On Active Deductive Databases: The Statelog Approach*

Georg Lausen Bertram Ludäscher Wolfgang May

Institut für Informatik, Universität Freiburg, Germany
{lausen,ludaesch,may}@informatik.uni-freiburg.de

Abstract. After briefly reviewing the basic notions and terminology of active rules and relating them to production rules and deductive rules, respectively, we survey a number of formal approaches to active rules. Subsequently, we present our own state-oriented logical approach to active rules which combines the declarative semantics of deductive rules with the possibility to define updates in the style of production rules and active rules. The resulting language *Statelog* is surprisingly simple, yet captures many features of active rules including composite event detection and different coupling modes. Thus, it can be used for the formal analysis of rule properties like termination and expressive power. Finally, we show how nested transactions can be modeled in Statelog, both from the operational and the model-theoretic perspective.

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 [Min96]. Today, there is a plethora of quite different application areas and semantics for rules. From a bird's-eye view, deductive and active rules may be regarded as two ends of a spectrum of database rule languages:

Fig. 1. Spectrum of database rule languages (adapted from [Wid93])

On the one end of the spectrum, *deductive rules* provide a concise and elegant representation of intensionally defined data. Recursive views and static integrity constraints can be specified in a declarative and uniform way using deductive rules, thereby extending the query capabilities of traditional relational languages like SQL. Moreover, the semantics developed for deductive rules with negation are closely related to languages from the field of knowledge representation and

* **In: *Transactions and Change in Logic Databases*, B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, LNCS 1472, Springer, 1998**

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.

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

Contributions and Overview. In this paper, we introduce to the different rule paradigms in databases in Figure 1, and survey a number of formal and logical approaches to active rules. We then present a specific state-oriented logical approach to active rules called *Statelog*, which yields a precise formal semantics for active rules. It can be shown that certain production rules and deductive rules are special cases of Statelog rules, and that many (but not all) features of active rules like composite event detection and different coupling modes can be specified in Statelog. Thus, Statelog can serve as a unified logical framework for active and deductive rules, in which fundamental properties of rules like termination behavior, complexity, and expressiveness can be studied in a rigorous way. The main technical contribution of this paper is the continuation and refinement of [LML96], where a model-theoretic Kripke-style semantics for Statelog in the context of a nested transaction model has been presented.

The paper is organized as follows: In Section 2, we first introduce the basic terminology and features of active rules, and then briefly relate them to production rules and deductive rules, respectively. Section 3 is a short survey on some formal approaches to active rules. In Section 4, we introduce *Statelog*, a state-oriented extension of Datalog and summarize the main results. A key idea of the approach is to add a state argument to every predicate: for example, $[s]\,p(x,y)$ intuitively means that $p(x,y)$ holds in state $[s]$ (this idea has come up several times; see Section 3.2). Starting from a simple model for flat transactions, the framework is extended subsequently to incorporate nested transactions (Section 5). Section 6 develops an abstract, conceptual model for Statelog with nested transactions by a model-theoretic Kripke-style semantics. Some concluding remarks are given in Section 7.

2 Rules in Databases

Rules in database programming languages come in many different flavors. In this section, we discuss language issues of the above-mentioned rule spectrum and highlight differences between the paradigms.

2.1 Active Rules

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

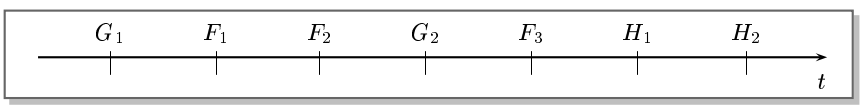
on ⟨*event*⟩ **if** ⟨*condition*⟩ **then** ⟨*action*⟩.

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*; typical operators are disjunction ($E_1|E_2$), sequence ($E_1;E_2$), conjunction (E_1,E_2), etc. (cf. [CKAK94,Sin95]). Alternatively, logics like past temporal logic (see e.g., [LS87,Cho95b]) may be used for the specification of composite events. 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].

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 *parameter contexts*, which were introduced for the SNOOP algebra in [CKAK94,CM94]. In order to illustrate the different parameter contexts, consider the composite event $E := ((F,G);H)$, which occurs if H occurs after both F and G have occurred. Assume 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.

Different parameter contexts are motivated by applications where constituent events should be consumed by the composite event in a certain way. The following parameter contexts have been proposed [CKAK94]:

- *Recent:* In this context, only the most recent occurrences of constituent events are used; all previous occurrences are lost. In the above event history, 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: 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 $\{G_1, F_1, F_2, G_2, F_3, H_1\}$.
- *Unrestricted:* In this context, constituent events are not consumed, but are reused arbitrarily many times. For the above history, E is reported for all twelve possible combinations of $\{F_j, G_k, H_l\}$.

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.

Systems. Most of the active database systems provide a low-level, *procedural* semantics of rules; see e.g. [WC96a] for an overview on a number of systems. Early precursors of active rules have been introduced in CODASYL, System R, and OPS5. More recent systems include Postgres, Starburst, Ariel, Heraclitus, ODE, and SAMOS. HiPAC has been influential by establishing the ECA-rule paradigm; follow-on projects are Sentinel [CKAK94] (with its powerful event specification language SNOOP) and REACH. Regarding commercial systems, the current SQL3 proposal offers so-called *declarative constraints* used specifically for maintaining referential integrity, and general-purpose *triggers* [CPM96,ISO97].

A-RDL [SK96] is closely related to deductive databases: intensional relations are defined by means of deductive rules. Delta relations record the net effect of changes to *edb*-relations during execution of a transaction. Active behavior is encoded via rules in an if-then style.

Chimera [CFPT96] distinguishes between declarative and procedural expressions: Declarative expressions are used in query primitives, integrity constraints,

and view declarations; transactions are specified using procedural expressions for actions and declarative expressions in conditions.

2.2 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. In particular, the production rule language OPS5 [BFKM85] 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.[1] 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-based 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 (*I-Datalog*) 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*$^{\neg\neg}$[2] 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 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 I-Datalog termination is guaranteed, this is no longer the case of Datalog$^{\neg\neg}$: 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].

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

[2] Another noninflationary semantics called *P-Datalog* is obtained by only keeping the newly derived tuples in each iteration; see also Section 4.5.

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 2.3.

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.

2.3 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 and have come up with now well-established solutions: The *stratified*, *well-founded*, and *stable semantics* [ABW88,VG89,GL88] are generally accepted as intended and intuitive semantics of logic programs with negation. 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 rule definitions. 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$^{\neg}$.

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

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, Coral, FLORID, Glue-Nail, LDL, LOLA, and XSB-Prolog (cf. [RH94,Min96,SP97]).

3 Formal Approaches to Active Rules

Whereas the meaning of deductive rules is based on solid logical foundations, the meaning of the more low-level and operationally intricate active rules is often hard to understand and predict—especially, if the semantics is only given informally. This has lead to numerous research towards formal foundations of active rules. In the sequel, we discuss some of these approaches; due to lack of space and the focus on logic-based approaches, we can only provide a rough and necessarily incomplete summary.

3.1 Analysis of Rule Properties

Although there is a great variety of execution models for active rules, certain fundamental properties like termination and complexity come up repeatedly and have been studied in the context of the respective execution models:

Termination, Confluence, and Determinism. [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].

Expressive Power and Complexity. [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 **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.

3.2 Logic-Based Formalizations of Rule Semantics

Whereas the above-mentioned works focus on analysing rule properties in some specific execution model, a lot of research aims at formalizing and characterizing the semantics of active rules in the first place. Once a formal model has been established, abstract properties like termination or expressiveness can be studied.

Situation Calculus Based. In [BL96,BLT97] a language $\mathcal{L}_{active}$ for active rules is developed, which allows to formalize and reason about the behavior of active rules. The language borrows from $\mathcal{L}_1$ [BGP97], an extension of the action description language $\mathcal{A}$ [GL93] used for modeling actual and hypothetical actions and situations, which in turn is based on the situation calculus. The main constructs of $\mathcal{L}_{active}$ are *causal laws* describing which fluents are added or deleted by an action, *executability conditions* stipulating when actions can be executed, and *active rules* defining a triggering event, an evaluation mode, a conjunctive precondition, and a sequence of actions. The automaton-based semantics of $\mathcal{L}_{active}$ uses transition diagrams with states (labeled by sets of fluents) and transitions (labeled by actions) to specify the meaning of an active database description in $\mathcal{L}_{active}$. A translation of $\mathcal{L}_{active}$ into logic programs is presented using a situation calculus notation. The generated rules are non-stratified, and the choice operator of [SZ90] (which is based on stable models) is used for nondeterministically selecting one rule among all rules that may be fired in a situation. Like the situation calculus, $\mathcal{L}_{active}$ focuses more on reasoning about the effect of actions than on the computationally easier task of executing them.

State-Oriented Datalog Extensions. By extending Datalog with a notion of state, (re)active production rules and deductive rules can be handled in a unified way, thereby combining the advantages of active and deductive rules. Two such (closely related) Datalog extensions are *XY-Datalog* [Zan93,Zan95,MZ97] and *Statelog* [LHL95,LML96] (see [KLS92] for an early precursor of the latter). The specification of operational aspects like composite event detection and coupling modes is possible in the logical language since the rules allow access to different database states—even complex execution models for nested transactions can be handled in this way as shown in detail for Statelog in subsequent sections.

However, the more procedural aspects are introduced into the language, the more intricate the representation of these features in the logical framework becomes. XY-Datalog and Statelog are themselves closely related to $Datalog_{1S}$ [Cho95a], a query language for temporal databases.

Logic-Based Formalization of Operational Semantics. In [FWP97], a framework for the integration of the different *operational semantics* of active and deductive rules is developed. The meaning of ECA rules is specified using distinct specification languages for events, conditions, and actions, respectively. The operational semantics for these ECA sublanguages and their interplay is formalized by means of deductive rules. More precisely, the database history (i.e., the ordered set of database states) is modeled using timestamped atoms, and the meaning of events, conditions, and actions is defined based on the *event calculus* in [Kow92]. The approach has been used for the formalization of the active rule component which is added to the deductive object-oriented database system ROCK & ROLL [BFP+95].

In contrast to Statelog and XY-Datalog, which provide a single *unified language* for active and deductive rules, [FWP97] *integrate* the different *operational semantics* of active and deductive rules using a common (deductive) specification formalism.

Another approach to logic-based formalization of active rule semantics is presented in [FT95], where so-called *Extended ECA rules* are used to encode the operational semantics of an ADB. Using these user-readable EECA rules, existing ADBs can be compared and classified. The precise meaning of EECA rules is obtained by translating them into a logical core language which specifies procedural details like event consumption, coupling modes, etc. Unlike in Statelog, this logical language is *not* meant to be handled by the rule programmer, but is considered as an internal representation which is used by the execution model, and thus is on a lower-level than the EECA rules.

Production Rule Semantics. Many approaches to active rules are based on a forward chaining execution model in the style of production rules, e.g., [AWH95], [PV95], and [Zan93,LHL95] above.[4] This is particularly true also for the *PARK semantics* of active rules [GMS92], which can be conceived as an inflationary fixpoint semantics extended by a mechanism to handle update literals $+L$ and $-L$, denoting insertion and deletion of L, respectively. Similar as in Statelog (Section 4), update literals correspond to events if they occur in the body, and to actions if they occur in the head. A main benefit of the approach is the simple and precise semantics with its flexible conflict resolution policy, the latter being a parameter to the PARK semantics. The Statelog approach also allows flexible conflict resolution policies, however, they are not treated as a black box as in

[4] Due to their declarative semantics with explicit states the latter may also be evaluated top-down, say in XSB-Prolog [SSW94]; however, the bottom-up view is more natural in this context.

[GMS92], but can be programmed *within* the logical rule language (Section 5.5 and [Lud98]).

3.3 Update Languages vs. Active Rules

Since active behavior can be specified by defining new updates as an automatic response to previous updates, active rule languages and languages for updates share essential features. A prominent logical framework for updates is *Transaction Logic* $\mathcal{T}_\mathcal{R}$ [BK94], a language that deals on a high level of abstraction with the phenomenon of state change in logic databases. The focus is on the composition of complex updates, whereas primitive updates (so-called *elementary transitions*) are not part of $\mathcal{T}_\mathcal{R}$ but regarded as parameters which are supplied by a transition oracle. In $\mathcal{T}_\mathcal{R}$—like in most languages based on top-down evaluation—updates are expressed in the body. These approaches are often tuple-oriented, so the specification of set-oriented updates (bulk-updates) becomes an issue. [WF97] present an update language based on *deferred updates* which solves this problem. Other well-known approaches in the deductive database community subsumable under the "updates in the body" paradigm are the early works on DLP [MW88] (based on dynamic logic), and LDL updates [NT89]. In contrast, frameworks with semantics similar to production rules typically express updates in the head of rules (cf. Sections 2.2 and 4). A main difference between update languages and active rules is that in the former, updates are initiated explicitly by the user, whereas the latter specify how rules initiate update (trans)actions automatically in response to occurring events.

Bibliographic Notes

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 *Extended ECA* rules, expressing the above-mentioned semantic dimensions, which are then translated into an internal core language. [PCFW95] surveys work on formal specification of active database functionality. [DGG95] presents the active database management system manifesto. In [Day95], a survey on the accomplishments of research in active databases is given. The workshops [WC94] and [BH95] were dedicated particularly to active rules; the workshop series [PW93,Sel95,GB97] also has major sections on active rules.

4 Extending Datalog with States: Flat Statelog

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). The semantics of active rules is often defined only in an informal and procedural way, making it very difficult to understand and predict the behavior of rules. 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 introduce *Statelog*, a logical framework for active rules which precisely and unambiguously defines the meaning of 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 (e.g., tuple-level execution), but covers many essential features including immediate and deferred execution and composite events.

In this section, we present "flat" Statelog, which is based on a linear state space and corresponds to a flat transaction model. Using a hierarchical state space, a framework incorporating nested transactions is developed in Section 5. For simplicity of presentation, we postpone a detailed description of the signature (delta relations, control relations, etc.) to Section 5.2; the intended use of relations will be clear from the context.

4.1 Basic Execution Model

The basic execution model of Statelog is illustrated in Figure 2: States are identified by the natural numbers $\mathbb{N}_0$; the *k-th final state* is denoted by $f_k \in \mathbb{N}_0$.

Assume f_i is the current final state of the database. The database remains in this state as long as no new external events occur. Queries are executed against f_i and may involve base relations, derived relations (i.e., local views on the current state), or historical information (using certain auxiliary relations; see Section 4.4). Observe that intermediate states are depicted as small circles, whereas bigger circles correspond to final states, i.e., which are materialized and directly accessible to the user. *External actions* correspond to outputs of the active rule program and are reported at final states. As described below, a stream of incoming *external events* is conceived as a sequence $\mathcal{E}_0, \mathcal{E}_1, \ldots$ of sets of events which induces (i) a sequence of transactions between (user-visible) final states $f_0, f_1, \ldots$, and (ii) a stream of outgoing *external actions* $\mathcal{A}_0, \mathcal{A}_1, \ldots$

The (simultaneous) occurrence of a set of external events $\mathcal{E}_i$ is modeled by asserting, at the i-th final state f_i, a finite set of facts[5]

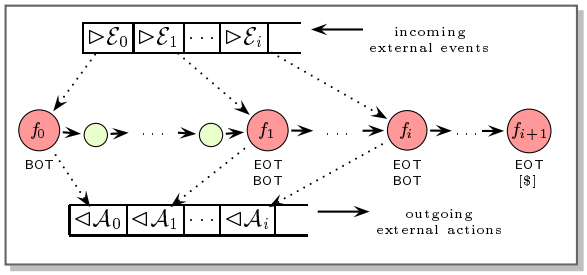
$$\mathcal{E}_i = \{[f_i] \rhd e(\bar{x}) \mid \text{event } e(\bar{x}) \text{ has occurred}\} .$$

[5] For clarity, relations for external events ("input") and external actions ("output") are prefixed with "$\rhd$" and "$\lhd$", respectively; states are often bracketed: $[s]$.

In general, using this new "seed" information, the rules of a Statelog program define a sequence of (intermediate) *transitions*

$$f_i \rightsquigarrow f_i{+}1 \rightsquigarrow f_i{+}2 \rightsquigarrow \cdots \rightsquigarrow f_i{+}k = f_{i+1}$$

until the *transaction* starting at f_i ends in the next final state f_{i+1}. Events occurring between f_i and f_{i+1} are mapped to the new final state f_{i+1}. It should be clear from Figure 2 that the logic model of $P \cup \mathcal{D} \cup \mathcal{E}_0 \cup \cdots \cup \mathcal{E}_k$ (where $\mathcal{D}$ denotes the initial database) is "add-only", i.e., past states cannot be changed, and new events influence only the current and future states. System-defined predicates BOT and EOT can be used to distinguish between the different kinds of states. 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. Another more general model is presented in Section 5.



Classification of states

state	BOT	EOT
initial	*true*	*false*
intermediate	*false*	*false*
past final	*true*	*true*
actual final [$]	*false*	*true*

Fig. 2. Mapping of external events to final states and classification of states

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

A *Statelog database* $\mathcal{D}[k]$ at state $k \in \mathbb{N}_0$ 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 $\mathcal{D}$.

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

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

where the head H is a Datalog atom, B_i are Datalog literals (atoms A or negated atoms $\neg\, A$), and $k_i \in \mathbb{N}_0$. The *leap* $l_i := k_0 - k_i$ of B_i is the distance between the state referred to in the head and the state for B_i in the body. 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 $i = 1, \ldots, n$. If $k_0 = k_i$ for all $i = 1, \ldots, n$, 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 *Statelog program* is a finite set of progressive Statelog rules.

4.3 Semantics

Every Statelog program may be conceived as a standard logic program by viewing the Statelog atom $[S+k]\,p(t_1, \ldots, t_n)$ as syntactic sugar for $p(S+k, t_1, \ldots, t_n)$.[6] In this way, notions (e.g., *local stratification*) and declarative semantics (e.g., *perfect model* $\boldsymbol{M}_P$) developed for deductive rules can be applied directly to Statelog.

Here, we adopt the *state-stratified semantics*[7] as the canonical model 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* [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 rules 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* [Zan93] and *ELS-stratification* [KRS95]. Together with the requirement of progressiveness, state-stratification implies local stratification [Lud98]:

Theorem 1 (State-Stratification)
Let P be a constant-free and progressive *Statelog program. Then,*

$$P \text{ is state-stratified} \;\Leftrightarrow\; P \text{ is locally stratified.} \qquad \square$$

Thus, if P is locally stratified, there exists a unique *perfect model* $\boldsymbol{M}_{P \cup \mathcal{D}}$ [Prz88], for any Statelog database $\mathcal{D}$ $(= \mathcal{D}[0])$.

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

r_1 : [S] del:emp(E,Sal,D) ← [S] del:dept(D,_), emp(E,Sal,D) .
r_2 : [S+1] emp(E,Sal,D) ← [S] emp(E,Sal,D), ¬ del:emp(E,Sal,D) .

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 5.2). r_2 is a *frame rule* specifying that only those

[6] Here, p denotes any type of relation (base, control, ...); see Section 5.2.
[7] See Section 6 for a model-theoretic Kripke-style semantics.

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 $r_2 \stackrel{0}{\rightarrow} r_1 \stackrel{1,\neg}{\rightarrow} r_2$ is not local (i.e., does not occur within a single state). □

Note that state-stratification does *not* imply local stratification for *non-progressive* rules (hence the progressiveness requirement in Theorem 1):

Example 3 (Non-Progressive Rules) The program

P: $\quad [S]\ \mathsf{p} \leftarrow [S{+}1]\ \neg\, \mathsf{p}.$

is state-stratified since it contains no local cycle. However, the truth-value of $[n]\,\mathsf{p}$ depends negatively on the *unfounded* sequence $[n{+}1]$, $[n{+}2]$, ..., so P is not locally stratified: in the well-founded model, p is *undefined*. □

4.4 Composite Events

Although the Statelog language is surprisingly simple, various kinds of composite events and consumption modes can be expressed, as shown in [MZ95] using a closely related variant of 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 detd:R store detected events. Using slight modifications of these rules, different event consumption modes can be accomplished:

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*. Note that in this variant the most recent occurrence of F can take part in *several* occurrences of E. However, if $\neg\, F(_)$ *and* $\neg\, E(_,_)$ are added to the second rule, then every occurrence of E consumes all constituent events, so F can take part only in one occurrence of E.

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, one can show (see [Lud98]) that composite events with chronicle contexts are *not* expressible in pure Statelog and require appropriate extensions (e.g., timestamping as in [MZ95]).

4.5 Formal Results

Using Statelog as a unified logical language for active and deductive rules allows to study abstract rule properties like termination, expressive power, and complexity (see also Section 3.1). Here, we only sketch the main results; see [LLM98] and [Lud98] for details.

For notational convenience, we do not distinguish between base relations and external events below, but assume w.l.o.g. that a Statelog database $\mathcal{D}[k]$ includes the set of external events occurring in state k as a set of facts.

Termination. We say that a Statelog program P *terminates for* $\mathcal{D}$, if the sequence of database states induced by P and $\mathcal{D}$ becomes stationary—more precisely: if for some n_0 and all $n \geq n_0$: $\boldsymbol{M}_{P\cup\mathcal{D}}[n] = \boldsymbol{M}_{P\cup\mathcal{D}}[n{+}1]$, where the *snapshot* $\boldsymbol{M}_{P\cup\mathcal{D}}[n]$ at state n of the perfect model $\boldsymbol{M}_{P\cup\mathcal{D}}$ is defined as

$$\boldsymbol{M}_{P\cup\mathcal{D}}[n] := \{p(\bar{x}) \mid \boldsymbol{M}_{P\cup\mathcal{D}} \models [n]\, p(\bar{x})\} \ .$$

If P terminates for $\mathcal{D}$, then $\boldsymbol{M}_{P\cup\mathcal{D}}[\$]$ is used as a generic notation for the unique final state; otherwise, we agree to set $\boldsymbol{M}_{P\cup\mathcal{D}}[\$] := \emptyset$.

Let $\textsc{Term}_{P,\mathcal{D}}$ denote the set of pairs $(P, \mathcal{D})$ such that P terminates for $\mathcal{D}$; $\textsc{Term}_{\exists\mathcal{D},P}$ and $\textsc{Term}_{\forall\mathcal{D},P}$ denote the set of Statelog programs which terminate for *some* and *all* databases $\mathcal{D}$, respectively. Then one can show [Lud98]:

Theorem 2 (Termination) *1.* $\textsc{Term}_{\exists\mathcal{D},P}$ *and* $\textsc{Term}_{\forall\mathcal{D},P}$ *are undecidable.*
2. $\textsc{Term}_{P,\mathcal{D}}$ *is* **PSPACE***-complete (with* $n = |\mathcal{D}|$*).*
3. $\textsc{Term}_{P,\mathcal{D}}$ *can be decided using a Statelog program* $P^{\downarrow}$ *which (i) terminates for all* $\mathcal{D}$*, and (ii) is effectively constructible from* P*.* □

Note that (*3*) means, in a sense, that Statelog programs allow to "speak" about their termination behavior at run-time (i.e., for given $\mathcal{D}$). However, since testing for termination may be prohibitively expensive, it is desirable to identify efficient classes of terminating programs:

One such class is *G-Statelog* (*guarded Statelog*) where each update rule is required to have a positive occurrence of an external event in the body, thereby guaranteeing that such rules can be applied only once at the beginning of a transaction. Another more powerful class is *Δ-Statelog* (*Δ-monotonic Statelog*): here, the basic idea is to enforce termination by preventing oscillation of updates (i.e., repeated insertion and deletion of the same tuple). Since Statelog programs operate on finite structures, the corresponding constructions guarantee termination in **PTIME**.

Different Rule Semantics in Statelog. As shown above and in Section 5, Statelog allows to handle typical features of active rules at the right end of the spectrum in Figure 1, like composite event detection and (re)active programming of updates. Moreover, several of the more declarative languages in the middle and further to the left of the spectrum turn out to be *special cases* of Statelog rules:

- *Production rules*: Let *I-Datalog* and *P-Datalog* denote the inflationary and noninflationary (or *partial*) semantics for Datalog$^\neg$, respectively.
- *Deductive rules*: The declarative semantics for Datalog$^\neg$ programs are denoted by *S-Datalog* (stratified Datalog) and *WF-Datalog* (well-founded Datalog), respectively.

These semantics have a very natural representation in Statelog [Lud98]; see Figure 3: Observe that the noninflationary P-Datalog semantics only transfers those tuples to the new state, which are derived anew. In contrast, the inflationary I-Datalog semantics propagates all previously derived tuples through all states. S-Datalog rules can be represented directly by local state-stratified rules. Finally, one way to represent the alternating fixpoint computation [VG89] of WF-Datalog is as shown in Figure 3: this encoding yields a terminating program iff the well-founded model is total; however, using a "doubled" encoding, it is easy to obtain a program which explicitly computes the true, false, and undefined atoms and always terminates.

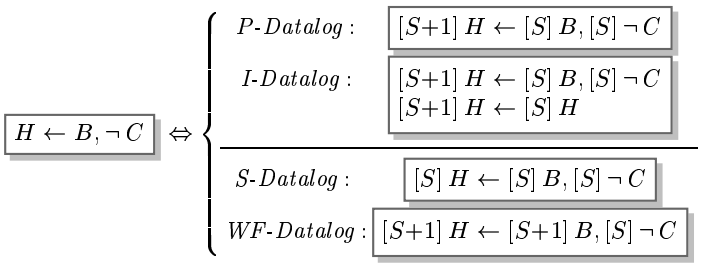


Fig. 3. Encoding schema for different rule semantics

Expressive Power and Complexity. With every Statelog program P one can associate different database mappings: The most important one describes the *transaction expressiveness*, i.e., the mapping $\Gamma_P^{\$} : \mathcal{D}[0] \mapsto \boldsymbol{M}_{P \cup \mathcal{D}}[\$]$ from the initial database state to the final state.

A natural question is: What kind of database mappings can be expressed using Statelog wrt. transactions? For languages involving intermediate states like XY-Datalog and Statelog, one can also consider *transition expressiveness*, i.e., the class of database mappings from one state to the immediate successor state.

Figure 4 summarizes the main results wrt. transition and transaction expressiveness: In the middle, well-known classes of database transformations are depicted (cf. [AHV95]). To the left and right, the equivalent (wrt. transitions and transactions, respectively) Statelog variants are depicted:

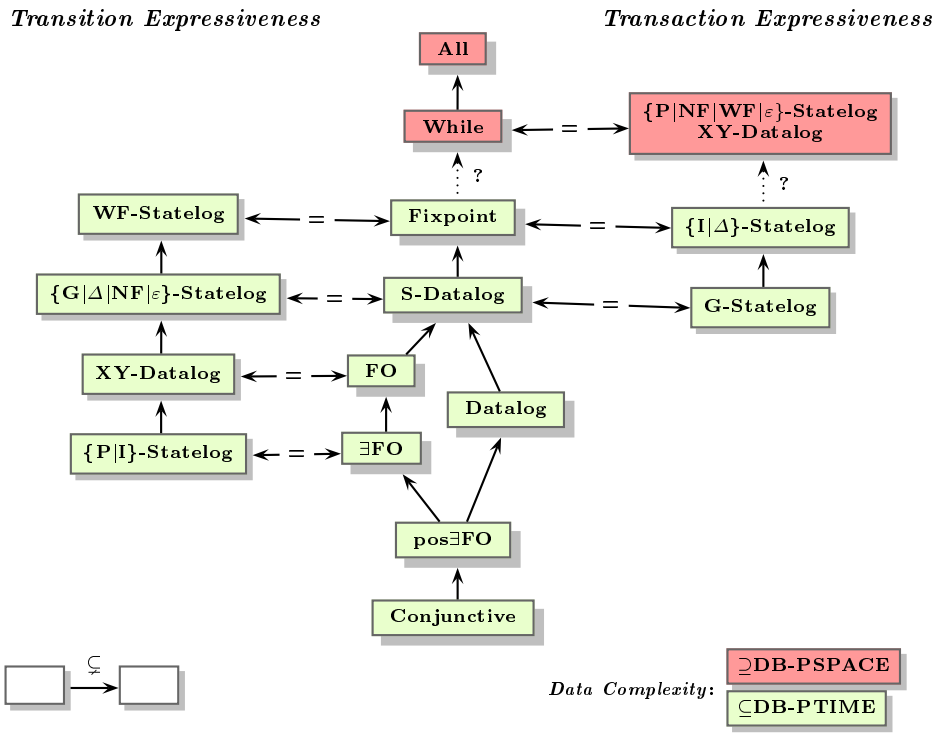


Fig. 4. Summary of expressiveness results [Lud98]

WF-Statelog is a Statelog variant where rules need *not* be stratified, but may involve well-founded negation; G-, Δ-, NF-, P-, I-, and ε-Statelog denote guarded, Δ-monotonic, normal form, noninflationary, inflationary, and unrestricted Statelog, respectively [LLM98,Lud98]. In NF-Statelog, for example, rules may only be 1-progressive or local. The expressiveness results in Figure 4 can be established using rewritings into NF-Statelog and the above encodings of different Datalog$^\neg$ semantics. The nice match between Statelog classes and the known query classes also yields the corresponding complexity results: Statelog *transitions* are always evaluable in **PTIME**, whereas *transactions* may require **PSPACE** in general. An efficient (i.e., **PTIME**-evaluable) class of transactions is given by Δ-Statelog, a class of terminating Statelog programs which—unlike inflationary languages—allows both insertions *and* deletions.

5 Nested Transactions in Statelog

The Statelog programs considered so far define a single transaction from the current state to the new final state for any given database (which includes a set of external events). External events occurring subsequently correspond to new facts being added and initiate the next transaction. Thus, the Statelog execution

model depicted in Figure 2 corresponds to a flat transaction model. In the sequel, we show how Statelog can be extended to model *nested transactions*. With nested transactions, Statelog provides a unified framework for modeling several advanced concepts in active databases, e.g., sophisticated coupling modes, event consumption policies, and trigger firing policies.

The following example, adapted from [MW88,Che95], motivates why structuring capabilities and a refined transaction model may be useful:

Example 4 (To Hire or Not to Hire)
Consider relation emp from Example 2. We want to hire an employee only if the average salary *after* the update does not exceed a certain limit. Such a "post-conditional" update may be expressed in flat Statelog as follows:[8]

$$[S]\ \mathsf{ins{:}emp(E,Sal,D)},\ [S{+}1]\ \mathsf{checksal(D)} \leftarrow [S]\ \triangleright\mathsf{hire_if_possible(E,Sal,D)}.$$
$$[S]\ \mathsf{check_ok} \leftarrow [S]\ \mathsf{checksal(D)},\ \mathsf{avg(D,\ AvgSal)},\ \mathsf{AvgSal} < 50000.$$
$$[S{+}1]\ \mathsf{del{:}emp(E,Sal,D)} \leftarrow [S]\ \triangleright\mathsf{hire_if_possible(E,Sal,D)},\ [S{+}1]\ \neg\,\mathsf{check_ok}.$$

On occurrence of the external event $\triangleright$hire_if_possible(E,Sal,D), employee E is preliminarily inserted and the new average salary is checked in $[S{+}1]$. If it exceeds the admissible amount, the insertion is undone by the last rule. □

The above program specifies the desired transaction, yet there are some potential pitfalls and drawbacks with this solution:

- Undoing the effect of changes (here: the compensation of insertions by corresponding deletions) has to be programmed by the rule designer. However, it is often desirable to *automatically* propagate the failure of a subtransaction like checksal.
- There is no structure which allows grouping of semantically closely related rules.
- The effects of *ephemeral changes* [Zan95], i.e., changes whose effect is undone later within the same transaction, and *hypothetical changes* are visible to other rules, since there is no encapsulation of effects of semantically related rules. E.g., if $\triangleright$hire_if_possible(...) occurs in $[S]$, the delete request ins:emp may trigger other active rules, although in $[S{+}2]$ the update is revoked. This may lead to unjustified (re)actions by other rules, similar to those described in [Zan95].

To avoid these problems, the transaction concept considered so far has to be refined: First, specific system-defined rules can be used to automatically undo the effect of failed transactions. Moreover, the second and third item can be resolved by grouping rules into certain modules which encapsulate rule effects. In principle, a flat transaction model would be sufficient here. However, it is often natural and more adequate to model certain tasks as *subtransactions* which

[8] For simplicity, we view the predicate holding the average salary avg(D,AvgSal) of a department D as a built-in.

are nested within the calling transaction: For example, hire(E,Sal,D) may be a subtransaction of a top-level transaction main; the salary check in turn may be a subtransaction of hire (see Example 6).

5.1 Hierarchical State Space

In order to model nested transactions and handle the problems described above, [LML96] propose the concept of Statelog *procedures*. A Statelog procedure π is a named and possibly parameterized set of Statelog user rules. In this sense, the flat Statelog programs considered so far can be seen as parameterless anonymous Statelog procedures. When π is called at run-time, it defines a transaction T_π by issuing primitive updates (through delta relations) and/or calling other procedures which in turn may define subtransactions, etc. T_π either terminates successfully (indicated by a special predicate committed:π), or aborts. When π calls another procedure μ, a subtransaction T_μ is started whose results are either incorporated into T_π, if T_μ commits, or discarded otherwise. From the point of view of the calling transaction T_π, the subtransaction T_μ is atomic, therefore requests derived directly within T_π and those submitted by T_μ should be indistinguishable. This is achieved by certain system-defined rules.

The behavior of μ is *encapsulated*, since deltas defined by T_μ are only visible within T_μ, but not in other (concurrent) transactions. Subtransactions execute in isolation and in an *all-or-nothing* manner, i.e., no results of T_μ will be visible in T_π if T_μ aborts. Note that this does *not* mean that T_π also aborts—on the contrary, π can detect the failure of T_μ (via aborted:μ) and issue alternative or compensating actions, or retry the execution of μ later.

In order to model the isolated execution of a Statelog procedure π as a (sub)transaction T_π, a unique name space for each (parameterized) invocation of $\pi(\bar{x})$ has to be introduced. This is accomplished by extended state terms and frame terms. The latter provide the *transaction frame* in which π executes.

The execution of Statelog procedures as nested transactions induces a hierarchical structure of the state space instead of the linear structure considered before (cf. Figure 5). Every state term encodes the complete transaction hierarchy from the top-level transaction down to the current transaction. States on the same level are grouped into transaction frames. The model-theoretic foundation of this concept is given by Kripke structures with different accessibility relations, see Section 6.

Given a fixed set Π of procedure names of a Statelog program, the set of *state identifiers* $\mathbb{S}_\Pi$ and *frame identifiers* $\mathbb{F}_\Pi$ are defined as the least sets such that

- $[\varepsilon] \in \mathbb{F}_\Pi$,
- $[f.n] \in \mathbb{S}_\Pi$, if $[f] \in \mathbb{F}_\Pi$ and $n \in \mathbb{N}_0$,
- $[s.\pi(\bar{x})] \in \mathbb{F}_\Pi$, if $[s] \in \mathbb{S}_\Pi, \pi \in \Pi$, and $\bar{x} \in U^\omega$.[9]

[9] $U^\omega := \bigcup_{i \in \mathbb{N}} U^i$.

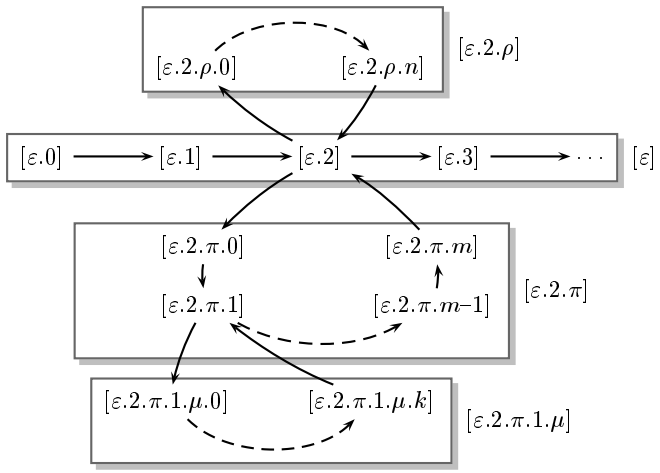


Fig. 5. States and frames

Here, the arity of π matches that of $\bar{x} = x_1, \ldots, x_k$, and U is the underlying domain. *State terms* and *frame terms* are defined similarly but may also involve variables. State and frame identifiers induce a hierarchically structured state space: The *initial frame* $[\varepsilon]$ denotes the *top-level transaction*, its initial state is $[\varepsilon.0]$. Let $[s]$ denote the current state. Then for a procedure call $\pi(\bar{x})$, the frame of the subtransaction induced by the execution of $\pi(\bar{x})$ is $[f] = [s.\pi(\bar{x})]$ and the first state of the transaction is $[f.0] = [s.\pi(\bar{x}).0]$. The successor state of $[f.n]$ (on the same level) is $[f.(n{+}1)]$. The grouping of states into a frame $[f] \in \mathbb{F}_\Pi$ is defined as

$$[f] := \{\, [f.n] \mid n \in \mathbb{N}_0 \,\},$$

which implies that every state $[f.n]$ belongs to exactly one frame $[f]$. Using this representation, the frames $[s.\pi(\bar{x})]$ and $[s.\mu(\bar{y})]$ induced by *different* parallel procedure calls of π and μ in the *same state* $[s]$ can be uniquely identified (if the name of the procedure is the same, at least the parameters are different). Similarly, frames of transactions induced by the *same procedure call* from *different states*, $[s_1.\pi(\bar{x})]$ and $[s_2.\pi(\bar{x})]$, can also be distinguished. The constructor "." is left-associative: e.g., $[s_1.\pi_1.s2.\pi_2] = [((s_1.\pi_1).s2).\pi_2]$.

5.2 Signature

In order to model the specific features of active rules, we introduce several types of relations (their precise semantics and interplay will be specified by system-defined Statelog rules below): The set of relation symbols of a given schema $\mathbf{R}$ is given as the disjoint union of the following sets:

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

Base and Derived Relations. The *extensional database* $edb(\mathbf{R})$ comprises the base relations which are stored in the database. In user-defined rules, *edb*-relations may only occur in the body; they are updated via *delta relations* from $\delta(\mathbf{R})$. In contrast, derived relations belong to the *intensional database* $idb(\mathbf{R})$ 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 $p \in edb(\mathbf{R})$ there are *delta relations* $\mathsf{del}{:}p, \mathsf{ins}{:}p \in \delta(\mathbf{R})$. Delta relations (or just *deltas*) denote update requests to delete or insert the corresponding tuples into p, respectively. For simplicity, we write $\mathsf{mod}{:}p(x/y)$ instead of $\mathsf{del}{:}p(x), \mathsf{ins}{:}p(y)$.

Procedure Calls. $\Pi(\mathbf{R})$ denotes the set of *procedure names*. A procedure π with parameters $\bar{x}$ is "called" by deriving $\pi(\bar{x})$ in the head of a rule.

Protocol Relations. For every base relation $p \in edb(\mathbf{R})$ there are *protocol relations* $\mathsf{deld}{:}p, \mathsf{insd}{:}p \in prot(\mathbf{R})$ (for inserted and deleted, resp.) which store the accumulated *net effect* of a sequence of updates. They can be used for several purposes, e.g., to enforce termination, as an auxiliary store for aborting transactions, or for returning the net effect of a subtransaction [Lud98].

Note that from the above-mentioned relations, only those from $idb(\mathbf{R})$, $\delta(\mathbf{R})$, $\Pi(\mathbf{R})$, and $\triangleleft(\mathbf{R})$ are user-definable; the relations from $edb(\mathbf{R})$ and $prot(\mathbf{R})$ are maintained by the system.

Control Relations. $ctl(\mathbf{R})$ contains special *control relations* like BOT, EOT, running, and abort for transaction control, and auxiliary relations for the detection of composite events. Additionally, $\mathsf{aborted}{:}\pi(\bar{x})$ or $\mathsf{committed}{:}\pi(\bar{x})$ indicate if a subtransaction has been aborted or committed.

5.3 User-Defined vs. System-Defined Rules

In the Statelog core language there is no distinction between user-defined and system-defined rules (e.g., the program given in Example 2 explicitly contains the frame rule r_2). 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 frame rules and transaction 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 (Example 5).

5.4 User-Defined Rules

We require that all *user-defined* Statelog rules are local; thus, state terms may be omitted. Moreover, only relations from $idb(\mathbf{R})$, $\delta(\mathbf{R})$, $\Pi(\mathbf{R})$, $\lhd(\mathbf{R})$ and certain distinguished relations from $ctl(\mathbf{R})$ are allowed to occur in rule heads of a user program. 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.

Programs and Procedures. A Statelog *program* is a finite collection of Statelog procedures. A Statelog *procedure* π is an expression of the form

$$\textbf{proc}\ \pi(X_1, \ldots, X_n)\ \{H_1 \leftarrow B_1\,;\ \ldots\ ;\ H_k \leftarrow B_k\,\}$$

where $X_1, \ldots, X_n$ are the *parameters* of π which may occur in the Statelog user rules $H_i \leftarrow B_i$. Every program contains a distinguished procedure main.

As in the case of flat Statelog, the meaning of rules is given by the declarative semantics of their representation as a logic program. Especially, if rules are locally stratified, a unique perfect model exists.[10]

Depending on the relation symbol in the head of a rule, the following cases can be distinguished:

Views:	$p(\bar{X}) \leftarrow \ldots$	if $p \in idb(\mathbf{R})$
Change Requests:	$\mathsf{ins}{:}p(\bar{X}) \leftarrow \ldots$	if $\mathsf{ins}{:}p \in \delta(\mathbf{R})$
	$\mathsf{del}{:}p(\bar{X}) \leftarrow \ldots$	if $\mathsf{del}{:}p \in \delta(\mathbf{R})$
Procedure Calls:	$\pi(\bar{X}) \leftarrow \ldots$	if $\pi \in \Pi(\mathbf{R})$
External Actions:	$\lhd a(\bar{X}) \leftarrow \ldots$	if $\lhd a \in \lhd(\mathbf{R})$
Transaction Control:	$\mathsf{abort} \leftarrow \ldots$	

External events are allowed only in the body of rules of main, whereas actions may occur in all procedures, but are only allowed in rule heads. Since *edb*-relations are not directly user-definable, all changes to base relations have to be accomplished through insert and delete requests. The materialization of these requests is implemented by frame rules as described below.

[10] A logic programming semantics for Statelog with nested transactions is presented in [LML96]. Note that in order to guarantee local stratification, also dependencies through subtransactions have to be considered.

Visibility of User-Defined Rules. Let $P(\pi)$ denote the (user-defined) rules of a procedure π. Apart from $P(\pi)$, the visible user-defined rules $P([S.\pi(\bar{x})]\,)$ in frame $[S.\pi(\bar{x})]$ include *idb*-relations of the calling transaction:

Definition 1 The set of *visible rules* $P([F]\,)$ of a frame is defined as

$$\begin{array}{ll} P([\varepsilon]\,) & := P(\mathsf{main}) \\ P([F.n.\pi(\bar{x})]\,) & := P(\pi) \;\cup \{p(\ldots) \leftarrow B \in P([F]\,) \mid p \in \mathit{idb}(\mathbf{R})\} \\ & \qquad\qquad \text{for all } n \in \mathbb{N}_0, \pi \in \Pi(\mathbf{R}). \end{array}$$

□

Thus, *idb*-relations are communicated to subtransactions by passing their defining rules, whereas *edb*-relations are communicated to subtransactions by copying their extensions into the initial state of a subtransaction.

5.5 System-Defined Rules

System-defined frame and procedure rules implement the intended semantics of request relations, protocol relations, and procedure calls. All changes are encapsulated within the current transaction frame and invisible everywhere else until the transaction commits. State terms are used in the specification of transitions and transaction management.

Starting a Transaction. 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 the top-level transaction (which coincides with the final state of the previous transaction). All events occurring subsequently are associated with the next initial state.

Frame Rules specify the correct handling of update requests and transitions. For all $p \in \mathit{edb}(\mathbf{R})$, there are the following frame rules:

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

Thus, updates to base relations are executed *immediately* in the transition to the successor state, unless the end of transaction is detected. Clearly, instead of this immediate coupling between condition evaluation and action execution, one could also specify *deferred* execution. Then a different set of frame rules accumulates delete requests and executes all of them at the end of transaction. Frame rules also propagate the *edb* to the subsequent transaction:

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

Protocol relations $\mathsf{insd}{:}p, \mathsf{deld}{:}p \in \mathit{prot}(\mathbf{R})$ store the accumulated *net effect* of changes during a transaction:

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

While there are pending change requests, a transaction is running:

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

A fixpoint is reached when there are no more changes, so EOT is signaled:

$$\begin{array}{l}
[S]\ \ \mathsf{EOT} \leftarrow [S]\ \ \mathsf{BOT}, \neg\, \mathsf{running}. \\
[S{+}1]\ \ \mathsf{EOT} \leftarrow [S]\ \ \mathsf{running}, \neg\, \mathsf{abort}, [S{+}1]\ \ \neg\, \mathsf{running}.
\end{array}$$

The internal event abort terminates a transaction prematurely:

$$[S]\ \ \mathsf{EOT} \leftarrow [S]\ \ \mathsf{abort}.$$

Apart from user-defined aborts, a transaction aborts if inconsistent requests are raised:

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

Other conflict resolution policies can also be easily specified: For example, if the previous rule is omitted, the above frame rules give priority to insertions whenever insertions and deletions occur simultaneously. Similarly, if one adds the goal $\neg\, \mathsf{del}{:}p(\bar{X})$ to the above frame rules with $\mathsf{ins}{:}p(\bar{X})$ in the body, then deletions will have higher priority.

Procedure Rules implement the semantics of procedure calls, i.e., the execution of subtransactions.
For all $\pi \in \mathit{\Pi}(\mathbf{R}), p \in \mathit{edb}(\mathbf{R})$, there are the following rules:

A procedure call creates the initial state of a new frame, signals BOT and initializes the *edb*-relations:

$$\begin{array}{l}
[S.\pi(\bar{X}).0]\ \ \mathsf{BOT} \leftarrow [S]\ \ \pi(\bar{X}). \\
[S.\pi(\bar{X}).0]\ \ p(\bar{Y}) \leftarrow [S]\ \ p(\bar{Y}), \pi(\bar{X}).
\end{array}$$

The processing of the results is implemented by rules checking the successful termination of the subtransactions and evaluating their protocol relations. Since these contain the changes made by the subtransactions, their extensions are copied into the request relations of the parent transaction:

$$\begin{array}{l}
[S]\ \ \mathsf{ins}{:}p(\bar{Y}) \leftarrow [S]\ \ \pi(\bar{X}), [S.\pi(\bar{X}).N]\ \ \mathsf{insd}{:}p(\bar{Y}), \mathsf{EOT}, \neg\, \mathsf{abort}. \\
[S]\ \ \mathsf{del}{:}p(\bar{Y}) \leftarrow [S]\ \ \pi(\bar{X}), [S.\pi(\bar{X}).N]\ \ \mathsf{deld}{:}p(\bar{Y}), \mathsf{EOT}, \neg\, \mathsf{abort}.
\end{array}$$

Thus, update requests reported to the parent by π are indistinguishable from those derived directly (provided π commits, i.e., $\mathsf{EOT} \wedge \neg\mathsf{abort}$ holds).

Parent transactions also perform some bookkeeping about committed and aborted subtransactions:

$$\begin{array}{ll} [S]\ \mathsf{committed}{:}\pi(\bar{X}) & \leftarrow [S]\ \pi(\bar{X}), [S.\pi(\bar{X}).N]\ \ \mathsf{EOT}, \neg\,\mathsf{abort}. \\ [S]\ \mathsf{aborted}{:}\pi(\bar{X}) & \leftarrow [S]\ \pi(\bar{X}), [S.\pi(\bar{X}).N]\ \ \mathsf{EOT}, \mathsf{abort}. \end{array}$$

The user can formulate application-specific aspects of transaction management, e.g., that the parent transaction should abort, if the child aborts:

$$\mathsf{abort} \leftarrow \pi(\bar{X}), \mathsf{aborted}{:}\pi(\bar{X}).$$

Serial Conjunction. It is sometimes convenient, especially in the context of procedures and nested transactions, to provide the user with a special connective "$\otimes$" denoting a serial version of conjunction.[11] To this end, we define

$$A_0 \otimes \cdots \otimes A_n \leftarrow B_0 \otimes \cdots \otimes B_m$$

as a shorthand notation for the Statelog rule

$$[S{+}m]\, A_0, \ldots, [S{+}m{+}n]\, A_n, [S]\, \mathsf{running}, \ldots, [S{+}m{+}n]\, \mathsf{running} \leftarrow [S]\, B_0, \ldots, [S{+}m]\, B_m.$$

For example, a rule of the form $A_0 \otimes A_1 \leftarrow B$, where A_0 and A_1 are primitive actions (like $\mathsf{ins}{:}p$, $\mathsf{del}{:}p$) or procedure calls, informally means: "*if B is true, then do A_0 followed by A_1*". Conversely, the rule $A \leftarrow B_0 \otimes B_1$ intuitively says: "*if B_0 was previously true, and B_1 holds now, then do A*".

5.6 Examples

Example 5 (Hypothetical Updates) In order to implement a hypothetical deletion of an employee, we can use the rule

$$\mathsf{del{:}emp(E,Sal,D)} \otimes \mathsf{ins{:}emp(E,Sal,D)} \leftarrow \triangleright\mathsf{hyp_del_emp(E)},\ \mathsf{emp(E,Sal,D)}.$$

When $\triangleright\mathsf{hyp_del_emp(E)}$ occurs, employee E is removed from the database and immediately inserted afterwards. If we want to determine if E is an "indispensable" employee, i.e., one whose deletion would result in an unpopulated department, we can use the rule:

$$\mathsf{ins{:}indispensable(E)} \leftarrow (\triangleright\mathsf{hyp_del_emp(E)},\ \mathsf{emp(E,Sal,D)}) \otimes \neg\,\mathsf{emp(_,_,D)}.$$

The hierarchical transaction model allows a flexible treatment of several interesting features of databases, like the following:

[11] The symbol "$\otimes$" is borrowed from [BK94], where it denotes a similar connective.

- Static integrity constraints can be implemented by aborting transactions (Example 6).
- Checking the admissibility of changes and blocking inadmissible ones: for any fact $p(\bar{x})$ that should be guaranteed, derive ins:$p(\bar{x})$. Every request to delete it causes an inconsistency.
- Ephemeral updates: every transaction can try some updates, check their results and decide whether it should commit or abort (Example 6).
- Hypothetical updates: every transaction can work on relations which are deleted before committing without having any effect at commit-time. By this it can create a hypothetical scenario, check the outcome and report the consequences. This can be used to evaluate several alternatives in parallel.

Finally, we are sufficiently equipped to revisit Example 4 in the extended framework. Observe that the problems of the flat approach mentioned at the beginning of Section 5 are resolved here:

Example 6 (To Hire or Not to Hire, Cont'd)
We specify the hire and checksal transactions by procedures:

proc main {hire(E,Sal,D) ←▷hire_if_possible(E,Sal,D)}.

proc hire(E,Sal,D) {ins:emp(E,Sal,D) ⊗ checksal(D) ← BOT;
abort←aborted:checksal(D)}.

proc checksal(D) {abort←avg(D,AvgSal)>50000}.

Whenever ▷hire_if_possible occurs, a subtransaction hire(...) is initiated. At the beginning of transaction (and only then), hire adds the new employee followed by a call to checksal. If checksal aborts then hire also aborts, resulting in an unchanged database. Otherwise hire commits and the insertion is realized. The following signatures are defined:

$$edb(\mathbf{R}) = \{\mathsf{emp}\}, \quad \delta(\mathbf{R}) = \{\mathsf{ins{:}emp}, \mathsf{del{:}emp}\},$$
$$prot(\mathbf{R}) = \{\mathsf{insd{:}emp}, \mathsf{deld{:}emp}\}, \quad \Pi(\mathbf{R}) = \{\mathsf{main}, \mathsf{hire}, \mathsf{checksal}\},$$
$$ctl(\mathbf{R}) = \{\mathsf{BOT}, \mathsf{running}, \mathsf{EOT}, \mathsf{abort}, \mathsf{aborted{:}checksal}, \mathsf{aborted{:}hire}\}.$$

Figure 6 depicts the state space which is created when hire(john,60000,d1) is called (and eventually aborted, since the average after the hypothetical update exceeds 50000).

Frames are represented by shadowed boxes, states are represented by ordinary boxes. In all states, the upper entry denotes the state term, the data below the first horizontal line are facts which are derived by frame rules or local rules, and the data below the second line (if it exists) are facts which are derived from results of subtransactions. □

Example 7 (The Christmas-Problem)
Consider a relation emp(E, BirthDay, Sal) with the obvious meaning. We want to implement the following, informally given procedure: *Every employee shall be given a salary raise by 5% at her birthday; on Christmas every employee shall get an extra $1000.* This is accomplished in flat Statelog as follows [LHL95]:

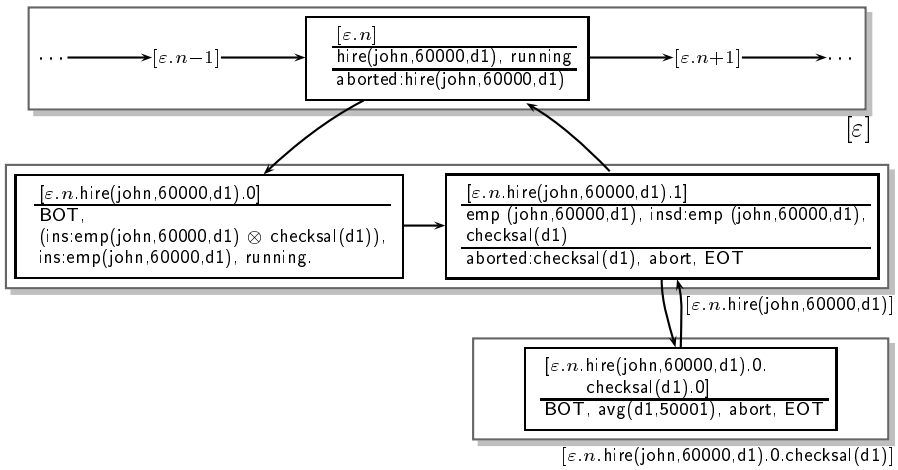


Fig. 6. Frames and Database States

$[S{+}1]$ mod:emp(E,Bday,Sal/Sal1) ←
    $[S]$ ▷daily, date(Day), Day=Bday, emp(E,Bday,Sal), Sal1:= Sal*1.05).
$[S{+}1]$ mod:emp(E,Bday,Sal/Sal1) ←
    $[S]$ ▷daily, date(Day), xmas(Day), emp(E,Bday,Sal), Sal1:= Sal+1000.

These rules work fine unless there is some employee whose birthday is on Christmas: Then two inconsistent modify-requests are generated, and the subsequent state is not well-defined. In a flat model, the problem could be solved by complete case splitting or by a rule using three states (however, this raises the problem that the intermediate state should not trigger other rules). In the structured model, the sequential composition inc_xmas ⊗ inc_bday can be used by the top-level transaction incsal to specify the order of execution:

proc main {incsal(Day) ← ▷daily, date(Day)}.
proc incsal(Day) {inc_xmas(Day) ⊗ inc_bday(Day) ←BOT}.
proc inc_xmas(Day) {mod:emp(E,Bday,Sal/Sal1) ←
        BOT,xmas(Day),emp(E,Bday,Sal), Sal1:= Sal+1000}.
proc inc_bday(Day) {mod:emp(E,Bday,Sal/Sal1) ←
        BOT, Day=Bday, emp(E,Bday,Sal), Sal1:= Sal*1.05}.

If inc_xmas(Day) ⊗ inc_bday(Day) were replaced by the simultaneous conjunction inc_xmas(Day) , inc_bday(Day), then two conflicting requests would be derived and the transaction would be aborted automatically by the corresponding system-defined rules. □

Using the nested transaction model, different rule schemata for modeling certain coupling modes, event consumption policies, and trigger firing policies can be defined, thereby providing a concise, formal specification of these features.

Example 8 (Control Features) *Instead-triggers* can be modeled by replacing a procedure call $\rho(\bar{x})$ by another call $\pi(\bar{x})$: when a call $\rho(\bar{x})$ is derived, it is discarded immediately, and the call $\pi(\bar{x})$ is derived instead. The procedure rules from Section 5.5 are modified, for every $\pi \in \Pi(\mathbf{R})$, as follows:

$$[S.\pi(\bar{X}).0]\ \mathsf{BOT} \leftarrow [S]\ \pi(\bar{X}), \neg\, \mathsf{discard}{:}\pi(\bar{X}).$$
$$[S.\pi(\bar{X}).0]\ p(\bar{Y}) \leftarrow [S]\ p(\bar{Y}), \pi(\bar{X}), \neg\, \mathsf{discard}{:}\pi(\bar{X}).$$

Now the instead-trigger *"$\pi(\bar{x})$ instead of $\rho(\bar{x})$"* is expressed by the (local) rule

$$\pi(\bar{X}), \mathsf{discard}{:}\rho(\bar{X}) \leftarrow \rho(\bar{X}).$$

Before- and *after-triggers* can be modeled as specialized instances of instead-triggers based on *serial conjunction*: Consider a trigger of the form *"before $\rho(\bar{x})$ do $\pi(\bar{x})$"*. This can be accomplised by the following rules (ρ' contains the same rules as ρ):

$$\pi_\rho(\bar{X}), \mathsf{discard}{:}\rho(\bar{X}) \leftarrow \rho(\bar{x}).$$
$$\textbf{proc}\ \pi_\rho(\bar{X})\ \{\pi(\bar{X}) \otimes \rho'(\bar{X}) \leftarrow \mathsf{BOT}\}.$$

Other control features which can be handled include different event consumption modes (see Section 4.4) and deferred (instead of immediate) coupling (cf. Section 5.5). □

5.7 Operational Semantics

The above rule system defines a partial order on state identifiers: Within a frame, $[f.n] \prec [f.m]$ if $n < m$; additionally, $[f.n] \prec [f.n.\pi.k] \prec [f.(n{+}1)]$ for $\pi \in \Pi(\mathbf{R})$ and all $k \in \mathbb{N}$. With the additional requirement on user-defined rules that there are no negative dependencies from relations of $\delta(\mathbf{R})$ to any other relation of the same state, $\prec$ can serve as a base for computing the individual database states as follows:

1. compute $[f.i]\,(\mathit{edb}(\mathbf{R}) \cup \mathit{prot}(\mathbf{R}))$ from $[f.(i{-}1)]\,(\mathit{edb}(\mathbf{R}) \cup \delta(\mathbf{R}) \cup \mathit{prot}(\mathbf{R}))$,
2. compute $[f.i]\,(\mathit{idb}(\mathbf{R}) \cup \delta(\mathbf{R}) \cup \Pi(\mathbf{R}))$,
3. compute $\{[f.i.\pi(\bar{x}).\mathbb{N}] \mid [f.i]\,\pi(\bar{x})$ holds$\}$ recursively,
4. add the resulting requests from $\{[f.i.\pi(\bar{x}).n]\,\mathit{prot}(\mathbf{R}) \mid [f.i]\,\pi(\bar{x})$ and $[f.i.\pi(\bar{x}).n]\,\mathsf{EOT}$ holds$\}$ to $[f.i]\,\delta(\mathbf{R})$,
5. extend $[f.i]\,(\mathit{idb}(\mathbf{R}) \cup \delta(\mathbf{R}) \cup \Pi(\mathbf{R}))$, based on the additions to $[f.i]\,\delta(\mathbf{R})$ from step 4 ("add-only", since these relations do not depend negatively on $\delta(\mathbf{R})$),
6. compute $\{[f.i.\pi(\bar{x}).\mathbb{N}] \mid [f.i]\,\pi(\bar{x})$ has been derived in the previous step$\}$,
7. iterate steps 4–6 until a fixpoint is reached,
8. if not $[f.i]\,\mathsf{EOT}$, proceed with step 1 for $[i{+}1]$.

Note that with the above schema, $[f.i]\,(\mathit{edb}(\mathbf{R}) \cup \mathit{prot}(\mathbf{R}))$ are computed once in the first step. Therefore, the database in the initial state $[f.i.\pi(\bar{x}).0]$ of subframes is identical for all subframes $[f.i.\pi(\bar{x})]$, independent from the iteration step in which the procedure call has been derived.

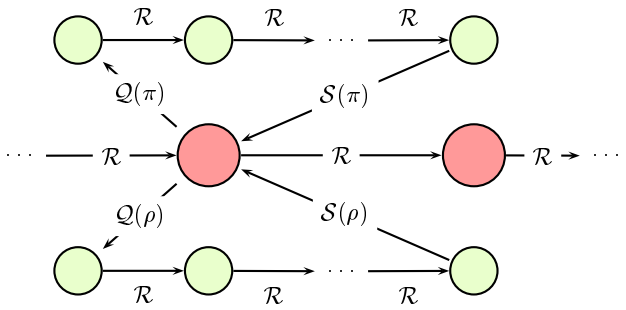


Fig. 7. Hierarchical Kripke Structure

6 Kripke-Style Semantics

A conceptual model for Statelog with nested transactions is established using a Kripke-style semantics, thereby providing states as "first-class citizens" of the logical framework. State identifiers are mapped to states of a Kripke structure which formalizes the relationships between individual states in terms of accessibility relations. The Kripke semantics yields an abstract and natural model of the hierarchical state space and can serve as a basis for the specification and verification of properties of a database system. We show that the rules given in the previous section (where states are "reified", i.e., encoded into the language) are correct wrt. the abstract Kripke semantics.

Definition 2 (Statelog Kripke Structure) A *Statelog Kripke structure* over a given Statelog signature $\mathbf{R}$ is a tuple $\mathcal{K} = (\mathcal{G}, \mathcal{U}, \mathcal{R}, \mathcal{Q}, \mathcal{S}, \mathcal{M}, \mathcal{P})$, (cf. Figure 7) where

$\mathcal{G}$ is a set of states,
$\mathcal{U}$ is the universe,
$\mathcal{R} \subseteq \mathcal{G} \times \mathcal{G}$ is an accessibility relation modeling the temporal successor relation.
$\mathcal{Q}, \mathcal{S} \subseteq \mathcal{G} \times \Pi(\mathbf{R}) \times \mathcal{U}^\omega \times \mathcal{G}$ are two labeled accessibility relations between states representing the procedure-call and -return relation, respectively.
$\mathcal{M}$ is a function which maps every state to a first-order interpretation over $\mathbf{R}$ with universe $\mathcal{U}$,
$\mathcal{P}$ is a function which maps every $g \in \mathcal{G}$ to a set of local rules (the rules visible in g). □

Here, the Statelog Kripke *frame* $(\mathcal{G}, \mathcal{R}, \mathcal{Q}, \mathcal{S})$ models the relationships between states and frames: Every *computation path* $(g_1, g_2, \ldots)$ in a Statelog Kripke structure s.t. $\mathcal{R}(g_i, g_{i+1})$ for all i corresponds to a frame in the hierarchical state space (note that $\mathcal{R}(g, g)$ denotes that a subtransaction has reached a fixpoint, i.e., then, $\mathcal{M}(g) \models \mathsf{EOT}$ holds). $\mathcal{Q}$ denotes the frame-subframe-relation, i.e., $\mathcal{Q}(g, \pi(\bar{x}), g')$ means that the first state of the subtransaction induced by a call of procedure π with arguments $\bar{x}$ is g'. $\mathcal{S}(g', \pi(\bar{x}), g)$ means that g' is the final state of the subtransaction induced by a call of procedure π with arguments $\bar{x}$ in g. Thus, results of subtransactions are communicated along $\mathcal{S}$.

The following characterization covers the intended semantics of the Statelog frame rules:

Definition 3 A Statelog Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{U}, \mathcal{R}, \mathcal{Q}, \mathcal{S}, \mathcal{M}, \mathcal{P})$ over a signature $\mathbf{R}$ is a *model* of a Statelog program P over $\mathbf{R}$ and an initial database $\mathcal{D}$ if

- $\mathcal{U}$ is the active domain of P and $\mathcal{D}$.
- There is a $g_0 \in \mathcal{G}$ s.t. $\mathcal{M}(g_0)|_{edb(\mathbf{R})} = \mathcal{D}$, and $\mathcal{M}(g_0)|_{prot(\mathbf{R})} = \emptyset$, and there is no g s.t. $\mathcal{Q}(g, _, g_0)$ or $\mathcal{R}(g, g_0)$ (existence of an initial state).
- External events are only mapped to the initial state of transactions on the highest hierarchical level, i.e., $\mathcal{M}(g)|_{\triangleright(\mathbf{R})} \neq \emptyset$ only if $\mathcal{R}^*(g_0, g)$ and $g = g_0$ or $\mathcal{M}(g) \models \mathsf{EOT}$.[12]
- The relation $\mathcal{R}$ models the temporal successor relation, i.e.,

  $$\mathcal{R}(g, h) \Rightarrow \mathcal{P}(h) = \mathcal{P}(g)$$

  and the following relationship between $edb(\mathbf{R})$ and $\delta(\mathbf{R})$ in g and $edb(\mathbf{R})$ and $prot(\mathbf{R})$ in h holds:

  $$\begin{array}{l}\mathcal{R}(g, h) \Rightarrow \text{ for all } p \in edb(\mathbf{R}) : \\ \quad \mathcal{M}(h)(p) = (\mathcal{M}(g)(p) \cup \mathcal{M}(g)(\mathsf{ins}{:}p)) \setminus \mathcal{M}(g)(\mathsf{del}{:}p) \text{ and} \\ \quad \mathcal{M}(h)(\mathsf{insd}{:}p) = (\mathcal{M}(g)(\mathsf{insd}{:}p) \cup \mathcal{M}(g)(\mathsf{ins}{:}p)) \setminus \mathcal{M}(g)(\mathsf{del}{:}p) \text{ and} \\ \quad \mathcal{M}(h)(\mathsf{deld}{:}p) = (\mathcal{M}(g)(\mathsf{deld}{:}p) \cup \mathcal{M}(g)(\mathsf{del}{:}p)) \setminus \mathcal{M}(g)(\mathsf{ins}{:}p)\end{array}$$

  and $\mathcal{R}$ is total, i.e., for every $g \in \mathcal{G}$ there is a $g' \in \mathcal{G}$ s.t. $\mathcal{R}(g, g')$.
- $\mathcal{Q}$ models the subtransaction calls: for all $g \in \mathcal{G}$, $\pi \in \Pi(\mathbf{R})$, $\bar{u} \in \mathcal{U}^\omega$:

  $$\mathcal{M}(g) \models \pi(\bar{u}) \;\Leftrightarrow\; \text{there is an } h \text{ s.t. } \mathcal{Q}(g, \pi(\bar{u}), h) \ ,$$

  and $\mathcal{Q}(g, \pi(\bar{u}), h)$ implies that $\mathcal{M}(h) \models \mathsf{BOT}$, $\mathcal{M}(h)|_{edb(\mathbf{R})} = \mathcal{M}(g)|_{edb(\mathbf{R})}$, $\mathcal{M}(h)|_{prot(\mathbf{R})} = \emptyset$, and

  $$\mathcal{P}(h) = P(\pi) \ \cup \{p(\ldots) \leftarrow B \in \mathcal{P}(g) \mid p \in idb(\mathbf{R})\} \ .$$

- $\mathcal{S}$ models the return-from-subtransaction relation:
  for all $g, g', h' \in \mathcal{G}$, $\pi \in \Pi(\mathbf{R})$, $\bar{u} \in \mathcal{U}^\omega$:

  $$\mathcal{Q}(g, \pi(\bar{u}), g') \text{ and } \mathcal{R}^*(g', h') \text{ and } \mathcal{M}(h') \models \mathsf{EOT} \;\Leftrightarrow\; \mathcal{S}(h', \pi(\bar{u}), g) \ .$$

- For every $g \in \mathcal{G}$,

  $$\mathcal{M}(g) = \boldsymbol{M}_{\mathcal{P}(g)}(\mathcal{M}(g)|_{edb(\mathbf{R}) \cup prot(\mathbf{R}) \cup \triangleright(\mathbf{R})} \cup \mathcal{C}(g))$$

  where $\mathcal{C}(g)$ is the set of requests which are contributed to g by subtransactions (and communicated along $\mathcal{S}$), given as

  $$\begin{array}{l}\mathcal{C}(g)(\mathsf{ins}{:}p) := \{\bar{u} \mid \text{there are } g' \in \mathcal{G}, \pi \in \Pi(\mathbf{R}), \bar{v} \in \mathcal{U}^\omega \text{ s.t. } \mathcal{S}(g', \pi(\bar{v}), g) \text{ and} \\ \qquad\qquad \mathcal{M}(g') \models \neg\, \mathsf{abort} \text{ and } \bar{u} \in \mathcal{M}(g')(\mathsf{insd}{:}p)\} \\ \mathcal{C}(g)(\mathsf{del}{:}p) := \{\bar{u} \mid \text{there are } g' \in \mathcal{G}, \pi \in \Pi(\mathbf{R}), \bar{v} \in \mathcal{U}^\omega \text{ s.t. } \mathcal{S}(g', \pi(\bar{v}), g) \text{ and} \\ \qquad\qquad \mathcal{M}(g') \models \neg\, \mathsf{abort} \text{ and } \bar{u} \in \mathcal{M}(g')(\mathsf{deld}{:}p)\} \ .\end{array}$$

□

[12] as usual, $\mathcal{R}^*$ denotes the transitive closure of $\mathcal{R}$.

Proposition 3 *In every Statelog Kripke structure which is a model of a Statelog program P, the following holds:*

- *The temporal successor relation $\mathcal{R}$ is deterministic modulo external events: for all $g, h, h' \in \mathcal{G}$:*

$$\mathcal{R}(g,h) \text{ and } \mathcal{R}(g,h') \Rightarrow \mathcal{M}(h)|_{\mathbf{R}\setminus\triangleright(\mathbf{R})} = \mathcal{M}(h')|_{\mathbf{R}\setminus\triangleright(\mathbf{R})} \ .$$

- *For every $g \in \mathcal{G}$ in a non-top level frame (i.e., on a level where there are no external events) s.t. $\mathcal{M}(g) \models \mathit{BOT}$, there is a unique computation path $(g_0, g_1, \ldots, g_n)$ with $g_i \models \neg\, \mathit{EOT}$ for all $i < n$ and $g_n \models \mathit{EOT}$.*
- *for every Statelog program P, database $\mathcal{D}$, and sequence $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, \ldots$ of sets of external events, there is a unique Kripke model of P with an initial state $\mathcal{M}(g_0) = \boldsymbol{M}_{\mathsf{main}}(\mathcal{D} \cup \mathcal{E}_0)$ and a top-level computation path $(g_0, g_1, \ldots)$ s.t. $\mathcal{M}(g_i) \models \mathit{EOT} \Leftrightarrow i \in \{f_1, f_2, \ldots\}$ (f_i denotes the i-th final state; see Section 4.1); in this case: $\mathcal{M}(g_{f_i}) \models \mathcal{E}_i$.* □

$\mathcal{S}^{-1}$ can also be regarded as a relation describing the effects of subtransactions: For $\pi \in \Pi(\mathbf{R})$ and $\bar{u} \in \mathcal{U}^\omega$,

$$\pi(\bar{u})(g) := h \in \mathcal{G} \text{ s.t. } (h,g) \in \mathcal{S}(\pi(\bar{u}))$$

is the result of executing $\pi(\bar{u})$ in state g. The relationship between $\mathcal{M}(g)$ and $\mathcal{M}(\pi(\bar{u})(g))$ is important for expressing (correctness) properties of subtransactions. With this, $\mathcal{C}(g)$ can be characterized in terms of the effects of the subtransactions which are issued in g:

$$\begin{aligned}
\mathcal{C}(g)(\mathsf{ins}{:}p) = \{\bar{u} \mid\ & \text{there is a } \pi \in \Pi(\mathbf{R}), \bar{v} \in \mathcal{U}^\omega \text{ s.t. } \bar{v} \in \mathcal{M}(g)(\pi) \text{ and} \\
& \mathcal{M}(\pi(\bar{y})(g)) \models \neg\, \mathsf{abort} \text{ and } \bar{u} \in \mathcal{M}(\pi(\bar{y})(g))(\mathsf{insd}{:}p)\} \ , \\
\mathcal{C}(g)(\mathsf{del}{:}p) = \{\bar{u} \mid\ & \text{there is a } \pi \in \Pi(\mathbf{R}), \bar{v} \in \mathcal{U}^\omega \text{ s.t. } \bar{v} \in \mathcal{M}(g)(\pi) \text{ and} \\
& \mathcal{M}(\pi(\bar{y})(g)) \models \neg\, \mathsf{abort} \text{ and } \bar{u} \in \mathcal{M}(\pi(\bar{y})(g))(\mathsf{deld}{:}p)\} \ .
\end{aligned}$$

Equipped with a notion of states, computation sequences, and subtransactions, the semantics of the individual subsets of a Statelog signature (cf. Section 5.2) can be described in terms of state transitions and (sub)transactions:

Theorem 4 (Adequacy) *Statelog Kripke structures are an adequate model of the intended semantics of nested transactions based on the elementary actions ins and del:*

- *$\mathcal{R}$ models the temporal successor relation:*
  - *edb-relations are changed exactly via requests: for all $p \in edb(\mathbf{R})$, $g, h \in \mathcal{G}$: if $(g,h) \in \mathcal{R}$, then*

$$\mathcal{M}(h)(p) = (\mathcal{M}(g)(p) \setminus \mathcal{M}(g)(\mathit{del}{:}p)) \cup \mathcal{M}(g)(\mathit{ins}{:}p) \ .$$

  - *In all states, the protocol relations contain the non-revoked changes of the corresponding subtransactions: For all $(g,h) \in \mathcal{QR}^*$ and all $p \in edb(\mathbf{R})$:*

$$\mathcal{M}(h)(p) = (\mathcal{M}(g)(p) \cup \mathcal{M}(h)(\mathit{insd}{:}p)) \setminus \mathcal{M}(h)(\mathit{deld}{:}p) \ .$$

- *$\mathcal{Q}$ models the subtransaction calls:*
  - *In the initial states of subtransactions, the edb is the same as in the calling state and the protocol relations (procedure knowledge) are empty.*
  - *The definition of the IDB in a subtransaction contains the definition of the IDB in the calling transaction.*
- *$\mathcal{S}$ models the return-from-subtransaction relation:*
  - *For all $g, h \in \mathcal{G}$, if $\mathcal{S}(h, _, g)$ and $\mathcal{M}(h) \not\models \mathsf{abort}$, then for all $p \in edb(\mathbf{R})$:*

$$\mathcal{M}(g)(\mathsf{ins}{:}p) \supseteq \mathcal{M}(h)(\mathsf{insd}{:}p) \text{ and } \mathcal{M}(g)(\mathsf{del}{:}p) \supseteq \mathcal{M}(h)(\mathsf{deld}{:}p) \ .$$

- *Internal semantics of states (Perfect model of $\mathcal{P}(g)$):*
  - *Insert/delete requests are derived by user-defined rules or contributed by subtransactions.*
  - *IDB and subtransaction calls are derived by user-defined rules.* □

The declarative semantics of the perfect model of a Statelog program and the operational semantics given in Section 5.7 for computing a Statelog model state-by-state coincide with the presented Kripke semantics:

Theorem 5 (Equivalence) *Fix a Statelog program P, an initial database $\mathcal{D}$, and a sequence $\mathcal{E} = (\mathcal{E}_0, \mathcal{E}_1, \ldots)$ of sets of external events. For every Statelog Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{U}, \mathcal{R}, \mathcal{Q}, \mathcal{S}, \mathcal{M}, \mathcal{P})$ which is a model of P, there is a partial mapping $\eta : \mathbb{S}_\Pi \to \mathcal{G}$ such that:*

- *$\mathsf{dom}(\eta) = \{[s] \mid \boldsymbol{M}_{P \cup \mathcal{D} \cup \mathcal{E}} \models [s]\, \mathsf{running} \vee \mathsf{EOT}\}$, i.e., the "used" states.*
- *For all literals L over $\mathbf{R}$, $\mathcal{M}(\eta(s)) \models L$ iff $\boldsymbol{M}_{P \cup \mathcal{D} \cup \mathcal{E}} \models [s]\, L$.*
- *For every $n \in \mathbb{N}_0$, $[f.n] \in \mathsf{dom}(\eta)$: $\mathcal{P}(\eta([f.n])) = P([f])$.* □

7 Summary and 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 programming is carried out in real world applications, theoretical foundations of active rules are still rare. In the first part of the paper, we have introduced the basics of active rules and related them to production rules and deductive rules, respectively. After discussing a number of formal approaches to active rules, 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 [LLM98,Lud98]. Although the basic execution model of flat Statelog 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. It can be shown [Lud98] that 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]. 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]). Finally, we have shown how the flat transaction model can be extended to handle nested transactions using a hierarchical state space. In this extended framework, low-level procedural constructs like *before-* and *instead-triggers* can be formalized in an intuitive way. Finally, a model-theoretic semantics based on labeled Kripke-structures has been developed for Statelog with nested transactions, which provides a conceptual, implementation-independent model for active rule behavior.

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 (TODS)*, 20(1):3–41, March 1995.

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

[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.

[BLT97] C. Baral, J. Lobo, and G. Trajcevski. Formal Characterization of Active Databases: Part II. In F. Bry, K. Ramamohanarao, and R. Ramakrishnan, editors, *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, pp. 247–264, Montreux, Switzerland, 1997. Springer.

[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.

[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.

[Che95] W. Chen. Programming with Logical Queries, Bulk Updates and Hypothetical Reasoning. In B. Thalheim, editor, *Workshop Semantics in Databases*, Prague, January 1995. Technische Universität Cottbus.

[Cho95a] J. Chomicki. Depth-Bounded Bottom-Up Evaluation of Logic Programs. *Journal of Logic Programming*, 25(1):1–31, October 1995.

[Cho95b] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[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.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14:1–26, 1994.

[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 M. Berndtsson and J. Hansson, editors, *1st Intl. Workshop on Active and Real-Time Database Systems (ARTDB)*, Workshops in Computing, pp. 3–22, Skövde, 1995. Springer.

[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.

[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.

[GB97] A. Geppert and M. Berndtsson, editors. *Proc. of the 3nd Intl. Workshop on Rules in Database Systems (RIDS)*, number 1312 in LNCS, Skövde, Sweden, 1997.

[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.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Intl. Conference on Logic Programming (ICLP)*, pp. 1070–1080, 1988.

[GL93] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[GMS92] G. Gottlob, G. Moerkotte, and V. S. Subrahmanian. The PARK Semantics for Active Rules. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Intl. Conference on Extending Database Technology*, number 1057 in LNCS, Avignon, France, 1992. Springer.

[ISO97] ISO-ANSI Working draft. SQL3, 1997. ISO/IEC JTC 1/SC 21/WG 3.

[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 Database a Practical Technology: a step forward. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 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 (VLDB)*, pp. 251–262, Vancouver, 1992.

[Kow92] R. A. Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, 12(1&2):121–146, 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. Conference on Management of Data (COMAD)*, pp. 221–238, Pune, India, 1995. Tata McGraw-Hill.

[LLM98] G. Lausen, B. Ludäscher, and W. May. On Logical Foundations of Active Databases. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 12, pp. 389–422. Kluwer Academic Publishers, 1998.

[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.

[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.

[Lud98] B. Ludäscher. *Integration of Active and Deductive Database Rules*. PhD thesis, Institut für Informatik, Universität Freiburg, 1998. infix-Verlag, Sankt Augustin, 1998, ISBN 3-89601-445-5.

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

[MW88] S. Manchanda and D. S. Warren. A Logic-Based Language for Database Updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[MZ95] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In Ling et al. [LMV95], pp. 19–37.

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

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[PCFW95] N. W. Paton, J. Campin, A. A. A. Fernandes, and M. H. Williams. Formal Specification of Active Database Functionality: A Survey. In Sellis [Sel95], pp. 21–37.

[PDW+93] N. W. Paton, O. Díaz, M. H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of Active Behaviour. In Paton and Williams [PW93], pp. 40–57.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann, 1988.

[PV95] P. Picouet and V. Vianu. Semantics and Expressiveness Issues in Active Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, 1995.

[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.

[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.

[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 Geppert and Berndtsson [GB97], pp. 1–19.

[SSW94] K. F. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engin. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 442–453, 1994.

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

[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.

[WF97] C.-A. Wichert and B. Freitag. Capturing Database Dynamics by Deferred Updates. In *Intl. Conference on Logic Programming (ICLP)*, Leuven, Belgium, 1997. MIT Press.

[Wid93] J. Widom. Deductive and Active Databases: Two Paradigms or Ends of a Spectrum. In Paton and Williams [PW93].

[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.