Structural Issues in Active Rule Systems

James Bailey, Guozhu Dong and Kotagiri Ramamohanarao

Dept. of Computer Science, University of Melbourne, Parkville Vic. 3052, Australia
E-mail: {jbailey,dong,rao}@cs.mu.oz.au

Abstract. Active database systems enhance the functionality of traditional databases through the use of active rules or 'triggers'. There is little consensus, though, on what components should be included in a rule system. In this paper, the expressive power of some simple active database rule systems is examined and the effect of choosing different features studied. Four important parameters of variation are presented, namely the rule language, the external query language, the meta rule language and the pending rule structure. We show that each of these is highly influential in determining the expressiveness of the rule system as a whole, and that an appreciation of them can serve as a basis for understanding the broader picture of system behaviour.

1 Introduction

Traditional database systems provide a mechanism for storing large amounts of data and an interface for manipulating and querying this data. They are, however, passive in the sense that their state can only change as a result of outside influences. In contrast, an active database is a system providing the functionality of a traditional database and additionally is capable of reacting automatically to state changes, both internal and external, without user intervention. This functionality is achieved by active rules or triggers. Applications have been found in areas such as workflow management, view management and constraint maintenance [7, 4, 3]. Additionally, many different prototype systems have been built. Despite this, less progress has been made with regard to the theory of active database rules. An understanding of how various features of rule syntax and semantics can affect the properties of active database rules is still in its infancy. Our aim in this paper is to illustrate the expressiveness of simple active rule systems and note the effect of making certain changes in their functionality. We study four dimensions of variation, namely

- The mechanism used to record pending rules for execution
- The rule language
- The external query/update language
- The meta rule language

We measure the power of a rule system by the set of external event histories that it can recognise. This metric helps us focus upon the potency of active rules as a programming language mechanism. It differs from most work on active databases, since the attention is less on using rules to react to changes in the database, but rather on using them as a tool to carry out computation. Through the use of this model, we are able to demonstrate two key results. The first is that even a very basic rule language can have power comparable to a Turing machine. The second is that the expressiveness of the rule system as a whole is acutely sensitive to a small change in any of the above dimensions. Each of these has important implications for language designers.

The remainder of this paper is structured as follows. The next subsection presents some related work and Section 2 presents the definitions to be used in the paper. Section 3 looks at the implications of varying the rule execution structure. In Section 4, meta rules are introduced and analysed. Section 5 looks at changing the rule and query language and it brings a database perspective to the results we have obtained. Lastly, we provide some conclusions and look at future directions.

1.1 Related Work

In [11], the concept of the relational machine is presented as useful for simulating an active database. It is essentially a Turing machine which has restricted access to a relational store via first order queries and is designed to capture the spirit of a database query language embedded in a host programming language such as C. An active database system is modelled by two relational machines, one replicating the external query system and the other duplicating the set of active rules. Using this model, statements can be made about the power of various simplified prototype systems. Our work in this paper is essentially complementary to this approach. Our aim is not so much to construct an all embracing formalism for active databases, but rather to focus on some of the elements that affect the power of the rule system. Also, we treat certain aspects not covered in [11] such as syntactic events. Our work is also complementary to [9], where a programming language which employs the delayed update or *delta* is defined. This can be used to express the semantics of some active database systems.

In [10], methods for specifying meta rules to manage execution of the rule set as a whole are presented. Although we also consider meta rules, our interest is primarily in the additional computational power they can add to a rule system and not on how to use them for static reasoning about rule behaviour.

2 Preliminaries

We begin by presenting the core active rule language used in this paper. It follows the so-called ECA format

on event *if* condition *then* action

The following 0-1 language was used as a simple example language in [2].

Definition 1. 0-1 Language

- Events are of the form U(X) which we understand to mean 'update the variable X'. They are thus triggered by an assignment statement on this variable[1].
- A condition is a conjunction of simple conditions. A simple condition is a test of the form *Var=0* or *Var=1*
- An action is a sequence of simple actions. A simple action is an assignment of the form *Var=0* or *Var=1*. □

Thus a typical 0-1 rule might be

On U(A)
If $C = 0 \wedge D = 1 \wedge T = 1$
Then $T = 0$; $B = 1$

The basic execution model used is:
0. A sequence of external events occurs, each of which may trigger some rules. Control is then passed to the rule system.
1. If there are no triggered rules then exit
2. Select a rule to execute from the pending rule structure
3. Evaluate the condition of the selected rule
4. If the condition is true then execute the action of the selected rule

The action executed in step 4 can cause further (internal) events which trigger other rules and these will be added in turn to the pending rule structure. Thus the steps 1-2-3-4 can potentially loop forever.

We now define what we mean by the power of a set of active database rules. The definition focuses on the power of rules as a programming language construct. The rule system is seen as a recogniser for external event sequences. Using this definition we can describe rule expressiveness in terms of formal language theory. As a result, it becomes easy to compare the expressiveness of different constructs and various corollaries on decidability can be obtained for free.

Definition 2. Rule Power
Suppose we have an alphabet E of external events and a set of rules R. Then L(R) denotes the set of external event sequences accepted by R. A sequence $w \in E^*$, known as an *external event history*, is said to be accepted by R if the computation of R, after input of the external event history w, halts in an accepting state.[2] □

[1] We don't require that the new value has to be different from the old one for an update to be registered.

[2] A state of the rule system is described by the values of its 0-1 variables. We designate a number of these states as *accepting states.*

Definition 3. Let H_p represent a set of external event histories. A rule set R is said to characterise H_p if L(R) $= H_p$. □

We are interested in situations such as H_p = the set of regular histories or H_p = the set of recursively enumerable histories.

3 Pending Rules

The first feature we investigate is the nature of the pending rule structure. The most straightforward choice is to make it a set and whenever a rule needs to be selected from it, the one with highest priority is chosen. We assume rules are totally ordered by priority. The implications of a set are that it can contain only one instance of any particular rule and thus there is a bound on the size of the set. More complex choices are to use a queue (like HiPAC [8]) or a stack (like NAOS [6]) to record the pending rules. These may contain multiple instances of a rule and thus are unbounded in size. Rule selection is done by taking the rule on top of the stack or queue. When a rule is triggered it is placed on top of the stack or on the bottom of the queue and if more than one rule is activated at once then they are placed on in order of highest priority.
We now state an interesting and perhaps surprising result about rules in a 0-1 language.

Theorem 4. *A 0-1 trigger system with a queue characterises the set of recursively enumerable external event histories.*

Proof(sketch): We will show that we can build a set of 0-1 triggers with queue to recognise any external event history that a Turing machine can. We show equivalence to Post machines [12] instead of Turing machines however. A Post machine has exactly the same power as a Turing machine and is like a pushdown automaton which uses a queue instead of a stack. It consists of an alphabet of input symbols and a number of states including a START state. In each state one can move to another state after reading and removing a symbol from the front of the queue and/or possibly adding an element(s) to the end of the queue. The machine doesn't have a separate input tape unit, but rather the input string is initially loaded into the queue before execution. Acceptance of a string is defined by whether the machine halts in an accepting state.

A Post machine's transition is of the form *(state, symbol, state', symbol')*

- *state* is the machine's current state
- *symbol* is the symbol on top of the queue
- *state'* is the new state the machine will go to
- *symbol'* is the symbol to place on the bottom of the queue

To translate this machine into 0-1 rules, we define the following variables.

- A special variable V_{accept} to indicate an accepting state

– A special variable V_ϵ, this will allow us to deal with the situation when the empty word is put on to the queue
– A special variable V_{flag} to help with mutual exclusion
– For each machine symbol a, the variable V_a
– For each machine state p, the variable V_p

We group transitions together according to symbol. Suppose the group for symbol a is the following:

$$(p,\, a,\, p_1,\, w_p)$$
$$(q,\, a,\, q_1,\, w_q)$$

These can be translated into the following rules.

R_a	R_{a_p}	R_{a_q}
On U(V_a)	On U(V_a)	On U(V_a)
If true	If V_p=1 and V_{flag}=1	If V_q=1 and V_{flag}=1
V_{flag}=1	Then V_p=0 ; V_{p_1}=1 ;	Then V_q=0 ; V_{q_1}=1 ;
	$V_{w_p} = 1$; V_{flag}=0	$V_{w_q} = 1$; V_{flag}=0

The variable V_{flag} ensures that only one of R_{a_p} and R_{a_q} is executed. Rule R_a resets V_{flag} so that other rules may use it. These rules are ordered from left to right so that R_a has the highest priority. If p is an accepting state, then we also include the action $V_{accept} = 1$ in rule R_{a_p}, similarly for state q and rule R_{a_q}.

We also need a rule in order to empty the queue when an accepting state is entered. Its priority is less than R_a and larger than R_{a_p} and R_{a_q}.

$R_{a_{empty}}$
On U(V_a)
If V_{accept}=1
Then V_{flag}=0

We have thus shown how the state transitions of the Post machine can be replicated by 0-1 rules. To complete the picture, we assume the rules are initially placed in the queue by a sequence of external events firing (this corresponds to the Post machine's input string) as described in section 2 and the variables V_s (corresponding to the START state s) and V_{accept} are initialised to 1. We also designate the START state as an accepting state since this allows us to accept ϵ (the empty event history). A 0-1 rule computation halts once the queue is empty.
□

Since we can simulate a Turing machine, it then immediately follows that

Corollary 5. *Termination is undecidable for a 0-1 trigger system with a queue.*
□

The next two theorems consider what happens when we replace the queue by a stack or a set. We observe that there is a dramatic loss of power for these structures. One reason for this is to do with the way rules are placed on the pending structure. Execution of rules can only begin once the entire external event history (in the form of rules) has been put in the pending rule structure. Hence the stack/set is being used as both a source of the history and also as an aid to computation. Contrast this situation with the operation of a machine such as a pushdown automaton, where a separate read only input is available. Here, the input string does not 'interfere' with intermediate computations on the stack. We could eliminate this interference by changing our semantics so that control is passed to the trigger system after each external event, but we would then want the ability to be able to terminate with a non-empty stack/set so it could then process the next external event and this would violate the spirit of active rule execution.

Theorem 6. *A 0-1 trigger system with a stack characterises the set of regular external event histories.* □

Theorem 7. *A 0-1 trigger system with a set can accept any external event history which can be described by a formula using the connectives* $\wedge$, $\neg$ *and* $\vee$ *to combine statements of the form* $\diamondsuit\ e_k$. $\diamondsuit\ e_k$ *holds at position* j *in a history iff the event* e_k *has occurred at position* j *or some preceding position.* □

Once again, using results from formal language theory we can state

Corollary 8. *Termination is decidable for a 0-1 trigger system with a set or a stack*

4 Meta Rules

Meta rules are used for managing the behaviour of the set of active rules. We have already seen an example of a meta rule in the form of the priority mechanism used to order the set of rules. We now consider complex events which can be thought of as a type of meta level construct for combining events.

Many active rule languages have a facility for specifying complex events. These are combinations of various primitive events. One needs to be careful, however, about specifying their semantics, since even seemingly simple operators may have a variety of interpretations [5].

The operator we will consider is the sequence operator. An event E=e1;e2 occurs if the event e1 followed by the event e2 occurs. The event consumption semantics we choose is a cumulative one and is intuitively 'match an e2 with each unconsumed e1 before it' (in real life this could correspond to tracking all deposits preceding a big withdrawal). Figure 1 illustrates this with six different occurrences of the event E. The numeric labels on the arcs indicate the complex event ordering i.e 1 occurs before 2, 2 occurs before 3 etc. When multiple rules

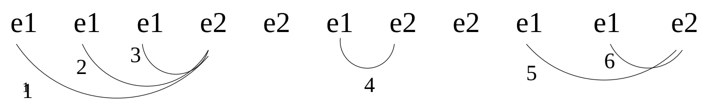

Fig. 1. Cumulative Consumption Semantics

are activated at once (e.g 1,2 and 3), they are pushed onto the stack in reverse order of firing (i.e 3 then 2 then 1).

Suppose we assume that our rule system has the power to recognise a complex event of the type just discussed. The following theorem tells us that it makes the system as powerful as when we had a queue earlier.

Theorem 9. *A 0-1 trigger system with a stack and the cumulative event sequence operator characterises the set of recursively enumerable external event histories.*

Proof(sketch): The proof is similar to that of Theorem 4. We will only need to show that it is possible to place a rule at the bottom of the stack to show that the stack can behave like a queue. For each rule in the rule set, we associate with it an identifier event. So for Rule $R1$ we have E_{R1}, Rule $R2$ we have E_{R2} etc. We also have a special event called E_{bottom}. Using these events we create another set of rules that are activated on complex events.
For rule $R1$ we would create another rule $R1'$ of the form

$R1'$
On $E_{R1}; E_{bottom}$
if C1'
then A1'

where C1' and A1' are the same as $R1$'s condition and action respectively.

Now suppose we wish to place rule $R5$ on the bottom of the stack and the stack already has a number of rules on it. We first set a variable called the consumption flag. This will make sure the condition of each rule in the stack is false and will also cause an event to be fired when that rule is considered. e.g If R1 is on top of the stack in consumption mode, then it will be removed and the event E_{R1} will be fired. Similar events will be fired for every other rule on the stack. Once the bottom of the stack is reached (detectable by an appropriate marker rule[3]), we do two things. Firstly we fire the event E_{R5} (since we want to put rule $R5$ on the bottom of the stack), then we fire the event for the marker rule followed by event E_{bottom} and lastly we turn consumption mode off. The firing of E_{bottom} causes all the rules with complex events defined above to be placed

[3] For this, it is necessary to assume the external event history is always begun by a distinguished event e_{marker} which places the marker rule in the stack.

back on the stack in reverse order of the firing of their E_{R_k} event. Thus the stack is the same as it was originally, but with rule $R5$ on the bottom. □

There are clearly many other types of meta rules which can be defined. A couple of examples are meta rules which prohibit two rules occupying the pending set simultaneously and meta rules which require a particular rule to be in the pending set for another rule to be added to it. The extra power meta rules provide to the rule system lies in their ability to control the flow of the system in non standard ways.

Remark: Suppose we have a meta rule which deterministically removes one instance of a certain rule from the pending set whenever a particular rule is added. Then it is straightforward to recognise the external event history $\{e_1^n e_2^n \mid n \geq 1\}$ using a 0-1 rule system with stack.

5 Language Variation

5.1 Trigger Language

We now turn our attention to the timing of activation of the components in an E-C-A rule. Current active database systems address this by incorporating the notion of coupling modes [8]. Each rule can be triggered in either of two modes.

- Immediate: The rule is placed into the pending rule structure immediately after the event occurs and control is then transferred to the rule set by the external query system (or if the event was generated internally by the rule system, then it will retain control).
- Deferred: If a rule is triggered, then it is placed into the pending rule structure only after the structure has become empty (until which time the rule can be thought of as occupying a separate 'deferred' pending structure). If a rule is triggered by an external event, then control is not passed to the rule system unless the event is the last operation in the history. Deferred mode corresponds to postponing rule execution until the end of a transaction, just before the commit phase.

In our semantics described in Section 2, we effectively assumed deferred coupling mode for rules activated triggered by external events and immediate coupling mode for rules triggered by internal events. If we relax this restriction, then we can get increased rule power.

Theorem 10. *A 0-1 trigger system with a stack and the option of immediate and deferred coupling modes for all rules, characterises the set of recursively enumerable external event histories.*

Proof(sketch): As in Theorem 9, we will just show that it is possible to place a rule at the bottom of the stack. First, suppose that for every rule we define two variants, one in immediate mode and one in deferred mode. Assume also, that we have a flag indicating whether we currently want to activate rules in deferred mode or in immediate mode. Suppose we want to place Rule $R5$ at the bottom of the stack, we set the flag to deferred and this will ensure that every rule on the stack is reactivated in deferred mode and then removed. Thus the stack will be emptied and conceptually we'll have a new stack containing all the rules activated in deferred mode, but with their original order reversed. We now activate rule $R5$ in deferred mode (we can set a flag to remember to do this as soon as we reach a marker rule indicating the bottom of the stack) and then carry out the deferred activation process once again, for each element on the stack. We then reset the flag to indicate immediate mode. We now have a new stack with the order as it was initially and rule $R5$ on the bottom. □

5.2 External Query Language

Query Augmentation We now address the question of whether the type of rule languages presented are useful in a database context. On the surface it would seem not, since neither the condition or action involves any reference to or manipulation of database relations. We show, however, that such languages can be useful provided events can be triggered in a certain way.

We look at whether the rule system can allow a given external query language to obtain answers to queries that it couldn't normally. Assume we are using a relational database and let us consider the query *even* on a unary relation T

$$\textit{even}(\mathrm{T}) = \text{true if } |\mathrm{T}| \text{ is even and false otherwise}$$

This query cannot be expressed by query languages such as *fixpoint, while or* $while_N$ on unordered databases [1]. It is possible to express this query using 0-1 active rules with stack and immediate coupling mode, however, if events can be generated in a tuple oriented fashion (i.e an event is triggered for each instance in the binding set).

Suppose a user asks the query *even*(T). Then this is translated into the statements

parity=1
add(tmp(X)) :- T(X)
if parity=1 then return *true*
else return *false*

The active database rules shown in figure 2 are instrumental in constructing the answer to the query. Assume R3 has higher priority than rule R2

As many instances of R1 will be placed on the stack as there are tuples in the relation T. As rules are removed from the stack for execution, they toggle the parity variable. We can thus determine the answer to the *even* query and this idea can be extended to performing tests such as $|T1| = |T2|$ etc.

R1	R2	R3
On add to relation tmp	On E2	On E3
If true	If parity=0	If parity=1 and flag=true
Then fire E2 ; fire E3;	Then parity=1;	Then parity=0
flag=true	flag=false	

Fig. 2. Rules for the parity query

Note that although this tuple oriented activation of events is deterministic, it would not be so if the rules were able to retain parameters containing information on how they were activated (e.g if the first instance of a rule was triggered by the tuple 'Fred'). We would then have to assume tuples to be accessed in some predefined order if we wished to retain determinism.

By the assumptions made in [11], it is not possible to express a query such as *even* using several major active database prototypes, yet we have shown how it can be done using tuple oriented triggering. In [13], it is shown how to express the query using the production rule language RDL1, which uses condition-action rules, but this language is not deterministic however.

Role of Events The preceding discussion raises the question of just what the role of events is in the 0-1 language. In section 3 they were primarily used as a convenient mechanism for controlling the activation of other rules. It is possible, however, to achieve the same functionality just with Condition-Action rules, provided we carefully choose our semantics. We will consider a C-A rule to be activated if its condition makes a transition from false to true. Suppose we want to simulate the E-C-A rule $R1$ by a C-A rule $R1^{'}$.

$R1$	$R1^{'}$
On E	
If C	If C and flag=true
Then A	Then A

For rule $R1^{'}$ to be triggered, we perform the action 'flag=false;flag=true'. It is a moot point whether we've gained anything by doing this, since this method of activation is an event in everything except name. Indeed we may even have lost power, since it is unclear whether C-A rules with this semantics may compute the *even* query.

First Order Extensions We now briefly consider the implications of adding the ability to execute relational operations to our 0-1 rule language. Suppose that our 0-1 rules with queue can issue a first order query to a relational store and can assign the result of a first order query to the store (call this rule language 0-1_{FO}). We can then claim that this system is equivalent computationally to the relational machine used by [11]. This provides an interesting perspective, since the results in [11] show that the active database system HiPAC [8] can be modelled by a relational machine. Therefore our 0-1_{FO} active rule language would have the ability to 'simulate' this complex prototype system, subject to [11] 's simplifying assumptions.

6 Conclusions and Future Work

We have examined some of the key features in an active database system and have seen that they can have a considerable impact on expressiveness. This is summarised in Figure 3 (the question marks in the set row indicate the problem is open at this time). We have also seen that even simple rule languages can

	Standard Configuration	Unrestricted Coupling	Cumulative Event
Set	Past Temporal Formula	?	?
Stack	Regular	Rec. Enum.	Rec. Enum.
Queue	Rec. Enum	Rec. Enum.	Rec. Enum.

Fig. 3. Rule Power Summary

be very powerful computationally in the presence of features such as a queue or complex events. This potential power can be used effectively for database queries, provided events can be generated in a sophisticated manner. More importantly, this power implies that many questions in regard to active behaviour will be undecidable.
In our future work, we plan to investigate the following directions:

- The effect of allowing conditions to look at more than one version of the database
- The effect of further types of meta rules
- The features needed in order to increase the power of a 0-1 rule system with a set
- The computational complexity of certain configurations of rule sets
- Investigating the relationship between the systems presented and various 'exotic' grammar types such as ordered grammars, timed grammars and grammars with control rules

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, Lecture Notes in Computer Science 985, pages 165–181. Springer-Verlag, 1995.

3. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 566–577, Brisbane, Australia, 1990.
4. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 577–589, Barcelona, Spain, 1991.
5. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1):1–26, 1994.
6. C. Collet, T. Coupaye, and T. Svensen. Naos: Efficient and modular reactive capabilities in an object oriented database system. In *Proceedings of the 20th International Conference on Very Large Data bases*, pages 132–143, Santiago, Chile, 1994.
7. U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, pages 204–214, Atlantic City, 1990.
8. U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, 1988.
9. R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 455–468, 1991.
10. H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick. Managing rule conflicts in an active database. In *Proceedings of the 14th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Montreal, Canada, 1996.
11. P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proceedings of the 14th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 126–138, San Jose, California, 1995.
12. E. Post. Finite combinatory processes-formulation I. *Journal of Symbolic Logic*, 1:103–105, 1936.
13. E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In *Proceedings of The International Conference on Database Theory*, pages 205–222, 1988.