Nested Transactions in a Logical Language for Active Rules

Institut für Informatik — Report 80*

Bertram Ludäscher Wolfgang May Georg Lausen

Institut für Informatik, Universität Freiburg
Am Flughafen 17, 79110 Freiburg, Germany
{ludaesch,may,lausen}@informatik.uni-freiburg.de

Abstract. We present a hierarchically structured transaction-oriented concept for a rule-based active database system. In [LL94, LHL95], we have proposed *Statelog* as a unified framework for active and deductive rules. Following the need for better structuring capabilities, we introduce *procedures* as a means to group semantically related rules and to encapsulate their behavior. In addition to executing elementary updates, procedures can be called, thereby defining (sub)transactions which may perform complex computations. A Statelog procedure is a set of ECA-style Datalog rules together with an import/export interface. System-immanent frame and procedure rules ensure both propagation of facts and processing of results of committed subtransactions. Thus, Statelog programs specify a nested transaction model which allows a much more structured and natural modeling of complex transactions than previous approaches. Two equivalent semantics for a Statelog program P are given: (i) a logic programming style semantics by a compilation into a logic program, and (ii) a model-theoretic Kripke-style semantics. While (ii) serves as a *conceptual model* of active rule behavior and allows to reason about properties of the specified transactions, (i) – together with the appropriate execution model – yields an operational semantics and can be used as an implementation of P.

1 Introduction

The need for a logically defined and intuitive semantics has been recognized as one of the major theoretical problems in the area of active databases. The active database manifesto, for example, requires as an essential feature that *"... rule execution must have a clear semantics, ie must define when, how, and on what database state conditions are evaluated and actions executed"* [DGG95]. Nevertheless, researchers continue to complain about the unpredictable behavior of active rules and the lack of a uniform and clear semantics.[2]

To overcome these difficulties, it has been suggested to use the logical foundations of deductive databases – with certain extensions – as a declarative semantics for active rules [Zan93, ZS94, Zan95, LL94, LHL95]. The main benefits of this approach are better understanding, maintainability and reasoning about rules when compared to the usual implementation-dependent operational semantics. However, as we will show in Section 2, the existing "mergers" of active and deductive rules are not sufficient to model complex (trans)actions in a natural way, since they (i) lack structuring capabilities, and (ii) do not encapsulate the effect of semantically related rules. In particular, they neglect the fact that complex database transactions can be adequately modeled by *nested transactions* where the parent transaction may consult the outcome of subtransactions in order to perform its own complex tasks. As a solution, we propose the extension of our declarative framework for active rules [LHL95] by the concept of update *procedures*. Procedures execute as

* *A slightly shorter version appears in Proc. Intl. Workshop Logic in Databases (LID), San Miniato, Pisa, Italy, 1996, LNCS, Springer*

[2] *"The unstructured, unpredictable, and often nondeterministic behavior of rule processing can become a nightmare for the database rule programmer"* [AWH95]. See also [Wid94, WC96, PCFW95, DHW95, FT96].

(closed) *nested transactions* whereas previous rule based approaches were limited to flat transactions. A Statelog procedure consists of a set of ECA-style Datalog rules each of which defines either

- a non-state-changing *query*, ie a (potentially recursive) view, or
- an *action*, ie
  - a *primitive update* request (insert, delete, modify),
  - a *complex update* request (procedure call), or
  - an *external action* to be issued by the database system, or
- a transaction control predicate.

System-immanent *frame* and *procedure rules* provide a declarative specification of state transitions and integrity preserving policies within the logical language without bothering the user with those problems.

The paper is structured as follows. The remainder of this section is devoted to an introduction to (flat) Statelog. Section 2 introduces the main ideas of procedures and nested transactions and their realization in Statelog. In Section 3 the syntax of the language is defined, Section 4 provides some examples. A logic programming semantics for Statelog is presented in Section 5 using a compilation from Statelog to logic programs. Section 6 defines a model-theoretic Kripke-style semantics which provides the connection between the intuitive understanding of procedure calls and the underlying state-oriented conceptual model. We give an overview on related work in Section 7 and conclude in the last section.

1.1 Statelog: Datalog and States

In this section, we introduce the basic ideas underlying *flat Statelog*[3] [LHL95]. The extended framework with procedures and nested transactions is described in Section 2.

While for query processing a "one-state logic" like Datalog is sufficient, active state-changing rules require access to different states and delta relations. In Statelog, this is accomplished by *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. The domain of S is $\mathbb{N}_0$, ie relations of a Statelog program evolve over the *linear state space* $\mathbb{N}_0$. S may only occur in state terms. A Statelog *rule* is of the form

$$[S+k_0]H(\bar{X}) \;\leftarrow\; [S+k_1]B_1(\bar{X}_1), \ldots, [S+k_n]B_n(\bar{X}_n) \quad ,$$

where the head $H(\bar{X})$ is an atom, and $B_i(\bar{X}_i)$ are atoms or negated atoms. A rule is *progressive*, resp. *local*, if $k_0 \geq k_i$, resp. $k_0 = k_i$, for all $i \in \{1, \ldots, n\}$. Since past states cannot be changed, we require that all rules are progressive.

Here and in the sequel, we denote by $\bar{X}$ a vector $X_1, \ldots, X_m$ of arguments (variables or terms) of suitable arity; ground terms are denoted by lower case letters.

State Transitions and Frame Rules. The user is relieved from handling states explicitly and may define only actions (including change requests to EDB relations) and views by local rules. The actual state change from $[S]$ to $[S+1]$ is specified by system-generated *frame and procedure rules* (Section 3.2). E.g. the following frame rules define the effect of insert and delete requests in flat Statelog:

$$[S+1]\ \mathsf{R(X)} \leftarrow [S]\ \mathsf{R(X)},\ [S]\ \neg\mathbf{del}\text{:}\mathsf{R(X)}.$$
$$[S+1]\ \mathsf{R(X)} \leftarrow [S]\ \mathbf{ins}\text{:}\mathsf{R(X)}.$$

Here R is an EDB relation, while **del:**R, **ins:**R, **mod:**R denote user-definable *request relations* (also called *delta relations*, or *deltas*) which are used to issue update requests.

[3] We refer to the language described in [LHL95] as *flat Statelog*, since the state space is $\mathbb{N}_0$, ie a flat structure. In contrast, *Statelog with procedures* uses a hierarchical state space to model the execution of nested transactions.

Execution Model. In addition to EDB, IDB, and request relations, there are relations which model the interface to the external application domain: *External events* $\triangleright ev(\bar{x})$ occurring within a certain "atomic" time interval are mapped to the current state $[S]$. E.g. an external temporal event may be denoted as $\triangleright$daily(Date), or, raised by some monitoring device, it may specify an event from the real world like $\triangleright$runway_clear(R), etc. *External actions* $\triangleleft a(\bar{x})$ are requests to perform some action in the application domain (like $\triangleleft$move(Thing,From,To)). It is assumed that external actions issued by the database system have no side-effects on the state of the database.

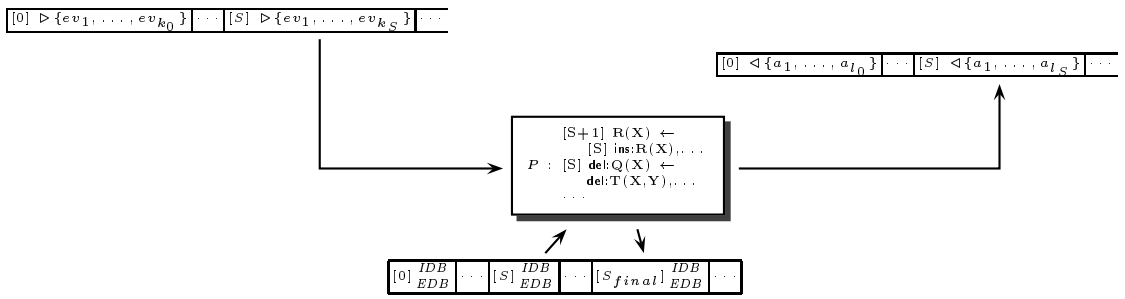

Fig. 1. Mapping of External Events to External Actions and Database States

Triggered by the occurrence of one or more external events $\triangleright ev_i$ in $[S]$, the corresponding rules become activated. According to the additional conditions given in the rule bodies, the database is queried and the actions specified in the rule heads are performed using frame rules (for *internal actions*, ie update requests) or signaled to the outside (external actions). In the subsequent state $[S+1]$, $\triangleright ev_i$ is regarded as *consumed*. Thus, for a current database state $[S]D$ and a set of events, the logical semantics of a program P yields a sequence of intermediate states, a set of external actions, and a new database state $[S_{final}]D$, see Fig. 1. In flat Statelog, a *transaction* beginning at $[S]$ *terminates* when there are no changes to successive states, ie when $[S_{final}]D = [S_{final}+1]D$.

2 Transaction-Oriented Hierarchical Structuring

The need for structuring capabilities and a more elaborated transaction model can be exemplified as follows (this example is adopted from [MW88, Che95]):

Example 1 (To Hire or Not to Hire). The employee Emp with salary Sal should be hired for department Dept provided the average salary *after* the update does not exceed a certain limit. This may be expressed in flat Statelog (or similarly in the related XY-Datalog approach [Zan93]) as follows:[4]

$[S]$ newemp(Emp,Sal,Dept), $[S+1]$ checksal(Dept) $\leftarrow$ $[S]$ $\triangleright$hire(Emp, Sal, Dept).
$[S]$ **ins:**empsal(Emp,Sal), **ins:**empdep(Emp,Dept) $\leftarrow$ $[S]$ newemp(Emp,Sal,Dept).
$[S+1]$ **del:**empsal(Emp,Sal),**del:**empdep(Emp,Dept) $\leftarrow$
    $[S]$ $\triangleright$hire(Emp,Sal,Dept), $[S+1]$ $\neg$check_ok.
$[S]$ check_ok $\leftarrow$ $[S]$ checksal(Dept), avg(Dept, Amt), Amt $<$ 50000.

When an external event $\triangleright$hire(Emp,Sal,Dept) occurs in the current state $[S]$, the new employee is preliminarily inserted and the new average salary is checked in $[S+1]$ (rules 1 and 2). If it exceeds the admissible amount, the effect of the insert is undone (rule 3).

[4] It is assumed that the average avg(Dept,Amt) for each department is given by other rules. A conjunction $H_1, H_2 \leftarrow B$ in the head of a rule is equivalent to two rules $H_1 \leftarrow B$ and $H_2 \leftarrow B$, ie denotes simultaneous "execution".

Problems. Although the above program specifies the desired transaction, there are some potential pitfalls and drawbacks with this "flat" approach:

- Undoing the effect of changes (here: the compensation of insertions by corresponding deletions) has to be programmed by the rule designer. However, it is often desirable to *automatically* propagate the failure of a subtransaction like checksal.
- There is no structure which allows grouping of semantically closely related rules. E.g. it is useful to view the insertion using newemp (rule 2) as an atomic subtransaction callable by the top-level transaction hire.
- The effects of *ephemeral changes* [Zan95], ie 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 ▷hire(...) occurs in $[S]$, the delete requests **ins:**empsal and **ins:**empdep may trigger other active rules, although in $[S+2]$ the updates are revoked. This may lead to unjustified (re)actions by other rules, similar to those described in [Zan95].

2.1 Procedures and Nested Transactions

In order to solve these problems, we propose the concept of Statelog *procedures*. A procedure π is a set of *local* Statelog rules with an import/export interface describing which relations are visible and updatable by π. When π is called at runtime, it defines a transaction T_π by issuing primitive updates (through request relations) and/or calling other procedures which in turn define subtransactions etc. T_π terminates either successfully, ie if commit is true in some state, or aborts. When π calls another procedure ρ, a subtransaction T_ρ is started whose results are either incorporated in T_π, if T_ρ commits, or discarded otherwise. From the point of view of the calling transaction T_π, the subtransaction T_ρ is atomic, therefore requests derived directly within T_π and those submitted by T_ρ should be indistinguishable. This is achieved by frame and procedure rules (Section 3.2).

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

The way in which procedures execute (ie as nested transactions) induces a hierarchical structure of the state space. The model-theoretic foundation of this concept is given by Kripke structures with different accessibility relations, see Section 6. We represent this hierarchy by *transaction frames* and *complex state terms* which extend the flat state terms $[S+k]$ of [LHL95].

The usual partitioning of the signature into base relations (EDB) and derived relations (IDB) is carried over to the hierarchical concept: base relations are passed from the current state to successor states (modulo the changes given by deltas) while IDB relations are not passed on but are rederived when needed. *All* user-defined changes to base relations have to be done via *requests*, ie by using request relations *ins:R*, *del:R*, *mod:R* (also called *delta relations*, or *deltas*).

Protocol relations insd:R, *deld:R*, *modd:R* (for *inserted*, *deleted*, *modified*, respectively) accumulate the *net effect* of user-defined requests and are automatically maintained by the system. Requested changes become effective in the transition to the successor state. Finally, there is a set Π of procedure names and transaction control relations *BOT*, *EOT*, *abort* etc.

In this structured model, Example 1 can be specified as follows ($\pi \otimes \rho$ denotes sequential composition, ie first do π, then do ρ[5]):

Example 2 (To Hire or Not to Hire revisited). The procedure hire defines an atomic transaction: First it calls newemp to insert the employee into the database, then it calls checksal to check the average salary. If this exceeds a certain amount, the transaction aborts. In the rules of hire it is specified that in this case hire should also abort, making no effects visible to its parent transaction.

[5] The symbol "⊗" is borrowed from [BK93], where it is called *serial conjunction*.

```
proc hire(Emp,Sal,Dept); ▽empsal,empdep; △empsal,empdep;
  initial:  newemp(Emp,Sal,Dept) ⊗ checksal(Dept) ← .
  always: abort ← aborted:checksal(Dept).
endproc
proc newemp(Emp,Sal,Dept); △empsal,empdep;
  initial:  ins:empsal(Emp,Sal) ←.
            ins:empdep(Emp,Dept) ←.
endproc
proc checksal(Dept); ▽empdep,empsal;
  initial:  abort ← avg(Dept,Amt), ¬Amt<50000.
endproc
```

The symbols "▽" and "△" denote import resp. export of relations (Section 2.3). The declarations initial, always (and final) specify when the corresponding rules should be executed, ie in the first state of the subtransaction, in every state, or in the last state, respectively; see Section 3.1 for details. hire may be called automatically from the top-level transaction using a rule of the form

$$\text{hire(Emp,Sal,Dept)} \leftarrow \triangleright\text{hire_someone(Emp,Sal,Dept)}.$$

Whenever the external event ▷hire_someone occurs, hire is executed as an atomic transaction. Fig. 3 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).

2.2 Hierarchical State Space

In the hierarchical context, state terms are more complex and extend those of flat Statelog: 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*. Given a set Π of procedure names, the syntax of frame terms $\mathcal{F}(\Pi)$ and state terms $\mathcal{Z}(\Pi)$ over Π is defined recursively:

1. $[\varepsilon]$ is a *frame term*.
2. $[F.n]$ is a *state term*, if $[F]$ is a frame term and $n \in \mathbb{N}_0$.
3. $[Z.\pi(\bar{x})]$ is a *frame term*, if $[Z]$ is a state term, $\pi \in \Pi$ is an n-ary procedure name, and $\bar{x}$ is a vector of n terms from the underlying Herbrand universe.

With these, a hierarchically structured state space is constructed (Fig. 2): the *initial frame* $[\varepsilon]$ denotes the *top-level transaction*, its initial state is $[\varepsilon.0]$. Let $[Z]$ be 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 $[F] = [Z.\pi(\bar{x})]$, the first state of the transaction is $[F.0] = [Z.\pi(\bar{x}).0]$. The successor state of $[F.n]$ (on the same level) is $[(F.n)+1] := [F.(n+1)]$. The grouping of states into frames is defined as

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

which implies that every state $[F.n]$ belongs to exactly one frame $[F]$.

Using this representation, the frames $[Z.\pi(\bar{x})]$ and $[Z.\rho(\bar{y})]$ induced by *different* parallel procedure calls of π and ρ in the *same state* $[Z]$ 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*, $[Z_1.\pi(\bar{x})]$ and $[Z_2.\pi(\bar{x})]$, can also be distinguished.

2.3 Signatures and Visibility

When procedures execute in parallel or as nested transactions, the question arises which "versions" of relations should be visible within a transaction. In Statelog this issue is resolved using the hierarchical state space:

Assume two procedures π and ρ are called simultaneously in the same state, say $[F.2]$ (Fig. 2). This creates two different frames $[F.2.\pi]$ and $[F.2.\rho]$, thereby allowing π and ρ to maintain their

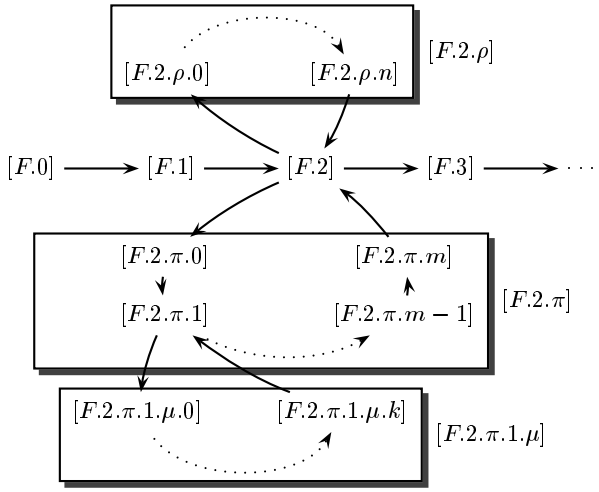

Fig. 2. States and Frames

own "view" of relations. Initially, ie in $[F.2.\pi.0]$ and $[F.2.\rho.0]$, π and ρ "see" the same versions of relations. π and ρ may update their versions subsequently and return the accumulated changes to $[F.2]$ as soon as they commit – here, in $[F.2.\pi.m]$ and $[F.2.\rho.n]$. For handling contradictory requests, conflict resolution policies can be specified by appropriate frame rules. Real parallelism is supported since frames maintain their own versions of relations, and simultaneously called procedures like π and ρ can execute independently and in parallel.

Given a frame $[F]$, we denote by Σ_F the signatures visible in $[F]$. In particular, Σ_F^{EDB} and Σ_F^{IDB} denote base resp. derived relations visible in $[F]$. For the top-level frame $[\varepsilon]$ this induces signatures $\Sigma_\varepsilon^{EDB} := EDB$ and $\Sigma_\varepsilon^{IDB} := IDB$, where EDB and IDB denote the global database scheme.

The local signature $\Sigma_{F.n.\pi(\bar{x})}$ of a frame $[F.n.\pi(\bar{x})]$ is defined in terms of imported and private relations of the procedure π which defines that frame:

Let Σ_π^{Imp} and Σ_π^{Exp} denote the signatures of *imported relations* and *exported relations* of π, respectively. All relation names occurring in π but neither in Σ_π^{Imp} nor in Σ_π^{Exp} are assumed private and belong to the *private signature* of π which is split into *private base relations* Σ_π^{EDB} and *private derived relations* Σ_π^{IDB}.

Σ^{EDB} and Σ^{IDB} denote the union of all EDB/IDB relations used in procedures of a given program. Using private and imported relations, the visible relations of a frame are defined as follows:

Definition 1 (Visible EDB/IDB Relations). Let $[F'] := [F.n.\pi(\bar{x})]$ be the frame created by the procedure call $\pi(\bar{x})$ in $[F.n]$. Then the signature of *visible EDB/IDB relations* of the new frame $[F']$ is given by

$$\Sigma_{F'}^{EDB} := (\Sigma_F^{EDB} \cap \Sigma_\pi^{Imp}) \cup \Sigma_\pi^{EDB} , \qquad \Sigma_{F'}^{IDB} := (\Sigma_F^{IDB} \cap \Sigma_\pi^{Imp}) \cup \Sigma_\pi^{IDB}$$

□

Therefore, $[F']$ "sees" all imported EDB/IDB relations which are visible in the frame $[F]$ of the calling transaction and all private EDB/IDB relations of π.

EDB relations are imported by taking over their extensions from the calling state into the initial state of the subtransaction (rules (D) in Section 3.2). In contrast, IDB relations are imported by including their defining rules into the rule set of the subtransaction frame (cf. Definition 5).

The export of an EDB relation is accomplished by "copying" the contents of the protocol relations of the final state of a subtransaction into the request relations of the parent transaction (rules (E) in Section 3.2).[6]

[6] This provides a natural facility for implementing hypothetical updates: A procedure imports a relation

User-defined rules may change EDB relations only through *request relations*. *Protocol relations* accumulate all non-revoked requests, ie the net effect of changes of a subtransaction is automatically maintained by the system. The extensions of the protocol relations are translated into requests for the calling transaction when the subtransaction commits.

Definition 2 (Request and Protocol Relations). Let Σ_F^{EDB} be the signature of visible EDB relations in a frame $[F]$. Then the signatures Σ_F^{Req} and Σ_F^{Prot} of *request relations* and *protocol relations* of $[F]$ are defined as

$$\Sigma_F^{Req} := \{ins{:}R, del{:}R \mid R \in \Sigma_F^{EDB}\}\ ,\ \Sigma_F^{Prot} := \{insd{:}R, deld{:}R \mid R \in \Sigma_F^{EDB}\}$$

□

Note that of the above-mentioned signatures, only Σ_F^{IDB} and Σ_F^{Req} are user-definable; the relations from Σ_F^{EDB} and Σ_F^{Prot} are maintained by the system.

The interface to the application domain is provided by sets E of *external event names* and A of *external action names*. These induce signatures for external events and actions:

Definition 3 (External Events and Actions). Given sets E and A of external events resp. actions, the signatures for external events and actions are

$$\Sigma^{Ev} := \{\triangleright e \mid e \in E\} \quad , \quad \Sigma^{Act} := \{\triangleleft a \mid a \in A\} \quad .$$

□

Σ^{Ev} is visible (read-only) only within the distinguished procedure **main** (Section 3) defining the top-level transaction $[\varepsilon]$, while Σ^{Act} is visible (write-only) in all frames.

Finally, the global signature contains additional relations for handling procedure calls and transaction management:

Definition 4 (Transaction Management). For a given set Π of procedure names, the signature $\Sigma^{Proc} := \Pi$ is used to represent procedure calls. Transaction control is provided through the signature

$$\Sigma^{Subtr} := \{aborted{:}\pi, committed{:}\pi \mid \pi \in \Sigma^{Proc}\}$$

of relations indicating which subtransactions have committed or aborted, and through the 0-ary relations in

$$\Sigma^{Ctl} := \{BOT, running, EOT, alive, abort\} \quad .$$

□

All relations in Σ^{Subtr} and Σ^{Ctl} are globally visible (but in general have different extensions for each frame). Σ^{Act} and Σ^{Proc} are completely user-defined, abort is partly user-defined, the others are internally defined.

The signature Σ comprises all previously mentioned signatures.

3 Syntax: Programs and Rules

In this section, we describe the syntax of user-defined rules and built-in frame and procedure rules. The logic programming semantics of programs is presented in Section 5.

without exporting it. Then it can operate on this relation without making changes visible to any other transaction.

3.1 User-Defined Rules

Programs and Procedures. A Statelog *program* is a finite set of Statelog procedures. There is a distinguished 0-ary procedure **main** (which is used to define the top-level transaction for the initial frame $[\varepsilon]$). An n-ary Statelog *procedure* π is of the form

$$\begin{array}{l}\textbf{proc } \pi(A_1,\ldots,A_n);\quad \nabla I_1,\ldots,I_k;\quad \triangle O_1,\ldots,O_l;\\ \quad \textbf{initial: } P_{initial}(\pi);\quad \textbf{always: } P_{always}(\pi);\quad \textbf{final: } P_{final}(\pi)\\ \textbf{endproc}\end{array}$$

where the arguments A_i of π are variables that may occur in the rules of $P_{...}(\pi)$. The relations $I_i \in \Sigma^{EDB} \cup \Sigma^{IDB}$ and $O_j \in \Sigma^{EDB}$ denote the imported, resp. exported relations[7]. The $P_{...}(\pi)$ are finite sets of Datalog rules (possibly with negation) of the following form:

$P_{initial}(\pi)$ is the set of *initial rules*. These are only enabled in the initial state of the transaction T_π defined by π and may be used for initialization purposes.
$P_{always}(\pi)$ is the set of *permanent rules*, applicable in all states of T_π.
$P_{final}(\pi)$ defines *final rules* which can be applied only in the last state of T_π. They have to be of the form "*abort* $\leftarrow$ *ic-condition*" and may be used for integrity maintenance: if an inconsistency is detected (*ic-condition* becomes true), the current transaction is automatically aborted.

Rules. The user may define rules only through the sets $P_{...}(\pi)$ above, therefore, all user-definable rules have standard Datalog syntax. Depending on the relation symbol in the head of a rule, the following cases can be distinguished:

Views:	$V(\bar{X})$	$\leftarrow \ldots$	for all $V \in \Sigma^{IDB}$
Change Requests:	*ins:*$R(\bar{X})$	$\leftarrow \ldots$	for all $R \in \Sigma^{EDB}$
	del:$R(\bar{X})$	$\leftarrow \ldots$	for all $R \in \Sigma^{EDB}$
Procedure Calls:	$\pi(\bar{X})$	$\leftarrow \ldots$	for all $\pi \in \Sigma^{Proc}$
External Actions:	$\lhd A(\bar{X})$	$\leftarrow \ldots$	for all $\lhd A \in \Sigma^{Act}$
Transaction Control:	*abort*	$\leftarrow \ldots$	where *abort* $\in \Sigma^{Ctl}$

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 by rules, all changes to base relations have to be accomplished through insert and delete requests. The materialization of these requests is implemented by frame rules which are described in the following section.

Visibility. Every procedure π defines a set of *internal rules* $P(\pi)$ implementing the desired semantics of initial, always and final declarations. For every frame $[Z.\pi(\bar{x})]$ there is a set $P([Z.\pi(\bar{x})])$ of visible local rules, namely rules of π and rules for imported IDB relations.

Definition 5. For a procedure π, the set of *internal rules* $P(\pi)$ is defined as

$$\begin{array}{rl}P(\pi) := & \{h \leftarrow b, BOT \mid h \leftarrow b \in P_{initial}(\pi)\}\\ & \cup \{h \leftarrow b, alive \mid h \leftarrow b \in P_{always}(\pi)\}\\ & \cup \{h \leftarrow b, EOT \mid h \leftarrow b \in P_{final}(\pi)\}\ .\end{array}$$

Using these, the set of *visible local rules* $P([F])$ of a frame is defined as

$$\begin{array}{ll}P([\varepsilon]) & := P(\boldsymbol{main})\\ P([F.n.\pi(\bar{x})]) & := P(\pi) \cup\\ & \{h \leftarrow b \in P([F]) \mid h \in \Sigma_F^{IDB} \cap \Sigma_\pi^{Imp}\} \text{ for all } n \in \mathbb{N}_0, \pi \in \Sigma^{Proc}\end{array}$$

□

The way IDB relations are treated reflects the intention that derived relations are imported by importing their defining rules, whereas EDB relations are imported by taking over their extensions into the initial state of a subtransaction.

[7] W.l.o.g., we assume that relation names are unique, even when the arity is ignored.

3.2 System-Defined Rules

System-generated frame and procedure rules implement the intended semantics of request 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. Let $[F]$ be the current frame. Then the following rules are visible (labels to the right of rules will be used in the compilation into a logic program in Section 5):

Frame Rules. Frame rules specify the correct handling of update requests and transitions. For all EDB relations R visible in F, the following frame rules are also visible:

Updates on EDB relations are executed in the transition to the successor state. EDB relations are propagated to the successor state as long as EOT does not hold:

$$\begin{array}{l}[Z+1]\ R(\bar{X}) \leftarrow [Z]\ \mathit{ins{:}}R(\bar{X}), \neg EOT. \\ [Z+1]\ R(\bar{X}) \leftarrow [Z]\ R(\bar{X}), \neg \mathit{del{:}}R(\bar{X}), \neg EOT.\end{array} \qquad (B)$$

The non-revoked updates of $[F]$ are accumulated in protocol relations:

$$\begin{array}{l}[Z+1]\ \mathit{insd{:}}R(\bar{X}) \leftarrow [Z]\ \mathit{ins{:}}R(\bar{X}), \neg EOT. \\ [Z+1]\ \mathit{insd{:}}R(\bar{X}) \leftarrow [Z]\ \mathit{insd{:}}R(\bar{X}), \neg \mathit{del{:}}R(\bar{X}), \neg EOT. \\ [Z+1]\ \mathit{deld{:}}R(\bar{X}) \leftarrow [Z]\ \mathit{del{:}}R(\bar{X}), \neg EOT. \\ [Z+1]\ \mathit{deld{:}}R(\bar{X}) \leftarrow [Z]\ \mathit{deld{:}}R(\bar{X}), \neg \mathit{ins{:}}R(\bar{X}), \neg EOT.\end{array} \qquad (B)$$

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

$$\begin{array}{l}[Z]\ \mathit{running} \leftarrow [Z]\ \mathit{ins{:}}R(\bar{X}), \neg R(\bar{X}). \\ [Z]\ \mathit{running} \leftarrow [Z]\ \mathit{del{:}}R(\bar{X}), R(\bar{X}).\end{array} \qquad (C)$$

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

$$\begin{array}{l}[Z]\ EOT \leftarrow [Z]\ BOT, \neg \mathit{running}. \\ [Z+1]\ EOT \leftarrow [Z]\ \mathit{running}, \neg \mathit{abort}, [Z+1]\ \neg \mathit{running}.\end{array} \qquad (A)$$

The internal event abort terminates a transaction prematurely:

$$[Z]\ EOT \leftarrow [Z]\ \mathit{abort}. \qquad (A)$$

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

$$[Z]\ \mathit{abort} \leftarrow [Z]\ \mathit{ins{:}}R(\bar{X}), \mathit{del{:}}R(\bar{X}). \qquad (C)$$

States of a frame are *alive* if the transaction really uses them:

$$\begin{array}{l}[Z]\ \mathit{alive} \leftarrow [Z]\ BOT. \\ [Z+1]\ \mathit{alive} \leftarrow [Z]\ \mathit{running}, \neg EOT.\end{array} \qquad (A)$$

Procedure Rules. Procedure rules implement the semantics of procedure calls, ie the execution of subtransactions. For all procedures π, in a frame $[F]$ the following procedure rules are visible: A procedure call creates the initial state of a new frame, signals BOT and initializes all imported relations:

$$[Z.\pi(\bar{X}).0]\ BOT \leftarrow [Z]\ \pi(\bar{X}). \qquad (A)$$

$$\text{for all } R \in \Sigma_\pi^{Imp} \cap \Sigma_F^{EDB}\text{:} \quad [Z.\pi(\bar{X}).0]\ R(\bar{Y}) \leftarrow [Z]\ R(\bar{Y}), \pi(\bar{X})\ . \qquad (D)$$

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 according to the export specification:

$$\begin{array}{l}\text{for all } R \in \Sigma_\pi^{Exp} \cap \Sigma_F^{EDB}: \\ [Z]\ ins{:}R(\bar{Y}) \leftarrow [Z]\ \pi(\bar{X}), [Z.\pi(\bar{X}).N]\ insd{:}R(\bar{Y}), EOT, \neg abort. \\ [Z]\ del{:}R(\bar{Y}) \leftarrow [Z]\ \pi(\bar{X}), [Z.\pi(\bar{X}).N]\ deld{:}R(\bar{Y}), EOT, \neg abort.\end{array} \quad (E)$$

Thus, $[Z.\pi(\bar{x}).n]\ insd{:}R(\bar{y})$ and $[Z.\pi(\bar{x}).n]\ deld{:}R(\bar{y})$ with $[Z]\ \pi(\bar{x})$, $[Z.\pi(\bar{x}).n]\ EOT$, $\neg\ abort$ are equivalent to requests $[Z]\ ins{:}R(\bar{y})$ resp. $[Z]\ del{:}R(\bar{y})$ which are derived directly.
Parent transactions also perform some bookkeeping about committed and aborted subtransactions:

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

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

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

3.3 Sequential Composition

To provide sequential execution of procedures as a built-in, the signature is extended with a connective "$\otimes$", which may be only used *in the head* of user-defined rules, e.g.

$$(A \otimes B) \leftarrow\ body\ .$$

means *first do A, then do B*, if *body* is true. $(A \otimes B)$ is compiled into two internal rules:

$$\begin{array}{l}[Z]\ A, running \leftarrow [Z]\ (A \otimes B). \\ [Z+1]\ B \leftarrow [Z]\ (A \otimes B), \neg abort.\end{array}$$

The previous scheme generalizes to the polyadic case $A_1 \otimes \cdots \otimes A_k$ in the obvious way.

Sequential composition is not only useful to serialize the execution of procedures, but also for directly manipulating relations. E.g. the rule

$$del{:}R(\bar{X}) \otimes ins{:}R(\bar{X}) \leftarrow body\ .$$

generates exactly one intermediate state in which $\bar{X}$ is not in R. This can be useful in defining hypothetical updates, e.g. to test this intermediate state and see what would happen if $R(\bar{X})$ were deleted.

4 Examples

The hierarchical transaction model with import and export declarations allows a flexible treatment of several interesting features of databases, like for example the following:

- Static integrity constraints can be implemented by using the final rules for aborting transactions (Example 2).
- 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 2).
- Hypothetical updates: every transaction can work on relations which are imported but not exported 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.

Example 3 (To Hire or Not to Hire: State Space and Database).
The program given in Example 2 creates the frames and database states given in Fig. 3. Frames are presented by shadowed boxes, states are presented by ordinary boxes. In all states, the upper entry gives 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.
In this example it is assumed that the average salary exceeds the admissible amount, so that the transaction hire(john,60000,d1) aborts, making no effects visible to its parent transaction.

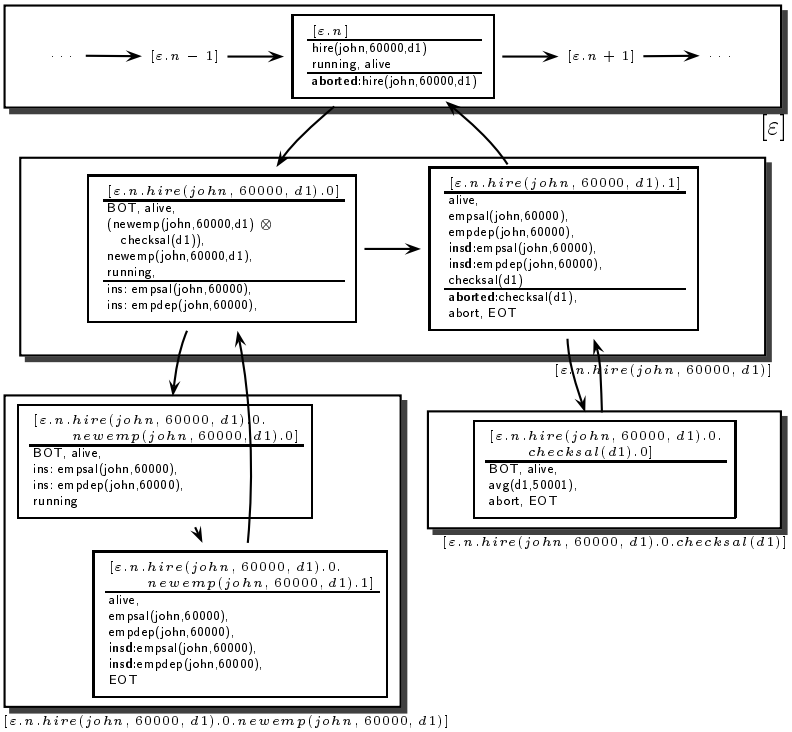

Fig. 3. Frames and Database States (cf. Example 2)

Example 4 (The Christmas-Problem). Consider a relation empl(Employee, BirthDay, Salary) with the obvious meaning. We want to implement the following, informally given procedure: *Every employee shall be given a salary raise by 5% at his/her birthday; on Christmas every employee shall get an extra $1000.* This is accomplished in flat Statelog as follows [LHL95]:

$[S+1]$ **mod:**empl(E,Bday,Sal$\rightsquigarrow$Sal1) $\leftarrow$
  $[S]$ $\triangleright$daily, date(Day), Day=Bday, empl(E,Bday,Sal), Sal1:= Sal*1.05).
$[S+1]$ **mod:**empl(E,Bday,Sal$\rightsquigarrow$Sal1) $\leftarrow$
  $[S]$ $\triangleright$daily, date(Day), xmas(Day), empl(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 $\otimes$ inc_bday is used by the top-level transaction incsal to specify the order of execution:

proc main; **always:** incsal(Day) $\leftarrow$ $\triangleright$daily, date(Day). **endproc**

proc incsal(Day); ∇empl; Δempl;
  initial: inc_xmas(Day) $\otimes$ inc_bday(Day) $\leftarrow$.
endproc

proc inc_xmas(Day); ∇empl; Δempl;
  initial: mod:empl(E,Bday,Sal $\rightsquigarrow$Sal1) $\leftarrow$
    xmas(Day),empl(E,Bday,Sal), Sal1:= Sal+1000.
endproc

proc inc_bday(Day); ∇empl; Δempl;
  initial: mod:empl(E,Bday,Sal $\rightsquigarrow$Sal1) $\leftarrow$
    Day=Bday, empl(E,Bday,Sal), Sal1:= Sal*1.05.
endproc

If "$\otimes$" 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 corresponding frame rules.

5 Logic Programming Semantics

In this section, we define the declarative semantics of a Statelog program P by a compilation into a logic program P^*, which in turn defines a certain canonical Herbrand-style model.

5.1 Compilation Scheme

The basic idea of the compilation is to code state terms into predicates, ie every term $[Z]R(\bar{X})$ is transformed into a term $R([Z], \bar{X})$. In the following definitions, π ranges over the finite set of procedure names which is given a priori by the user program (ie, the rule scheme is applied for all $\pi \in \Sigma^{Proc}$). Moreover, all $\in$-expressions may be defined by rules in the obvious way, but are omitted to avoid unnecessary details.

In a first step, state and frame terms, the visibility of relations (Section 2.3), and the import of IDB rules are defined:[8]

Definition 6 (Frames and States). If a procedure π (with arguments $\bar{X}$) is called, a frame and all necessary states are created:

$\mathsf{state}([\varepsilon.0])$.
$\mathsf{state}([Z.\pi(\bar{X}).0]) \leftarrow \pi([Z], \bar{X})$.
$\mathsf{state}([Z+1]) \leftarrow \mathsf{state}([Z]), \mathsf{alive}([Z])$.

$\mathsf{frame}([\varepsilon])$.
$\mathsf{frame}([Z.\pi(\bar{X})]) \leftarrow \pi([Z], \bar{X})$.

□

Definition 7 (Visible EDB). The relation $\mathsf{visible} : \mathcal{F} \times \Sigma^{EDB}$ defines Σ_F^{EDB} (Definition 1), ie $\mathsf{visible}([F], R)$ means that R is a visible EDB relation in frame $[F]$:

$\mathsf{visible}([\varepsilon],R) \leftarrow R \in \Sigma_{main}^{EDB}$.
$\mathsf{visible}([Z.\pi(\bar{X})],R) \leftarrow \mathsf{frame}([Z.\pi(\bar{X})]),\ R \in \Sigma_\pi^{EDB}$.
$\mathsf{visible}([F.N.\pi(\bar{X})],R) \leftarrow \mathsf{frame}([F.N.\pi(\bar{X})]), \mathsf{visible}([F],R),\ R \in \Sigma_\pi^{Imp}$.

□

Definition 8 (Visible IDB). The relation $\mathsf{imports} : \mathcal{F} \times \Sigma^{IDB} \times \Sigma^{Proc}$ defines which IDB *rules* are visible in a frame, ie $\mathsf{imports}([F], R, P)$ means that in frame $[F]$ the IDB rules for R from procedure P are imported:

$\mathsf{imports}([\varepsilon], R, \pi) \leftarrow R \in \Sigma_{main}^{IDB}$.
$\mathsf{imports}([Z.\pi(\bar{X})], R, \pi) \leftarrow \mathsf{frame}([Z.\pi(\bar{X})]),\ R \in \Sigma_\pi^{IDB}$.
$\mathsf{imports}([F.N.\pi(\bar{X})], R, P) \leftarrow \mathsf{frame}([F.N.\pi(\bar{X})]), \mathsf{imports}([F], R, P),\ R \in \Sigma_\pi^{Imp}$

□

The main step consists of a compilation of the various rules from Section 3 (nesting of procedure calls, import of EDB/IDB relations, materialization of requests, etc.) into a logic program P^*. In particular, the restricted visibilities of relations have to be considered:

Definition 9 (Compilation $P \mapsto P^*$). Apart from the preceding rules, the compiled program P^* of a Statelog program P contains the following rules:

1a. On the highest level the rules of the main part are activated:
for all rules $h(\bar{X}_0) \leftarrow b_1(\bar{X}_1), \ldots, b_n(\bar{X}_n) \in P(\boldsymbol{main})$:

$$h([\varepsilon.N], \bar{X}_0) \leftarrow b_1([\varepsilon.N], \bar{X}_1), \ldots, b_n([\varepsilon.N], \bar{X}_n),\ \mathsf{state}([\varepsilon.N]).$$

[8] Expressions of the form $[F.N]$ denote standard terms in the obvious way, ie $f_{[]}(f_{dot}(F, N))$.

1b. In every frame, the rules of the corresponding procedure are activated:
for every $\pi \in \Sigma^{Proc}$ and all rules $h(\bar{X}_0) \leftarrow b_1(\bar{X}_1), \ldots, b_n(\bar{X}_n) \in P(\pi)$:

$$h([Z.\pi(\bar{Y}_0).N], \bar{X}_0) \leftarrow b_1([Z.\pi(\bar{Y}_0).N], \bar{X}_1), \ldots, b_n([Z.\pi(\bar{Y}_0).N], \bar{X}_n), \\ \mathsf{state}([Z.\pi(\bar{Y}_0).N]).$$

2. In all frames the appropriate defining rules of imported IDB relations are used: for every $\pi \in \Sigma^{Proc}$ and all rules $h(\bar{X}_0) \leftarrow b_1(\bar{X}_1), \ldots, b_n(\bar{X}_n) \in P(\pi)$:

$$h([F.N], \bar{X}_0) \leftarrow b_1([F.N], \bar{X}_1), \ldots, b_n([F.N], \bar{X}_n), \\ \mathsf{state}([F.N]),\ \mathsf{imports}([F], h, \pi).$$

3. The application of frame and procedure rules from Section 3.2 has to be restricted:

3a. In every state, all rules marked with (A) are activated: for all such rules $[Z_0]\ h(\bar{X}_0) \leftarrow [Z_1]\ b_1(\bar{X}_1), \ldots, [Z_n]\ b_n(\bar{X}_n)$ with state variable Z in all Z_i:

$$h([Z_0], \bar{X}_0) \leftarrow b_1([Z_1], \bar{X}_1), \ldots, b_n([Z_n], \bar{X}_n),\ \mathsf{state}([Z]).$$

3b. Frame rules marked with (B) are restricted to visible EDB relations: for all such rules $[Z+1]\ h(\bar{X}_0) \leftarrow [Z]\ b_1(\bar{X}_1), \ldots, b_n(\bar{X}_n)$ with EDB relation R:

$$h([F.N+1], \bar{X}_0) \leftarrow b_1([F.N], \bar{X}_1), \ldots, b_n([F.N], \bar{X}_n), \\ \mathsf{state}([F.N]),\ \mathsf{visible}([F], R).$$

3c. Frame rules marked with (C) are also restricted to visible EDB relations: for all such rules $[Z]\ h \leftarrow [Z]\ b_1(\bar{X}_1), \ldots, b_n(\bar{X}_n)$ with EDB relation R:

$$h([F.N]) \leftarrow b_1([F.N], \bar{X}_1), \ldots, b_n([F.N], \bar{X}_n),\ \mathsf{state}([F.N]),\ \mathsf{visible}([F], R).$$

3d. Procedure rules marked with (D) are restricted to imported EDB relations: for all such rules $[Z.\pi(\bar{X}).0]\ R(\bar{Y}) \leftarrow [Z]\ R(\bar{Y}), \pi(\bar{X})$ with EDB relation R:

$$R([Z.\pi(\bar{X}).0], \bar{Y}) \leftarrow R([Z], \bar{Y}), \pi([Z], \bar{X}),\ \mathsf{state}([Z]),\ R \in \Sigma_\pi^{Imp}.$$

3e. Procedure rules marked with (E) are restricted to exported EDB relations: for all such rules $[Z]\ h(\bar{Y}) \leftarrow [Z]\ \pi(\bar{X}),\ [Z.\pi(\bar{X}).N]\ b_1(\bar{Y}), \ldots, b_n(\bar{Y})$ with EDB relation R:

$$h([Z], \bar{Y}) \leftarrow \pi([Z], \bar{X}),\ b_1([Z.\pi(\bar{X}).N], \bar{Y}), \ldots, b_n([Z.\pi(\bar{X}).N], \bar{Y}), \\ \mathsf{state}([Z.\pi(\bar{X}).N]),\ R \in \Sigma_\pi^{Exp}.$$

□

Note that the generated rules may be safely evaluated in a bottom-up style, since all rules are range-restricted provided the user-defined rules are range-restricted themselves.[9] Furthermore, by Definition 6 frames and states are only created when needed by the computation.

5.2 Semantics and Termination

The semantics of a Statelog program P depends on an EDB and a set EB of external events which have occurred in the current state and is given as a model $\mathcal{M}$.

Similar to [LHL95] one can find a syntactical condition which ensures that rules are *state-stratified* (ie, stratified within a state). Since all rules are progressive, this implies that P^* is locally stratified and therefore has a unique *perfect model* [Prz88]. In case rules are not necessarily state-stratified, the *well-founded model* [VG89, VGRS91] provides a natural and generally accepted semantics. As it extends the perfect model semantics – ie coincides with the perfect model on locally stratified semantics – we use it as the canonical model $\mathcal{M}$:

Definition 10 (Event Base). The *event base* $EB := \{ev(\bar{x}) \mid ev(\bar{x}) \text{ is signaled}\}$ is the set of all external events signaled in the current state.

□

[9] A rule r is *range-restricted* if every variable in r occurs positively in the body of r.

Definition 11 (Semantics of P). The *semantics* $\mathcal{M}(P, EDB, EB)$ of a Statelog program P w.r.t. a database EDB and an event base EB is the well-founded model of

$$P^* \cup \{\ r([\varepsilon.0], \bar{x}) \leftarrow .\ \mid r(\bar{x}) \in EDB\} \cup \{\ \triangleright ev([\varepsilon.0], \bar{x}) \leftarrow .\ \mid ev(\bar{x}) \in EB\}$$ □

Termination of rules can be guaranteed by enforcing that only finite models are actually generated:[10]

Definition 12 (Termination). Given an EDB and an event base EB, a program P *terminates* if there are only finitely many states in $\mathcal{M}(P, EDB, EB)$. □

Since the chosen model-theoretic semantics $\mathcal{M}$ is deterministic, confluence is implied. Moreover, from the following lemma the uniqueness of the final state – if one exists – follows directly.

Lemma 13. *In every frame* $[F]$ *there is at most one state* $[F.m]$ *such that* $\mathcal{M}(P, EDB, EB^n) \models EOT([F.m])$. *In this case* $[F.m{+}1]$ *is an empty state, and there are no states* $[F.m']$ *with* $m' > m{+}1$.

Proof. Frame rules are deactivated when EOT holds, thus EDB and protocol relations are empty in $[F.m{+}1]$. From $\mathcal{M} \models EOT([F.m])$ follows $\mathcal{M} \models \neg alive([F.m{+}1])$, disabling the local rules from $P([F])$ in $[F.m{+}1]$. Thus IDB and request relations are empty, no procedure is called in $[F.m{+}1]$, and no procedure return rule can insert any requests into $[F.m{+}1]$. Hence neither *running*, BOT, EOT, nor *alive* are derivable, so $[F.m{+}1]$ is really an empty state. Thus, $\mathcal{M} \models \neg state([F.m{+}2])$.

Theorem 14. *If a program* P *terminates w.r.t. an* EDB *and* EB*, there is a unique final state* $[\varepsilon.m]$ *such that* $\mathcal{M}(P, EDB, EB) \models EOT([\varepsilon.m])$.

In case of termination, the final state $[\varepsilon.m]$ represents the effect of executing the transaction given by EB and P on the database EDB: if $\mathcal{M} \models \neg abort([\varepsilon.m])$, then $[\varepsilon.m]$ is the new database state reached after executing this transaction. If $\mathcal{M} \models abort([\varepsilon.m])$, then the transaction aborts, and the database remains unchanged.

Note, that the converse of Theorem 14 does not hold, since EOT can be derived even if infinitely many states are nonempty. For example, the following program creates an infinitely deep nesting of procedure calls of π but derives EOT on the top-level in the first state:

proc main; **initial:** $\pi \leftarrow .$, abort$\leftarrow .$ **endproc**
proc π; **initial:** $\pi \leftarrow .$ **endproc**

There are different ways to enforce termination of rule processing, even though the problem of deciding whether a program P terminates for all databases is undecidable in general. One way, similar to that of [Zan95], is to enforce termination at runtime by adjusting frame (and procedure) rules in such a way, that changes may not be revoked. In the presence of procedures, one has the additional requirement, that the *procedure call graph* induced by P is acyclic (local rules may be recursive, of course).

Another approach, pursued in flat Statelog, is the class of *Δ-monotone programs* which guarantees termination *at compile-time* [LHL95]. A similar notion can be defined for Statelog with procedures, but is beyond the scope of this paper.

6 Kripke-Style Semantics

In this section, a model-theoretic Kripke-style semantics is given, which interprets the state space as a suitable Kripke structure. It provides the connection between the intuitive understanding of procedure calls and the state-oriented model obtained by the logic programming semantics and can serve as a basis for formal verification. A class of Kripke structures appropriate to model nested transactions is defined together with the notion of a *minimal Kripke model* of a Statelog program w.r.t. an EDB and an EB. Then the equivalence of the Herbrand-style model (Section 5) and the minimal Kripke model is shown, showing the adequacy of the concept.

The Kripke-style semantics is presented in its two-valued version, thus covering all "well-behaved" computations, ie those where all states are completely defined. This is the case if and only if the well-founded model is total.

[10] Another approach is to use a *finite representation* of infinite models, cf. [CI93].

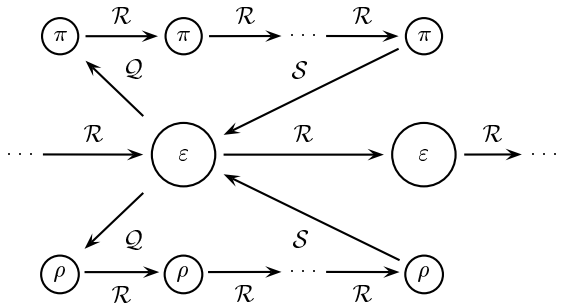

Fig. 4. Hierarchical Kripke Structure

6.1 Statelog Kripke Structures

Definition 15 (Statelog Kripke Structure). A *Statelog Kripke structure* over a given Statelog signature Σ is a tuple $\mathcal{K} = (\mathcal{G}, \mathcal{A}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{U}, \mathcal{M}, \mathcal{P})$, (cf. Fig. 4) where

$\mathcal{G}$ is a set of states,
$\mathcal{A}$ (actions) is a set of procedure names,
$\mathcal{Q}, \mathcal{S} \subseteq \mathcal{G} \times \mathcal{A} \times \mathcal{U}^\omega \times \mathcal{G}$, are two marked accessibility relations between states representing the procedure-call resp. -return relation: $\mathcal{Q}(g, \pi(\bar{x}), g')$ 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 have to be communicated along $\mathcal{S}$.
$\mathcal{R} \subseteq \mathcal{G} \times \mathcal{G}$ is another accessibility relation modeling the temporal successor relation. Let $\mathcal{R}^*$ denote the reflexive transitive closure of $\mathcal{R}$.
$\mathcal{U}$ is the universe of elements,
$\mathcal{M}$ is a function which maps every state to a first-order interpretation over Σ 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).

□

To obtain a simpler notation, every state $g \in \mathcal{G}$ is identified with the corresponding first-order structure $\mathcal{M}(g)$.

Definition 16. A Statelog Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{A}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{U}, \mathcal{M}, \mathcal{P})$ over a signature Σ is a *model* of a Statelog program P over the same signature if

- $\mathcal{A}$ is the set Π of procedure names occurring in P
- External actions are only present in the initial state on the highest hierarchical level: for all $g \in \mathcal{G}$ with $\mathcal{P}(g) \neq P(\boldsymbol{main})$ or $\exists g' : \mathcal{R}(g', g)$, $g|_{\Sigma^{Ev}} = \emptyset$.
- States with no temporal predecessor which are not targets of a procedure call, are initial states on the highest hierarchical level: for all $g \in \mathcal{G}$ with $\{h \mid \mathcal{R}(h, g)\} = \emptyset$ and $\{(h, a) \mid \mathcal{Q}(h, a, g)\} = \emptyset$: $\mathcal{P}(g) = P(\boldsymbol{main})$ and there exists at least one state with this property.
- States which have no temporal predecessor or which are targets of a procedure call are beginnings of transactions and their protocol relations are empty:
  for all $g \in \mathcal{G}$ with $\{h \mid \mathcal{R}(h, g)\} = \emptyset$ or $\exists h, \pi, \bar{x} : \mathcal{Q}(h, \pi(\bar{x}), g)$:
  $g \models BOT$ and $g|_{\Sigma^{Prot}} = \emptyset$.
- Every $g \in \mathcal{G}$ is a model of the corresponding set of local rules: $g \models \mathcal{P}(g)$.
- $\mathcal{Q}$ represents exactly the procedure calls:
  for all $g \in \mathcal{G}$, $\pi \in \mathcal{A}$, $\bar{x} \in \mathcal{U}^\omega$: $g \models \pi(\bar{x}) \Leftrightarrow \exists h : \mathcal{Q}(g, \pi(\bar{x}), h)$.
- $\mathcal{S}$ represents exactly the return-from-subtransaction relation:
  for all $g, g', h' \in \mathcal{G}$, $\pi \in \mathcal{A}$, $\bar{x} \in \mathcal{U}^\omega$:

$$\mathcal{Q}(g, \pi(\bar{x}), g') \wedge \mathcal{R}^*(g', h') \wedge \mathcal{R}(h', h') \Leftrightarrow \mathcal{S}(h', \pi(\bar{x}), g) .$$

- The temporal accessibility relation $\mathcal{R}$ models the relationship between the EDB and request relations of one state and the EDB and protocol relations of the successor state: for all $g, h \in \mathcal{G}$:

$$\begin{aligned}\mathcal{R}(g,h) \;\Leftrightarrow\; & \mathcal{P}(g) = \mathcal{P}(h) \text{ and for all } R \in \Sigma^{EDB}: \\ & h(R) = (g(R) \cup g(ins{:}R)) \setminus g(del{:}R) \text{ and } g(ins{:}R) \cap g(del{:}R) = \emptyset \text{ and} \\ & h(insd{:}R) = (g(insd{:}R) \cup g(ins{:}R)) \setminus g(del{:}R) \text{ and} \\ & h(deld{:}R) = (g(deld{:}R) \cup g(del{:}R)) \setminus g(ins{:}R) \quad .\end{aligned}$$

- The (marked) relation $\mathcal{Q}$ models the procedure calls:
for all $g, g' \in \mathcal{G}$, $\pi \in \mathcal{A}$, $\bar{x} \in \mathcal{U}^\omega$:

$$\begin{aligned}\mathcal{Q}(g, \pi(\bar{x}), g') \;\Rightarrow\; & \mathcal{P}(g') = P(\pi) \cup \{h \leftarrow b \in \mathcal{P}(g) \mid h \in \Sigma^{IDB} \cap \Sigma_\pi^{Imp}\} \text{ and} \\ & \text{for all } R \in \Sigma^{EDB}: g'(R) = g(R) \text{ if } R \in \Sigma_\pi^{Imp}, \\ & \qquad\qquad g'(R) = \emptyset \text{ if } R \notin \Sigma_\pi^{Imp} \quad .\end{aligned}$$

- The (marked) relation $\mathcal{S}$ models the feedback from subtransactions:
for all $g \in \mathcal{G}$, $R \in \Sigma_{EDB}$, $\pi \in \mathcal{A}$:

$$\begin{aligned} & g(ins{:}R) \supseteq \textstyle\bigcup_{\{g' \mid \exists \pi, \bar{x}: \mathcal{S}(g', \pi(\bar{x}), g) \wedge g' \not\models abort \wedge R \in \Sigma_\pi^{Exp}\}} g'(insd{:}R) \quad , \\ & g(del{:}R) \supseteq \textstyle\bigcup_{\{g' \mid \exists \pi, \bar{x}: \mathcal{S}(g', \pi(\bar{x}), g) \wedge g' \not\models abort \wedge R \in \Sigma_\pi^{Exp}\}} g'(deld{:}R) \quad , \\ & g(aborted{:}\pi) = \{\bar{x} \in \mathcal{U}^\omega \mid \exists g': \mathcal{S}(g', \pi(\bar{x}), g) \wedge g' \models abort\} \quad , \\ & g(committed{:}\pi) = \{\bar{x} \in \mathcal{U}^\omega \mid \exists g': \mathcal{S}(g', \pi(\bar{x}), g) \wedge g' \not\models abort\} \quad . \end{aligned}$$

- All $\mathcal{M}(g)$ are minimal s.t. the above-mentioned conditions hold. □

Definition 17. Let $\mathcal{C}(g)$ be the subset of requests which are contributed to g by subtransactions:

$$\begin{aligned}\mathcal{C}(g) := \{ & ins{:}R(\bar{x}) \mid \exists g' \in \mathcal{G}, \pi \in \mathcal{A}, \bar{y} \in \mathcal{U}^\omega: \\ & \mathcal{S}(g', \pi(\bar{y}), g) \wedge R \in \Sigma_\pi^{Exp} \wedge g' \models insd{:}R(\bar{x}) \wedge \neg abort\} \cup \\ \{ & del{:}R(\bar{x}) \mid \exists g' \in \mathcal{G}, \pi \in \mathcal{A}, \bar{y} \in \mathcal{U}^\omega: \\ & \mathcal{S}(g', \pi(\bar{y}), g) \wedge R \in \Sigma_\pi^{Exp} \wedge g' \models deld{:}R(\bar{x}) \wedge \neg abort\}\end{aligned}$$ □

Lemma 18. *The temporal successor relation $\mathcal{R}$ is deterministic:*

$$\text{for all } g, h, h' \in \mathcal{G}: \; \mathcal{R}(g,h) \wedge \mathcal{R}(g,h') \;\Rightarrow\; h = h' \quad .$$

Definition 19. A *computation path* in a Statelog Kripke structure $\mathcal{K}$ is a sequence $(g_1, g_2, \ldots)$ with $\mathcal{R}(g_i, g_{i+1})$ for all i. □

Computation paths in the Kripke model correspond to frames in the Herbrand model. Since $\mathcal{R}$ is deterministic, in every model $\mathcal{K}$ of P, for every $g \in \mathcal{G}$ there is exactly one maximal (infinite, but possibly becoming stationary) computation path through g.

Definition 20. The non-extendable sequences in $\mathcal{R}^*$ are collected in a relation

$$\mathcal{R}^\wedge(g,h) \;:\Leftrightarrow\; \mathcal{R}^*(g,h) \wedge \neg \exists h' \neq h: \mathcal{R}(h,h') \quad ,$$

and for $\pi \in \Sigma^{Proc}$, $\bar{x} \in \mathcal{U}^\omega$, $g \in \mathcal{G}$ such that $g \models \pi(\bar{x})$, let

$$(\pi(\bar{x}))(g) := h \in \mathcal{G} \text{ such that } \exists g' \in \mathcal{G}: \mathcal{Q}(g, \pi(\bar{x}), g') \wedge \mathcal{R}^\wedge(g', h)$$

denote the result of executing $\pi(\bar{x})$ in state g. □

Using this definition, $\mathcal{C}(g)$ can be characterized without explicitly mentioning $\mathcal{S}$:

$$\begin{aligned}\mathcal{C}(g) = \{ & ins{:}R(\bar{x}) \mid \exists \pi \in \mathcal{A}, \bar{y} \in \mathcal{U}^\omega: \\ & g \models \pi(\bar{y}) \wedge R \in \Sigma_\pi^{Exp} \wedge (\pi(\bar{y}))(g) \models insd{:}R(\bar{x}) \wedge \neg abort\} \cup \\ \{ & del{:}R(\bar{x}) \mid \exists \pi \in \mathcal{A}, \bar{y} \in \mathcal{U}^\omega: \\ & g \models \pi(\bar{y}) \wedge R \in \Sigma_\pi^{Exp} \wedge (\pi(\bar{y}))(g) \models deld{:}R(\bar{x}) \wedge \neg abort\} \quad .\end{aligned}$$

In the following, for a set $\mathcal{I}$ of facts and a logic program P, let $\Phi_P(\mathcal{I})$ denote the set of true atoms in the well-founded model of $P \cup \mathcal{I}$.

Theorem 21. *For every Statelog program P, database EDB, and event base EB, there is a unique* minimal *Kripke model (ie with a minimal number of states) of P with a distinguished initial state $g_0 \in \mathcal{G}$ such that $\mathcal{P}(g_0) = \boldsymbol{main}$ and $\mathcal{M}(g_0) = \Phi_{\boldsymbol{main}}(EDB \cup EB)$.*

Corresponding to Theorem 14, we have

Theorem 22. *If the minimal model $\mathcal{K}$ of a Statelog program P, a database EDB, and an event base EB is finite and $\mathcal{R}$ has no cycles of length > 1, then there is a unique computation path $(g_0, g_1, \ldots, g_n, g_n, \ldots)$ with $g_n \models EOT$.*

6.2 Adequacy of Statelog Kripke Structures

Theorem 23 (Adequacy). *Statelog Kripke structures are an adequate model of the intended intuitive semantics of nested transactions:*

- *EDB relations are changed exactly via requests:*
  for all $R \in \Sigma^{EDB}$, $\bar{x} \in \mathcal{U}^\omega$, $g, h \in \mathcal{G}$:
  if $(g,h) \in \mathcal{R}$ then $h \models R(\bar{x}) \;\Leftrightarrow\; (g \models R(\bar{x}) \wedge g \not\models del{:}R(\bar{x})) \vee g \models ins{:}R(\bar{x})$.
- *Every state contains all requests contributed by subtransactions:*
  for all $g \in \mathcal{G}$: $g \supseteq \mathcal{C}(g)$.
- *IDB relations are derived locally by user-defined rules: for all $g \in \mathcal{G}$, $R \in \Sigma^{IDB}$, $\bar{x} \in \mathcal{U}^\omega$:*
  $g \models R(\bar{x}) \;\Leftrightarrow\; R(\bar{x}) \in \Phi_{\mathcal{P}(g)}(g|_{\Sigma^{EDB}} \cup \mathcal{C}(g))$.
- *Requests are derived by user-defined rules or contributed by subtransactions:*
  for all $g \in \mathcal{G}$, $R \in \Sigma^{IDB}$, $\bar{x} \in \mathcal{U}^\omega$:
  $g \models ins{:}R(\bar{x}) \;\Leftrightarrow\; ins{:}R(\bar{x}) \in \Phi_{\mathcal{P}(g)}(g|_{\Sigma^{EDB}} \cup \mathcal{C}(g))$ (Analogously for del:R).
- *In all states the protocol relations contain all non-revoked changes of the corresponding subtransactions. For imported EDB relations, they subsume the differences between the EDB in the state where the subtransaction was initiated and the current state, while they represent exactly the EDB for non-imported relations: for all $g, h \in \mathcal{G}$:*

$$(g,h) \in \mathcal{QR}^* \;\Rightarrow\; \forall R \in \Sigma^{EDB} \cap \Sigma^{Imp}_\pi : h(R) = (g(R) \cup h(insd{:}R)) \setminus h(deld{:}R)$$
$$\text{and } \forall R \in \Sigma^{EDB} \setminus \Sigma^{Imp}_\pi : h(R) = h(insd{:}R) \quad .$$

6.3 Equivalence of Both Semantics

For every program P, database EDB and event base EB, the Herbrand-style model of P, $\mathcal{M}(P, EDB, EB)$ can be split into states by its state term components and can be mapped bijectively to the minimal model $\mathcal{K}$ of P.

Definition 24. For a Statelog signature Σ, a Herbrand interpretation $\mathcal{H}$ over $\Sigma \cup \mathcal{Z}(\Sigma^{Proc})$ is contained in a Statelog Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{A}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{U}, \mathcal{M}, \mathcal{P})$ if $\mathcal{A} = \Sigma^{Proc}$, $\mathcal{U}$ is the underlying domain of $\mathcal{H}$, and there is a (partial) mapping $\eta : \mathcal{Z}(\Sigma^{Proc}) \to \mathcal{G}$ such that

- for all $[z] \in \mathcal{Z}(\Sigma^{Proc})$:
  if $\{(p, \bar{x}) \mid p \in \Sigma, \bar{x} \in \mathcal{U}^\omega, \mathcal{H} \models p([z], \bar{x})\} \neq \emptyset$ then $[z] \in \mathbf{dom}(\eta)$.
- for all $[z] \in \mathbf{dom}(\eta)$, $p \in \Sigma$, $\bar{x} \in \mathcal{U}^\omega$: $\eta([z]) \models p(\bar{x}) \;\Leftrightarrow\; \mathcal{H} \models p([z], \bar{x})$.
- for all $[z] \in \mathbf{dom}(\eta)$: $\mathcal{H} \models \neg EOT([z]) \;\Leftrightarrow\; \mathcal{R}(\eta([z]), \eta([z+1]))$.
- for all $[z] \in \mathbf{dom}(\eta)$: $\mathcal{H} \models EOT([z]) \;\Leftrightarrow\; \mathcal{R}(\eta([z]), \eta([z]))$.
- for all $[z] \in \mathbf{dom}(\eta)$, $\pi \in \Sigma^{Proc}$, $\bar{x} \in \mathcal{U}^\omega$:
  $\mathcal{H} \models \pi([z], \bar{x}) \;\Leftrightarrow\; \mathcal{Q}(\eta([z]), \eta([z.\pi(\bar{x}).0]))$.
- for all $[z] \in \mathbf{dom}(\eta)$, $\pi \in \Sigma^{Proc}$, $n \in \mathbb{N}_0$, $\bar{x} \in \mathcal{U}^\omega$:
  $(\mathcal{H} \models \pi([z], \bar{x}) \wedge \mathcal{H} \models EOT([z.\pi(\bar{x}).n])) \;\Leftrightarrow\; \mathcal{S}(\eta([z.\pi(\bar{x}).n]), \eta([z]))$.

□

The following theorem states that the model obtained from the logic programming semantics is equivalent to the Kripke structure representing the model-theoretic semantics:

Theorem 25. *Let P be a program, EDB a database, and EB an event base such that $\mathcal{M}(P, EDB, EB)$ is total. Then $\mathcal{M}(P, EDB, EB)$ is contained in the minimal Kripke model $\mathcal{K}$ of P via a surjective mapping η.*

Proof. Set $\eta([\varepsilon.0]) := g_0$, $\eta([F.N+1]) := g$ such that $\mathcal{R}(\eta([F.N]), g)$ (well-defined by Lemma 18), $\eta([Z.\pi(\bar{x}).0]) := g$ such that $\mathcal{Q}(\eta([Z]), \pi(\bar{x}), g)$ for those $[Z]$ to be contained in $\mathbf{dom}(\eta)$ according to Definition 24.

In particular, the unique final state $[\varepsilon.m]$ of the Herbrand model $\mathcal{M}$ is the same as the stationary state of the unique computation path beginning in g_0 in the minimal Kripke Model $\mathcal{K}$. Thus, the logic programming semantics is also adequate w.r.t. the intuitive semantics.

7 Related Work

The idea of using state terms to refer to different states in logical rules has come up several times, e.g. in *XY-Datalog* [Zan93, ZAO93], to allow a unified semantics for active and deductive rules, and in [KLS92, LL94] as a means to specify updates in a declarative way. Flat Statelog [LHL95], XY-Datalog, and the temporal query languages $Datalog_{1S}$ and *Templog* [Cho90, AM89, Bau95] are closely related, since they all extend Datalog by a linear state space. In contrast, our present approach uses a branching hierarchical state space (similar to that of $Datalog_{nS}$ [CI93] which does not deal with active rules and procedures, however). The presented Kripke semantics extends that of [LS93] which is now a special case restricted to flat sequential computations.

[Zan95] proposes a "transaction-conscious" stable model semantics to cope with the problem that occurs when *ephemeral changes* (changes whose effect is undone within the same transaction) trigger active rules. Thus, to avoid unintended behavior, only durable changes should be visible to active rules. In our approach this problem is solved in a different way by the concept of "atomically executing" procedures which encapsulate their changes until the end of transaction. Thus, only the net effect of a subtransaction may trigger rules in the calling transaction.

Transaction Logic $\mathcal{T}_\mathcal{R}$ [BK93, BK94] deals, on a high level of abstraction, with the phenomenon of state changes in logic databases and employs a powerful model theory and proof theory. Primitive updates (so-called *elementary transitions*) are not part of $\mathcal{T}_\mathcal{R}$, but a parameter which is supplied by a transition oracle. In contrast, Statelog semantics provides a complete specification of changes from primitive updates to complex transactions and has a standard logic programming and Kripke-style semantics. Both languages can be combined by "plugging in" Statelog procedures in the transition oracle of $\mathcal{T}_\mathcal{R}$.

The concept of nested transactions in Statelog is similar to that of *HiPAC* [DBB+88, DBC96]. Statelog declarations *initial, always* and *final* allow execution of rules at specific points within a transaction and thus can be used in a similar way as *coupling modes* in HiPAC. E.g. integrity maintenance may be deferred until EOT by declaring the corresponding rules *final*.

The idea to structure rule sets using procedures or modules has already been introduced in the area of logic programming. E.g. [BMPT94, BT94] develop a modular design for logic programs including union, intersection, and encapsulation. However, they do not deal with active rules and state change, so their concept does not cover sequential composition, transactions etc.

[FT96] proposes *Extended ECA* rules as a common framework for a large number of existing active database systems and prototypes. In existing systems, the semantics of programs depends on the implicitly given operational semantics. These implicit assumptions are made apparent by encoding them in user-readable EECA rules. *Heraclitus[Alg,C]* [GHJ+93, GHJ96], is an extension of C which incorporates the relational algebra and elevates deltas to be "first-class citizens" of the database programming language. It allows to combine deltas and to express hypothetical updates, however no logical semantics is given.

Related to our work are approaches dealing with updates in deductive databases. Often, the rule semantics depends on a certain evaluation strategy, e.g. [Abi88, AV91, SK96] (bottom-up), or [MW88, Che95] (top-down), whereas e.g. [MBM95] is – like Statelog – independent of a certain strategy. However, these works do not cover the ECA-rule paradigm of active databases or the

concept of nested transactions. Although Statelog allows a very intuitive "bottom-up reading" of rules (cf. Example 1), evaluation may also be done top-down due to the presence of explicit state terms $[S]$ and $[S+1]$. This is in contrast to approaches like [Abi88, AV91] or [MW88, Che95], which refer to different states only *implicitly*. Thus their semantics is more tied to *either* bottom-up *or* top-down evaluation, respectively.

8 Conclusion

In recent work, the benefits of an integration of active and deductive rules have become apparent [Zan95, MZ95, LHL95]. First of all, a logical framework unambiguously specifies the semantics of rules – a necessary precondition to verify and reason about the behavior of rules. For example, the semantics of transactional events like abort and commit is completely specified in our logical framework. Moreover, properties like termination or expressive power can be investigated, as in [LHL95], *independent* of a given implementation. This complements work on termination and confluence of active rules which focuses more on specific systems like e.g. [AWH92, AWH95, BCP95, KU96].

In this paper, we have presented *Statelog*, based on a concept which integrates transaction-oriented programming of complex (trans)actions with logical foundations of deductive rules in a seamless way. This framework is an extension of flat Statelog [LL94, LHL95], and uses procedures as a means to structure rules and to encapsulate their behavior. Statelog programs have a declarative and deterministic semantics which is given (i) by a compilation into a standard logic programming semantics, which yields a (naive) implementation of the language, and (ii) by a Kripke-style semantics which describes a conceptual and implementation-independent model of active rule behavior. Procedures execute isolated and in an all-or-nothing style. The underlying nested transaction model facilitates parallel execution of concurrent transactions and allows to specify complex transactions in a natural way using subtransactions. We plan to extend the prototypical implementation of flat Statelog [Ham95] to the full language including procedures.

References

[Abi88] S. Abiteboul. Updates, a new frontier. In *ICDT*, Springer LNCS 326, pp. 1–18, 1988.

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Comp.*, 8(3), 1989.

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *JCSS*, 43, 1991.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *SIGMOD*, 1992.

[AWH95] A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *TODS*, 20(1):3–41, 1995.

[Bau95] M. Baudinet. On the expressiveness of temporal logic programming. *Information and Computation*, 117(2), 1995.

[BCP95] E. Baralis, S. Ceri, and S. Paraboschi. Run-time detection of non-terminating active rule systems. In Ling et al. [LMV95].

[BK93] A. J. Bonner and M. Kifer. Transaction logic programming. In D. S. Warren, editor, *ICLP*. MIT Press, 1993.

[BK94] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Comp. Sci.*, 133, 1994.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM TOPLAS*, 16(4):1361–1398, July 1994.

[BT94] A. Brogi and F. Turini. Semantics of meta-logic in an algebra of programs. In *LICS*, pp. 262–270, Paris, France, July 1994.

[Che95] W. Chen. Programming with logical queries, bulk updates and hypothetical reasoning. In B. Thalheim, ed., *Proc. of the Workshop Semantics in Databases*, Prague, 1995. TU Cottbus.

[Cho90] J. Chomicki. Polynomial time query processing in temporal deductive databases. *PODS*, 1990.

[CI93] J. Chomicki and T. Imieliński. Finite representation of infinite query answers. *TODS* 18(2), 1993.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining Active Databases and Timing Constraints. In *SIGMOD*, 1988.

[DBC96] U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC Project. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann, 1996.

[DGG95] K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of adbms features. In Sellis [Sel95].

[DHW95] U. Dayal, E. Hanson, and J. Widom. Active database systems. In W. Kim, ed., *Modern Database Systems: The Object Model, Interoperability, and Beyond*, Ch. 21. ACM Press, 1995.

[FT96] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *TODS*, 1996. to appear.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs *et al.* On implementing a language for specifying active database execution models. In *VLDB*, 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *TODS*, 1996. To appear.

[Ham95] U. Hamann. Ein System zur Beschreibung und Ausführung von Änderungsoperationen in einer zustandsorientierten Erweiterung von Datalog. Master's thesis, Institut für Informatik, Universität Freiburg, 1995.

[KLS92] M. Kramer, G. Lausen, and G. Saake. Updates in a rule-based language for objects. *VLDB*, 1992.

[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 ICDE*, 1996.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proc. 7th Intl. Conf. on Management of Data (COMAD)*, Pune, 1995. Tata McGraw-Hill. ftp://ftp.informatik.uni-freiburg.de/documents/reports/report78/report78.ps.gz.

[LL94] G. Lausen and B. Ludäscher. Updates by reasoning about states. In *2nd Intl. East-West Database Workshop*, Workshops in Computing, Klagenfurt, Austria, 1994. Springer.

[LMV95] T. W. Ling, A. O. Mendelzon, and L. Vieille, editors. *DOOD*, Springer LNCS 1013, 1995.

[LS93] G. Lausen and G. Saake. A possible world semantics for updates by versioning. In *Proc. of 4th Workshop on Modelling Database Dynamics*, Volkse, 1993. Springer.

[MBM95] D. Montesi, E. Bertino, and M. Martelli. Transactions and updates in deductive databases. Technical Report 2, Dipartimento di Scienze dell'Informazione, Università di Milano, 1995.

[MW88] S. Manchanda and D. S. Warren. A logic-based language for database updates. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 363–394. 1988.

[MZ95] I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In *4th DOOD*, LNCS 1013, 1995.

[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].

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 191 – 216. Morgan Kaufmann, 1988.

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

[SK96] E. Simon and J. Kiernan. The a-rdl system. In Widom and Ceri [WC96], Chapter 5.

[VG89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *PODS*, 1989.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620 – 650, July 1991.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[Wid94] J. Widom. Active databases. In M. Stonebraker, ed. *Readings in Database Systems, 2nd edition.* Morgan Kaufmann, 1994. Introduction to Chapter 4.

[Zan93] C. Zaniolo. A unified semantics for active and deductive databases. *Proc. of the 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, 1993. Springer.

[Zan95] C. Zaniolo. Active database rules with transaction conscious stable model semantics. In Ling et al. [LMV95].

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}++$ Approach. In S. Ceri, K. Tanaka, and S. Tsur, eds., *DOOD*, Springer LNCS 760, 1993.

[ZS94] C. Zaniolo and R. Sadri. A simple model for active rules and their behavior in deductive databases. *Proc. 2nd ICLP Workshop on Deductive Databases and Logic Programming*, Santa Margherita Ligure, Italy, 1994.

A Built-in "modify"-semantics

A first ad-hoc version transforms a modification into a deletion of the old tuple and an insertion of the new tuple:

$$ins{:}R(X') \leftarrow R(X), mod{:}R(X, X')$$
$$del{:}R(X) \leftarrow R(X), mod{:}R(X, X')$$

A drawback of this concept even for a flat state space is that a modification where the new tuple is equal to the old tuple causes an inconsistency. This can be avoided by "turning off" the rules in such cases:

$$ins{:}R(X') \leftarrow R(X), mod{:}R(X, X'), \neg R(X')$$
$$del{:}R(X) \leftarrow R(X), mod{:}R(X, X'), X \neq X'$$

However, the drawback remains that in the presented model the further decomposition in *insd*- and *deld*-protocol relations results in a different treatment of modifications which are derived directly by rules and those which are derived indirectly via subtransactions.

A better way to handle modifications is to treat them similar to insertions and deletions as update *primitives*. First, the signature from Definition 2 has to be extended with *mod* and *modd*:

$$\Sigma_F^{Req} := \{ins{:}R, del{:}R, mod{:}R \mid R \in \Sigma_F^{EDB}\}\ ,\ \Sigma_F^{Prot} := \{insd{:}R, deld{:}R, modd{:}R \mid R \in \Sigma_F^{EDB}\}\ .$$

Next, frame and procedure return rules have to be adjusted. Since the handling of requests is application-dependent, there are several possibilities. The following combinations of requests have to be handled yielding alternative possibilities of successor states:

1.: maximal, set-oriented, *mod* can override existing tuples.
2.: updates must be executable independently. Every source of ambiguity causes an inconsistency.
3.: consistent, *mod* has an effect only if the source tuple is true, *mod* can override.

	(1)	(2)	(3)
$R(x), R(y), ins{:}R(x), mod{:}R(x,y)$	$R(x), R(y)$	*abort*	*abort*
$R(x), \neg R(y), ins{:}R(x), mod{:}R(x,y)$	$R(x), R(y)$	*abort*	*abort*
$\neg R(x), R(y), ins{:}R(x), mod{:}R(x,y)$	$R(x), R(y)$	*abort*	$R(x), R(y)$
$\neg R(x), \neg R(y), ins{:}R(x), mod{:}R(x,y)$	$R(x)$	*abort*	$R(x)$
$R(x), R(y), del{:}R(x), mod{:}R(x,y)$	$R(y)$	*abort*	$R(y)$
$R(x), \neg R(y), del{:}R(x), mod{:}R(x,y)$	$R(y)$	*abort*	$R(y)$
$\neg R(x), R(y), del{:}R(x), mod{:}R(x,y)$	$R(y)$	$R(y)$	$R(y)$
$\neg R(x), \neg R(y), del{:}R(x), mod{:}R(x,y)$			
$R(x), R(y), R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(y), R(z)$	*abort*	*abort*
$R(x), R(y), \neg R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(y), R(z)$	*abort*	*abort*
$R(x), \neg R(y), R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(y), R(z)$	$R(y), R(z)$	$R(y), R(z)$
$R(x), \neg R(y), \neg R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(y)$	$R(y)$	$R(y)$
$\neg R(x), R(y), R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(z)$	*abort*	$R(z)$
$\neg R(x), R(y), \neg R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(z)$	$R(z)$	$R(z)$
$\neg R(x), \neg R(y), R(z), mod{:}R(x,y), mod{:}R(y,z)$	$R(z)$	$R(z)$	$R(z)$
$\neg R(x), \neg R(y), \neg R(z), mod{:}R(x,y), mod{:}R(y,z)$			
$R(x), R(y), mod{:}R(x,y), del{:}R(y)$	$R(y)$	*abort*	*abort*
$R(x), \neg R(y), mod{:}R(x,y), del{:}R(y)$	$R(y)$	*abort*	*abort*
$\neg R(x), R(y), mod{:}R(x,y), del{:}R(y)$			
$\neg R(x), \neg R(y), mod{:}R(x,y), del{:}R(y)$			
$R(x), R(y), mod{:}R(x,y), ins{:}R(y)$	$R(y)$	*abort*	$R(y)$
$R(x), \neg R(y), mod{:}R(x,y), ins{:}R(y)$	$R(y)$	*abort*	$R(y)$
$\neg R(x), R(y), mod{:}R(x,y), ins{:}R(y)$	$R(y)$	$R(y)$	$R(y)$
$\neg R(x), \neg R(y), mod{:}R(x,y), ins{:}R(y)$	$R(y)$	$R(y)$	$R(y)$

There are some more possibilities, e.g., executing the three types of updates in any sequential order, with or without overriding. We think, that one important restriction is to keep updates local, ie no inserted tuple can be modified in the same step, no modified tuple can be further modified, and no result of any modification can be deleted.

It is even conceivable to choose different strategies for different EDB relations, dependent on the intended semantics of the individual relations.

Since in Section 3.2 we proposed a handling corresponding to (3), the rules are given in this style: For all EDB relations R visible in F, the rules marked with (B) from Section 3.2 have to be replaced by

$$
\begin{array}{l}
[Z+1]\ R(X) \leftarrow [Z]\ R(X), \neg[Z]\ \mathit{del{:}R}(X), \forall X' : \neg[Z]\ \mathit{mod{:}R}(X, X'), \neg \mathit{EOT}. \\
[Z+1]\ R(X) \leftarrow [Z]\ \mathit{ins{:}R}(X), \neg \mathit{EOT}. \\
[Z+1]\ R(X) \leftarrow [Z]\ R(X'), \mathit{mod{:}R}(X', X), \neg \mathit{EOT}. \\
\\
[Z+1]\ \mathit{insd{:}R}(X) \leftarrow [Z]\ \mathit{ins{:}R}(X), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{insd{:}R}(X) \leftarrow [Z]\ \mathit{insd{:}R}(X), \neg[Z]\ \mathit{del{:}R}(X), \forall Y : \neg[Z]\ \mathit{mod{:}R}(X, Y), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{insd{:}R}(Y) \leftarrow [Z]\ \mathit{insd{:}R}(X), [Z]\ \mathit{mod{:}R}(X, Y), \neg \mathit{EOT}. \\
\\
[Z+1]\ \mathit{deld{:}R}(X) \leftarrow [Z]\ \mathit{del{:}R}(X), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{deld{:}R}(X) \leftarrow [Z]\ \mathit{deld{:}R}(X), \neg[Z]\ \mathit{ins{:}R}(X), \forall Y : \neg[Z]\ (\mathit{mod{:}R}(Y, X) \wedge R(Y)), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{deld{:}R}(X) \leftarrow [Z]\ \mathit{modd{:}R}(X, Y), [Z]\ \mathit{del{:}R}(Y), \neg \mathit{EOT}. \\
\\
[Z+1]\ \mathit{modd{:}R}(X, Y) \leftarrow [Z]\ R(X), \mathit{mod{:}R}(X, Y), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{modd{:}R}(X, Y) \leftarrow [Z]\ \mathit{modd{:}R}(X, Y), \neg[Z]\ \mathit{del{:}R}(Y), \forall Y' : \neg[Z]\ \mathit{mod{:}R}(Y, Y'), \neg \mathit{EOT}. \\
[Z+1]\ \mathit{modd{:}R}(X, Y') \leftarrow [Z]\ \mathit{modd{:}R}(X, Y), [Z]\ \mathit{mod{:}R}(Y, Y'), \neg \mathit{EOT}.
\end{array}
\tag{B}
$$

The rules marked with (C) have to be replaced by

$$
\begin{array}{l}
[Z]\ \mathit{running} \leftarrow [Z]\ \mathit{ins{:}R}(X), \neg R(X). \\
[Z]\ \mathit{running} \leftarrow [Z]\ \mathit{del{:}R}(X), R(X). \\
[Z]\ \mathit{running} \leftarrow [Z]\ \mathit{mod{:}R}(X, Y), R(X), X \neq Y.
\end{array}
\tag{C}
$$

The rule for checking update-consistency, which is also marked with (C) in Section 3.2 has to be replaced by a set of rules handling of inconsistent update requests in presence of modify requests.

$$
\begin{array}{l}
[Z]\ \mathit{abort} \leftarrow [Z]\ \mathit{ins{:}R}(\bar{X}), \mathit{del{:}R}(\bar{X}). \\
[Z]\ \mathit{abort} \leftarrow [Z]\ \mathit{ins{:}R}(\bar{X}), \mathit{del{:}R}(\bar{X}). \\
[Z]\ \mathit{abort} \leftarrow [Z]\ R(\bar{X}), \mathit{ins{:}R}(\bar{X}), \mathit{mod{:}R}(\bar{X}, \bar{Y}). \\
[Z]\ \mathit{abort} \leftarrow [Z]\ R(\bar{X}), \mathit{del{:}R}(\bar{Y}), \mathit{mod{:}R}(\bar{X}, \bar{Y}).
\end{array}
\tag{E}
$$

The procedure return rules which are marked with (E) have to be adjusted to

$$
\begin{array}{l}
\text{for all } R \in \Sigma_\pi^{Exp} \cap \Sigma_F^{EDB} : \\
[Z]\ \mathit{ins{:}R}(X) \leftarrow [Z]\ \pi(X), [Z.\pi(X).N]\ \mathit{insd{:}R}(X), \mathit{EOT}, \neg \mathit{abort}. \\
[Z]\ \mathit{del{:}R}(X) \leftarrow [Z]\ \pi(X), [Z.\pi(X).N]\ \mathit{deld{:}R}(X), \mathit{EOT}, \neg \mathit{abort}. \\
[Z]\ \mathit{mod{:}R}(X, Y) \leftarrow [Z]\ \pi(X), [Z.\pi(X).N]\ \mathit{modd{:}R}(X, Y), \mathit{EOT}, \neg \mathit{abort}.
\end{array}
\tag{E}
$$

Since the new rules fit into the scheme of procedure and frame rules, the compilation scheme from Section 5 can be applied, providing a logical semantics also for this case.