Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 94 (2005) 71–77
Information Processing Letters
www.elsevier.com/locate/ipl

Event Trace Independence of active behavior ☆

Angela Bonifati [a,*], Stefano Ceri [b,1], Stefano Paraboschi [c]

[a] *Icar CNR, Via P. Bucci 41C, I-87036 Rende, Italy*
[b] *DEI, Politecnico di Milano, P.za L. da Vinci 32, I-20133 Milano, Italy*
[c] *DIGI, Università di Bergamo, Viale Marconi 5, I-24044 Dalmine, Italy*

Received 12 April 2004; received in revised form 10 December 2004

Available online 16 January 2005

Communicated by S.E. Hambrusch

Abstract

We present the *Event Trace Independence* (*ETI*), a novel property of active rules exhibiting a behavior independent of the specific event sequence that had caused a state transition. When employed in a distributed setting, this property supersedes the classical property of confluence, which is not sufficient herein. We show that ETI is in general undecidable and provide a sufficient condition, called *invertibility*, which offers a practical way to demonstrate ETI.

Keywords: Rule analysis; Active databases; Rule termination; Events; Databases

1. Introduction

Event Trace Independence (ETI) is a novel property that characterizes active databases. The property holds for a given rule system and rule sets when, for any arbitrary state change from an initial to a final state, the behavior of the system is independent of the particular state sequence that produced the final state after rules activation. This property is particularly relevant in the case of asynchronously distributed active systems whenever the changes on a site are gathered in a trace, which is forwarded to another site after the occurrence of the changes. Changes occurred on a site are collected in an *event trace* and exported to another site, on which they may fire a set of actions. The action is a generic *stored procedure*, i.e., code that is located on a site and executed as response to a special event. These systems include (but are not limited to) classical distributed databases, data warehouses, replicated systems, mediators, caching systems, recent P2P architectures and the Web. This problem is also relevant in the context of recent technology developed for XML. Sig-

☆ Research presented in this paper is supported by Esprit project WebSI.
* Corresponding author.
E-mail addresses: bonifati@icar.cnr.it (A. Bonifati), ceri@elet.polimi.it (S. Ceri), parabosc@unibg.it (S. Paraboschi).
URL: http://www.icar.cnr.it/angela.
[1] Work is partially supported by CESTIA-CNR (Milano).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.12.014

nificant examples, such as Apache Jelly, Macromedia DreamWeaver, JSP.Net, basically embed method calls inside the documents and make these calls activated upon the click of a mouse or the occurrence of a particular event.

Event traces as described above can be captured at run-time, packed and then shipped to another machine. Or, alternatively, they can be generated by comparing two different versions of the same data [1]. Indeed, besides containing the sequence of occurred events, each event trace also describes the transition between two states of the system. There are usually different event traces describing the same state transition. In a distributed asynchronous context, such as that of the systems described above, event traces may be used liberally, e.g., after merging with other traces. Thus, one would like to guarantee that the active behavior triggered *elsewhere* by those event traces is independent of the trace, but only dependent on the occurred state transition. This is different from what was studied in centralized active databases [2–6], where the event trace has usually been fixed and known in advance. Two properties of *termination* and *confluence* have been defined for sets of database rules. Termination guarantees that any transaction execution associated with rule execution is completed, whereas confluence guarantees that such completion produces the same final state.

However, in a distributed context working asynchronously, such as that of the Web or in any other of the above settings, termination and confluence are no longer sufficient to explain the active behavior. When events may have occurred outside the scope of the system or may not have been traced with the same tool, the behavior of rules in the current site still needs to be guaranteed as safe.

In the remainder, we assume reactive behaviors produced by active rules and we adopt the relational model as the underlying data model. Nevertheless, the results can be applied to any different event-based reactive processing among those discussed.

Statement of the problem. The property of *event trace independence* complements the notion of confluence in all the situations when transaction execution is not known in terms of the sequence of the atomic events, but only in terms of the initial state and the final state, produced by the sequence. To better appreciate the difference between confluence and event trace independence, consider Fig. 1. The (bold) segments (labeled with *ET*) between the database state DB_0 and the state DB_i represent the event traces, while those (labeled with *RE*) between the state DB_i and each final database state DB_{fi} represent the rule executions triggered by the events of the traces. Note that each point in the figure represents a database state, that, in case of the event traces, is due to a user update and, in case of rule executions, is caused by a rule activation.

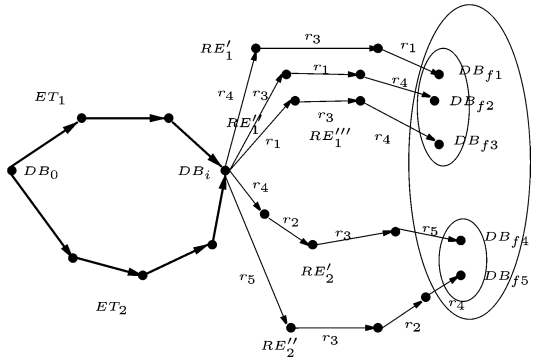

Fig. 1. An overview of confluent and ETI rule sets.

Observe that in classical confluence, for a given event trace, e.g., ET_1, what is required by the property is that the rule executions raised by this event trace ET_1 (in the figure, RE'_1, RE''_1 and RE'''_1, respectively) should converge to the same final state $DB_{f1} = DB_{f2} = DB_{f3}$ (enclosed in the small oval). Instead, in case of the ETI property, for each event trace between DB_0 and DB_i (in the figure only two of them are illustrated, ET_1 and ET_2), different rule executions can be raised (the previous rule executions for ET_1, and RE'_2, RE''_2 for ET_2). What is demanded by ETI is a stronger convergence, i.e., the convergence of all the final states into a unique one ($DB_{f1} = DB_{f2} = DB_{f3} \equiv DB_{f4} = DB_{f5}$, enclosed in the big oval).

Contributions. Event trace independence is a difficult problem, because in addition to confluence, we must guarantee the uniqueness of the final state while enumerating all the possible event traces occurring between two states. We show that ETI is undecidable. Therefore, we devise a sufficient condition, called *rule invertibility*, which guarantees ETI. This is similar to the use of *rule commutativity* to guarantee confluence [2]. However, commutativity holds for a fixed event trace and is no longer applicable here. More-

over, invertibility can be used as a proof mechanism to demonstrate event trace independence.

2. Events, rules and event traces

Let us assume to have a database schema $DB = \{R_1[A_1], \ldots, R_n[A_n]\}$, where each R_i is a *relation* name on a set A_i of attribute names a_j. A database state at time w is an instance of the database, namely $DB_w = \{\mathcal{I}(R_1[A_1]), \ldots, \mathcal{I}(R_n[A_n])\}$, $\{t_1, \ldots, t_n\} \in \mathcal{I}(R_i[A_i])$.

Sequences of tuple events, i.e., events applied to the database tuples, can be applied to a database state. More formally, we first define the sequences of tuple events:

Definition 2.1. An *event trace* is a sequence of tuple events, where a *tuple event* e is defined as $\tau(R_i, t_j)$, where R_i is a relation, t_j a tuple, and $\tau \in \{\texttt{insert}, \texttt{delete}\}$.[2]

Next, we first define the application of a tuple event to a database state and then the application of a sequence of tuple events.

Definition 2.2. A tuple event $e = \tau(R_i, t_j)$ applied to a database state DB_i, $DB'_i = DB_i \oplus e$, yields a new database state DB'_i, as follows:

- If $\tau = \texttt{insert}$ and $t_j \notin DB_i$, then $DB'_i = DB_i \cup t_j$. If $t_j \in DB_i$, then $DB'_i = DB_i$ and the tuple event is not applicable to *DB*.
- If $\tau = \texttt{delete}$ and $t_j \in DB_i$, then $DB'_i = DB_i - \{t_j\}$. If $t_j \notin DB_i$, then $DB'_i = DB_i$ and the tuple event is not applicable to *DB*.

Definition 2.3. An event trace $ET = e_1, \ldots, e_n$ applied on a database state DB_0 yields a final state DB_n, $DB_n = DB_0 \oplus ET$, iff each event of the sequence yields a database state $DB_i = DB_{i-1} \oplus e_i$.

It follows that, if an event in ET is not applicable, then the entire event trace is not applicable to *DB*. We will show in Proposition 2.9 that the notion of applicability is indeed central to the property of ETI. As a consequence, only applicable event traces can be inverted and characterize a significant class of ETI rules (as shown in the remainder in Theorem 4.2).

Since we are mostly interested in the result of the application of an event trace, we denote by $ET_{DB_0 \to DB_n}$ a generic event trace which applied on DB_0 produces DB_n; i.e., $ET_{DB_0 \to DB_n} \leftrightarrow DB_n = DB_0 \oplus ET$.

To understand how active databases are sensitive to event traces, we briefly recall the definition of active rules.

Definition 2.4. An event of an event trace $ET = e_1, \ldots e_n$, raises an *Active Rule* $r = (E, C, A)$ iff:

E: $\{\tau(R_j) \mid e_i \in ET \wedge e_i = \tau(R_j, t_i)\}$, where each $\tau(R_j)$ represents a type of event being monitored and e_i is a tuple event as defined above. Variables may be associated with the tuples on which the event occurred.

C: (*predicate*)* where the *predicate* is a predicate using variables defined in the event or new variables associated with the tuples.

A: $\{e_k \mid k > n \wedge e_k \in ET'\}$, where the tuple events in the action form a new event trace ET'.

It follows quite straightforwardly that an active database $\langle DB, R\rangle$ is a database *DB* enriched with a collection of active rules R. Moreover, we adopt a general rule execution mechanism, which works as follows:

(i) the event trace ET is applied on the database state DB_i ($\forall e_i \in ET \mid e_i$ is applicable);
(ii) all the rules that have the type of event e_i in their event part E, are *triggered* and the pair (r, e_i) is inserted into the *conflict set Cs*;
(iii) if $Cs = \emptyset$, rule processing terminates;
(iv) a rule r is extracted from *Cs*, following a given *Conflict Resolution Policy*;[3]
(v) the rule condition C is evaluated.

If the condition holds, the action A is executed and the events e_k thus generated are applied to DB_i and appended to the event trace ET'. Moreover, we *do not*

[2] We disregard without loss of generality the update events, which equivalently correspond to deletes followed by inserts.

[3] We are not proposing here a particular conflict resolution policy; we just assume a simple one, based on rule priorities.

assume a particular coupling mode for rules, as *immediate, deferred* or *decoupled* modes are all suitable. Deferred (decoupled respectively) rule execution implies that evaluation of rule conditions is done at the end of the current transaction (in a next transaction respectively), whereas immediate rule execution evaluates conditions as earlier as possible within the current transaction. The property of ETI holds independently of the coupling mode used by the rules, as long as applicability of event traces is guaranteed.

While defining the ETI property, we assume that rule execution eventually reaches a *quiescent* state where no rule is triggered—and therefore the rule execution mechanism terminates. More precisely:

Definition 2.5. Given a set of rules R, an initial state DB_n and an event trace ET, the function RuleExec(R, DB_n, ET) returns the quiescent state DB_f produced by rule execution.

2.1. Properties of event traces

We start our theory on event traces by giving a definition of *equivalence*.

Definition 2.6. $ET_i \equiv ET_j \leftrightarrow (\exists DB_0 \mid (DB_0 \oplus ET_i) = (DB_0 \oplus ET_j)) \vee (\exists ET_k \mid (ET_i \equiv ET_k) \wedge (ET_j \equiv ET_k))$.

Equivalence of event traces is reflexive, symmetric, and transitive. From now on, $ET_{DB_0 \to DB_n}$ denotes a family of equivalent event traces, applicable on DB_0 and yielding DB_n.

Definition 2.7. Given an event e, its inverse inv(e) is the single event that reverses the effect of e.

In the relational model, the inverse of `insert`(R, t) is `delete`(R, t); the inverse of `delete`(R, t) is `insert`(R, t).

Analogously, we define the inverse of an event trace.

Definition 2.8. Given an applicable event trace ET in $ET_{DB_0 \to DB_n}$, the inverse event trace inv(ET) is the event trace composed of the inverse of each event in ET, in reverse order (i.e., given $ET = \{e_1, \ldots, e_n\}$, inv($ET$) = {inv($e_n$), …, inv($e_1$)}).

The inverse event trace produces the original database state if applied after ET. This is shown in the following:

Proposition 2.9. *Given an applicable event trace* $ET_{DB_0 \to DB_n}$, *the inverse event trace* inv(ET) *is also applicable in* DB_n *and its application produces* DB_0.

Proof (*Sketch*). We split ET into two parts: a sequence of $n-1$ events $\{e_1, \ldots, e_{n-1}\}$ and the last event e_n. inv(ET) can be split into inv(e_n) and {inv(e_{n-1}), …, inv(e_1)} (the second fragment of inv(ET) is the inverse of the first fragment of ET). The last event of ET is exactly inverted by the first event of inv(ET). Thus, since the last event of ET was applicable and led to the final state DB_n, the first event of inv(ET) is also applicable in DB_n. By induction, the inversion can be applied to the complete inverse trace and eventually lead to the initial state DB_0. □

Definition 2.10. Given an applicable event trace ET in $ET_{DB_0 \to DB_n}$, an event trace equivalent to inv(ET) is a *complement* of ET.

By definition of event trace equivalence, the complement, if applicable, transforms DB_n into DB_0. The following lemma holds, whose proof is omitted for brevity.

Lemma 2.11. *The complement of the complement of an event trace ET is equivalent to ET*.

3. Event Trace Independence

We are now able to introduce the novel property.

Definition 3.1 (*ETI*). Given a rule set R, the rule set is *Event Trace Independent* (*ETI*) iff ∀ pair of states $\langle DB_0, DB_n \rangle$: $(\exists! DB_{\text{fin}} = \text{RuleExec}(R, DB_n, ET), \forall ET \in ET_{DB_0 \to DB_n})$.

It follows from the above definition that the function RuleExec does not depend on the specific event trace chosen among all the equivalent ones. At this point, we would like to make clear that the classical notion of confluence is different from ETI. Indeed, for the only purpose of comparison, we recall its definition.

Definition 3.2 (*Confluence*). Given a rule set R, the rule set is *confluent* iff $\forall DB_n, \forall ET \in ET_{DB_0 \to DB_n}$: $(\exists! DB_{\text{fin}} = \text{RuleExec}(R, DB_n, ET))$.

3.1. Undecidability of ETI

Demonstrating that a rule set has the ETI property is in general a difficult problem. Thus, analogously to what happened for termination and confluence, it can be proven that ETI is undecidable.[4]

Theorem 3.3. *ETI is undecidable for a set of rules written in the language of Definition* 2.4.

Proof (*Sketch*). The proof is done by reduction to an instance $\mathcal{P}$ of the *Post Correspondence Problem* (PCP). We have devised a set of rules which is ETI iff $\mathcal{P}$ has a solution. Since PCP is known as undecidable, the ETI property cannot be decidable as a consequence. □

Undecidability means that we cannot build a complete procedure for checking whether a set of rules is ETI. Analogously to termination and confluence, we identify a subclass of rule systems for which efficient techniques lead to prove ETI. Proving ETI is significantly more difficult than proving termination. Many techniques for termination are based on a relatively simple analysis of the structure of each rule, and eventually reduced to a combinatorial problem on graphs (e.g., [2,4,5]). Conversely, both confluence and ETI require a careful consideration of rule behavior. Moreover, confluence is less critical for rule applications than ETI itself, as active rule systems usually apply a fixed ordering on rules, and thus confluence directly derives from termination. In next section, we will give *sufficient conditions* for ETI.

4. Rule invertibility

We characterize a subclass of rules for which the property of *Rule Invertibility* holds. Intuitively, rule invertibility identifies the rule systems where the result of the application of separate event traces is equivalent to the application of a single event trace representing the same global state transition.

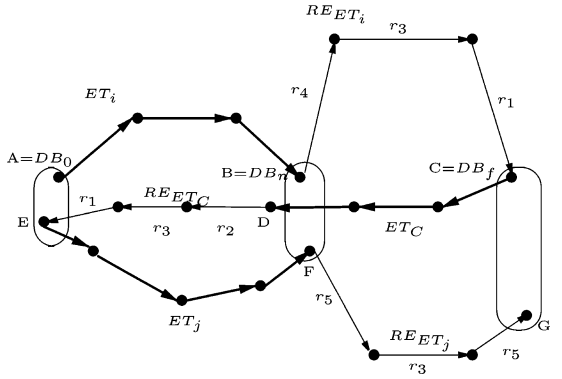

Fig. 2. The sequence of states A, B, C, D, E, F, G.

Definition 4.1. A rule system is *invertible* if, for any initial state DB_n and applicable event trace ET followed by rule execution that reaches state $DB_R = \text{RuleExec}(R, DB_n, ET_{DB_0 \to DB_n})$,

(i) a complement $\overline{ET}$ can be applied to the database state DB_R and
(ii) the state produced by rule execution is the original state DB_0, e.g., $DB_0 = \text{RuleExec}(R, \text{RuleExec}(R, DB_n, ET_{DB_0 \to DB_n}) \oplus ET_{DB_n \to DB_0}, ET_{DB_n \to DB_0})$.

We now demonstrate that an invertible rule system is ETI, which represents the chief result of the paper.

Theorem 4.2. *Invertible rule systems are ETI.*

Proof. We consider two generic (equivalent) event traces ET_i and ET_j representing the transition from DB_0 to DB_n. We introduce an event trace ET_c complement of ET_i. We can now apply the event traces in the order ET_i, ET_c, ET_j, with every application of event traces followed by rule execution. The system will step through the sequence of states (see Fig. 2):

A: DB_0; B: DB_n $(= DB_0 \oplus ET_i)$;
C: DB_f $(= \text{RuleExec}(R, DB_n, ET_i))$;
D: $DB_f \oplus ET_c$; E: $\text{RuleExec}(R, \text{D}, ET_c)$;
F: $\text{E} \oplus ET_j$; G: $\text{RuleExec}(R, \text{F}, ET_j)$.

[4] The fact that ETI rules have as pre-conditions termination and confluence does not imply that ETI rules are also undecidable.

We can group the above transitions in two different ways. First, we can consider the path A, B, C, D, E, which represents the sequence of states reached by the database after the application of ET_i and ET_c, each followed by the corresponding rule reactions. Since ET_c is the complement of ET_i and the rule system is invertible, the final result of the execution of the rules is to return to the original database state DB_0 (i.e., A and E coincide; the fact that ET_i and ET_j are equivalent implies that also B and F coincide). Then, the behavior of the rule system will be equivalent to the one obtained applying ET_j directly to DB_0. We now can consider the path C, D, E, F, G, which represents the sequence of states reached by the database after the application of ET_c and ET_j, each followed by the corresponding rule reactions. From Lemma 2.11, ET_i is the complement of ET_c and, since ET_i and ET_j are equivalent, ET_j is a complement of ET_c. The application of ET_j after ET_c produces, for rule invertibility, the database state before the application of ET_c (i.e., C and G coincide). This means that the behavior of the system is the same that would be obtained by applying only ET_i. Then, the behavior exhibited by applying ET_i is the same that would be obtained applying ET_j. Since ET_i and ET_j are arbitrary event traces representing the transition between two arbitrary database states $DB_0 \rightarrow DB_n$, this proves that the rule system is ETI. □

However, the inverse of Theorem 4.2 does not hold as ETI rules characterize a larger family of rule sets than invertible ones. We formalize this result in the next theorem, whose proof is straightforward and omitted for brevity.

Theorem 4.3. *ETI rule systems are not always invertible.*

4.1. Approach to the demonstration of rule invertibility and invertible rule applications

It is possible to separate the events of an event trace *ET* into two sets: the *minimal part* and the *redundant part*. Event traces may contain pairs of `insert-delete` events operating on the same table and characterizing the same tuple. The application of such a pair of events does not modify the database state. All such pairs are put into the redundant part of the event trace, redundant(ET); the events in ET which are not in redundant(ET) are put into the *minimal* event trace, min(ET). In order to prove that a set of rules is invertible, we can operate in two steps:

(i) show that the redundant part of ET, redundant(ET), does not have an impact on the final state produced by rule execution;
(ii) show that every event e appearing in min(ET) activates a rule whose effect is nullified by the rule activated by $\bar{e}$ appearing in min($\overline{ET}$).

The above approach leads to a general proof technique for the ETI property, which is based on invertibility (hence, it is only sufficient). The technique consists in separating every possible trace in its redundant part and its minimal part, and then showing that the application of rules triggered by any redundant part has no side effects. Indeed, the application of rules triggered by any equivalent trace in the minimal set produces the same quiescent state.

Apart from the specific demonstration technique presented above, the concept of rule invertibility is important because it characterizes many rule systems and clarifies their nature. Examples of invertible rule applications are *global rollback-based maintenance rules* (i.e., rules that force a global rollback if the final state of a transaction is inconsistent), and all *view maintenance rules* (i.e., rules maintaining all kinds of views, ranging from select–project–join (SPJ) views, views with negation and/or aggregates, views with recursion).

5. Conclusions

We introduced the event trace independence as an interesting novel property that characterizes active behavior. We showed that such property is essential when the data repository does not manage the actual changes to the data. We have shown that ETI is undecidable and envisioned a proof technique, based on *rule invertibility*, to demonstrate it. We argue that ETI is applicable in several fields and is particularly fascinating in a web context, where event streams occur at remote sites and a diff algorithm must be used to envision the trace. In such a context, ETI also implies the independence from the used diff algorithm.

References

[1] G. Cobena, S. Abiteboul, A. Marian, Detecting changes in XML documents, in: Proc. of ICDE, San Jose, USA, 2002, pp. 41–52.
[2] A. Aiken, J. Widom, J.M. Hellerstein, Static analysis techniques for predicting the behavior of active database rules, ACM Trans. Database Syst. 20 (1) (1995) 3–41.
[3] E. Baralis, S. Ceri, J. Widom, Better termination analysis for active databases, in: RIDS, Edinburgh, Scotland, 1993, pp. 163–179.
[4] E. Baralis, S. Ceri, S. Paraboschi, Compile-time and run-time analysis of active behaviors, IEEE Trans. Knowledge Data Engrg. (TKDE) 10 (3) (1998) 353–370.
[5] E. Baralis, J. Widom, An algebraic approach to static analysis of active database rules, ACM Trans. Database Syst. 25 (3) (2000) 269–332.
[6] L. van der Voort, A. Siebes, Termination and confluence of rule execution, in: Proc. Internat. Conf. on Information and Knowledge Management (CIKM), Washington, DC, 1993, pp. 245–255.