12

A Customizable Model for the Semantics of Active Databases

Sara Comai& Piero Fraternali
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza L. Da Vinci, 32 - I-20133 Milano, Italy
Giuseppe Psaila
Politecnico di Torino, Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24 - I-10129 Torino, Italy
Letizia Tanca
Università di Bari, Dipartimento di Informatica
Via Re David 5 - I-70125 Bari, Italy
e-mail: {fraternali,psaila,tanca}@elet.polimi.it

Abstract

Active databases couple database technology with rule-based programming and offer a natural way to develop applications in which the shared properties of data are centralised instead of being scattered among application programs. Unfortunately, designing, implementing and evolving applications based on the active database paradigm is still a very difficult task, mostly due to the poor understanding of the semantic issues underlying active rules. This paper provides a categorization of the dimensions of active database semantics and provides a uniform model to describe rules with different behaviours, capable of expressing the features of most commercial system and research prototypes.

Keywords

Active Databases, Rules, Semantic Model.

1 INTRODUCTION

An active database is a database system extended with active rules, i.e., stored procedures automatically invoked by the DBMS when specific events occur.

The best known form of active rule is called Event-Condition-Action (ECA); ECA rules have a threefold pattern: the *event part* specifies the event(s) that must occur for the rule to be awakened; the *condition part* express a precondition for executing the rule; the *action part* actually contains the code to execute.

When its events occur, a rule is said to be *triggered*; when the condition has been evaluated the rule is said to be *considered*; if the consideration succeeds the rule is *executed*.

R. Meersman et al. (eds.), *Database Applications Semantics*

Active rules are useful for a variety of tasks, including security, integrity maintenance (Ceri et al.(1994b), Ceri and Widom (1990)), view materialization (Ceri and Widom (1991)), reactive control, heterogeneous databases integration and so on. However, rule programming is normally considered a low-level and difficult task, mostly due to the many subtleties of rule execution semantics and the intricate way in which rules interact.

Differently from the field of deductive databases, where clean, declarative semantic models have been defined for deductive rules with different expressive power, active rules still lack a well-defined semantics (not to say a uniform syntax!). In fact, several active databases systems and prototypes have been designed, and partially or completely implemented. Unfortunately, they often show different behaviours for rules with a similar form. This depends on the fact that they have been designed in a totally independent way, without the support of a common theory dictating the semantics of ECA rules.

This paper is a first step in the direction of understanding the dimensions that contribute to the behaviour of an active database and gives a simple model to express in a uniform way the execution semantics of rules from different commercial and prototype systems (Ceri and Manthey (1993), Gehani and Jagadish (1991), SQL3, Stonebraker et al. (1990), Widom et al. (1991), Widom and Finkelstein (1990)).

The main idea is to encode existing rules into an intermediate format, called EECA (Extended Event-Condition-Action) syntax, that makes explicit the options choosen for the dimensions that characterize the behavior of active databases : the *granularity* of updates and of rule applications, the modes of *coupling*, first between Event and Condition, and then between Condition and Action, the *atomicity of action execution* in rules, the *relationship of rule processing to transaction processing*, the *conflict resolution* mechanism, the time and scope of *event consumption*, the handling of *event net effect*, and the inspection of *transaction history*. Furthermore, EECA rules are translated into a common low-level language, called *core language*, to express the operational semantics of rules written in this language. As a consequence, the encoding of rules into the EECA and Core formats requires a deep understanding of the specific active database system. Then, a proper definition of the semantics of database updates is given, followed by the Core rule execution semantics, i.e. a *unique* execution model which exploits the information on rule semantics available in the Core format. The final semantics of the active database is then obtained as the fixpoint of a simple transformation based on this execution model. The logical style of the rule Core format allows this part to parallel in many ways the well-known work on the semantics of deductive databases, although the similarity is superficial, since active databases show intrinsical procedural features that make it very difficult to describe them in a purely declarative style.

Though there is a conspicuous body of research on active databases about prototype and system implementation (Ceri et al. (1995), Gehani and Jagadish (1991), Gehani et al. (1992), Hanson (1992), McCarthy and Dayal (1989), Widom et al. (1991)), potential or actual applications (Baralis et Al. (1994), Ceri et al. (1994), Ceri et al. (1994b), Ceri and Widom (1990), Ceri and Widom (1991)), and different paradigm integration, less work has appeared on the topic of modeling the behaviour of reactive systems by means of a formal semantics.

In Zaniolo (1993) the notion of stratification of logic programs is extended to develop a semantic framework for updates (called X-Y stratification), in order that the semantics of a set of active rules is represented by the perfect model of the (logic) program that

encodes the active rules. Another approach (Bonner adn Kifer (1993)) gives a powerful extension of classical predicate logic, called *Transaction Logic* ($\mathcal{T}_{\mathcal{R}}$), that establishes a declarative semantics for a wide range of database updates; then, active rules and their triggering events are expressed as $\mathcal{T}_{\mathcal{R}}$ statements and the semantics of the reactive system admits both a model- and a proof-theoretic characterization. A denotational semantics is presented in (Widom (1992)) to describe the features of a particular active database system (Starburst); reactive processing semantics is represented by a meaning function that computes the least-fixpoint of a transformation describing the atomic step of rule computation (analogous to the Elementary Production Step introduced in Section 5.1). Finally, we mention the work Zhou and Hsu (1990), to which our approach of expressing the semantics of rule processing as a set of relations is inspired.

Each of these works is interesting as a preliminary attempt to formalize the behaviour of active databases; however, the spectrum of considered features is still limited when compared with the wide variety offered by real systems, in particular the procedural aspects like conflict resolution, triggering modes, transaction history inspection and event net effect criteria; besides, all these works show a somewhat unnatural way of treating the notion of *event*, which we think should be treated as a first-class concept.

The paper is structured as follows: Section 2 illustrates the dimensions of active rule semantics, Section 3 introduces the main components of the semantic model, Section 4 describes the semantics of updates and transaction in an active database, Section 5 presents the Core rule execution semantics, finally Section 6 draws the conclusions.

2 DIMENSIONS IN ACTIVE DATABASE SYSTEMS

In this section we review the dimensions that characterize the behaviour of an active database and illustrate them through several examples from known active database systems.

2.1 Granularity

Updates in a database system are traditionally classified into *instance-oriented* (also called *tuple-oriented*) and *set-oriented*, depending on whether the target of the update is a single database entity or a set thereof.

This distinction naturally extends to rules, which react to events caused by database updates.

Hence, it is possible to speak of tuple or set-oriented rules, depending on whether rules react separately for each distinct updated item (as, for example, in Postgres (Stonebraker et al. (1990)), or are triggered by the collective modification and react only once (as, for example, in Starburst (Widom and Finkelstein (1990)), This dimension is not confined to relational systems: in the object-oriented case, rules may react either to single object updates (as in Ode, Gehani and Jagadish (1991)), or to events affecting sets of objects of the same class (this semantics is taken in Chimera, Ceri et al. (1995)). In the SQL3 standard and in most commercial relational active databases (Informix, Ingres, Oracle), both types of rules can be defined.

Note that a set-oriented update language does not necessarily imply set-oriented rules:

in Postgres the set-oriented update language POSTQUEL is used, but rules react to single tuple-level updates.

Example I: The rule below computes the average salary of all departments, triggered by the insertion of new employees. If a command is issued to give a 10% salary raise to several (or all) employees, the rule may either be activated once and compute the final value of the average salary of the departments (e.g. in Starburst and Chimera, or in commercial relational systems providing the FOR EACH STATEMENT option), or be activated as many times as the number of updated employees and possibly compute intermediate average values after each object update (e.g. in Ode, in Postgres, and in commercial relational systems providing the FOR EACH ROW option).

In the SQL3 standard, triggers are activated by set-oriented SQL update commands, but may have either tuple- or set-oriented granularity. Thus, the active rule has the form:

```
CREATE TRIGGER Compute_Average
AFTER INSERT ON Employee
WHEN True
update Department D
  set avg_salary =
    (select avg(sal) from Employee E where E.deptno=D.deptno)
FOR EACH ROW (STATEMENT)
```

2.2 Coupling Modes

The notion of coupling mode of ECA rules was first introduced in McCarthy and Dayal (1989), to describe two different things: how rule activation, condition evaluation and action execution are synchronized; and how condition evaluation and action execution are related to the transaction where the triggering event is generated from the standpoint of concurrent execution. In this paper, as in Kappel et al. (1994), the rule coupling mode has only the former meaning; transaction semantics is considered as a separate, independent dimension, treated in Section 2.4.

E-C Coupling Mode

The Event-Condition (E-C) coupling mode describes the time relationship between triggering event(s) *produced by the user transaction* and condition evaluation.

Two different choices are available:

- *Immediate Coupling*: the condition is evaluated as soon as the update unit that produced the triggering event(s) terminates. This option encompasses a wide variety of behaviours, depending on what a system considers to be a "non-interruptable update unit". Implemented proposals range from rules reacting istantaneously to instance-oriented updates, to systems where fairly complex sequences of updates are considered as non-interruptable.
- *Delayed Coupling*: the evaluation of the condition part is not performed immediately upon termination of the non-interruptable update unit that produced the triggering event, but is delayed until some other "event" takes place. This may be an arbitrary, application-dependent event (Kappel et al. (1994)), or a "special" event, e.g., the at-

tempt of the transaction to commit. This latter case is traditionally called *deferred* E-C coupling (McCarthy and Dayal (1989)). *

C-A Coupling Mode

The Condition-Action (C-A) coupling mode of a rule describes the time relationship between condition evaluation and the instant when the action is executed. The same options (*immediate* and *delayed*) as for E-C coupling apply, though immediate C-A coupling is almost generally chosen. When actions are executed at the end of the transaction in which the condition is evaluated, C-A coupling is traditionally called *deferred.*

Also the coupling between event occurrence and action execution (E-A coupling) can be defined (Gehani and Jagadish (1991)), as the weakest between the E-C and C-A coupling. Thus, Postgres rules, which have immediate E-C and C-A coupling, also have immediate E-A coupling, whereas Starburst rules are deferred, due to their deferred E-C coupling.

2.3 Atomicity of Rule Execution

When a triggering event is generated from within the action part of an executing rule, rule processing may proceed in two ways: either the rule that produced the triggering event is suspended, to process other possibly triggered rules, or the triggering event is "frozen" until the termination of action execution. In the former case we say that rule actions are interruptable, in the latter that they execute atomically.

Rule processing with interruptable rules somewhat resembles recursive procedure invocation: the executing process that produced the event is suspended, and resumed only when all "recursively" triggered rules have been processed.

Note that atomic rule execution may override the standard coupling between events and condition. In general, existing systems allow the coupling mode induced by rule atomicity to be looser than the transaction E-C coupling. For example, in Ode ordinary rules react immediately to events produced by transactions, while their activation is postponed when events are produced during the action part of *constraints* (a variety of Ode's active rules).

We now show an example of the different semantics yielded by different choices with respect to rule atomicity.

Example II: Rule *Propagate_Joe's_salary* equals Sam's salary to Joe's and Bob's salary to Sam's.

Rule *Raise_Sam's_salary* gives Sam a 10% increase after each update to Sam's salary, until it reaches $5000. If rules are interruptable (like in Postgres), after the execution of both rules, Bob is paid as much as Sam. Otherwise, he gets the same salary as Joe: this occurs in most systems. The syntax used in this example is that of Postgres.

```
define rule Propagate_Joe's_salary
on replace to EMP.salary where
    EMP.name = "Joe"
then do
```

*Deferred EC coupling could be considered a special case of immediate coupling, where a whole transaction is non-interruptable by rule processing; however, this would exclude immediate rules, which in most systems coexist with deferred rules.

```
    replace EMP (salary = NEW.salary) where
      EMP.name = "Sam"
    replace EMP (salary = E.salary) where
      E.name = "Sam" and
      EMP.name = "Bob"

define rule Raise_Sam's_salary
on replace to EMP.salary where
    EMP.name = "Sam" and EMP.salary<5000
then do
    replace EMP (salary = 1.1*NEW.salary) where
      EMP.name = CURRENT.name
```

2.4 Relationship to Transaction

This dimension is relevant for those systems that support the notion of *transaction*, i.e., of a sequence of data manipulation and query operations, which satisfy the usual ACID (Gray and Reuter (1994)) requirements.

By relationship to transaction we mean the relationship between the transaction where the triggering event of a rule is generated and the (possibly different) transactions in which the condition is evaluated and the action executed.

Two situations are possible:

- *Same transaction*: the condition is evaluated and the action is executed in the same transaction where the triggering event arises. This implies that an abort caused by the rule action produces an abort of the whole transaction.
- *Different transaction(s)*: condition evaluation and/or action execution are processed in a single separate transaction or even in two distinct transactions. This implies that the order in which conditions are evaluated and actions are executed is not under the responsibility of the rule execution engine, but is part of the concurrency control system.

The latter mode is also called *decoupled* or *separate* (McCarthy and Dayal (1989)), and admits two sub-cases:

- *Dependent sub-transaction(s)*: the separate transaction is a sub-transaction (also called *nested transaction*) of the one in which the triggering event occurs. Also in this case, failure of the sub-transaction entails failure of the main transaction (this semantics is found, e.g., in HiPac).
- *Independent transaction(s)*: the separate transaction is not a sub-transaction of the one in which the triggering event occurs. Failure of the sub-transaction does not imply failure of the main transaction (this semantics is found, e.g., in Ode.

This paper only deals with the same-transaction case, since for the moment our formal model of updates and transactions does not capture the concurrent execution of multiple transactions. We plan to address this issue in future work.

2.5 Conflict Resolution

In most systems, more than one rule can be in a triggered state simultaneously, either because one event can trigger multiple rules or because rules are not considered immediately after the occurrence of their triggering event. Hence, the need arises for a policy to select the rule(s) to be considered and possibly executed, among the triggered ones, i.e. a *conflict resolution* policy. Two alternatives are available:

- *Serial execution*: a single rule is chosen.
- *Parallel execution*: all triggered rules are considered and/or executed in parallel, e.g., by scheduling a separate transaction to evaluate the condition and/or execute the action.

Serial execution implies a criterion to select among triggered rules; this may be either deterministic (e.g., a total priority order among rules), or non-deterministic (e.g., a partial priority order or no ordering at all).

Since the concurrency issues related to the parallel execution of multiple rules are outside the scope of this paper, in the following we restrict our semantic model to the case of serial processing.

2.6 Event Consumption

When a triggered rule is processed (its condition is evaluated and/or its action is executed), the events responsible of the triggering may undergo different treatments, which we call *event consumption modes.*

Two issues are relevant with respect to event consumption: whether processed events retain their capability of triggering rules and, in the case they do not (i.e., are *consumed*), when consumption takes place.

Scope of Event Consumption

After the processing of a triggered rule, there are three possible ways to deal with its triggering events:

- *No consumption*: the triggering events are unaffected by condition evaluation and/or action execution; in particular, they retain the possibility of triggering rules. This situation arises in the so-called condition based rule systems, where rules (implicitly) activated by an event remain triggered until their condition becomes false. This is the case, for example, of the OPS5 production rule system.
- *Local consumption*: the triggering events no longer activate the processed rule, but still trigger other rules not yet considered. We say that such events are "consumed" locally to the processed rule. This option is the most frequently adopted by the existing active database systems.
- *Global consumption*: the triggering events no longer trigger the processed rule, nor all (or a subset of) the other rules not yet processed. We say that such events are "consumed" for the processed rule and for those rules no longer triggered by them. This option is used in the Postgres system, where a rule may be defined as an exception to another one triggered by the same event. When the common triggering event occurs, both the exception and the general rule are activated; but the exception rule is consid-

ered first and its processing "detriggers" the general rule. In this case, we say that the exception rules consumes the triggering event of the general rule.

Time of Event Consumption

The consumption of triggering events can take place either after a rule has been considered (irrespective of the result of condition evaluation) or, more restrictively, only when a rule is actually executed. The latter option implies that an unsuccessful evaluation of the rule condition is not considered as a "processing" of the triggering events, which remain pending until the rule is eventually executed. This choice is taken in the first specification of Starburst's semantics, described in Widom and Finkelstein (1990), whereas in (Widom et al. (1991), Widom (1992)), Starburst rules consume events at consideration time†. In most other systems, consumption takes place after condition evaluation.

Example III: A Starburst rule is defined that reacts to salary updates, whose condition is satisfied only if Bob's salary is modified. In Starburst, the event part of a rule is evaluated with respect to the triggering events occurred since the last execution of the rule, or, if the rule has never been executed, since the beginning of the transaction. With the semantics of Widom and Finkelstein (1990), event consumption is local and at the time of rule execution, thus it may happen that rule *Updated_Bob* is activated by an update to somebody else's salary and remains triggered for the rest of the transaction, even if Bob's salary is not altered. With the semantics of Widom (1992) (from now on called Starburst II), where event consumption is defined to take place at condition evaluation, the rule is excluded from the set of triggered rules after the first unsuccessful consideration.

```
create rule Updated_Bob
when updated emp.salary
if exists (select * from new updated emp.salary
      where name = "Bob")
then print("Bob's salary has been updated")
```

2.7 DB Past States and Event Net Effect

In most rule systems it is possible for a rule to explicitly query information concerning transaction execution history. Two types of information can be queried: past states of the database and past events.

Past Data

Access to past states of data affected by events is normally achieved by extending the syntax of the query language used in the rules with special keywords that alter the normal evaluation of database queries, so that a state of the database prior to the current one is queried. In principle, all the intermediate states since the origin of the transaction and up to the current one may be the target of such inspection. However, in practice, only particular states are addressed by existing systems, namely the state before the execution of the transaction (*pre-transaction state*), the state before the last consideration of the

†Since other systems illustrated here also show this behaviour, we consider the semantics of Widom and Finkelstein (1990), to demonstrate the whole range of possibilities.

rule (*pre-consideration state*) and the state before the occurrence of the event(s) that triggered the rule and have not been processed yet (*pre-event state*).

For instance, in Chimera, the interpretation of the predicate *old* is based on the so-called *rule consuming mode*: when it is *event_consuming*, it returns the value at the time of the last consideration, or the pre-transaction value, if the rule is evaluated for the first time; when it is *event_preserving*, it refers to the pre-transaction value.

Items Affected by Past Events

Another way to influence the behaviour of a rule based on the history of the transaction is to permit the explicit query of items affected by occurred events, feature that is extremely important to define rules whose behaviour is based on the variation of data, needed, e.g., for integrity checking and incremental view maintenance.

This capability is achieved by extending the query language used in the rules with a set of predicates interpreted over items affected by occurred events, and by providing ad-hoc data structures containing information on events. A query may retrieve the items affected by all events since the start of the transaction, or be restricted to those events that have contributed to the most recent triggering of the rule. Another distinction is whether a rule is permitted to query items affected by all events, including those that do not trigger it, or only by those that trigger it.

For example, in Chimera, the *occurred* predicate has a twofold interpretation: if the rule is declared as *event-consuming* then only those events that have determined the most recent triggering of the rule are considered; if the rule is declared as *event-preserving*, all events since the transaction start are considered. Event queries are restricted to the events that trigger the rule, and the *occurred* predicate provides those objects affected by the queried events.

Event Net Effect

There are two possible ways to think about sequences of events affecting the same database information: either one may consider only the bare occurrence of an event, irrespective of what has happened afterwards (we say that we consider *occurred* events); or the occurrence of some events may be regarded as invalidating previous ones (e.g., a deletion after a modification), so we consider the *net effect* of a sequence of events, or simply *net events*. Criteria for computing net effect of events vary from one system to another. In Starburst they are the following, valid in the course of a transaction : if a tuple $X^{\ddagger}$ is created, and possibly updated, and subsequently deleted, the net effect is null; if a tuple is created and then updated several times, then the net effect is the creation of the final version of the tuple; if a tuple is updated and then deleted, then the net effect is simply the deletion of the tuple.

In the Ode system, where triggers are activated by events corresponding to generic method invocations on objects: if several events are detected that affect the same object, they are treated as a single event and the corresponding trigger is considered only once.

There are two possible *uses* of the net effect criteria: *Net effect for triggering*, when net events are used to determine whether a rule is triggered or not, as in Starburst, instead

‡The same criteria may be applied to the object oriented and relational case, in the latter case by providing relational tuples with a *tuple identifier* field, as actually done in Widom et al. (1991). The criteria for net effect in Chimera are basically the same as those of Starburst.

of occurred events (notice that the possibility arises that a once triggered rule is not triggered any more after the occurrence of an event, as in the case of a rule, triggered by the creation of an entity, that may be detriggered by the subsequent deletion of the same entity); *Net effect for event predicates*, when querying the events of a transaction, it is possible to target the query either to occurred or to net events (for example, Chimera supplies two distinct predicates for querying occurred and net events, whereas Starburst only considers net events).

3 MODELING ACTIVE DATABASES SEMANTICS

In this section we introduce two formalisms in order to describe the different semantics of active databases. The first one, called Extended Event-Condition-Action syntax (EECA), gives us the capability to put in evidence the options for the semantic dimensions which characterize active databases; notice that this formalism is not a new paradigm for active rules, but the word *extended* means that specific keywords are used to define semantic options. Thus, the EECA syntax is a high level language, independent of the implementation or of a particular event base schema.

Subsequently, after a formal definition of database and eventbase (Section 3.2), a *core* format is introduced (Section 3.3) : this is a logical and low level description of the semantics, independent of the implementation as well, but not of the eventbase schema. This format is the input for a general algorithm that is the same for all different rules. An EECA syntax description can be automatically translated into Core format, as shown in Comai et al. (1995), which also reports the EBNF syntax for EECA and Core rules; thus, any feature or keyword expressed in EECA is representable in Core syntax.

3.1 Extended Event-Condition-Action Rules

In this section we introduce the EECA syntax, by which the semantic dimensions present in the various systems can be formally encoded.

EECA rules are still composed of the three traditional parts, namely the event, condition and action part, plus an optional rule ordering specification and a number of ad-hoc notations expressing some of the abovementioned semantic dimensions.

EECA rules begin with a declarative section, where the following keywords are used:

- *Granularity*: a keyword **granularity** specifies the rule activation granularity and may take the values **instance-oriented** or **set-oriented** (abbreviated **I/O**, **S/O**).
- *Coupling mode*: a keyword **EC** specifies the E-C coupling mode and may assume the value **immediate** or **deferred**; the C-A coupling is assumed to be always **immediate**.
- *Atomicity*: the boolean keyword **interruptable** tells whether the rule's action is to be executed atomically (value = **False**) or can be interrupted (value = **True**).
- *Event Consumption*: a keyword **consumption-scope** specifies the event consumption scope and may assume the values **none, local, global-to{list-of-rules}**. A second keyword **consumption-time** specifies the event consumption time and may assume the values **consideration, execution**.

The **event part** is a *disjunction of event type names*. For simplicity, we consider only

atomic events (i.e., events corresponding to a single instantaneous occurrence) and make no assumption on the kind of events that can trigger a rule. If net effect criteria are present and rule triggering is done on net events, the event type name is prefixed by the keyword **net**.

The **condition part** is a *query on the database and/or the transaction history.* In principle, any query language can be assumed here. We require rule safety, i.e., the condition must be a safe formula and all variables used in the action part, which are not restricted by a query in the action part itself, must be properly restricted in the condition.

The **action part** is a *sequence of blocks*, delimited by the separator ";". Blocks act as *atomic update units*: if a rule is interruptable, rule processing can be started at the end of each block of the action part. A block in turn contains a sequence of query and/or update operations, separated by commas. Again, there is no need to assume a particular update language. When needed, we will use self-explaining database updates.

The database query language used to write rule conditions and actions is extended with predicates and keywords, to query the history of the transaction. For *Items affected by past events*, an occurrence of a literal **pending(E,X)**, where E is a set of event names, has the effect of binding the variable X to the identifiers of the database items (tuples or objects) affected by events of one of the types listed in E, occurred after the last *rule assertion point.* A rule assertion point (Widom (1992)) is either consideration, if consumption takes place after consideration, or execution, if consumption takes place after execution. E must be a subset of the event types specified in the event part of the rule. In practice, **pending(E,X)** considers only the events not yet processed by the rule. Conversely, the predicate **history(E,X)** has the effect of binding the variable X to the identifiers of the database items affected by events of one of the types listed in E, occurred since the beginning of the transaction. In this case, E can contain any event type. If the keyword **net** is prefixed to an event type in E in either predicate, then only net events of that type are considered.

After the action, an EECA rule may contain an optional priority specification, listing those rules that must always be executed before or after it. This is used to enable conflict resolution at run time.

Note that the relationship to transaction is not explicitly addressed by the EECA syntax, since in this paper we consider only a fixed value for this dimension: the *unique transaction* case.

3.2 Database and Eventbase

We distinguish two components of the passive repository of an active database: the *database*, which indeed stores the application data, and the *eventbase*, which records facts relevant to the database history. They are tightly connected by the definition of update on an active database, which is always specified in terms of its simultaneous effect on both data and events.

Definition: An *active database* $\mathcal{D}$ is a triple $\langle DB, EB, \mathcal{R} \rangle$, where DB is a passive database, EB is an eventbase and $\mathcal{R}$ is a set of active rules.

In the following, the DB is represented by a first order (f.o.) formula of a logical language that depends on the underlying data model of the passive database (e.g., the conjunction of a set of ground database atoms of Domain Relational Calculus in the case

of Relational Databases), whereas the EB is represented by a f.o. formula of the logical language defined below, specifically tailored for representing the contents of the eventbase. We also generalize the meaning of the term *state*, to stand for the pair ⟨*database state, eventbase state*⟩, unless otherwise specified.

The EB records two kinds of events: *internal* and *external.* Internal events concern database modifications: additions, deletions and updates are internal events. External events are instead events that are independent of the internal database functions. Examples are *commit* and *rollback* commands; clock events (e.g., "it's eight o'clock") may also be modeled as external events, inserted into the eventbase by the system.

Events in the EB are described by the following pieces of information:

- The *eid*, i.e., the unique identifier of occurred events.
- The *type*, i.e., the kind of events recognized by the reactive system. For instance, an event type *"create(P)"* means that a new element has been introduced in relation or class P.
- The *affected database information*, which links an event to the database items that were affected by it.
- The *timing information*, which shows the order of occurrence of events, e.g., by means of timestamps.
- The *scope*, which concerns the visibility of events to rules.
- The *net-validity* flag, a boolean flag indicating events that are to be considered as still valid, if net effect criteria are taken into account; net-validity may carry information either on rule triggering or about transaction history inspection.

This information is grouped in the relations *EVENT(eid, type, datainfo, ts, net)* and *ACTIVE(eid, scope, net)*, the former containing all events occurred during the transaction, the latter showing which events still contribute to trigger which rules.

Fig. 1 shows an instance of the eventbase, which records, in relation *EVENT*, five events. The first two events are caused by the execution of an object creation primitive affecting two objects (with OIDS *#417, #582*) of class P. The event identifiers are *e12, e13*, and both events have occurred at time *4*[§]. Relation *EVENT* also records events *e24, e25*, caused by the deletion of two objects of class C, occurred at time *8*, and event *e23* due to the deletion of object *#417* of class P, occurred at time *7*. Since object *#417* has been created and deleted in the same transaction, the net-validity flag is set to *"False"* for events *e12, e23*, denoting that their net effect is null.

Relation *ACTIVE* shows which events are *"active"* for which rules, meaning which events are still able to trigger rules. Rule R_1 sees events *e12, e23, e24, e25* as active; *e12, e23* have net-validity flag set to *"False"*, so they may not contribute to triggering R_1, if R_1 is triggered by net events. Event *e13* has already been consumed by rule R_1 and thus is no longer active for this particular rule and the corresponding tuple does not appear in relation ACTIVE. Instead, rule R_3 has consumed events *e24, e25* and may be triggered by events *e12, e13, e23*; however, if R_3 is triggered only by net events, *e12, e23* must not be considered.

[§]In this particular instance of the eventbase we are considering as simultaneous the bulk update to an entire class. In some other system, events recording updates to different objects may have different timestamps.

ACTIVE

EID	SCOPE	NET
e12	R_1	False
e23	R_1	False
e24	R_1	True
e25	R_1	True
e12	R_3	False
e13	R_3	True
e23	R_3	False

EVENT

EID	TYPE	DATAINFO	TS	NET
e12	"create(P)"	#417	4	False
e13	"create(P)"	#582	4	True
e23	"delete(P)"	#417	7	False
e24	"delete(C)"	#123	8	True
e25	"delete(C)"	#555	8	True

Figure 1 An example of eventbase instance.

We consider relations *ACTIVE* and *EVENT* as local to each transaction. It would be also possible to consider them as global, adding a further piece of information to determine the transaction that generated an event. An inclusion dependency is valid over the two relations: events recorded in table *ACTIVE* must always be present also in table *EVENT*.

The event language must permit to express event queries (EQ) and event updates (EU). It will be extensively used in the syntax of Core rules. The event query language can be any f.o. language comprising the predicates defined by the above relations. In the following, we mostly use Domain Relational Calculus.

For example, the event query

$$\exists\, EID, M, X, TS, N\ (\text{active}(EID, \text{“}R_i\text{”}, M) \wedge \text{event}(EID, \text{“create(P)”}, X, TS, N))$$

returns true if an insertion in relation or class P has occurred and is active for rule R_i, irrespective of net effect.

The event update language permits the following kinds of statements, corresponding to atomic operations on the EB:

- $\langle EID, TS\rangle$:= *+(“type”, “datainfo”)* inserts an event, described by a tuple [*eid, “type”, “datainfo”, ts, “True”*], into the eventbase relation *EVENT*; as a result of its execution, the variables EID and TS get bound to the system-generated values *eid* (the unique identifier of the event) and *ts* (the unique timestamp of the event), respectively. If the bindings for EID and TS are not to be recorded, the short notation *+(“type”, “datainfo”)* can be used. This instruction is used in the following to insert into the eventbase “special-purpose”, global events (e.g., the event *BeginAction*) that are not consumed by rules.
- $\langle EID, TS\rangle$:= *++(“type”, “datainfo”, “R”)*—shortly *++(“type”, “datainfo”, “R”)*—inserts an event into the eventbase and makes it active for rule R; in particular, it adds a tuple [*eid, “type”, “datainfo”, ts , “True”*] to relation *EVENT* and a tuple

[*eid, "R", "True"*] to relation *ACTIVE*. As a result of its execution, the variables EID and TS get bound to the system-generated values *eid* and *ts* respectively. This instruction is used to insert into the eventbase "consumable" events and to make them "visible" to certain rules.

- $\langle EID, TS \rangle$:= *++("type", "datainfo")*—shortly *++("type", "datainfo")*—inserts a single event into the eventbase and makes it active for all rules; in particular, it adds a tuple [*eid, "type", "datainfo", ts, "True"*] to relation *EVENT* and one tuple [*eid, "R_i", "True"*], for each defined rule R_i, to relation *ACTIVE*. The variables EID and TS have the same meaning as before. This instruction is used to insert into the eventbase "consumable" events, making them "visible" to all rules.
- *-(EID)* deletes the tuple referring to the event *EID* from relation *EVENT*. This instruction is used in the following to delete from the eventbase "special-purpose", global events (e.g., the event *BeginAction*), which are not present in relation *ACTIVE*.
- *-(EID, "R")* deletes the tuple referring to the event *EID* and to rule *R* from relation *ACTIVE*.
- *-net(EID)*, *-net(EID, "R")* act in a way similar as *-(EID)*, *-(EID, "R")*, but instead of deleting event EID, they set its net-validity flag to "False".

For example, the update formula -($e24$, R_1), applied to the eventbase instance of Fig. 1, deletes the tuple relative to *e24* and to rule R_1 from relation *ACTIVE*. Note that relation *ACTIVE* may contain multiple tuples recording the same event for different rules: this update affects only the version of the event "seen" by rule R_1.

In active databases that provide the capability of querying past data, the eventbase is extended by adding a relation *LOG(id, attribute,value, ts)*, whose tuples record the values assumed by the various attributes of a database entity at different times, from the start of the transaction onward.

3.3 Syntax of Core Active Rules

Also a Core active rule consists of the three usual parts: the event, condition and action parts, possibly followed by the before/after declaration needed to specify rule priorities.

The **event part** is any well-formed formula of a f.o. language comprising the eventbase predicates *event* and *active*. This formula expresses the semantics of triggering.

The **condition part** is divided into two sub-parts. The former, called *DB/EBQ*, is the translation of the condition part of the EECA rule; we assume that DB/EBQ is an arbitrary range-restricted (see Ullman (1982)) well-formed formula, written in any first order language comprising the abovementioned eventbase predicates, possibly the *log* predicate (used to access past database states), and the predicates defined in the passive database of the specific system. The latter component of the condition part, called *EBU*, makes explicit the event consumption semantics of the EECA rule. It may be either the null operation NOP, if the EECA rule does not consume events or consumption takes place at rule execution time, or an update to the eventbase, enforcing the appropriate consumption semantics.

Finally, the **action part** contains one or more non-interruptable update blocks. Each block in turn contains a sequence of elementary updates, which can affect both the database and the eventbase, and share variable bindings. Structuring the action into

blocks ensures that the execution of each block is never interrupted by reactive processing. Instead, if a rule action formed by a sequence of blocks is interruptable, rule processing can be started between two subsequent blocks.

Variable bindings can be transferred from one part of a Core rule to other parts, according to the following schema:

$$ebq(\vec{y}_1);\ db/ebq(\vec{x}_2, \vec{y}_2), ebu(\vec{x}_2) \rightarrow TU_1(\vec{x}_{3_1}); \ldots; TU_n(\vec{x}_{3_n})$$

where:

a. $ebq(\vec{y}_1)$ is the event part formula, and $\vec{y}_1$, with $\mid \vec{y}_1 \mid \geq 0$, are all the free variables of $ebq(\vec{y}_1)$. These variables, if present, are bound to the identifiers of triggering events to be used in other parts of the rule.
b. $db/ebq(\vec{x}_2, \vec{y}_2)$ is the formula in *DB/EBQ*, $\vec{x}_2 \subseteq \vec{y}_1$ (safety) are input variables, and $\vec{y}_2$ are output variables representing variables of the EECA rule whose bindings may be transferred from the condition to the action part.
c. $ebu(\vec{x}_2)$ is a sequence of updates to the eventbase, which may take the variables $\vec{x}_2$ as input. This allows to model event consumption at consideration time.
d. $TU_1(\vec{x}_{3_1}); \ldots; TU_n(\vec{x}_{3_n})$ are update blocks; $\vec{x}_{3_i} \subseteq \vec{y}_1 \cup \vec{y}_2$ (safety) are input variables, by which it is possible to refer in the action part to the identifiers of the events to be consumed (this allows to model consumption at execution time) and to the IDs of items retrieved by the condition, which must be acted upon in the action. The passage of bindings between blocks is not permitted.

3.4 Translation of Active Rules

We now illustrate how EECA rules are translated into Core rules. Then, the semantics of EECA rules is completely defined by formally specifying the execution semantics of Core rules, which we do in Section 5.

Event

The translation of the event part from EECA to Core format has the major goals of expressing the EECA event part, of preventing rules from being triggered when the execution of another non-interruptable rule is in progress, and of making explicit the granularity and EC coupling of rules.

The event part of a Core rule consists of the query $ebq(\vec{y}_1)$ on the eventbase, derived as follows:

a. If a rule R has **EC immediate**, $ebq(\vec{y}_1)$ is the conjunction of queries **1.** and **2.**:

1. $[\ \exists EID]\ \bigvee_{i=1}^{n}(\exists M_i\ active(EID, \text{“}R\text{”}, M_i) \wedge$
$\exists X_i, TS_i, N_i\ event(EID, \text{“}event\text{-}type_i\text{”}, X_i, TS_i, N_i))$

here each *“event-type$_i$”* is one of the event types that appear in the event part of the EECA rule. This formula becomes true, or produces bindings for the variable EID, iff the eventbase contains events of type *“event-type$_i$”*, for some i, active for rule R.

The variable EID (which we call *event variable*) is free for instance-oriented rules, existentially quantified for set-oriented rules. In the instance-oriented case, the rule triggering algorithm, illustrated in Section 5, produces a different rule activation for each event that satisfies $ebq(\vec{y}_1)$. Hence, the variable EID gets bound to the identifier of one of the triggering events of rule R, and this binding is exploited in the condition and action parts to perform event consumption. In the set-oriented case, the formula expresses a boolean condition and thus no bindings are computed. In this case, the rule triggering algorithm produces only one rule activation irrespective of the number of triggering events. When triggering events must be retrieved in the condition or action part of the Core rule, an explicit eventbase query must be used.
If the EECA rule is triggered by **net** events of type *"event-type$_i$"*, the i-th disjunct in the above formula becomes:

$$active(EID, \text{"}R\text{"}, \text{"}True\text{"}) \wedge \exists X_i, TS_i, N_i \; event(EID, \text{"event-type}_i\text{"}, X_i, TS_i, N_i)$$

which excludes from triggering those events whose net-validity flag with respect to net effect is *"False"*.

2. $\forall EID, X, TS, N \; \neg event(EID, \text{"}BeginAction\text{"}, X, TS, N)$

tests for the absence from the eventbase of the special-purpose event *BeginAction*. In this way, if the EVENT relation contains an occurrence of the *BeginAction* event (meaning that the execution of some rule has begun and is not yet concluded), then the rule is not triggered, even if its activating events have occurred. *BeginAction* events are produced by non interruptable rules, to "freeze" the processing of events generated during their execution (see Section 3.4).

b. If a rule R has **EC deferred**, $ebq(\vec{y}_1)$ is built as in case a., but the query

$$\exists EID, TS, M, N \; active(EID, \text{"}R\text{"}, M) \wedge \; event(EID, \text{"}commit\text{"}, \text{"}NULL\text{"}, TS, N)$$

is conjuncted, to test for the presence of an active occurrence of the external event *commit*. In this way, if the *EVENT* relation does not contain an occurrence of the *commit* event, active for rule R, then R is not triggered and thus excluded from reactive processing. Note that, since the *commit* event is not consumed (see Section 3.4 and 3.4), multiple executions of the same deferred rule in the transaction are possible. *Commit* events are typically produced by omonimous commands issued at the end of the transaction; in addition, it is possible to model explicit rule activation statements (like, e.g., the Chimera *savepoint* and the Starburst *start rule* commands), by making these commands generate a *commit* event, before actually committing the transaction.

Condition

The goal of condition translation from EECA to Core format is to express the EECA condition, making explicit possible transaction history inspection primitives, and to exhibit the event consumption semantics of rules with **consumption-time consideration**.

In the Core format, the condition part is composed by an arbitrary query formula

$db/ebq(\vec{x}_2, \vec{y}_2)$ to the database and/or the eventbase, which expresses the EECA condition part, and by one or more elementary updates ($ebu(\vec{x}_2)$), which make explicit the event consumption semantics. The query *db/ebq* is obtained from the EECA condition part by rewriting without modification the subformulas that query the database and by translating the EECA literals **pending** and **history**, as explained in Section 3.4.

Depending on the rule granularity and event consumption mode, $ebu(\vec{x}_2)$ contains different eventbase updates. The following holds for instance-oriented rules.

a. If the EECA rule has **consumption-scope local**, the translated Core rule contains an eventbase update of the form –(EID, R), where EID is the event variable defined in Section 3.4. In this way, the triggering event bound to EID is physically deleted after consideration from the part of relation *ACTIVE* seen by the translated Core rule *R*, and cannot subsequently retrigger R. Instead, it remains in the relation *EVENT* and can be retrieved by queries translating the EECA predicate **history**.
   If the EECA rule has **consumption-scope global-to** $\{R_1 \ldots R_n\}$, then an additional set of eventbase updates –(EID, R_i), for $i = 1 \ldots n$, is added. This detriggers also rules $\{R_1 \ldots R_n\}$.
b. If the EECA rule has **consumption-time execution** or **consumption-scope none**, $ebu(\vec{x}_2)$ is the null eventbase update NOP. In this way, the triggering events retain their "active" status, so that they still trigger the rule and can also be later retrieved by queries on the transaction history.

The translation of set-oriented rules is similar, except for the computation of the bindings for the event variable EID. In the instance-oriented case, EID is bound in the event part and the binding is used in the condition; conversely, in set-oriented rules, no binding for EID is passed from the event part, and triggering events to be consumed must be bound to EID in $ebu(\vec{x}_2)$.

Hence, the eventbase update of point a. becomes:

$\bigvee_{i=1}^{n}(\exists M_i\ active(EID, \text{“}R\text{”}, M_i)\ \wedge\ \exists X_i, TS_i, N_i\ event(EID, \text{“}event\text{-}type_i\text{”}, X_i, TS_i, N_i)),$
$-(EID, R)..$

If only net events of type *event-type*$_i$ are considered for triggering, the i-th disjunct in the above update becomes:

$active(EID, \text{“}R\text{”}, \text{“}True\text{”}) \wedge\ \exists X_i, TS_i, N_i\ event(EID, \text{“}event\text{-}type_i\text{”}, X_i, TS_i, N_i),$

so that only events with net-validity flag "True" of type *"event-type*$_i$*"* are retrieved and consumed.

Action

The translation of the action part of an EECA rule into Core format has three goals: to express the EECA action part, to enforce the atomic execution of non interruptable rules, and to exhibit the event consumption semantics of rules with **consumption-time execution**. Updates in the EECA action part are transferred without modification to the Core action part.

Non-interruptability of rules is achieved by generating an external event of type *BeginAction* at the beginning of the action, which is then removed at the end; to do so, the action is prefixed by an update, *+("BeginAction", "NULL")*, and ended by an update

$\exists$ *TS, N event(EID, "BeginAction", "NULL", TS, N), -(EID).*

This ensures that no rule can be awakened during the execution of a non interruptable rule, since the event part of all Core rules contains a query for the absence of events of type *BeginAction*, as shown in Section 3.4. Only after a non interruptable rule has finished, and removed the *BeginAction* event, rules can take part again to reactive processing.

If the **consumption-time** option of the rule is **execution**, then the eventbase updates for consumption illustrated in point a. of Section 3.4 must be added to the action part. Event consumption is performed before the first database update of the EECA action part, to ensure that interruptable rules that can recursively trigger themselves are activated after the "old" events have been consumed and thus "see" only the most recent events. Note that consuming triggering events at the beginning of the action does not hamper the possibility of querying them later in the action part (e.g., to retrieve the objects affected by them); since the EECA C-A mode is always immediate, the needed queries can be equivalently moved to the condition part and their result bound to variables referenced in the action part.

Net Effect Computation

We model net events as tuples of relation *EVENT* and *ACTIVE*, whose net-validity flag has value "True". When tuples are originally inserted into relation *EVENT* and *ACTIVE*, the net-validity flag is always "True". Then, the application of net effect criteria may lead to setting the flag to false for those events that have a null net effect.

In principle, any formalism can be employed to specify the computation of the net-validity flag of events . We have chosen to do that by means of a set of ad-hoc Core rules, called *auxiliary rules*, that recognize specific event patterns and apply the corresponding net effect criteria. They are defined with topmost priority, so that normal Core rules, executed after all triggered auxiliary rules have been processed, can "see" an always up-to-date version of the net-validity flags in relation *EVENT* and *ACTIVE*. This approach further simplifies the rule execution semantics, since auxiliary rules are treated as ordinary Core rules and there is no need to extend the rule execution model to cope with net effect. Also, net effect criteria are not hard-wired into the semantics, but are a parameter of the execution model: switching from one system to another, with different net effect criteria, simply requires to consider a different set of auxiliary rules, without modifying the model.

Translation of Past Event Queries

Objects affected by events occurred during a transaction are retrieved by means of the EECA predicates **pending** and **history**.

An occurrence of the literal **pending(E,X)** in a rule R, where **E** is a set of event types $\{\mathbf{E}_1, \ldots, \mathbf{E}_n\}$, is translated into the following eventbase query in Core syntax:

$$[\exists\, EID] \bigvee_{i=1}^{n} (\exists\, M_i\ active(EID, \text{“}R\text{”}, M_i) \wedge \exists\, TS_i, N_i\ event(EID, \text{“}E_i\text{”}, X, TS_i, N_i))$$

The event variable EID is free in instance-oriented rules, existentially quantified in set-oriented rules. In the former case, the binding computed in the event part is used for EID and the variable X becomes bound to the identifier of the item affected by the single event corresponding to one particular rule activation. In the latter, all events present in relation ACTIVE are considered and thus the query binds the variable X to the identifiers of items affected by all triggering events.

An occurrence of the literal **history(E,X)** in a rule R, where **E** is a set of event types $\{\mathbf{E}_1, \ldots, \mathbf{E}_n\}$, is translated into the following eventbase query in Core syntax:

$$\bigvee_{i=1}^{n} (\exists EID_i, TS_i, N_i \; event(EID_i, \text{“}E_i\text{”}, X, TS_i, N_i))$$

This query binds the variable X to the identifiers of items affected by events of the desired types, occurred during the transaction. If any of the event types in **E** is prefixed by the keyword **net**, then only "net-valid" tuples of relation *EVENT* must be considered for that event type; hence, the corresponding disjunct becomes formula number **1.** in the **pending** case, or formula number **2.** in the **history** case.

1. $active(EID, \text{“}R\text{”}, \text{“}True\text{”}) \wedge \exists TS_i, N_i \; event(EID, \text{“}E_i\text{”}, X, TS_i, N_i)$
2. $\exists EID_i, TS_i, \; event(EID_i, \text{“}E_i\text{”}, X, TS_i, \text{“}True\text{”})$

Example of Rule Translation into EECA and Core Format

Rule *Raise_Sam's_salary* gives Sam a 10% increase after each update to Sam's salary, until it reaches $5000. The syntax used in this example is that of Postgres.

```
define rule Raise_Sam's_salary
on replace to EMP.salary where
    EMP.name = "Sam" and EMP.salary<5000
then do replace EMP (salary = 1.1*NEW.salary) where
        EMP.name = CURRENT.name
```

In Postgres, rules have *instance oriented* granularity, *immediate* E-C coupling mode, and *local consumption* of events performed at *consideration time*; all these options are made explicit in the EECA rule.

```
define granularity I/O EC immediate interruptable True
       consumption-scope local consumption-time consideration
       rule Raise_Sam's_salary
event: replace(Emp.salary)
condition: pending(replace(Emp.salary),X),
           X.name="Sam", X.salary<5000 (*X=NEW tuple*)
action: replace(Emp.salary, X, 1.1*X.salary)
```

Then, the translation from EECA into Core format is performed. Notice that the Core version of rule *Raise_Sam's_salary*'s condition (DB/EBQ) contains a query binding the tuple affected by the triggering event to variable X and it may be fired just after the update of Sam's salary generates the corresponding event.

```
RULE Raise_Sam's_salary
(*CORE EVENT PART *)
  ∃M1 active(EID,..., M1) ∧
  ∃X1,TS1,N1 event(EID, "replace(Emp.salary)", X1, TS1, N1) ∧
  ∀EID2, X2, TS2,N2 ¬ event(EID2, "BeginAction", X2, TS2, N2);
(*CORE CONDITION PART: DB/EBQ *)
  ∃M3 active(EID, ..., M3) ∧
  ∃TS3,N3 event(EID, "replace(Emp.salary)", X, TS3, N3) ∧
  X.name="Sam" ∧ X.salary<5000,
(*CORE CONDITION PART: EBU *)
  -(EID, Raise_Sam's_salary)
⟶ (*CORE ACTION PART*)
  replace(Emp.salary, X, 1.1*X.salary)
```

4 SEMANTICS OF UPDATES AND TRANSACTIONS IN AN ACTIVE DATABASE

Prior to introducing the semantics of rule processing, we present the syntax and semantics of updates, scaling from *atomic updates* to *transactions.*

4.1 Preliminary Notions

In the sequel, we often use the notion of *variable substitution*: given an array of variables $\vec{x} = \langle x_1 \dots x_n \rangle$ with domains $D_1, \dots, D_n$, a *ground* substitution θ can be regarded as a tuple $\langle c_1 \dots c_n \rangle \in D_1 \times \dots \times D_n$, and a set of ground substitutions as a relation, whose attributes are just the variable names $x_1 \dots x_n$. Therefore, sets of ground substitutions can be manipulated by means of the usual relational operators.

Two special substitutions will be frequently used:

Definition 1:

- The *empty* substitution $\perp$ is a polymorphic symbol that stands for a substitution of any arity that assigns the null value to all its variables.
- The *void* substitution ι stands for a substitution with zero arity.

Given a set of ground substitutions Θ for the variables $\vec{x}$, the following definitions hold:

(1) $\Theta \times \{\iota\} = \Theta$ $\forall\Theta$, i.e., $\{\iota\}$ works as the identity element for sets of substitutions;
(2) $\Pi_{\vec{y}}\Theta = \{\iota\}$, if $\vec{y} \cap \vec{x} = \emptyset$;
(3) $\Theta \times \{\perp\} = \{\perp\}$ $\forall\Theta$, i.e., $\{\perp\}$ works as the zero element for sets of substitutions.

Note that definition (3) requires that the extension of a ground substitution with another substitution that assigns the null values to its variables be considered as the empty substitution.

We also make extensive use of binary and ternary relations :

Definition 2:
The composition operator $\circ$ for binary relations is defined as: $\gamma_1 \circ \gamma_2 \stackrel{\text{def}}{=} (\Pi_{1,4}\gamma_1 \bowtie_{2=1} \gamma_2)$.
The composition operator $\bullet$ for ternary relations is defined as $\gamma_1 \bullet \gamma_2 \stackrel{\text{def}}{=} (\Pi_{1,5,6}\gamma_1 \bowtie_{2=1} \gamma_2)$.

Definition 3: Let $s = (DB, EB)$ and $s' = (DB', EB')$ be two states; we say that s is equivalent to s' (i.e. $s \simeq s'$) if there exists a substitution of oids, eids and time-stamps

$$\xi = (oid_1/oid'_1, eid_1/eid'_1, ts_1/ts'_1 \ldots oid_n/oid'_n, eid_n/eid'_n, ts_n/ts'_n)$$

such that $DB\xi = DB'$ and $EB\xi = EB'$.

Given a state s, $\tau^{(1)}$ is a mapping from s to its equivalence class w.r.t. $\simeq$; given an equivalence class ec, $\tau^{(2)}$ is a mapping from ec to a representing state; then the composite function $\tau = \tau^{(2)} \circ \tau^{(1)}$ is a mapping from a state to the state representing its equivalence class; if $s_2 = \tau(s_1)$, this induces a substitution $\xi_{\tau(s_1)}$ such that $s_1\xi_{\tau(s_1)} = s_2$. Considering a ground substitution θ, we can obtain a new ground substitution θ' by applying $\xi_{\tau(s_1)}$ in the following way: $\theta\xi_{\tau(s_1)} = \theta'$, for any state s_1. In the following sections, to represent the active database semantics we make use of ternary relations γ containing tuples composed by a pair of states and a substitution : $\gamma = \langle s_1, s_2, \theta \rangle$; the notation $\gamma' = \tau(\gamma)$ stands for the relation obtained applying τ to every state s_2 of γ and composing $\xi_{\tau(s_2)}$ with the associated θ .

4.2 Atomic Updates

Atomic updates are the elementary means by which the state of an active database is changed. They can be distinguished in *state transitions*, which are single modifications to the database, and *elementary updates*, which specify the set-oriented application of a state transition, by means of input variables. Elementary updates and state transitions may be combined in sequences, to form *transaction units* and *transactions.*

State Transitions

The state of an active database evolves in time by means of *state transitions.* These are state changes with a twofold effect: they may alter the content of the passive database, and generate *events.* In addition, state transitions can have *output variables*, which contain the output of the transition. This happens, for example, in the case of a transition that creates a new object and returns as output the *oid* of that object, and in the case of a query.

Definition 4: (**Syntax**) A *state transition* is an expression $u(\vec{c}, \vec{y})$, where $\vec{c}$ is an array of constants and $\vec{y}$ is an array of output variables, with $|\vec{c}| \geq 0, |\vec{y}| \geq 0$.

Definition 5: (**Pre-Semantics**) The execution pre-semantics of a state transition $u(\vec{c}, \vec{y})$ is specified by a ternary relation $\mathcal{PS}_{u(\vec{c},\vec{y})}$, which contains tuples of the type $\langle (DB1, EB1), (DB2, EB2), \theta \rangle$, where $(DB1, EB1), (DB2, EB2)$ are states of the active database and θ is a *ground* substitution for the variables $\vec{y}$. In $\mathcal{PS}_{u(\vec{c},\vec{y})}$, the pair $(EB1, EB2)$ is such that if there are events in $EB2$ which were not present in $EB1$, their time stamp must be greater than the time stamp of any event in $EB1$ (event generation is monotonic).

$\langle (DB1, EB1), (DB2, EB2), \theta \rangle \in \mathcal{PS}_{u(\vec{c},\vec{y})}$ intuitively means that the execution of $u(\vec{c}, \vec{y})$ in the state $(DB1, EB1)$ yields a new state $(DB2, EB2)$ and binds the variables $\vec{y}$ to the values specified by the ground substitution θ. For state transition without output variables, θ is ι.

In the following, $(DB1, EB1)$ is called the *before state*, $(DB2, EB2)$ is called the *after state*, and θ is called the *output* of $u(\vec{c}, \vec{y})$. For the sake of brevity, we sometimes write *bs* and *as*, when $(DB1, EB1)$ and $(DB2, EB2)$ are understood.

$\mathcal{PS}_{u(\vec{c},\vec{y})}$ contains a tuple for every after state which can be obtained executing the state transition starting from each possible before state of $u(\vec{c}, \vec{y})$. Note that in many cases the execution of a state transition can lead to different after states obtained from the same before state; such a fact is represented by means of tuples with the same *bs* and different *as*. At the data level, if the different after states are equivalent according to Def. 3 this corresponds to the fact that two databases are the same *up to OID isomorphism* (in object oriented databases).

In contrast, when different *as*'s are not equivalent w.r.t. $\simeq$, we have a truly non-deterministic state transition.

The absence from $\mathcal{PS}_{u(\vec{c},\vec{y})}$ of tuples with before state equal to $(DB1, EB1)$ means that $(DB1, EB1)$ is not a possible before state of $u(\vec{c}, \vec{y})$. In other words, when applied to $(DB1, EB1)$, $u(\vec{c}, \vec{y})$ *fails.*

Definition 6:(Semantics) The execution semantics of a state transition $u(\vec{c}, \vec{y})$ is specified by a ternary relation $\gamma_{u(\vec{c},\vec{y})}$ obtained from $\mathcal{PS}_{u(\vec{c},\vec{y})}$ as follows :

$$\gamma_{u(\vec{c},\vec{y})} = \tau(\mathcal{PS}_{u(\vec{c},\vec{y})});$$

also in $\gamma_{u(\vec{c},\vec{y})}$, the constraint on the pairs $(EB1, EB2)$ must be satisfied : if there are events in $EB2$ which were not present in $EB1$, their time stamp must be greater than the time stamp of any event in $EB1$.

We can view the application of τ as the particular way chosen by the real active database system to execute a state transition, when a state transition can produce many different but equivalent *as*'s from the same *bs*. In this situation the system chooses only one of the possible sets of oids for the objects in the database, and this is expressed by the representative of the class containing all the equivalent *as*'s, provided by τ. In general, after the application of τ we obtain a smaller relation, because all equivalent after states are collapsed into one after state.

Examples of State Transitions

For the purpose of illustration, we describe below a set of possible generic state transitions, w.r.t. a generic object oriented database. Notice that with ϕ and ψ we indicate f.o. formulas asserting a change in the database state. With $\langle DB1,\ EB1 \rangle$ we indicates a valid before state. If the state transition is applied to an invalid before state, it fails.

1. $\gamma_{create(C,S,O)} = \{\langle (DB1, EB1), (DB1 + \psi, EB2), \langle O/OID \rangle \rangle\}$
2. $\gamma_{modify(C.A,O,V)} = \{\langle (DB1, EB1), (DB1 + \phi, EB2), \iota \rangle\}$
3. $\gamma_{query(C,S,O)} = \{\langle (DB1, EB1), (DB1, EB2), \langle O/OID \rangle \rangle\}$
4. $\gamma_{commit} = \{\langle (DB1, EB1), (DB1, EB2), \iota \rangle\}$
5. $\gamma_{rollback} = \{\langle (DB1, EB1), (DB1, EB2), \iota \rangle\}$.

1. A new object with initial state S is added to class C and assigned a certain OID;

$\langle O/OID \rangle$ binds the output variable O to OID; EB2 is the eventbase state produced by the following EB updates: $+ + \langle$ *"create(C)", O*$\rangle$.
2. The value V is assigned to attribute A of object O belonging to class C and the old value is no longer valid; EB2 is the eventbase state produced by the following EB updates: $+ + \langle$ *"modify(C.A)", O*$\rangle$.
3. Before database state (DB1) does not change after the state transition execution, $\langle O/OID \rangle$ binds the output variable O to OIDs indicating objects in class C respecting selection condition S (the aimed result of the query). EB2 is the eventbase state produced by the following EB updates: $+ + \langle$ *"query(C)", O*$\rangle$.
4. A **commit** state transition simply adds a *commit* event to the event base with the EB updates: $+\langle$ *"commit", "NULL"*$\rangle$; as a consequence, deferred rules are allowed to start.
5. A **rollback** state transition is very similar to a **commit** state transition, except for the event type *rollback* added to the eventbase.

Elementary Updates

State transitions can be applied in a set-oriented way, producing *elementary updates.*

Definition 7: (Syntax) An elementary update is an expression $u(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are arrays of variables, with $|\vec{x}| > 0$, such that for every ground substitution $\sigma \neq \perp$ of $\vec{x}$ it holds that $u(\vec{x}, \vec{y})\sigma$ is a state transition. Variables in $\vec{x}$ are called input variables, whereas variables in $\vec{y}$ are called output variables.

Thus, an elementary update allows to express the bulk application of state transitions: the transitions to be executed are determined by a set of ground substitutions for the input variables of the elementary update.

The (set-oriented) execution semantics $\gamma_{u(\vec{x},\vec{y})}$ of an elementary update $u(\vec{x}, \vec{y})$ is defined in terms of that of state transitions. First, we give the semantics of the special case $u(\vec{x}, \vec{y})\ \perp$. The intuitive meaning of such an expression is that an elementary update is evaluated, which misses the values for its input variables: such a case is considered a failure and its semantics is denoted by the empty relation, i.e., $\gamma_{u(\vec{x},\vec{y})\perp} = \emptyset$. In other words, for every possible before state, no valid after state is reached by executing $u(\vec{x}, \vec{y})\ \perp$.

Now we can introduce the semantics of an elementary update with respect to a set of bindings Σ for the input variables (possibly containing the empty substitution $\perp$).

Definition 8: (Semantics) Given an elementary update $u(\vec{x}, \vec{y})$ and given a set $\Sigma = \{\sigma_1 \ldots \sigma_n\}$ of ground substitutions for $\vec{x}$, let $\Sigma_j = \sigma_1^j \ldots \sigma_n^j$ be one of the possible $n!$ orderings of Σ. The semantics of the set-oriented execution of $u(\vec{x}, \vec{y})$ with respect to Σ is given by the ternary relation $\gamma_{u(\vec{x},\vec{y})\Sigma}$ defined as follows:

$$\gamma_{u(\vec{x},\vec{y})\Sigma} \equiv \tau\Big(\bigcup_{j=1}^{n!}(\Pi_{bs,as}\beta_n^j) \times \alpha_n^j\Big), \qquad \text{where:}$$

$$\beta_1^j = \gamma_{u(\vec{x},\vec{y})\sigma_1^j} \qquad \beta_{i>1}^j = \beta_{i-1}^j \bullet \gamma_{u(\vec{x},\vec{y})\sigma_i^j}$$
$$\alpha_1^j = \Pi_\theta\gamma_{u(\vec{x},\vec{y})\sigma_1^j} \qquad \alpha_{i>1}^j = \alpha_{i-1}^j \cup \Pi_\theta\beta_i^j.$$

Relation β_n^j contains the after states generated by the evaluation of the state transitions $u(\vec{x}, \vec{y})\sigma_1^j \ldots u(\vec{x}, \vec{y})\sigma_n^j$, corresponding to the ordering $\Sigma_j = \sigma_1^j \ldots \sigma_n^j$, whereas relation α_n^j

collects all the ground substitutions for the output variables produced in the sequential execution.

Hence, $\bigcup_{j=1}^{n!}(\Pi_{bs,as}\beta_n^j) \times \alpha_n^j$ produces all the possible after states that can be obtained by applying in an arbitrary order the tuple-oriented evaluation of $u(\vec{x},\vec{y})\sigma_i$ for each σ_i in Σ and a set of bindings for the output variables that is the union of those generated by the tuple-oriented evaluations. It contains one tuple for each after state and output ground substitution generated by the bulk application of $u(\vec{x},\vec{y})$. In particular, if the bulk application of an elementary update is non-deterministic, i.e., if it yields different results for different orders Σ_j, then relation $\bigcup_{j=1}^{n!}(\Pi_{bs,as}\beta_n^j) \times \alpha_n^j$ cointains a set of tuples for each different order.

Remind that the mapping τ is obtained by composing $\tau^{(1)}$, which identifies the equivalence class of a state, and $\tau^{(2)}$, that chooses the representing state of an equivalence class. When τ is applied to a state transition, $\tau^{(2)}$ chooses the representing states in such a way that time stamp generation is monotonic (by definition of state transition semantics). When τ is applied to elementary updates, $\tau^{(2)}$ is the same mapping applied to every state transition $u(\vec{x},\vec{y})\sigma_i^j$, then the time stamp generation is still monotonic in an elementary update. Thus :

Proposition 1: For each tuple in $\gamma_{u(\vec{x},\vec{y})\Sigma}$, if there are events in $EB2$ which were not present in $EB1$, their time stamp is greater than the time stamp of any event in $EB1$.

We say that an elementary update is *set-oriented deterministic* if its relation $\gamma_{u(\vec{x},\vec{y})\Sigma}$ contains only one *as* for each *bs*. This fact happens when the elementary update generates many different but equivalent after states; the application of τ (and thus the choice of the representative state) reflects the way chosen by the system to compute the elementary update.

We prefix the term *set-oriented* to the word *deterministic*, because such an elementary update is not strictly deterministic, since different executions of the same elementary update on the same before state can follow different ways leading to equivalent after states.

We say that the elementary update is *set-oriented non-deterministic* when, for each before state, $\gamma_{u(\vec{x},\vec{y})\Sigma}$ contains more than a tuple. This fact means that the elementary update can lead to more than one acceptable after state which are not equivalent, and the application of τ cannot find a common after state for each before state.

Note that, since the sequential execution is represented by the iterated composition of the relations corresponding to the state transitions (with the condition that the after state produced by the $i-th$ transition must be a valid before state of the $(i+1)-th$ one) $\gamma_{u(\vec{x},\vec{y})\Sigma}$ does not contain partial applications of $u(\vec{x},\vec{y})$, i.e.:

Proposition 2: If, for some ordering $\Sigma_j = \sigma_1^j \ldots \sigma_n^j$, some state transition $u(\vec{x},\vec{y})\sigma_i^j$ fails, then $\Pi_{bs,as}\beta_n^j \times \alpha_n^j = \emptyset$ (i.e., no tuple corresponding to Σ_j appears in $\gamma_{u(\vec{x},\vec{y})\Sigma}$).

Also, the lack of input bindings for some input substitution causes the entire set-oriented execution of an elementary update to fail, i.e.:

Proposition 3: If $\perp \in \Sigma$, then $\gamma_{u(\vec{x},\vec{y})\Sigma} = \emptyset$.

Thus, elementary updates are truly set-oriented: either they succeed for all the input substitutions or they fail.

Finally, by definition, if $|\vec{y}|=0$, it holds that $\Pi_\theta \gamma_{u(\vec{x},\vec{y})\Sigma} = \{\iota\}, \forall u(\vec{x},\vec{y})$ *and* $\forall\Sigma$.

Note that a state transition can be considered as an elementary update applied to a singleton set of ground substitutions, i.e., $u(\vec{c},\vec{y}) \equiv u(\vec{x},\vec{y})\langle\vec{x}/\vec{c}\rangle$. Thus, from now on we will use relation $\gamma_{u(\vec{x},\vec{y})\Sigma}$ to represent the semantics of both elementary updates and state transitions.

Examples of Elementary Updates

Consider the update $u(x_1,y_1) = create(C,x_1,y_1)$ (see Section 4.2 for its semantics) and the set of ground substitutions $\Sigma = \{\langle x_1/T_1\rangle, \langle x_1/T_2\rangle\}$.
Let $\Sigma_1 = \langle\langle x_1/T_1\rangle, \langle x_1/T_2\rangle\rangle$ be an order leading to the final state $bs + \psi_1 + \psi_2$, and $\Sigma_2 = \langle\langle x_1/T_2\rangle, \langle x_1/T_1\rangle\rangle$, an order leading to $bs + \psi'_2 + \psi'_1$.

$$\gamma_{create(C,x_1,y_1)\Sigma} = \tau \left(\begin{array}{ll} \{ & \langle bs, bs+\psi_1+\psi_2, y_1/o_1\rangle \\ & \langle bs, bs+\psi_1+\psi_2, y_1/o_2\rangle \\ & \langle bs, bs+\psi'_2+\psi'_1, y_1/o'_2\rangle \\ & \langle bs, bs+\psi'_2+\psi'_1, y_1/o'_1\rangle \ \} \end{array} \right) =$$

$$= \begin{array}{ll} \{ & \langle bs, bs+\psi_1+\psi_2, y_1/o_1\rangle \\ & \langle bs, bs+\psi_1+\psi_2, y_1/o_2\rangle \ \} \end{array}$$

because $bs+\psi_1+\psi_2 \simeq bs+\psi'_1+\psi'_2$ and we suppose that τ chooses $bs+\psi_1+\psi_2$ as the representative of the equivalence class; since for each before state $\gamma_{create(C,x_1,y_1)\Sigma}$ contains only one after state, the elementary update is set-oriented deterministic.

Let us consider now the update $u(x_1,x_2) = modify(C.A, x_1, A \cup x_2)$ (see Section 4.2) and the set of ground substitutions $\Sigma = \{\langle x_1/o_1, x_2/e_1\rangle, \langle x_1/o_1, x_2/e_2\rangle\}$.
Let $\Sigma_1 = \langle\langle x_1/o_1, x_2/e_1\rangle, \langle x_1/o1, x_2/e_2\rangle\rangle$, leading to the state $bs + \phi_1 + \phi_2$, and $\Sigma_2 = \langle\langle x_1/o_1, x_2/e_2\rangle, \langle x_1/o_1, x_2/e_1\rangle\rangle$, leading to $bs + \phi'_2 + \phi'_1$.

$$\gamma_{modify(C.A,x_1,x_2)\Sigma} = \tau \left(\begin{array}{ll} \{ & \langle bs, bs+\phi_1+\phi_2, \iota\rangle \\ & \langle bs, bs+\phi'_2+\phi'_1, \iota\rangle \ \} \end{array} \right) = \{ \ \langle bs, bs+\phi_1+\phi_2, \iota\rangle \ \}$$

because $bs+\phi_1+\phi_2 \simeq bs+\phi'_1+\phi'_2$ and we suppose that τ chooses $bs+\phi_1+\phi_2$ as the representative of the equivalence class; since for each before state $\gamma_{modify(C.A,x_1,x_2)\Sigma}$ contains only one after state, the elementary update is set-oriented deterministic.

4.3 Complex Updates

Transaction Units

In this Section we introduce the notion of a transaction unit, i.e., a sequence of atomic updates possibly sharing variable bindings.

Definition 9: (Syntax) A *transaction unit* $TU(\vec{x})$ is a *sequence* $u_1(\vec{t_1},\vec{y_1}),\ldots,u_n(\vec{t_n},\vec{y_n})$, where: u_i is an atomic update, $\vec{t_i}$ is an array of terms[¶], the $\vec{y_i}$ are possibly empty arrays of distinct variables, each variable in $\vec{t_i}$ either appears in one $\vec{y_j}$, with $j<i$, or is a variable of $\vec{x}$.

According to the above definition, a transaction unit may have only input parameters.

[¶] If u_i is a state transition, $\vec{t_i}$ is an array of constants $\vec{c_i}$; otherwise it is an array of variables $\vec{x_i}$.

The reason for this will be clarified in Section 4.3. Moreover, for a transaction unit to be evaluable, the input parameters must be bound to constant values outside the transaction unit. The elementary updates that appear in the transaction unit have input (and possibly output) variables; each of them may draw the bindings for its input variables either from the global input variables $\vec{x}$, or from the output variables of some elementary update or state transition that precedes it in the sequence.

The execution semantics of a transaction unit with respect to a set Σ of ground substitutions for its input variables is formally specified by a binary relation $\gamma_{TU(\vec{x})\Sigma}$:

Definition 10: (**Semantics**) Given a transaction unit $TU(\vec{x}) = u_1(\vec{t}_1, \vec{y}_1), \ldots, u_n(\vec{t}_n, \vec{y}_n)$ and a (possibly empty) set Σ of ground substitutions for $\vec{x}$, the execution semantics of TU with Σ is given by:

$$\gamma_{TU(\vec{x})\Sigma} = \Pi_{bs,as}\beta_n, \quad \text{where}$$

$$\begin{array}{lll} \beta_1 & = & \gamma_{u_1(\vec{x},\vec{y})\Theta_1} \qquad \beta_{i>1} = \beta_{i-1} \bullet \gamma_{u_i(\vec{x},\vec{y})\Theta_i} \\ \alpha_i & = & \Pi_\theta \beta_i \\ \Theta_i & = & \Pi_{\vec{x}_i}\Sigma \times \Pi_{\vec{x}_i}\alpha_1 \ \ldots \times \ldots \Pi_{\vec{x}_i}\alpha_{i-1}. \end{array}$$

Relation $\gamma_{TU(\vec{x})\Sigma}$ contains tuples of the form $\langle bs, as \rangle$, where bs is a possible before state of TU and as is an after state. These tuples are computed by executing the sequence of elementary updates and/or state transitions listed in the transaction unit; at intermediate steps, the bindings for the input variables $\vec{x}_i$ of an elementary update are given by the cartesian product of the projections over $\vec{x}_i$ of Σ and of all the bindings computed at preceding steps.
Since a join with an empty relation (corresponding to a failing elementary update) is empty :

Proposition 4: If an elementary update or state transition u_i fails, all the subsequent steps and hence the whole transaction unit fails.

Since $\Theta_i = \{\perp\}$ yields $\gamma_{u_i(\vec{x}_i,\vec{y}_i)\Theta_i} = \emptyset$ and thus $\gamma_{TU(\vec{x})\Sigma} = \emptyset$:

Proposition 5: If an elementary update $u_i(\vec{x}_i, \vec{y}_i)$ lacks the bindings for its input variables, because at some previous step an empty substitution has been returned for some of the $\vec{x}_i$, the whole transaction unit fails.

Since time-stamp generation is monotonic, precedences among events generated by different steps of the same transaction unit are not altered. Indeed, Definition 6 and Proposition 1 ensure that time-stamp generation in state transitions and elementary updates is monotonic. A transaction unit is obtained as a chain of elementary updates, then it is monotonic w.r.t. the time-stamp generation process.

Proposition 6: In a transaction unit, the time-stamp generation process is monotonic.

Example of Transaction Unit

Consider the transaction unit

$$TU(x_1, x_2) = \mathit{create}(C1, x_1, y_1), \mathit{create}(C2, x_2, y_2), \mathit{modify}(C2.A, y_2, A \cup y_1)$$

and the set of ground substitutions $\Sigma = \{\langle x_1/T_1, x_2/T_2\rangle,\ \langle x_1/T_1', x_2/T_2'\rangle\}$ (see Section 4.2), in an active database.

The evaluation of TU with respect to Σ proceeds as follows: first two objects, say o_1, o_1' are created in class $C1$, with state T_1, T_1' and the bindings $\langle y_1/o_1\rangle$, $\langle y_1/o_1'\rangle$ are produced; then, two objects, say o_2, o_2' are created in class $C2$, with state T_2, T_2' and the bindings $\langle y_2/o_2\rangle$, $\langle y_2/o_2'\rangle$ are generated. Finally, the *oids* of o_1, o_1' are added to the object-valued attribute A of both objects o_2, o_2'. Below we show the various steps of the computation of the relation $\gamma_{TU(\vec{x})\Sigma}$; to do that, we make use of the relations $\gamma_{create(C,x_1,y_1)\Sigma}$ and $\gamma_{modify(C.A,x_1,x_2)\Sigma}$ of Section 4.2.

$$\Theta_1 = \Pi_{x_1}\Sigma = \{\langle x_1/T_1\rangle, \langle x_1/T_1'\rangle\}$$

$$\beta_1 = \gamma_{create(C1,x_1,y_1)\Theta_1} = \{\langle bs, bs+\psi_1+\psi_1', y_1/o_1\rangle, \langle bs, bs+\psi_1+\psi_1', y_1/o_1'\rangle\}$$

$$\Theta_2 = \Pi_{x_2}\Sigma \times \Pi_{x_2}\Pi_\theta\beta_1 = \Pi_{x_2}\Sigma \times \{\iota\} = \{\langle x_2/T_2\rangle, \langle x_2/T_2'\rangle\}$$

$$\beta_2 = \beta_1 \bullet \gamma_{create(C1,x_2,y_2)\Theta_2} = \{\langle bs, bs+\psi_1+\ldots+\psi_2', y_2/o_2\rangle, \langle bs, bs+\psi_1+\ldots+\psi_2', y_2/o_2'\rangle\}$$

$$\Theta_3 = \Pi_{y_1,y_2}\Sigma \times \Pi_{y_1,y_2}\Pi_\theta\beta_1 \times \Pi_{y_1,y_2}\Pi_\theta\beta_2 = \{\iota\} \times \Pi_{y_1,y_2}\Pi_\theta\beta_1 \times \Pi_{y_1,y_2}\Pi_\theta\beta_2 = \{\langle y_1/o_1, y_2/o_2\rangle, \langle y_1/o_1, y_2/o_2'\rangle, \langle y_1/o_1', y_2/o_2\rangle, \langle y_1/o_1', y_2/o_2'\rangle\}$$

$$\beta_3 = \beta_2 \bullet \gamma_{modify(C2.A,y_2,A\cup y_1)\Theta_3} = \{\langle bs, bs+\psi_1\ldots\psi_2'+\phi_1\ldots\phi_2', \iota\rangle\}$$

$$\gamma_{TU(\vec{x})\Sigma} = \Pi_{bs,as}\beta_3 = \{\langle bs, bs+\psi_1+\psi_1'+\psi_2+\psi_2'+\phi_1+\phi_1'+\phi_2+\phi_2'\rangle\ \}$$

Note that, in the above transition unit, the elementary update $modify(C2.A, y_2, A \cup y_1)$ has no output variables; hence $\Pi_\theta\beta_3 = \{\iota\}$.

Transactions

A transaction $T(\vec{x})$ is a sequence of transaction units, where $\vec{x}$ is a possibly empty array of input variables.

Definition 11: (Syntax) A *transaction* $T(\vec{x})$ is an expression $TU_1(\vec{x}_1), \ldots, TU_n(\vec{x}_n)$, where: $TU_i(\vec{x}_i)$ is a transaction unit and each variable in $\vec{x}_i$ appears in $\vec{x}, \forall i = 1 \ldots n$.

Transactions are, together with Core rules (presented in Section 3.3), the fundamental units of computation of an active database. They are applied to an *original state* and, if their computation terminates, lead to a *final state.* During the computation of a transaction, the active database system may react to the occurrence of events and trigger the execution of active rules. Anywhere in the execution of a transaction a *rollback* statement may be issued, whose effect is that of producing a final state of the active database that coincides with the original state. Conversely, if no rollback is issued and the computation of active rules terminates, the transaction is committed to a final state, which reflects the updates performed by the transaction and those made by the reactive system.

The semantics of a transaction is intuitively represented by a binary relation that associates each original state to the set of reachable final states.

Definition 12: (**Semantics**) The execution semantics of a transaction $T(\vec{x})$ with respect to an active database $\mathcal{D}$ and a set of ground substitutions Σ for $\vec{x}$ is given by the binary relation $\Gamma_{T(\vec{x})\Sigma,\mathcal{D}}$ defined as follows:

$$\Gamma_{T(\vec{x})\Sigma,\mathcal{D}} \equiv \bigcirc_{i=1}^{n}(\gamma_{TU_i(\vec{x}_i)\Sigma_i} \circ \Gamma_{\mathcal{D},i})$$

where $\Sigma_i = \Pi_{\vec{x}_i}\Sigma$ and $\Gamma_{\mathcal{D},i}$ is the *rule processing* relation, formally defined in Section 5.

We anticipate from next section that the rule processing relation $\Gamma_{\mathcal{D}_i}$ contains pairs of states $\langle bs, as\rangle$, in which as is the final state of a finite chain of states, each produced by the execution of an active rule, starting from bs.

Relation $\Gamma_{T(\vec{x})\Sigma,\mathcal{D}}$ contains all the tuples $\langle s_{or}, s\rangle$ such that s is a final state of the computation of $T(\vec{x})\Sigma$, starting from s_{or}. If any of the steps that constitute $\Gamma_{T(\vec{x})\Sigma,\mathcal{D}}$ either fails or does not terminate, the computation of the transaction is undefined and relation $\Gamma_{T(\vec{x})\Sigma,\mathcal{D}}$ contains no tuples corresponding to the original state s_{or}.

Given a state s_{or} of the active database and a transaction T, we call *rule starting point* of the computation of T starting from s_{or} a state s such that, for some i, $s \in \Pi_{as}(\gamma_{\mathrm{TU}_1(\vec{x}_1)\Sigma_1} \circ \Gamma_{\mathcal{D},1} \circ \ldots \circ \gamma_{\mathrm{TU}_i(\vec{x}_i)\Sigma_i} \circ \Gamma_{\mathcal{D},i})$, i.e., s is one of the after states produced by a transaction unit TU_i.

The execution of a transaction comprising a sequence of n transaction units proceeds as follows: the first transaction unit TU_1 is executed, with the original state s_{or} as before state. If the execution of the transaction unit does not fail, the first rule starting point is reached, from which rule processing is started. Rule execution may lead to a state in which no other rule is applicable; we call such a state *quiescent*. If a quiescent state is reached, the second transaction unit is executed on such state. Thus, the successful execution of a transaction comprising n transaction units goes through n rule starting points, each corresponding to the after state of one transaction unit. The computation proceeds in this way, by alternating transaction unit execution and reactive processing, until the final transaction unit is executed. Normally, this is a *commit* statement, which provokes events of type *commit* to be added to the eventbase, which in turn ensure that also *deferred* rules can be triggered in the final activation of the rules (see Section 3.4). Anyway, after the last transaction unit, rule processing is awakened for the last time and the quiescent state eventually reached is the after state of the transaction. This coincides with the original state, if the eventbase contains an active occurrence of the *rollback* event.

The reason for disallowing the passage of variable bindings from one transaction unit to another is that the execution semantics of transactions consisting of multiple units is such that rule processing is activated between the execution of two subsequent units; thus, variable bindings computed by a former unit may no longer correspond to the content of the database, when used by a subsequent unit.

5 ACTIVE RULE EXECUTION SEMANTICS

Active rule semantics can be thought of as the representation of the execution of a forward chaining production rule system, which in turn is very similar to the bottom-up computation of a logic program in a deductive database.

However, the case of active rules is complicated by the notion of **time**, which is totally

absent from the context of traditional production rule systems (such as OPS5), or deductive systems. Time arises in active databases mainly in two contexts: in the treatment of events, and in the awakening of rule computation. If we want to represent active rule semantics by means of some kind of fixpoint, the computation of such fixpoint, and the involved data structures, we have to take time into account.

We propose a data structure, called *tracepoint*, which can be regarded as an extension of the notion of state of the active database, with the purpose of capturing the additional information needed to represent transaction execution semantics.

Definition 13: A *tracepoint* is a quadruple of the form: $\langle s_{or}, s, \tau, \eta \rangle$, where

- $s_{or} = (DB_{or}, EB_{or})$ is the original state.
- $s = (DB, EB)$ is the current state.
- τ is a set of pairs $\{\langle R, \Theta_1 \rangle\}$, where R is an active rule and Θ_1 is a set of non-empty ground substitutions for the variables $\vec{y}_1$ in the event part of R; we call τ the *triggering set* (also called *conflict set*), because it contains all the rules triggered by a given eventbase instance, along with the possible variable bindings computed by the triggering process.
- η is a pair $\langle R, \Theta_3 \rangle$, where R is an active rule and Θ_3 is a set of non-empty ground substitutions for the variables $\vec{x}_3$ in the action part of R. We call η the *fire set*, because it contains the rule R to be fired and the variable substitutions that actually give the ground instantiations of the action part needed to fire such a rule.

5.1 Elementary Production Step

The basic step of the computation of an active database system is called *Elementary Production Step (EPS)* and is modeled by a binary relation $EPS_{\mathcal{D}}$, analogous to the relation defined by the *Immediate Consequence Operator* used in the fixpoint semantics of deductive databases.

Given an active database $\mathcal{D}$, $EPS_{\mathcal{D}}$ associates an input tracepoint to a (possibly empty) set of tracepoints, which represent all the possible states produced by the process of firing a single rule and executing all the rules possibly triggered by it. A possible execution trace of the reactive system is represented in $EPS_{\mathcal{D}}$ as an ordered set of tuples such that the second value of one tuple equals the first value of the subsequent one. Then, a terminating computation is represented by a finite such chain, whose last element has no possible successors in $EPS_{\mathcal{D}}$. This is formally captured by the notion of *trace fixpoint*, defined below.

Definition 14: A tracepoint $\langle s_{or}, s, \tau, \eta \rangle$ is said to be a *trace fixpoint* for $\mathcal{D}$ if $\nexists \langle s_{or}, s', \tau', \eta' \rangle$ such that $\langle \langle s_{or}, s, \tau, \eta \rangle, \langle s_{or}, s', \tau', \eta' \rangle \rangle \in EPS_{\mathcal{D}}$.

Thus, no further rule execution is possible, when the rule processing system enters a state corresponding to a trace fixpoint. Based on $EPS_{\mathcal{D}}$, the semantics of the execution of a set of rules is given by the *rule processing* relation, defined below.

Definition 15: Given an active database $\mathcal{D}$, the *rule processing* relation $\Gamma_{\mathcal{D}}$ is the binary relation defined as follows:

$$\langle bs, as\rangle \in \Gamma_{\mathcal{D}} \Leftrightarrow \exists\, \tau, \tau', \eta, \eta' \,(\, \langle \langle s_{or}, bs, \tau, \eta\rangle, \langle s_{or}, as, \tau', \eta'\rangle \,\rangle \in \overline{EPS_{\mathcal{D}}} \wedge \langle s_{or}, as, \tau', \eta'\rangle \text{ is a trace fixpoint of } EPS_{\mathcal{D}} \,)$$

where $\overline{EPS_{\mathcal{D}}}$ is the transitive closure of $EPS_{\mathcal{D}}$.

Thus, $\Gamma_{\mathcal{D}}$ contains tuples of the form $\langle \overline{bs}, \overline{as}\rangle$, in which $\overline{as}$ is the final state of a finite chain of tracepoints starting from $\overline{bs}$. As anticipated in the previous section, we call $\overline{as}$ a *quiescent* state to emphasize that reactive processing comes to an end when the state $\overline{as}$ is reached. If rule processing, started in state $\overline{bs}$, does not terminate or produces a failure, then relation $\Gamma_{\mathcal{D}}$ contains no tuples of the form $\langle \overline{bs}, \overline{as}\rangle$.

The elementary production step is the sequential execution of three steps, called *triggering, consideration* and *execution.* Correspondingly, the relation that describes it is obtained by sequentially composing ($\circ$) three relations, each describing a single step and by taking only distinct tracepoint pairs:

Definition 16: Given an active database $\mathcal{D}$, $EPS_{\mathcal{D}}$ is a binary relation defined as follows:

$$\langle\langle s_{or}, s_i, \tau_i, \eta_i\rangle, \langle s_{or}, s_j, \tau_j, \eta_j\rangle\rangle \in \mathrm{EPS}_{\mathcal{D}} \Leftrightarrow$$
$$\langle\langle s_{or}, s_i, \tau_i, \eta_i\rangle, \langle s_{or}, s_j, \tau_j, \eta_j\rangle\rangle \in TS_{\mathcal{D}} \circ CS_{\mathcal{D}} \circ ES_{\mathcal{D}} \wedge (s_j \neq s_i \vee \tau_j \neq \tau_i \vee \eta_i \neq \eta_j)$$

where $TS_{\mathcal{D}}$, $CS_{\mathcal{D}}$ and $ES_{\mathcal{D}}$ are the binary relations defined in Section 5.2, 5.3 and 5.4, respectively.

5.2 Rule Triggering

Rule triggering is the process of matching the active events stored in the eventbase with the event part of rules, to see which rules are activated. Rule triggering is done by performing a query on the eventbase for each rule.

The semantics of rule triggering is represented by the binary relation $TS_{\mathcal{D}}$ defined below.

Definition 17: Given an active database $\mathcal{D}$, $TS_{\mathcal{D}}$ is a binary relation between tracepoints defined as follows:

$$\langle\, \langle s_{or}, s_i, \tau_i, \eta_i\rangle, \langle s_{or}, s_j, \tau_j, \eta_j\rangle \,\rangle \in TS_{\mathcal{D}} \Leftrightarrow$$
$$(\ \textit{rollback} \in EB_i \ \wedge\ (DB_j = DB_{or}, EB_j = \emptyset, \tau_j = \emptyset, \eta_j = \emptyset)) \vee$$
$$(\textit{rollback} \notin EB_i \ \wedge\ (\langle s_{or}, s_j, \tau_j, \eta_j\rangle \in ts_{\mathcal{D}}(\langle s_{or}, s_i, \tau_i, \eta_i\rangle)))\|$$

According to this definition, relation $TS_{\mathcal{D}}$ contains a tuple $\langle\langle s_{or}, s_i, \tau_i, \eta_i\rangle, \langle s_{or}, s_j, \tau_j, \eta_j\rangle\rangle$ if s_i does not contain an active occurrence of the *rollback* event and $\langle s_{or}, s_j, \tau_j, \eta_j\rangle$ is a

‖The notation *"rollback"* $\in EB_i$ is a shorthand for the statement: " the query $\exists x, y, z$ *event(x, "rollback", -, y, z)* is satisfied in EB_i". An analogous definition is assumed for the statement *"rollback"* $\notin EB_i$.

```
INPUT: an active database D = ⟨DB, EB, R⟩; a tracepoint ⟨s_or, s, τ, η⟩;
OUTPUT: a new tracepoint.          (* R the set of Core rules *)
BEGIN
  τ := ∅;
  FORALL R IN R DO
    Θ1 := Π3 σ_{bs=s} γ_{ebq(ȳ1)};    (*Evaluate event query*)
    IF Θ1 ≠ ∅ THEN
      FORALL θ1 IN Θ1 DO
        τ := τ ∪ ⟨R, θ1⟩
      END (* FORALL*)
    END (*IF*)
  END (* FORALL*)
  RETURN ⟨s_or, s, τ, η⟩
END.
```

Figure 2 The Triggering Operator $ts_{\mathcal{D}}$.

possible output of an operator $ts_{\mathcal{D}}$, applied to $\langle s_{or}, s_i, \tau_i, \eta_i \rangle$. Tracepoints corresponding to eventbase states containing an active occurrence of the event *rollback* are treated in a special way: the subsequent tracepoint is uniquely defined as the one having the current DB state equal to the original one and empty EB instance (the EB is empty after a *rollback* command since we are considering the case of single transaction), set of triggered rules and fire set. As shown in Fraternali and Tanca (1993), this ensures the termination of the computation.

In $ts_{\mathcal{D}}$, the event query formulas of all rules are synchronously evaluated on the input EB instance. The outcome of such evaluation is then used to determine the new triggering set.

Figure 2 shows the computation of τ; the evaluation of the event part of a rule determines a set Θ_1 of substitutions, one for every different combination of events that triggers a rule. ** Then, if $\Theta_1 \neq \emptyset$, a pair $\langle R, \Theta_1 \rangle$ is added to the triggering set of every $\theta_1 \in \Theta_1$. Note that, if *ebq* is a closed formula, then $\Theta_1 = \{\iota\}$, i.e. the *void substitution* (which is a substitution with 0 arity), if *ebq* is true; otherwise, $\Theta_1 = \emptyset$.

5.3 Rule Consideration

After the set of triggered rules has been determined, rule consideration evaluates the condition part of triggered rules to select one for which the condition expressed by $db/ebq(\vec{x}_2, \vec{y}_2)$ is satisfied. Such a rule will be executed in the current run of the Elementary Production Step. In addition, rule consideration performs event consumption, if required.

The semantics of rule consideration is represented by the binary relation $CS_{\mathcal{D}}$ defined below.

**Θ_1 is obtained as the projection on the third argument of those tuples of relation $\gamma_{ebq(\vec{y}_1)}$ (the result of the event query), whose before state is equal to the input state of the algorithm.

Definition 18: Given an active database $\mathcal{D}$, $CS_{\mathcal{D}}$ is a binary relation between tracepoints defined as follows:

$$\langle \langle s_{or}, s_i, \tau_i, \eta_i \rangle, \langle s_{or}, s_j, \tau_j, \eta_j \rangle \rangle \in CS_{\mathcal{D}} \Leftrightarrow$$
$$(\tau_i (= \tau_j) = \emptyset \wedge s_j = s_i \wedge \eta_j = \emptyset) \vee$$
$$(\tau_i \neq \emptyset \wedge \langle s_{or}, s_j, \tau_j, \eta_j \rangle \in cs_{\mathcal{D}}(\langle s_{or}, s_i, \tau_i, \eta_i \rangle))$$

According to the above definition, $CS_{\mathcal{D}}$ associates to an input tracepoint with empty triggering set an output tracepoint with empty fire set and the same state. Otherwise, $CS_{\mathcal{D}}$ contains the tuples computed by applying an operator $cs_{\mathcal{D}}$, illustrated in Fig 3.

The operator $cs_{\mathcal{D}}$ consists of a cycle whose goal is to choose one rule to be executed. The selection is performed by picking one triggered rule and by testing for the satisfaction of its database query formula $db/ebq(\vec{x}_2, \vec{y}_2)$. The picking is done by a procedure *Choose*, which selects non-deterministically one of the triggered rules. [††] Each considered rule is removed from the set of triggered rules, independently of the outcome of condition evaluation.

For every picked rule, consideration is done by evaluating the ground query obtained by applying the variable bindings θ_1, if it exists ($\theta_1 \neq \iota$), to $db/ebq(\vec{x}_2, \vec{y}_2)$. The bindings computed by such a query are represented by the set of variable substitutions Θ_2.

The outcome of condition evaluation is the *fire set*, i.e., the set of ground substitutions Θ_3 that satisfy both R's event and the condition formula.

After the evaluation of the condition, event consumption is carried out, if required in the Core rule, by executing the eventbase update $ebu(\vec{x}_2)$ with the binding θ_1, if it exists. This ensures that the right triggering events are handled, in a way that complies with the rule consumption mode and granularity.

The consideration cycle ends when either all triggered rules have been considered without success or a rule is found whose condition holds, i.e., the fire set is non-empty.

5.4 Rule Execution

When the consideration step produces a non-empty fire set, the selected rule is executed by performing the sequence of update blocks in its action side. In analogy with the case of transactions, after the execution of each update block, rule processing is awakened, by recursively applying the Elementary Production Step[‡‡]. Conversely, if the consideration step produces an empty fire set, rule execution trivially returns the input tracepoint as output. Such a behavior is captured by the following definition.

Definition 19: Given an active database $\mathcal{D}$, $ES_{\mathcal{D}}$ is a binary relation defined as follows:

[††] Procedure *Choose* may be based on rule's priorities expressed in the Core rules, e.g., it may non-deterministically select one of the triggered rules with highest priority.

[‡‡] However, differently from the case of transactions, only recursive immediate rules must be awakened after the execution of an update block in the action side of a rule. In Section 3.4, we have illustrated a technique for the translation of EECA rules that achieves this goal.

INPUT: an active database $\mathcal{D} = \langle DB, EB, \mathcal{R} \rangle$; a tracepoint $\langle s_{or}, s, \tau, \eta \rangle$;
OUTPUT: a new tracepoint.
BEGIN
  $\eta := \emptyset$;
  REPEAT
    $\langle R, \theta_1 \rangle$:=*Choose*(τ); $\tau := \tau - \langle R, \theta_1 \rangle$; (* Pick next candidate *)
    WITH R DO
      $\Theta_2 := \Pi_3 \sigma_{bs=s} \gamma_{dbq/ebq(\vec{x}_2,\vec{y}_2)\theta_1}$; (* Evaluate condition *)
      $\Theta_3 := \Pi_{\vec{x}_3}(\{\theta_1\} \times \Theta_2)$; (* Compute bindings for action side *)
      $s := \Pi_2 \sigma_{bs=s} \gamma_{ebu(\vec{x}_2)\theta_1}$; (* Consume events, if required *)
    END (*WITH*)
  UNTIL $\Theta_3 \neq \emptyset \vee \tau = \emptyset$ (* Rule found or no more candidates *)
  IF $\Theta_3 \neq \emptyset$ THEN $\eta := \langle R, \Theta_3 \rangle$ END (*IF*)
  RETURN $\langle s_{or}, s, \tau, \eta \rangle$
END.

Figure 3 The Rule Consideration Step.

$$\langle \langle s_{or}, s_i, \tau_i, \eta_i \rangle, \langle s_{or}, s_j, \tau_j, \eta_j \rangle \rangle \in ES_{\mathcal{D}} \Leftrightarrow$$
$$(\eta_i = \emptyset \wedge \langle s_{or}, s_j, \tau_j, \eta_j \rangle = \langle s_{or}, s_i, \tau_i, \eta_i \rangle) \vee$$
$$(\eta_i \neq \emptyset \wedge \langle \langle s_{or}, s_i, \tau_i, \eta_i \rangle, \langle s_{or}, s_j, \tau_j, \eta_j \rangle \rangle \in (\bigcirc_{i=1}^{n-1} \gamma_{TU_i(\vec{x}_{3i})\Theta_3} \circ EPS_{\mathcal{D}}) \circ \gamma_{TU_n(\vec{x}_{3n})\Theta_3})$$

where[§§]

- $\eta_i = \langle R, \Theta_3 \rangle$, i.e., R is a rule that must be executed and Θ_3 is the associated set of ground substitutions for the action side variables;
- $TU_1 \ldots TU_n$ are the update blocks that appear in the action side of R. Note that the first of these update blocks must contain the eventbase update that expresses event consumption, in those Core rules that have it in the action part.
- $\bigcirc$ represents the iterative application of the composition operator $\circ$.

According to the above definition, $ES_{\mathcal{D}}$ is the identity relation, if no triggered rule has passed the consideration step and is ready to fire and hence the fire set computed by the consideration step is empty. In this case, rule processing reaches a quiescent state. Otherwise, $ES_{\mathcal{D}}$ is a relation that represents the actual execution phase.

The action part of the selected rule is executed by applying the ground substitutions Θ_3 in the fire set. Update blocks are executed in the order in which they appear in the rule action. Action execution proceeds in this way, by alternating update block execution and rule processing, until the final block is executed.

[§§]In the definition, with an abuse of notation, the symbol $\circ$ is used to compose relations between tracepoints with relations between states, by implicitly considering a state s equivalent to a tracepoint $\langle s_{or}, s_i, \emptyset, \emptyset \rangle$, and a tracepoint $\langle s_{or}, s_i, \tau_i, \eta_i \rangle$ equivalent to a state s_i.

Note that the final EB instance may be non-empty: in this case some events, either present in the input EB instance or generated during reactive processing, are still pending, since they have not been processed by any active rule.

6 CONCLUSIONS

This paper has presented a review of the most significant dimensions that define the behavior of rules of an active database and a semantic model of database updates and transactions in an environment where active rules can be autonomously executed by the database system during transaction processing.

The proposed model can be used to capture the semantics of rules from most existing systems, provided that such rules are at first translated into an intermediate format, called *EECA language*, and in a second time into a uniform target format, called *Core language*. This translation requires to identify the choices taken by an active database system with respect to the proposed semantic dimensions.

The semantics of Core rules is then given, together with the semantics of updates and transactions, by means of a unique execution model capable of expressing the behavior of rules featuring a wide spectrum of sophisticated primitives.

The proposed model will be used to investigate many important issues concerning active rules, like their expressive power, termination and determinism.

REFERENCES

A. Aiken, J. Widom, and J.M. Hellerstein, "Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism", Proc. ACM-SIGMOD Int. Conference, San Diego, May 1992.

"Allbase Manuals", June 1992.

Anwar, E., Maugis, L. and Chakravarthy, S. (1993) "A New Perspective on Rule Support for Object-Oriented Databases", Proc. ACM-SIGMOD Int. Conference, Washington DC, May 1993, 99-108.

Baralis, E., Ceri, S., Monteleone, G. and Paraboschi, S. (1994) "An Intelligent Database System Application: the Design of EMS", Proc. ADB '94 - International Conference on Applications of Databases, giugno 1994, Vadstena, Sweden, 172-89.

Beeri, C. and Milo, T. (1991) "A Model for Active Object Oriented Database", Proc. 17th Int. Conference on Very Large Data Bases, Barcelona, September, 1991, 337-49.

Bonner, A.J. and Kifer. M. (1993) "Transaction Logic Programming", University of Toronto, Tech. Rep. CSRI-270, revised July 1993.

Branding, H., Buchmann, A., Kudrass, T. and Zimmermann, J. (1993) "Rules in an Open System: The REACH Rule System", Proc. of First Workshop on Rules in Database Systems, WICS, Edinburgh, Scotland, Aug. 1993, Springer-Verlag, Berlin, 40-57..

Brownston, L., Farrell, R., Kant, E. and Martin, N. (1985) "Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming" Addison-Wesley, Reading, Massachusetts, 1985.

Cannan, S. and Otten, G. (1993) "SQL: The Standard Handbook", McGraw-Hill, 1993.

Ceri, S., Fraternali, P. and Paraboschi, S. (1994) "Constraint Management in Chimera", Bulletin of the Tech. Committee on Data Engineering, June 1994, Vol. 17, N. 2, 4-8.

Ceri, S., Fraternali, P., Paraboschi, S. and Tanca, L. (1994b) "Automatic generation of Production Rules for Integrity Maintenance", ACM TODS, Vol. 19, No. 3, September 1994.

Ceri, S., Fraternali, P., Paraboschi, S. and Tanca, L. (1995) "Active Rule Management in Chimera", in Widom and Ceri (1995).

Ceri, S., Fraternali, P., Paraboschi, S., and Widom, J. (1994d) "Active Database Systems", Proc. PUC-Rio DB Workshop on New Database Research Challenges, Rio de Janeiro, September 1994, 35-56.

Ceri, S. and Manthey, R. (1993) "Consolidated Specification of Chimera", ESPRIT Rep. IDEA.DE.2P.006.01, Nov. 1993.

Ceri, S. and Widom, J. (1990) "Deriving Production Rules for Constraint Maintenance", Proc. 16th Int. Conference Very Large Data Bases, Brisbane, Aug. 1990.

Ceri, S. and Widom, J. (1990b) "Deriving Production Rules for Constraint Maintenance", Technical Report, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Laboratorio di Calcolatori, 1990.

Ceri, S. and Widom, J. (1991) "Deriving Production Rules for Incremental View Maintenance", Proc. 17th Int. Conference Very Large Data Bases, Barcelona, Sept. 1991.

Chakravarthy, S., Anwar, E., Maugis, L. and Mishra, D. (1994) "Design of Sentinel: an object-oriented DBMS with event-based rules" Information and Software Technology, 1994 36, (9).

Chakravarthy, S., Blaustein, B., Buchmann, A.P., Carey, M., Dayal, U., Goldhirsch, D., Hsu, M., Jauhari, R., Ladin, R., Livni, M., McCarthy, D., McKee, R. and Rosenthal, A. (1989) "HiPAC: A Research Project in Active, Time-Constrained Database Management", Tech. Rep. XAIT-89-02, Xerox Advanced Information Technology, July 1989.

Comai, S., Fraternali, P., Psaila, G. and Tanca, L. (1995) "Semantics of Active Databases", Technical Report 014-95, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Laboratorio di Calcolatori, 1995.

Delcambre, L.M.L. and Etheredge, J.N. (1988) "The relational production language: a production language for relational databases", Proc. of the Second Int. Conf. on Expert Database Systems, pp 153-62.

Etzion, O. (1993) "Pardes- A Data-Driven Oriented Active Database Model", SIGMOD RECORD, Vol. 22, No. 1, March 1993, 7-14.

Fraternali, P., Montesi, D. and Tanca, L. (1994) "Active Database Semantics", Proc. ADC '94, Fifth Australasian Database Conference, University of Canterbury, New Zealand, 17-18th January, 1994.

Fraternali, P. and Tanca, L. (1993) "A Toolkit for the Design of Database Semantics", Technical Report 078-93, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Laboratorio di Calcolatori, 1993.

Gatziu, S., Geppert, A. and Dittrich, K.R. "Integrating Active Concepts into an Object-Oriented Database System", Proc. DBPL91 Conf., 1991, pp. 341-57.

Gehani, N.H. and Jagadish, H.V. (1991) "Ode as an Active Database: Constraints and Triggers", Proc. 17th Conference on Very Large Data Bases, Barcelona, September, 1991, pp. 327-36.

Gehani, N.H., Jagadish, H.V. and Shmueli, O. (1992) "Composite Event Specification in Active Databases: Model and Implementation", Proc. 18th Conference on Very Large Data Bases, Vancouver, British Columbia, Canada, 1992, 327-38.

Gehani, N.H., Jagadish, H.V. and Shmueli, O. (1992b) "Event Specification in an Active Object-Oriented Database", Proc. ACM SIGMOD Int. Conference, San Diego, May 1992, pp. 81-90

Gray, J. and Reuter, A. (1994) "Transaction processing: Concepts and Techniques", Morgan-Kaufmann, San Mateo, California, 1994.

Hanson, E. (1992) "Rule Condition Testing and Action Execution in Ariel", Proc. ACM-SIGMOD Int. Conference, San Diego, May 1992.

Hanson, E. and Widom, J. (1992) "An Overview of Production Rules in Database Systems",

IBM RJ 9023, October 1992, 83-119.

Kappel, G., Rausch-Schott, S. and Retschitzegger, W. (1994) "Beyond Coupling Modes: Implementing Active Concepts on Top of a Commercial ooDBMS", manuscript, 1994

"Informix Guide to SQL, Syntax", Version 6, March 1994, PartNo. 000-7597.

"Informix Guide to SQL, Tutorial", Version 6, March 1994, PartNo. 000-7598.

"Ingres Database Administrator's Guide", Release 6.3, November 1990.

"Interbase, DDL Reference Manual", 1990.

McCarthy, D. and Dayal, U. (1989) "The Architecture Of An Active Database System" Proc. ACM-SIGMOD Int. Conference, 1989.

"ORACLE7 Server Concepts Manual", Part No 6693-70, December 1992.

"ORACLE7 Server Application Developer's Guide", Part No 6695-70, December 1992.

Palopoli, L and Torlone, R. (1994) "Modeling Database Applications Using Generalized Production Rules", Proc. 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS '94), Houston, Texas, February, 1994, 30-45.

Paton, N.W., Diaz, O., Williams, M.H., Campin, J., Dinn, A. and Jaime, A. (1993) "Dimensions of Active Behaviour", Proc. of First Workshop on Rules in Database Systems, WICS, Edinburgh, Scotland, Aug. 1993, Springer-Verlag, Berlin, 40-57.

"VAX Rdb/VMS SQL Reference Manual", Digital, November 1991.

"SQL3 Document X3H2-94-080 and SOU-003", ISO-ANSI Working Draft, Database Language SQL, 1994.

Stonebraker, M., Jhingran, A., Goh, J. and Potamianos, S. (1990) "On Rules, Procedures, Caching, and Views in Data Base Systems", Proc. ACM-SIGMOD Int. Conference, Atlantic City, June 1990, 281-90.

Stonebraker, M. (1992) "The Integration of Rule Systems and Database Systems", IEEE Trans. on Knowledge an Data Engineering, Vol. 4, N. 5, Oct. 1992.

Tanca, L. (1991) "(Re)-action in deductive databases", Proc.2nd Int. Workshop on Intelligent and Cooperative Information Systems, Como, Italy, Oct. 1991.

"Transact-SQL User's Guide for Sybase", Release 10.0.

"Transact-SQL User's Guide for Sybase", Release 4.8, April 1992.

Ullman,, J.D. (1982) "Principles of database systems", Computer Science Press, Potomac, Maryland, 1982.

Urban, S.D., Karadimce, A.P. and Nannapaneni, R.B. (1992) "The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Databas", Proc. 8th Int. Conference on Data Engineering, Feb. 1992, Phoenix, Arizona, 565-72.

Widom, J. (1992) "A Denotational Semantics for the Starburst Production Rule Language", SIGMOD Record, Vol 21, N. 3, Sept. 92, 4-9.

Widom, J. and Ceri, S. (1995) "Active Database Systems", Eds., Morgan Kaufmann, San Mateo, California, to appear in August 1995.

Widom, J., Cochrane, R.J. and Lindsay, B.G. (1991) "Implementing Set-Oriented Production Rules as an Extension to Starburst", Proc. 17th Int. Conference on Very Large Data Bases, Barcelona, Sept. 1991, 275-85.

Widom, J. and Finkelstein, S.J. (1990) "Set-oriented Production Rules in Relational Database Systems", Proc. of ACM-SIGMOD, May 1990, 259-70.

Zaniolo, C. (1993) "A Unified Semantics for Deductive and Active Databases", Proc. of the 1st Int. Workshop on Rules in Database Systems, Edinburgh, August 1993.

Zhou, Y. and Hsu, M. (1990) "A theory for rule triggering systems", in Proc. of EDBT '90, LNCS 416, Springer-Verlag, Berlin, March 1990, 407-21

emp

name	age	salary	manager
John	22	2000	Frank
Fred	35	4000	Frank
Frank	29	4000	Jack
Bob	31	5000	Jack
Jack	28	5200	Tony
Tom	32	3800	Frank

a)

updated emp

name	age	salary	manager
John	22	2000	Frank
Fred	*35*	*4400*	*Frank*
Frank	29	4000	Jack
Bob	*31*	*5500*	*Jack*
Jack	28	5200	Tony
Tom	*32*	*4180*	*Frank*

b)

new updated emp

name	age	salary	manager
Fred	35	4400	Frank
Bob	31	5500	Jack
Tom	32	4180	Frank

c)

Figure 4 Starburst : a) the *emp* relation; b) The *updated emp* relation; c) the *new updated emp* relation.

APPENDIX 1 A COMPARATIVE EXAMPLE

In this section we want to show how our semantic model can be applied to a practical case in order to model the different behaviours of different systems.

In the following sections we will consider three systems, i.e. Starburst, which is a relational set-oriented database, Postgres, that is also a relational database, but its behaviour is not set-oriented, and Chimera, an object oriented and set-oriented database. The same example is adopted, to put in evidence the differences among the considered systems.

Let us now describe the example. The table in Figure 4.a is the initial state of the database. The *emp* relation contains data about employees, that is name, age, salary and manager. We want the system to compute the following transaction : select the employees older than 30 and increase their salary by 10%.

Not every new salary is valid : the company the employees belong to do not want the salary of an employee to be greater than his manager's salary. This is a constraint that can be checked by introducing a rule, which checks, for each updated employee, if his/her salary does not overtake the manager's salary, and if it is so, produces a warning towards the operator.

In our example, three tuples are affected by the transaction : those related to the employees Fred, Bob and Tom; the relation after the update is shown in Figure 4.b . Notice that all the three modified tuples violate the constraint. Thus, the active mechanism must put the rule in execution.

APPENDIX 1.1 Starburst

In this section we consider the Starburst version of our example. Starburst is a relational set-oriented database, based on SQL. The transaction assumes the following form :

```
begin transaction

update emp
set salary = salary*1.1
where age > 30

commit
```

Notice that the transaction is composed of one SQL update statement.
The rule *Starburst_verify_salary* is defined as follows.

```
create rule Starburst_verify_salary
when updated emp.salary
if ( select salary for new updated emp X ) >
   (select salary from emp Y where X.manager = Y.name )
then print( "employees ", X.name, " earn too much" )
```

This rule does not modify the *emp* table; it only prints a message alerting the operator that some employees do not respect the constraint.

In Starburst every rule execution is delayed to the transaction end and the system is also set-oriented; that means that in our example rule execution starts as an effect of the *commit* instruction; at this time the relation is up to date already. For this reason, the rule can see as instance of the so called *new updated emp* relation the relation in Figure 4.c . Such a relation contains only the tuples modified by the transaction. The rule checks the constraint joining the new updated *emp* and the whole *emp* relation (up to date).

Remember that Starburst is set-oriented, so the rule is executed only once, and only one message is printed.

```
employees Fred Bob Tony earn too much.
```

Let us start now modeling the behaviour according to our update model.

$$
\begin{aligned}
\gamma_{\text{update emp}} &= \{\langle bs, bs + \psi_{Fred}, \iota\rangle, \\
&\quad \langle bs, bs + \psi_{Bob}, \iota\rangle, \\
&\quad \langle bs, bs + \psi_{Tom}, \iota\rangle\} \\
\gamma_{TU()\Sigma} &= \{\langle s_0, s_0 + \psi_{Fred} + \psi_{Bob} + \psi_{Tom}\rangle\} \\
\gamma_{\text{commit}} &= \{\langle s_1, s_1 + \psi_{commit}\rangle\} \\
\Gamma_{\mathcal{D}} &= \{\langle s_{commit}, s_{commit} + \psi_{EB}\rangle\}
\end{aligned}
$$

At first, we provide the semantics of the update instruction, by means of the relation $\gamma_{\text{update emp}}$. This relation contains three tuples, one for each tuple update performed by

the system. Notice that ψ_{Fred} is the f.o. formula which asserts that the new state differs from the previous state for the new value of attribute *salary* in Fred's tuple (the reasoning is similar for ψ_{Bob} and ψ_{Tom}). In Starburst, operations do not provide output variables, so the third field of the relations is always the void substitution ι.

We use *bs* to indicate a generic before state which is valid for the application of the update indicated by the f.o. formula. The set oriented execution produces a result which is obtained as the composition of the three single updates.

In our example the transaction unit is formed only by the update instruction, then the transaction unit is modeled by the provided relation $\gamma_{TU()\Sigma}$. Notice the use of the state s_0, that indicates the actual initial state of the transaction.

After the execution of the transaction unit, the *commit* instruction is typed. The *commit* instruction can be considered as a transaction unit, because it modifies the EB generating *commit* events; such events allow delayed rules to be executed. The *commit* transaction unit is described in the relation γ_{commit}; the notation s_1 stands for the state produced by the update transaction unit, while the f.o. formula ψ_{commit} indicates that the EB is changed as a consequence of the *commit* instruction.

After the execution of the *commit* instruction, the triggering mechanism starts. The execution of the rule is modeled by relation $\Gamma_{\mathcal{D}}$; note that the f.o. formula ψ_{EB} indicates that only the EB has been modified by the rule execution; in fact the rule has consumed the triggering events, but has not produced new events. Notice the use of the state s_{commit}, that indicates the state after the *commit* transaction unit execution.

Finally the $\Gamma_{T()\Sigma,\mathcal{D}}$ modeling the whole transaction is obtained by composing the three previous relations. Looking at the composition formula is immediatly comprehensible that the active mechanism is activated only once at the end of the transaction. Also notice that the tuple indicates the initial state of the transaction and the state obtained at the end of the transaction.

$$\begin{aligned}\Gamma_{T()\Sigma,\mathcal{D}} &= \gamma_{TU()\Sigma} \circ \gamma_{\text{commit}} \circ \Gamma_{\mathcal{D}} = \\ &= \{\langle s_0, s_0 + \psi_{Fred} + \psi_{Bob} + \psi_{Tom} + \psi_{commit} + \psi_{EB}\rangle\}\end{aligned}$$

APPENDIX 1.2 Postgres

Postgres is based on the QUEL language, but the aspect of the transaction is not so different w.r.t. the SQL transaction of Starburst; notice the different style used for relation names (all capital letters).

```
begin transaction

replace EMP ( salary = salary * 1.1 )
where EMP.age > 30

commit
```

Moreover, the rule *Postgres_verify_salary* is different, because it refers explicitly to the modified tuple (with the keyword NEW) or to the state of the tuple prior to the modification (with the keyword CURRENT).

EMP (1)

name	age	salary	manager
John	22	2000	Frank
Fred	*35*	*4400*	*Frank*
Frank	29	4000	Jack
Bob	31	5000	Jack
Jack	28	5200	Tony
Tom	32	3800	Frank

EMP (2)

name	age	salary	manager
John	22	2000	Frank
Fred	35	4400	Frank
Frank	29	4000	Jack
Bob	*31*	*5500*	*Jack*
Jack	28	5200	Tony
Tom	32	3800	Frank

EMP (3)

name	age	salary	manager
John	22	2000	Frank
Fred	35	4400	Frank
Frank	29	4000	Jack
Bob	31	5500	Jack
Jack	28	5200	Tony
Tom	*32*	*4180*	*Frank*

Figure 5 Postgres : (1) the *EMP* relation after the first tuple update; (2) the *EMP* relation after the second tuple update; (3) the *EMP* relation after the third and final tuple update.

```
define rule Postgres_verify_salary
on replace to EMP.salary
where NEW.salary > EMP.salary and
      EMP.name = NEW.manager
then do
   print ( "employee ", NEW.name, " earns too much." )
```

In fact, although relational, Postgres is not set-oriented at all, because the triggering mechanism starts after every tuple update and the rule can see only the tuple updated immediately before its execution.

$$\gamma_{\text{replace Fred}} = \{\langle bs, bs + \psi_{Fred}, \iota\rangle\} \qquad \gamma_{TU_1()\Sigma} = \{\langle s_0, s_0 + \psi_{Fred}\rangle\}$$
$$\gamma_{\text{replace Bob}} = \{\langle bs, bs + \psi_{Bob}, \iota\rangle\} \qquad \gamma_{TU_2()\Sigma} = \{\langle s'_1, s'_1 + \psi_{Bob}\rangle\}$$
$$\gamma_{\text{replace Tom}} = \{\langle bs, bs + \psi_{Tom}, \iota\rangle\} \qquad \gamma_{TU_3()\Sigma} = \{\langle s'_2, s'_2 + \psi_{Tom}\rangle\}$$
$$\Gamma_{\mathcal{D}_1} = \{\langle s_1, s_1 + \psi_{EB_1}\rangle\}$$
$$\Gamma_{\mathcal{D}_2} = \{\langle s_2, s_2 + \psi_{EB_2}\rangle\}$$
$$\Gamma_{\mathcal{D}_3} = \{\langle s_3, s_3 + \psi_{EB_3}\rangle\}$$

In our model the triggering mechanism can start only between a transaction unit and the following one, then every Postgres update is modeled as a separate operation corresponding to a transaction unit. The states of the *EMP* relation after each of the three transaction units are represented in Figure 5.a, Figure 5.b and Figure 5.c respectively.

Then we provide the semantics of the three updates, by means of the three distinct relations $\gamma_{\text{replace Fred}}$, $\gamma_{\text{replace Bob}}$ and $\gamma_{\text{replace Tom}}$. Notice again the use of the generic *bs* state; it indicates every state which is valid for the application of the f.o. formula ψ_{Fred} (or ψ_{Bob} or ψ_{Tom}). Also in Postgres, operations do not provide output variables, so the third field of the relations is always the void substitution ι.

As each single tuple update corresponds to a transaction unit, applying them to three particular states the following relations describing the three different transaction units are obtained. Notice that here we have chosen one of the possible orderings of execution of the three tuple updates. This is non-deterministic in Postgres (probably based on the physical order of the tuples in the actual database?). Notice also the use of the three states s_0 (which indicates the initial state of the transaction), s'_1 (that indicates the state after the first execution of the rule), and s'_2 (which indicates the state after the second execution of the rule).

The three rule executions are modeled by the three relations $\Gamma_{\mathcal{D}_1}$, $\Gamma_{\mathcal{D}_2}$ and $\Gamma_{\mathcal{D}_3}$. Again notice the use of the three states s_1 (the state after the execution of the first transaction unit), s_2 (the state after the execution of the second transaction unit), and s_3 (the state after the execution of the third transaction unit).

Finally, the *commit* command is typed. The *commit* instruction can be considered as a transaction unit, because it modifies the EB.

The *commit* transaction unit is described in the relation γ_{commit}; the notation s'_3 stands for the state produced by the third execution of the triggering mechanism, while the f.o. formula ψ_{commit} indicates that the EB is changed as a consequence of the *commit* instruction.

$$
\begin{array}{lcl}
\gamma_{\text{commit}} & = & \{\langle s'_3, s'_3 + \psi_{commit}\rangle\} \\
\Gamma_{T()\Sigma,\mathcal{D}} & = & \gamma_{TU_1()\Sigma} \circ \Gamma_{\mathcal{D}_1} \circ \gamma_{TU_2()\Sigma} \circ \Gamma_{\mathcal{D}_2} \circ \gamma_{TU_3()\Sigma} \circ \Gamma_{\mathcal{D}_3} \circ \gamma_{\text{commit}} = \\
 & = & \{\langle s_0, s_0 + \psi_{Fred} + \psi_{EB_1} + \psi_{Bob} + \psi_{EB_2} + \psi_{Tom} + \psi_{EB_3} + \psi_{commit}\rangle\}
\end{array}
$$

Then the relation $\Gamma_{T()\Sigma,\mathcal{D}}$ describing the whole transaction is obtained by composing the seven previous relations in the correct order. Notice that looking at the composition formula it is immediate to see that the triggering mechanism is executed three times, that is after each transaction unit, each corresponding in our model to one tuple update in Postgres.

As a result of the execution, we show the three different messages that are printed, one for each rule execution.

```
employee Fred earns too much.
employee Bob earns too much.
employee Jack earns too much.
```

Again, we want to stress that the result is strongly based on the order in which tuples are stored in the relation *EMP*. If tuples were stored in another order (not necessarily in this example) the final result could be different.

APPENDIX 1.3 Chimera Version

Chimera is an Object Oriented, Active and Deductive database. As it is object oriented, we talk of the *emp* class, and each object in the database is identified by a unique OID (object identifier); the class *emp* is shown in Figure 6.a . The Chimera programming language is completely different from the other two ones considered here, because it is based on the first order logic. Moreover the instructions are specialized, i.e. there is an instruction for the selection and an instruction for the updates, and so on. For these reasons, the aspect of the transaction is completely different from the previous ones.

```
begin transaction;

select(X where emp( X ), X.age > 30), modify(emp.salary, X, salary*1.1);

commit;
```

In Chimera a single execution unit is called *transaction line.* Variables can be passed from an instruction to the following instructions in the same transaction line. Variables exist only in the transaction line.

Then, in our example the transaction contains one transaction line, composed by the selection of the objects we want to update and the update of these objects; the variable X is used to pass the objects from the first to the second operation. The state of the *emp* class after the execution of the transaction line is in Figure 6.b .

Let us now show the rule *Chimera_verify_salary.* Notice that the condition is expressed by means of a formula. In this formula the predicate *occurred* binds the variable X to the objects affected by the specified event. Since the trigger is declared as consuming, the predicate *occurred* returns the objects not considered by previous executions of the rule.

```
define immediate consuming trigger Chimera_verify_salary
events   modify( emp.salary )
condition
   occurred( modify( emp.salary ), X ),
   emp( M ), Y.manager = M.name, Y.salary > M.salary
actions
   display( Y ), print(" earn too much.")
end;
```

$$
\begin{aligned}
\gamma_{\text{modify(emp.salary, X, salary*1.1)}} &= \{\langle bs, bs + \psi(X), \iota\rangle\} \\
\gamma_{\text{select(X)}} &= \begin{array}{l}\{\langle bs, bs, \{X/45\}\rangle, \\ \langle bs, bs, \{X/23\}\rangle, \\ \langle bs, bs, \{X/694\}\rangle\}\end{array} \\
\gamma_{TU()\Sigma} &= \{\langle s_0, s_0 + \psi(45) + \psi(23) + \psi(694)\rangle\} \\
\Gamma_{\mathcal{D}} &= \{\langle s_1, s_1 + \psi_{EB}\rangle\}
\end{aligned}
$$

emp

OID	name	age	salary	manager
3	John	22	2000	Frank
45	Fred	35	4000	Frank
91	Frank	29	4000	Jack
23	Bob	31	5000	Jack
1042	Jack	28	5200	Tony
694	Tom	32	3800	Frank

a)

emp

OID	name	age	salary	manager
3	John	22	2000	Frank
45	*Fred*	*35*	*4400*	*Frank*
91	Frank	29	4000	Jack
23	*Bob*	*31*	*5500*	*Jack*
1042	Jack	28	5200	Tony
694	*Tom*	*32*	*4180*	*Frank*

b)

Figure 6 a) The Chimera *emp* class; b) The Chimera *emp* class after the transaction unit execution.

The semantics of the *select* instruction is provided by the relation $\gamma_{\text{select(X)}}$. The selection creates bindings for the output variable X, then the third field in the relation is not the void substitution but a substitution for X.

Also notice that the selection operation does not change the state of the database.

Moreover, the semantics of the *modify* instruction is provided. Notice that the we use a parametric f.o. formula ψ, which indicates that object with OID specified by X is updated.

During the execution of the transaction line, the OIDs bounded to X are passed to the modify instruction. A transaction line corresponds to a transaction unit in our model, and the execution of the transaction unit is described by the next relation. Notice the use of the state s_0, which indicates the initial state of the transaction.

Chimera is truly set-oriented, then the triggering mechanism starts only after the transaction unit has been executed. The relation $\Gamma_{\mathcal{D}}$ shows the semantics of the triggering execution, applied to the state s_1, that is the state after the execution of the transaction unit. With ψ_{EB} we indicate that only the EB is modified by the execution of the rule *Chimera_verify_salary*.

After the execution of the transaction unit and the consequent rule execution, the *commit* instruction is typed. The *commit* instruction can be considered as a transaction unit, because it modifies the EB generating *commit* events; such events allow delayed rules to be executed (delayed rules are allowed in Chimera and are called *deferred* rules, but in our example we do not use them). The *commit* transaction unit is described in the relation γ_{commit}; the notation s_1' stands for the state produced by the update transaction unit, while the f.o. formula ψ_{commit} indicates that the EB is changed as a consequence of the *commit* instruction.

$$\gamma_{\text{commit}} = \{\langle s_1', s_1' + \psi_{commit}\rangle\}$$

Finally, the execution semantics of the whole transaction is obtained by composing the three previous relations.

$$\begin{aligned}\Gamma_{T()\Sigma,\mathcal{D}} &= \gamma_{TU()\Sigma} \circ \Gamma_{\mathcal{D}} \circ \gamma_{commit} = \\ &= \{\langle s_0, s_0 + \psi(45) + \psi(23) + \psi(694) + \psi_{EB} + \psi_{commit}\rangle\}\end{aligned}$$

Chimera is set-oriented, then the rule produces only one message, because it has been executed only once.

```
| Fred |
| Bob  |
| Tom  |
--------    earn too much.
```

Questions & answers

Question []:

If an action is a decoupled transaction, a visible, inconsistent state is possible.

Answer [G. Psaila]:

This is beyond the scope of our paper.