ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

A Framework for the Specification of Active Rule Language Semantics

Danilo Montesi and Riccardo Torlone

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Framework for the Specification of Active Rule Language Semantics

Danilo Montesi*

Imperial College, Department of Computing
180 Queen's Gate – London SW7 2BZ, UK
Email: d.montesi@doc.ic.ac.uk

Riccardo Torlone

Università di Roma Tre, c/o IASI-CNR
Viale Manzoni, 30 – 00185 Roma, Italy.
Email: torlone@iasi.rm.cnr.it

Abstract

We present a formal framework that can be used to specify and study a number of different semantics for rule execution in active databases. We shall consider the core of several active rule languages that are already available (e.g., Ariel, Starburst and HiPAC) but whose rule execution is specified only by informal descriptions. The framework is based on a generic active rule language and relies on a transaction rewriting technique. This technique takes a user defined transaction, which is viewed as a sequence of basic database updates forming a semantic unit, and translates it into a new transaction that explicitly includes the additional updates due to active rule triggering. We show that this framework provides a basis for the theoretical analysis and the comparison of different execution models of active rules. Moreover, it allows us to formally investigate a number of important issues related to active rule processing, such as transaction equivalence, confluence and optimization, independently of a specific rule execution model.

1 Introduction

Active rules allow the user to specify actions to be taken automatically, when certain events occur and some conditions are met. It is widely recognized now that active rule processing is a powerful mechanism for the management of several important database activities (e.g., constraint maintenance and view materialization [3, 4]), and for this reason, active databases have been extensively investigated and experimented in the last years (see [10, 13] for extensive bibliographies). A number of active database systems are now already available: among the others we recall *Ariel* [9], *Starburst* [15], *Postgres* [14] and *HiPAC* [5, 11]. Indeed, each of these systems uses a different semantics for rule application. The main differences stem from the choice of when rules should be fired (e.g., within the original transaction or at the end of the transaction or in another transaction), how they should be fired (e.g., in parallel or sequentially or according to a fixed rule order), and the granularity of the computation (e.g., set-oriented or tuple-oriented). Other differences concern the complexity of specifiable events, conditions and actions as well as the binding mechanisms (if any) between the various components of an active rule. However, a crucial point is that in the various approaches, active rule execution is generally specified only by informal, natural-language descriptions. It follows that very often, when the number of rules increases, active rule processing becomes quickly complex and unpredictable [10].

In this paper we present a formal framework, based on a new approach to active rule processing [12], that allows us to formally describe a variety of different active rule semantics. We show that this framework can be used as a basis for the theoretical analysis and the comparison of different execution models for active rules. Moreover, it allows us to formally investigate a number of important issues related to active rule processing, such as transaction equivalence, confluence and optimization, independently of a specific rule execution model.

The framework is based on a rewriting technique, to be performed at compile time, that takes as input a user defined transaction T and a set of active rules expressed in a generic language, and produces a new transaction T' that “embodies” active rule semantics, in that T' explicitly includes the additional updates due to active processing. It follows that the execution of the new transaction in a passive environment corresponds to the execution of the original

*The work of this author has been partially supported by the EU *Human Capital and Mobility* grant N. ERBCHBGCT930365 “Compulog-group”.

transaction within the active environment defined by the given rules. The compilation procedure has two parameters that allows us to specify: (1) how the triggered rules are interleaved with updates of the user's transactions, and (2) the order in which triggered rules are selected for execution. We show that the active rule execution strategies of most active database systems can be described by just adjusting these two parameters.

Other approaches consider rewriting techniques [8, 14], but usually they apply in a restrictive context or are not formal. Conversely, we believe that this formal and simple approach can improve the understanding of several active concepts and make it easier to show results. In fact, the execution model of our transactions is based on a relational transaction model which has been extensively investigated [1]. This allows us to use known results for transaction equivalence for a formal investigation of important properties of active rule processing. First, we can check whether two transactions are equivalent in an active database. Then, due to the results on transaction equivalence, we are also able to provide results on confluence [2]. Finally, optimization issues can be addressed. We point out that with this approach, the various results on active rule processing are independent of the specific rule execution model. As a final remark, we note the proposed technique does not require any specific run-time support, and therefore it can be easily implemented on the top of a traditional database management system.

The rest of this paper is organized as follows. In Section 2 we define the basic framework that refers to a generic active database language. In Section 3 we show how this framework can be used to specify a number of active rule language semantics. In Section 4 we report on general results that can be derived for active rule processing. Finally, in Section 5, we summarize some conclusions. Because of space limitation, several technical details are omitted in this paper.

2 The basic framework

2.1 Preliminary notions

Let us fix a generic *database scheme* $\mathbf{R} = \{R_1(X_1), \dots, R_n(X_n)\}$, where the R_i 's are *relational schemes* and the X_i 's are sets of *attributes* $A_1, \dots, A_k$. We then denote by s an *instance* over $\mathbf{R}$, that is, a set of relations $s = \{r_1, \dots, r_n\}$ over $R_1, \dots, R_n$ respectively, and by $Inst(\mathbf{R})$ the set of all possible instances over $\mathbf{R}$.

A *relational atom* over a scheme $R(A_1, \dots, A_k)$ is an object of the form $R(t_1, \dots, t_k)$ where t_i , for $i = 1, \dots, k$, is a *term*, that is, a variable or a constant. A *built-in atom* is an object of the form $t_1 \phi t_2$, where t_1 and t_2 are terms and ϕ is a comparison operator. A *literal* is an atom (*positive literal*) or a negated atom (*negative literal*). A relational atom $R_i(t_1, \dots, t_k)$ without variables is true in a database instance $s = \{r_1, \dots, r_n\}$ if $R_i(t_1, \dots, t_k) \in r_i$. A *condition* is a set of literals. A *valuation* of a condition is a substitution of the variables in C with constants, that makes true all the atoms in C .

An *action* is a relational atom preceded by one of the symbols $\{+, -\}$. An *update* U has the form $A[C]$, where C is a condition and A is an action such that all the variables occurring in it also occur in C . An update is executed for those tuples (if any) that verify the specified condition. Specifically, the *effect* of an update U is a function $EFF(U) : Inst(\mathbf{R}) \rightarrow Inst(\mathbf{R})$ defined as follows:

- $EFF(+B[C])(s) = s \cup \{\theta(B) : \theta \text{ is a valuation of } C \text{ in } s\};$
- $EFF(-B[C])(s) = s - \{\theta(B) : \theta \text{ is a valuation of } C \text{ in } s\}.$

Note that, for sake of simplicity, we do not consider modify operations here.

2.2 Transactions and active rules

We define a *user transaction* as a collection of updates forming a semantic atomic unit for recovery and concurrency purposes: $T = U_1; \dots; U_k$. The effect of T over an instance s is then defined as follows:

$$EFF(T)(s) = EFF(U_1) \circ \dots \circ EFF(U_k)(s).$$

An *event* is a relational atom preceded by one of the symbols $\{\oplus, \ominus\}$. An event denotes the fact that a certain update operation has been performed on a database instance. Thus, the execution of the update $+R_i(X, Y)[X = a]$ generates the event $\oplus R_i(a, Y)$.

An *active rule* has the form:

$$L : (E_1 \vee \dots \vee E_x) \circ C \rightarrow A_1; \dots; A_y$$

where L is the label that is used to express priority, $(E_1 \vee \dots \vee E_x)$ is a disjunction of events, C is a condition and $A_1; \dots; A_y$ is a sequence of actions such that: (1) each variable that occurs in a negative literal in the condition also occurs in a positive literal or in the event part, and (2) each variable that occurs in the action also occurs in the condition part. An *active program* P is a set of active rules. An *active database* is a pair (s, P) where s is a database state and P is an active program. The above rule language includes the core of the languages used in the main active database systems (Ariel, Postgres, Starburst and HiPAC).

The intuitive semantics of the above active rule is: “if any of the events $E_1, \dots, E_x$ has occurred and there is no triggered rule with priority higher than L , evaluate the condition C and if it holds, perform the actions $A_1; \dots; A_y$ using the bindings of the event and the condition parts.” There are some important points that should be further specified: (1) the temporal relationship between the execution of the various components of a rule, (2) the parameters that can be passed from the event and the condition parts to the action part, (3) the order in which the rules are fired when a conflict occurs. The various systems may vary considerably on these points. We will show however that with the above generic language and with the transformation technique presented in the next section, we are able to express the main cases.

2.3 Transaction transformation

The key point in our approach is that the behavior of an active database with respect to a transaction $T = U_1; \dots; U_k$ is defined in terms of execution in a passive environment of a new transaction T' , induced by T , that “embodies” active rule semantics. We present now the technique that allows us to transform T in T' .

We will consider the main active rule execution models that have been proposed in the literature: immediate, deferred and decoupled. The immediate modality reflects the intuition that rules are processed as soon as they are triggered, while deferred modality suggests that a rule is evaluated and executed after the end of the original transaction [11]. Finally, the decoupled modality suggests that the triggered rules are all executed in a different transaction. According to these execution models, three different rewriting procedures will be given. Specifically, consider a user transaction as a sequence of updates:

$$T = U_1; \dots; U_k.$$

This transaction is transformed under the immediate modality into an induced one:

$$T^I = U_1; \bar{U}_1^P; \dots; U_k; \bar{U}_k^P$$

where $\bar{U}_i^P$ denotes the sequence of updates computed as *immediate reaction* of the update U_i with respect to a set of active rules P . This reaction can be derived by “unifying” the update U_i with the event part of the active rules. Clearly the obtained updates can themselves trigger other rules, hence this reaction is computed recursively. Note that under the *immediate modality* the induced transaction is an interleaving of user defined updates with rule actions.

Under the *deferred modality*, the induced transaction has the form:

$$T^{DE} = U_1; \dots; U_k; \bar{U}_1^P; \dots; \bar{U}_k^P.$$

Hence the *reaction* is *deferred* (or postponed) until the end of the user transaction. Here again the induced updates can themselves trigger other rules, and so the reactions of the original updates are recursively computed, but using the immediate modality.

Under the *decoupled modality*, there are two induced transactions. The former is:

$$T_1^{DC} = U_1; \dots; U_k$$

and corresponds to the user transaction while the reaction is decoupled to a different transaction:

$$T_2^{DC} = \bar{U}_1^P; \dots; \bar{U}_k^P.$$

Again, the reactions of the original updates are recursively computed. In the rest of this section we will formalize the above notion of reaction.

Let U be an update $A[C]$ and R be an active rule of the form

$$L : (E_1 \vee \dots \vee E_x) \circ C' \rightarrow A'_1; \dots; A'_y$$

that does not share any variable with U (this can be easily enforced by means of variable renaming). Then, we say that U *triggers* R because of E_i if:

1. there is an E_i , with $1 \leq i \leq x$, such that $E_i = \oplus B'$ and $A = +B$, or $E_i = \ominus B'$ and $A = -B$, and
2. there is a substitution θ , called *unifier*, such that $B = \theta(B')$.

If an update U triggers a rule $(E_1 \vee \dots \vee E_x) \circ C' \rightarrow A'_1; \dots; A'_y$ with θ as unifier, then we say that U *induces* the sequence of updates $\theta(A'_1[C']); \dots; \theta(A'_y[C'])$.

During translation we need to keep trace of the relationship between inducer and induced updates. This is done by subscribing the induced updates in order to encode the inducer, the inducer of the inducer and so on. For instance the update $U_{3,2,1}$ means that it was induced by the update $U_{3,2}$ that in turn was induced by U_3 . Thus the original update U_3 induces $U_{3,2}$ that induces $U_{3,2,1}$.

The following is a recursive algorithm that computes the reaction of a single update to a rule program which allows disjunctive events and sequences of actions in the rules. In the algorithm a rule $L : (E_1 \vee \dots \vee E_x) \circ C' \rightarrow A'_1; \dots; A'_y$ is simply denoted by R . Note that the algorithm is parametric with respect to the selection policy of the triggered rules that can be expressed through several SELECT functions.

Algorithm REACT

Input: An active program P and an update U_j .

Output: A sequence $\bar{U}_j^P$ of updates induced by U_j .

begin

$\bar{U}_j^P := \langle \rangle; i := 1; index := \langle j \rangle;$

$Triggered := \{ R \in P : R \text{ is triggered by } U_j \text{ because of } E_k \};$

while $Triggered$ is not empty **do**

$R := \text{SELECT}(Triggered);$

$\theta := \text{the unifier of } E_k \text{ and } U_j;$

$index := \text{append}(index, i);$

$\bar{U}_{index} := [\theta(A_1[C]); \dots; \theta(A_y[C])];$

$\bar{U}_j^P := \text{merge}(\bar{U}_j^P, \bar{U}_{index});$

$\bar{U}_j^P := \text{merge}(\bar{U}_j^P, \text{REACT}(P, \theta(A_1[C])); \dots; \text{REACT}(P, \theta(A_y[C])));$

$i := i + 1;$

$Triggered := Triggered - \{R\};$

endwhile

return $\bar{U}_j^P$

end.

In general, different outputs can be generated by the above algorithm depending on the order in which rules are selected by the SELECT function. Clearly the algorithm can be generalized in such a way that all the possible different induced sequences of updates are generated. Moreover, syntactical restriction can be given to guarantee its termination [12].

3 Active Rule Execution Semantics

In this section we describe how the framework defined in the previous section can be used to specify different proposed semantics for rule execution in active databases. In particular, we shall consider the core of the semantics of rule execution in the Ariel, Starburst and HiPAC systems.

3.1 Ariel

The basic features of Ariel are: sequences of updates in events and actions, set-oriented reaction, immediate modality and numeric priority among rules which may be not unique. This can be characterized with the following algorithm that specifies the execution model of Ariel rules.

Algorithm ARIEL

Input: An Ariel active program P and a user transaction $T = U_1; \dots; U_k$.

Output: A transaction T^A induced by T .

begin

$T^A := U_1; \text{REACT}(P, U_1); \dots; U_k; \text{REACT}(P, U_k);$
return T^A

end.

To each Ariel rule is assigned a numeric priority but the assignment need not be unique and therefore conflicts may occur. Conflict resolution in Ariel can be described by the following procedure.

Algorithm SELECT

Input: A set of triggered rules I_R .

Output: One rule in I_R .

begin

$I_{out} := \{R \in I_R \text{ with highest priority}\};$
if I_{out} *is not a singleton*
 then $I_{out} := \{R \in I_{out} \text{ most recently matched by changes}\};$
if I_{out} *is not a singleton*
 then $I_{out} := \{R \in I_{out} \text{ whose condition is the most selective}\};$
if I_{out} *is not a singleton*
 then $I_{out} := \text{a random rule } R \in I_{out};$
return I_{out}

end.

3.2 Starburst

The basic features of Starburst are: disjunction of updates as events, set-oriented reaction, deferred modality and relative priorities among rules. This can be characterized with the following algorithm that specifies the execution model of Starburst rules.

Algorithm STARBURST

Input: A Starburst active program P and a user transaction $T = U_1; \dots; U_k$.

Output: A transaction T^S induced by T .

begin

$T^S := U_1; \dots; U_k; \text{REACT}(P, U_1); \dots; \text{REACT}(P, U_k);$
return T^S

end.

In Starburst for any two rules, one rule can be specified as having higher priority than the other rule. It follows that rules are partially ordered. Conflict resolution in Starburst can be described by the following procedure.

Algorithm SELECT

Input: A set of triggered rules I_R .

Output: One rule in I_R .

begin

$I_{out} :=$ a serialization of the rules in I_R that satisfies the partial order;

return the first rule I_{out}

end.

3.3 HiPAC

The HiPAC system is indeed object-oriented but we describe its rule execution semantics in the relational model. This is however sufficient for our purposes. The basic features of HiPAC are: sequences of actions, set-oriented reaction, seven different coupling modalities and concurrent execution of rules. Three modalities can be characterized with the following algorithm that specifies the semantics of HiPAC rules for three different coupling modes between Event-Condition and Condition-Action parts of a rule.

Algorithm HiPAC

Input: An HiPAC active program P and a user transaction $T = U_1; \dots; U_k$.

Output: A two transactions T_1^H and T_2^H .

begin

case E-C Mode and C-A Mode **of**

Immediate – Immediate :

$T_1^H := U_1; \text{REACT}(P, U_1); \dots; U_k; \text{REACT}(P, U_k);$

Deferred – Deferred :

$T_1^H := U_1; \dots; U_k; \text{REACT}(P, U_1); \dots; \text{REACT}(P, U_k);$

Decoupled – Immediate :

$T_1^H := U_1; \dots; U_k; T_2^H := \text{REACT}(P, U_1); \dots; \text{REACT}(P, U_k)$

endcase;

return T^H

end.

HiPAC departs from other active database systems in that multiple triggered rules are executed concurrently. However, HiPAC includes a relative order on rules that influences the serialization of concurrent execution. Thus, conflict resolution can be described by simply attaching priorities to the induced transaction produced by the algorithm above.

4 General results on active rule processing

An update in an induced transaction is executed only if: (1) the inducing update has been effectively executed or (2) it has not been invalidated afterwards. This is coherent with the approach, common to all the proposals, to consider always the *net effect* of update execution while processing a transaction in an active database. Then, a new effect semantics for induced transactions needs to be defined according to that.

Let U be an update, and s be a database state. We denote with $\Delta^+(U, s)$ and $\Delta^-(U, s)$ the *changes* induced by U on s , that is, the tuples that U adds to and deletes from s , respectively. Now, let s' be another database state. We say that the effect of U on s is *visible* on s' if: (1) $\Delta^+(U, s) \cup \Delta^-(U, s) \neq \emptyset$, (2) $\Delta^+(U, s) \subseteq s'$, and (3) $\Delta^-(U, s) \cap s' = \emptyset$.

Now, let $T = U_1, \dots, U_k$ be an induced transaction and let $T|_j$ denote the transaction $U_1, \dots, U_j$ composed by the first j components of T ($j \leq k$). Then, the new effect function EFF' is recursively defined as follows, for $1 \leq i \leq k$ and $1 \leq j \leq k$.

$$\text{EFF}'(T|_j)(s) = \begin{cases} \text{EFF}'(T|_{j-1})(s) & \text{if } U_j \text{ is induced by } U_i \text{ and the effect of } U_i \text{ on } \text{EFF}'(T|_{i-1})(s) \\ & \text{is not visible on } \text{EFF}'(T|_{j-1})(s); \\ \text{EFF}(U_j) \circ \text{EFF}'(T|_{j-1})(s) & \text{otherwise.} \end{cases}$$

Note that the above “induced by” relationship between updates can be easily derived on the basis of the indexes associated with the updates.

4.1 Transaction Equivalence.

Two induced transactions T_1 and T_2 are equivalent if $\text{EFF}'(T_1) = \text{EFF}'(T_2)$. Note that this notion is parametric with respect to immediate, deferred or decoupled modalities as well as with the selection policy. Under certain hypotheses, we have defined in [12] a method for testing equivalence of induced transactions which derives from an equivalence result for passive transactions [1]. The method is based on a set of transformation rules $\mathcal{R}$ that allows us to transform an induced transaction into an equivalent one [12].

4.2 Confluence.

We say that an active program P is *confluent* in a certain execution model M with respect to a user transaction T , if each pair of induced transactions T_1^M and T_2^M that can be generated by the algorithm describing M , have the same effect on any active database over P . Moreover, we say that an active program P is *strongly confluent* if it is confluent with respect to any user transaction T . In [12] we have provided a practical method to test for confluence based on the results on equivalence. Specifically, we have shown that an active program P is confluent in a certain execution model M with respect to a user transaction T , if an induced transaction generated by the algorithm describing M can be transformed in all the other transactions that can be generated by this algorithm using the above rules $\mathcal{R}$. We have also defined and characterized a notion of *local* confluence that applies on the active program and provides a necessary and sufficient condition for strong confluence.

4.3 Optimization.

Optimizing induced transactions is particularly important since, with our approach, it yields a method for optimizing the overall activity of active rule processing. The notion of optimality that we use is based on syntactic aspects, that is length and complexity of a transaction, but it turns out to be appealing also for operational criteria such as the number of atomic updates performed by a transaction. Let $|T|$ denote the number of insertion/deletion operations involved in the transaction T . Let T and T' be two equivalent induced transactions. Then, we say that T is *simpler* than T' if $|T| < |T'|$.

The set of transformation rules $\mathcal{R}$ above mentioned can be grouped into two classes. The former contains simplification rules, which always yield a strictly simpler transaction, whereas the latter consists of commutativity rules, which do not affect the complexity of the translation but are useful in order to apply simplification rules. A *reduction* of a transaction T based on $\mathcal{R}$ is the transaction obtained from T by applying rules in $\mathcal{R}$ alternating simplification and commutativity rules as long as some simplification rule can be applied. In [12] we have shown that: the reduction process (1) generates a simpler transaction, (2) always terminates (in polynomial time), and (3) is essentially deterministic regardless of the order of application of the rules.

5 Conclusions

We have presented a framework based on transaction transformation techniques that allows us to reduce active rule processing to passive transaction execution. We have shown that many problems are easier to understand and to investigate from this point of view, as they can be tackled in a formal setting that naturally extends an already established framework for relational transactions. We have also shown that this framework can be used for the specification and the theoretical analysis of a number of different execution models of active rule languages.

We believe that the approach is also appealing from a practical point of view since a lot of work can be done statically, at compile time, and its implementation does not require any specific run-time support.

References

- [1] S. Abiteboul and V. Vianu. Equivalence and Optimization of Relational Transactions. *Journal of the ACM*, 35(1):70–120, January 1988.
- [2] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 59–68, 1992.
- [3] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the Sixteenth International Conf. on Very Large Data Bases, Brisbane*, pages 566–577, 1990.
- [4] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the Seventeenth International Conf. on Very Large Data Bases, Barcelona*, pages 577–589, 1991.
- [5] U. Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraint. In *ACM SIGMOD Record*, Vol. 17, No 1, pages 51–70, 1988.
- [6] N. Gehani and H. V. Jagadish. ODE as an active database: constraints and triggers. In *Proc. Seventeenth International Conf. on Very Large Data Bases, Barcelona*, pages 327–336, 1991.
- [7] G. Guerrini, D. Montesi, and G. Rodriguez. Implementing active rules in object-database systems. Technical Report n. 62–94, Politecnico di Milano, 1994.
- [8] P. W. P. J. Grefen. Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem. In *Proc. of the Nineteenth International Conf. on Very Large Data Bases, Dublin*, pages 581–591, 1993.
- [9] E. N. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, 1992.
- [10] E. N. Hanson and J. Widom. Rule Processing in Active Database Systems. In *International Journal of Expert Systems*, 6(1):83–119, 1993.
- [11] D.R. McCarthy and U. Dayal. The architecture of an Active Data Base Management System. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 215–224, 1989.
- [12] D. Montesi and R. Torlone. A Rewriting Technique for the Analysis and the Optimization of Active Databases. In *Proc. of the Fifth International Conference on Database Theory, pag 238-251, Springer-Verlag, Prague*, 1995.
- [13] M. Stonebraker. The integration of rule systems and database systems. *IEEE Trans. on Knowledge and Data Eng.*, 4(5):415–423, October 1992.
- [14] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching, and views in data base systems. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 281–290, 1990.
- [15] J. Widom and S. J. Finkelstein. Set-Oriented Production Rule in Relational Databases Systems. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 259–270, 1990.