal{D} ? Interestingly, this is indeed the case: for every Statelog program P there is a transaction equivalent program P^\downarrow which always terminates. Notice that this does not contradict Theorem 2, since P^\downarrow solves the problem only for a *given* database \mathcal{D} , i.e., at run-time.

The proof for the following theorem [23] is based on the idea of simultaneously evaluating P and a delayed version of P . This idea has been applied earlier in the context of Datalog^{¬¬} under the name *loop-free simulation* [2], and for partial fixpoint logic in [11]. As a generic notation, we write $\mathcal{M}_{P \cup \mathcal{D}}[\$]$ to denote the final database state if P terminates for \mathcal{D} ; otherwise we agree to set $\mathcal{M}_{P \cup \mathcal{D}}[\$] := \emptyset$.

Theorem 3 *For every Statelog program P with schema \mathbf{P} there is a transaction equivalent program P^\downarrow which always terminates. In particular, one can define relations *terminates* and *diverges* in P^\downarrow such that for all databases \mathcal{D} :*

$$\begin{aligned} \mathcal{M}_{P^\downarrow \cup \mathcal{D}}[\$] \models \text{terminates} &\Leftrightarrow \mathcal{M}_{P \cup \mathcal{D}} \text{ terminates for } \mathcal{D}, \\ \mathcal{M}_{P^\downarrow \cup \mathcal{D}}[\$] \models \text{diverges} &\Leftrightarrow \mathcal{M}_{P \cup \mathcal{D}} \text{ does not terminate for } \mathcal{D}. \end{aligned}$$

⁶In [2], the name Datalog^{¬*} is used for Datalog^{¬¬}.

Moreover, for all $p \in \mathbf{P}$ one can define relations p_t, p_f, p_u in P' such that

$$\begin{aligned} \mathcal{M}_{P \downarrow \mathcal{D}}[\$] \models p_t(\bar{x}) &\Leftrightarrow \exists n_0 \forall n \geq n_0 : \mathcal{M}_{P \cup \mathcal{D}} \models [n] p(\bar{x}), \\ \mathcal{M}_{P \downarrow \mathcal{D}}[\$] \models p_f(\bar{x}) &\Leftrightarrow \exists n_0 \forall n \geq n_0 : \mathcal{M}_{P \cup \mathcal{D}} \models [n] \neg p(\bar{x}), \\ \mathcal{M}_{P \downarrow \mathcal{D}}[\$] \models p_u(\bar{x}) &\Leftrightarrow \forall n_0 \exists n, m \geq n_0 : \mathcal{M}_{P \cup \mathcal{D}} \models [n] p(\bar{x}), [m] \neg p(\bar{x}). \end{aligned}$$

Thus P^\downarrow allows to “speak” about (non)termination of $P \cup \mathcal{D}$. In the case of nontermination, the relations p_t , p_u , and p_f can be used to determine the atoms which are *eventually true* in *every*, in *some* (but not every), or in *no state*, respectively.

The proof of Theorem 3 provides a theoretical construction to decide within Statelog whether a program P terminates for \mathcal{D} . Therefore, it is possible 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_u), or which are always true (those in p_t) can be useful in implementing more elaborated strategies. However, since Statelog transactions have the same expressive power as partial fixpoint logic (or, equivalently the language WHILE [1]), one can show [23]:

Theorem 4 *Given a Statelog program P and a database \mathcal{D} , deciding whether P terminates for \mathcal{D} is PSPACE-complete.*

Therefore, run-time detection can be infeasible for general Statelog programs. In the next section an approach for *enforcing* termination is presented, which also yields a tractable class of terminating Statelog programs.

7. Enforcing Termination. In Statelog two kinds of recursion can be distinguished:

- *Local recursion*, (i.e., involving locally recursive rules) 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,
- *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 local rules of Example 3 together with the corresponding frame rules are progressively recursive and lead to an oscillation

of q . Therefore, the basic idea to enforce rule termination is to restrict progressive recursion such that oscillation is avoided.

7.1. Guarded Rules are one possibility, where user-defined update rules are “guarded” by a positive goal of the form $\triangleright e(\bar{X})$:⁷

Definition 2 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 , or **del**: p in the head) is guarded.

According to the execution model in Fig. 2, 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. Many basic updates can be expressed in a concise and intuitive way using guarded rules:

Example 4 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:

$$\begin{aligned} \mathbf{ins}:p(\bar{X}) &\leftarrow \triangleright \text{insert_into_}p(\bar{X}). \\ \mathbf{del}:p(\bar{X}) &\leftarrow \triangleright \text{delete_from_}p(\bar{X}), p(\bar{X}), \varphi(\bar{X}). \\ \mathbf{del}:p(\bar{X}) &\leftarrow \triangleright \text{discard_}p, p(\bar{X}). \end{aligned}$$

The following rules swap the contents of two relations p and q of the same arity, whenever $\triangleright \text{swap_}p_q$ occurs:

$$\begin{aligned} \mathbf{del}:p(\bar{X}), \mathbf{ins}:q(\bar{X}) &\leftarrow \triangleright \text{swap_}p_q, p(\bar{X}). \\ \mathbf{del}:q(\bar{X}), \mathbf{ins}:p(\bar{X}) &\leftarrow \triangleright \text{swap_}p_q, q(\bar{X}). \end{aligned}$$

Here, we have to use frame rules for p and q which ignore conflicting insertions and deletions, i.e., yield a “no-op” in case of conflict.

7.2. 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

⁷The idea of guarded Statelog rules was presented earlier in [19].

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:

7.2.1. 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 3 A Statelog user program P is *compile-time Δ -monotonic* if for every $p \in \text{edb}(\mathbf{P})$: $(\text{ins}:p(\dots) \leftarrow \dots) \in P \Rightarrow (\text{del}:p(\dots) \leftarrow \dots) \notin P$.

Therefore, for every *edb*-relation p of P , $\text{ins}:p$ and $\text{del}:p$ 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.

Notice that we tacitly assume that frame rules (as presented above) 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 $P := \{\text{ins}:p \leftarrow \neg p\}$. P is Δ -monotonic and terminates since the inserted fact p is propagated by frame rules. However, if a system-defined frame rule like

$$[S+1] p \leftarrow [S] p, \neg \text{del}:p.$$

were *not* given, then p would oscillate between *true* and *false*: if $\neg p$ holds in $[n]$, then $\text{ins}:p$ is derived, so $[n+1] p$ holds. Thus, $\neg p$ is false in $[n+1]$, preventing the derivation of $[n+1] \text{ins}:p$. Therefore, p becomes false again at $[n+2]$ (since there is no frame rule for p), etc.

7.2.2. 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 $p \in \text{edb}(\mathbf{P})$ it contains the rules:

⁸A preliminary definition of essentially the same idea can be found in [24].

$$\begin{aligned}
[S+1] \mathbf{inc}:p &\leftarrow [S] \mathbf{ins}:p(\bar{X}), \neg p(\bar{X}). \\
[S+1] \mathbf{dec}:p &\leftarrow [S] \mathbf{del}:p(\bar{X}), p(\bar{X}). \\
[S+1] \mathbf{inc}:p &\leftarrow [S] \mathbf{inc}:p, \neg \text{EOT}. \\
[S+1] \mathbf{dec}:p &\leftarrow [S] \mathbf{dec}:p, \neg \text{EOT}. \\
[S] \mathbf{abort} &\leftarrow [S] \mathbf{inc}:p, \mathbf{dec}:p.
\end{aligned}$$

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, a request to insert into and a request 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 $\mathbf{insd}:p$ and $\mathbf{deld}:p$ 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

$$\begin{aligned}
[S] \mathbf{abort} &\leftarrow [S] \mathbf{insd}:p(\bar{X}), \mathbf{del}:p(\bar{X}). \\
[S] \mathbf{abort} &\leftarrow [S] \mathbf{deld}:p(\bar{X}), \mathbf{ins}:p(\bar{X}).
\end{aligned}$$

for every $p \in \text{edb}(\mathbf{P})$. 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 5 Efficient access to intensional relations like tc in Example 2 is obtained by *materializing* tc in some *edb*-relation, say mtc , thereby avoiding the need to compute tc on demand:

$$\begin{aligned}
r_1: \quad \mathbf{del}:mtc(X,Y) &\leftarrow \triangleright \text{discard_mtc}, mtc(X,Y). \\
r_2: \quad \mathbf{ins}:mtc(X,Y) &\leftarrow \triangleright \text{materialize_mtc}, e(X,Y). \\
r_3: \quad \mathbf{ins}:mtc(X,Y) &\leftarrow e(X,Z), mtc(Z,Y), \neg mtc(X,Y).
\end{aligned}$$

r_1 empties the materialized view upon the occurrence of an external event $\triangleright \text{discard_mtc}$, whereas r_2 and r_3 materialize the view when $\triangleright \text{materialize_mtc}$ occurs. If $\triangleright \text{materialize_mtc}$ occurs in a separate transaction after $\triangleright \text{discard_mtc}$ 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).

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 Statalog programs: Guarded and Δ -monotonic programs always terminate within **PTIME** [23].

8. Conclusions. Statalog provides a logical framework integrating deductive rules, production rules, and many essential features of active rules. Some features like chronicle contexts of composite events cannot be expressed directly, but require certain extensions like event queues or timestamping. Also, low-level procedural constructs like *before* and *instead of* triggers, which do not lend themselves to a logical semantics, are not covered. Using the Statalog framework, formal properties of active rules, e.g., termination, complexity, and expressive power can be studied (cf. [20, 23]). Here, we elaborated on the problem of handling rule termination: Not surprisingly, termination is undecidable at compile-time. At run-time, deciding termination is **PSPACE**-complete and can be accomplished within the language. Guarded and Δ -monotonic rules constitute an efficient (**PTIME**-computable) class of terminating programs.

It should be noted that the basic ideas of Statalog are orthogonal to the underlying data model and, thus, are not restricted to relational databases. Indeed, Statalog ideas have been used in [27, 26] to extend the deductive object-oriented database language F-logic [18] by states.

REFERENCES

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
2. S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of datalog. *Theoretical Computer Science*, 78(1):137–158, 1991.
3. S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.
4. 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, Mar. 1995.
5. 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*, pages 89–148. Morgan Kaufmann, 1988.
6. L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.
7. 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, *Proc. Intl. Conference on Very Large Data Bases (VLDB)*, pages 606–617, Santiago de Chile, 1994.

8. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14:1–26, 1994.
9. J. Chomicki. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming*, 25(1):1–31, Oct. 1995.
10. J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.
11. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer, 1995.
12. 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.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. Intl. Conference on Logic Programming (ICLP)*, pages 1070–1080, 1988.
14. A. Geppert and M. Berndtsson, editors. *Proc. of the 3rd Intl. Workshop on Rules in Database Systems (RIDS)*, number 1312 in LNCS, Skövde, Sweden, 1997.
15. 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.
16. 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. [21], pages 91–108.
17. G. Kiernan, C. de Maindreville, and E. Simon. Making deductive database a practical technology: a step forward. In *Proc. ACM Intl. Conference on Management of Data (SIGMOD)*, pages 237–246, 1990.
18. 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.
19. 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, pages 17–30, Klagenfurt, Austria, 1994. Springer.
20. G. Lausen, B. Ludäscher, and W. May. On logical foundations of active rules. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 12. Kluwer Academic Publishers, 1998. to appear.
21. T. W. Ling, A. O. Mendelzon, and L. Vieille, editors. *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1013 in LNCS, Singapore, 1995. Springer.
22. U. W. Lipeck and G. Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, pages 255–269, 1987.
23. B. Ludäscher. *Integration of Active and Deductive Database Rules*. PhD thesis, Institut für Informatik, Universität Freiburg, 1998. to appear.
24. B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proc. 7th Intl. Conference on Management of Data (COMAD)*,

- pages 221–238, Pune, India, 1995. Tata McGraw-Hill.
25. B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. In Pedreschi and Zaniolo [31], pages 196–222.
 26. W. May, B. Ludäscher, and G. Lausen. Well-founded semantics for deductive object-oriented database languages. In F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors, *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, pages 320–336, Montreux, Switzerland, 1997. Springer.
 27. W. May, C. Schleppehorst, and G. Lausen. Integrating dynamic aspects into deductive object-oriented databases. In Geppert and Berndtsson [14].
 28. J. Minker. Logic and databases: a 20 year retrospective. In Pedreschi and Zaniolo [31], pages 3–57.
 29. I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In Ling et al. [21], pages 19–37.
 30. N. W. Paton and M. H. Williams, editors. *Proc. of the 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Workshops in Computing, Edinburgh, Scotland, 1993. Springer.
 31. D. Pedreschi and C. Zaniolo, editors. *Proc. Intl. Workshop on Logic in Databases (LID)*, number 1154 in LNCS, San Miniato, Italy, 1996. Springer.
 32. K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122, Apr. 1994.
 33. P. Sampaio and N. Paton. Deductive object-oriented database systems: A survey. In Geppert and Berndtsson [14], pages 1–19.
 34. E. Simon and J. Kiernan. The a-rdl system. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, chapter 5, pages 111–149. Morgan Kaufmann, 1996.
 35. B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akad. Nauk. SSSR*, 70:569–572, 1950.
 36. A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 1–10, 1989.
 37. V. Vianu. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(I-II):215–259, 1997.
 38. J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum. In Paton and Williams [30].
 39. C. Zaniolo. A unified semantics for active and deductive databases. In Paton and Williams [30], pages 271–287.

B. Ludäscher has received his Diploma in Informatics from the University of Karlsruhe in 1993. Since then he has been a member of the database group headed by Prof. Dr. G. Lausen; first at the University of Mannheim, now at the University of Freiburg, where he has just completed his PhD thesis on the integration of active and deductive database rules. His research interests include rules in databases (active, deductive, object-oriented), logic programming, database theory, and Web query languages.

G. Lausen is head of the research group on Databases and Information Systems at the Albert-Ludwigs-University of Freiburg. He received a Diploma in Industrial Engineering, majoring in Computer Science/Operations Research, in 1978, his Ph.D. in 1982, and his Habilitation (Angewandte Informatik) in 1985 from the University of Karlsruhe. From 1986 to 1987 he was associate professor for Information Technology and its Integration Problems at the Technical University of Darmstadt. From 1987 to 1994 he was professor for Databases and Information Systems at the University of Mannheim and since 1994 at the University of Freiburg.

Georg Lausen has broad research interests in the areas of databases and information systems. His current work focusses on object-oriented and deductive databases, active databases, workflow modelling and analysis, and rule-based mining of the web.