Organizing Long-Running Activities with Triggers and Transactions

Umeshwar Dayal*
Meichun Hsu†
Rivka Ladin‡

Abstract

This paper addresses the problem of organizing and controlling activities that involve multiple steps of processing and that typically are of long duration We explore the use of triggers and transactions to specify and organize such long-running activities Triggers offer data- or event-driven specification of control flow, and thus provide a flexible and modular framework with which the control structures of the activities can be extended or modified We describe a model based on event-condition-action rules and coupling modes The execution of these rules is governed by an extended nested transaction model Through a detailed example, we illustrate the utility of the various features of the model for chaining related steps without sacrificing concurrency, for enforcing integrity constraints, and for providing flexible failure and exception handling

1 Introduction

In many computer-supported activities, a user request often involves multiple "steps" of processing, each of which may be serviced by a different server, possibly on a different node of a distributed system For example, a purchase order may be issued from an inventory clerk, then passed to a manager who approves it, and then passed to an accountant who makes proper accounting entries We will call the processing of requests that involve multiple application steps and that might be of long duration *long-running activities*

Traditionally, the control flow between processing steps was embedded in the application programs Thus, each application had to reimplement the control flows, the synchronization, and the handling of failures between steps

Various workflow models have been proposed to explicitly specify the flows of control in long-running activities, instead of embedding them in application programs (e g [Zism78], [DZ81], [Barr82], [CC82], [LR83], [BP83], [WL86]) This explicit representation of control flow facilitates understanding the structure of the long-running activities However, the early workflow models do not address the problems of data sharing, persistence, and failure recovery

Conventional database management systems (DBMSs) provide transactions as the atomic units of work An activity that runs as an atomic transaction is guaranteed to satisfy the concurrency atomicity, failure atomicity, and permanence properties However, executing a long-running activity as a single transaction is not strictly necessary in most cases, and can significantly delay the execution of short transactions For example, if purchase order processing is run as a single transaction, locks on the inventory records and the budget records may be held for a long time, severely limiting database concurrency When these steps involve several distributed servers, commit processing is also expensive, and the transaction can run only when all servers are available simultaneously

One approach to handling a long-running activity, therefore, is to have each step run as a transaction, thus, the long-running activity corresponds to multiple transactions In conventional transaction processing (TP) systems, the sequence of transactions to be run in response to a user request is embedded in application programs, and the TP system provides primitives for chaining these transactions (e g [McGe78]) However, there is no system support for handling failures or exceptions across the steps of the long-running activity Also, embedding the control structure in the application programs makes it difficult to dynamically modify the flow of control

*Address Digital Equipment Corp , One Kendall Square - Building 700, Cambridge, MA 02139, dayal@crl dec com

†Address Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, hsu@harvard harvard edu

‡Address Digital Equipment Corp , One Kendall Square - Building 700, Cambridge, MA 02139, rivka@crl dec com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise or to republish requires a fee and/or specific permission
© 1990 ACM 089791 365 5/90/0005/0204 $1 50

Several extended transaction models to support long-running activities have been proposed (e g [GS87], [KR88]) These models are primarily based on explicitly expressed control flow The control flow specifies the sequence of steps that must be executed to process a user request of a certain type Each step is executed as a transaction One salient feature of these models is the provision of an automatic *compensation* capability which offers failure atomicity for the user request a fixed compensation transaction is defined by the application programmer for each step of the activity

In this paper we explore more flexible ways for expressing control flows (including for expressing compensating steps) for long running activities We add the use of *rules* to organize long-running activities With the introduction of rules, the system can automatically trigger additional processing when some event or situation of interest occurs The triggered processing may occur in the same transaction as the triggering transaction or in a separate transaction Unlike the previously proposed extended transaction models, which are based on a fixed control sequence and a rigid compensation policy, rules are *data- or event-driven* This allows for a more modular specification of the control flow, it allows integrity constraints, exception conditions, etc to be defined separately from the "routine" steps of the application, and it enables the system to be more easily modified or extended to react to events that were not anticipated when the application programs were first developed

Consider the following example database rule [1]

```
Event  Update Qty_On_Hand(item)
Condition  Qty_On_Hand(item) < Limit(item)
Action  Submit_Order(item)
```

The person responsible for inventory control may have written this rule, and incorporated it into the database There may be several types of user-submitted updates that affect the quantity-on-hand of an item Writing the rule once obviates the need for the inventory controller to pin down all the possible sources of requests that may affect quantity-on-hand and to ensure that all these requests are run with a proper control flow specification that includes the appropriate Check_Quantity_On_Hand step and the Submit_Order step If additional applications that also update quantity-on-hand are added in the future, these new applications need not be aware of how inventory is reordered Therefore, this rule-based approach to expressing workflows may be preferred in some application scenarios

There have been a few proposals for augmenting a database with triggers or rules [Eswa76, Ston86, Syba87, KDM88, Sell89], primarily for defining integrity constraints or alerters, or for inferencing over a database However, these models typically consider only triggers caused by database updates, except for the ETM model [KDM88], which includes triggers caused by "abstract" events signalled by external processes Also, they have primarily focused on *immediate* or *deferred* modes of execution of triggers, in which the triggered actions execute within the same transaction as the triggering updates

In [Daya88a,b], we have proposed Event-Condition-Action (ECA) rules as a general formalism for modelling the functionality of an active DBMS Integrity constraints, access constraints, policies for handling derived data, alerters, etc , can all be expressed as ECA rules The model supports arbitrary events, not just database updates Also, it provides fine control over the execution of triggered actions relative to the triggering transactions the triggered actions may be coupled within the triggering transaction or decoupled in a separate transaction The detailed semantics of ECA rules are described in [DBM88] In [HLM88], we describe an *execution model*, based on an extended nested transaction model, to govern the concurrency and recovery properties of transactions with triggered ECA rules An architecture for an active DBMS that supports this model is described in [MD89], and optimization techniques are described in [RCBB89] A simple prototype implementation of the basic model is described in [Chak89]

In this paper, we describe the use of ECA rules and our execution model for specifying and organizing long-running activities In particular, we show the utility of the various coupling modes to chain related steps without sacrificing concurrency, to enforce integrity constraints, and to provide flexible failure and exception handling We extend the execution model of [HLM88] to take into account priority and sequencing requirements We introduce language constructs for specifying rules Finally, we illustrate the power and flexibility of the model through an example of a hospital information system We do not claim that the entire application should be implemented by rules Rather, the application still consists of procedures and transactions It is the control flow between them that we express by rules Also, we use rules to check integrity constraints or exception conditions and to invoke countermeasures

Section 2 contains a brief review of related work In section 3, we describe our ECA rule model and its execution semantics Section 4 gives language constructs for specifying rules Section 5 gives the example Section 6 summarizes our results and points out future directions

2 Related Work

Numerous workflow models for office (or business) procedures have been described in the literature [Zism78, DZ81, CC82, Barr82, Zloo82, LR8383, WL86] All these models include some notion of a task (sometimes called a procedure, action, or step) The flow of control between tasks is specified typically by augmented Petri nets or triggers

In the augmented Petri net models [Zism78, DZ81, LR83], tasks are represented by position nodes of the Petri

[1] Following the specification in [HLM88], a rule consists of a triggering event, a condition to be evaluated, and an action to be executed if the condition is satisfied This specification is further detailed in the next sections

net, and flows between tasks by transition nodes (i e , each transition node has one or more input position nodes and one or more output position nodes) Transition nodes may also have preconditions (predicates over the database) defined for them When a task completes, it marks its corresponding position node When all position nodes input to a transition node are marked, and its precondition is satisfied, the transition may fire The firing of a transition causes the tasks on its output positions to execute When these tasks complete, they mark their position nodes, and so on

The trigger models [CC82, Zloo82] allow more flexible workflows triggers are condition-action pairs, where the conditions depend on time or on database states, and the actions are tasks When a trigger's precondition becomes true, the tasks in its action are executed These tasks may, by updating the database, cause other triggers' conditions to become true, and so on

These workflow models do not precisely describe their atomicity, concurrency, and recovery semantics It is not clear whether all fired tasks execute in the same transaction as the firing task, whether each task executes as a separate transaction, or whether a task may itself be composed of several related (chained) transactions Also, it is not clear what happens when a task aborts do "downstream" tasks also abort? The "script" model of TAXIS [Barr82] generalizes Petri nets by making each transition a transaction (rather than an instantaneous event), but this is rather inflexible We want the actions fired by a transition sometimes to execute within the firing transaction, and at other times to execute separately

In [GS87], the notion of *saga* is proposed as a model for long-running activities The application programmer specifies that the activity is composed of a sequence of transactions $T_1, T_2, \ldots, T_n$ When T_i is finished, it is committed and then T_{i+1} is invoked If T_k fails, then T_k is aborted and the system automatically invokes compensating transactions $C_{k-1}, \ldots, C_1$, in that order A compensating transaction C_i logically compensates for the updates performed by T_i In general, the specification of the compensating transactions must be provided by the application programmer

The saga model is generalized in the *migrating transaction* model [KR88], where concurrent execution of component atomic transactions can be specified, thus generalizing the control flow from a linear sequence to an acyclic graph In addition, invariants on the database state that must be maintained to ensure the feasibility of running compensating transactions, can also be specified In [Reut89], *contracts* are proposed as an extension to the model of migrating transactions The steps of a contract may be arbitrary sequential programs, not necessarily transactions The manner in which each step is executed and the control flow among the steps are specified as part of the contract definition

These models provide fixed workflow and a rigid compensation policy Also, because they are not based on nested transactions, they neither support nesting nor concurrency within a step

Several extended transaction models were proposed to deal with long-running cooperative activities such as engineering design The NT/PV model [KS90] allows a long-running activity to be broken up into a partially ordered set of subactivities The split transaction model [PKH88] allows a user to commit changes to some of the data objects that his transaction updated, while the remainder of the transaction continues to operate on other objects The transaction groups model [FZ89] provides an extended set of non-restrictive and communication (notification) modes to faciliatate sharing of data among collaborators These models provide more sharing than the traditional transaction model, but they do not provide dynamic workflow, nor do they address the issue of compensation

Several systems have proposed or implemented triggers In System R [Eswa76], Postgres [Ston86] and Sybase [Syba87], triggered actions typically occur in the same transaction as the triggering updates They are either executed immediately (e g , *triggers* in Sybase and System R), or deferred to the end of the triggering transaction (e g , *assertions* in System R) In Postgres, triggered actions can also occur on demand (i e , lazy evaluation) However, these triggering models are not suitable for organizing long-running multi-transaction activities

In AP5 [Cohe86] and CPLEX [HC88], a class of triggers, called *automation rules*, are defined which are executed as separate transactions after the triggering updates are committed This automation rule construct was motivated by the need to organize long-running activities in software engineering The behavior of the automation rule corresponds to one of the coupling modes offered in the general model described in this paper

3 The Model

In this section, we describe our rule model, and its execution semantics The language constructs for specifying rules and transactions are outlined in the next section

3.1 The ECA Rules

An ECA Rule consists essentially of three parts an *event*, a *condition*, and an *action* When the active DBMS detects that an event has occurred (is *signalled*), it *evaluates* the condition, if the condition is *satisfied*, it *executes* the action

Events may be primitive or composite Primitive events are database operations (e g , update, end_transaction) that can be detected by components of the DBMS, or abstract events that are signalled by external processes (e g time events signalled by a timer process, hardware failures signalled by a diagnostics routine) Formal parameters may be defined for each event, these are bound to actual values at the time the event occurs (e g , the actual tuples inserted, deleted, or modified by a database operation), and these bindings are reported in the event signal Composite events are constructed from the primitive events using algebraic operations, which also prescribe

how composite events' signals are constructed from those of the constituent primitive events For example, a sequence of database updates performed by a transaction can be defined as a composite event whose signal contains the net set of tuples inserted or deleted by the sequence of updates

A condition is a query (closed or open) over the database, it may also refer to values (parameter bindings) in the event signal The condition is satisfied if the query evaluates to *true* or returns a non-empty answer

An action is a program that may include database operations and/or external operations (e g , invocation of an application program, notification of a user, waiting for input from a user) The context in which the action executes includes the database state, the values in the event signal, and the result computed by the condition The action may also signal events

Many other rule models (e g , [Forg82], [Ston86]) do not separate the event and condition parts of rules as we do The separation is useful Events and conditions play different roles events specify *when* to check if a rule should fire, conditions specify *what* to check This separation makes it possible to trigger actions based on application signals or specific operations, not just on database predicates It supports asymmetric rules, e g , to maintain the invariant Mike's salary = Bill's salary, we can define two separate rules if Mike's salary is updated, then update Bill's, but if Bill's salary is updated, abort the transaction This is impossible to do without events The separation also makes optimization possible check conditions when specific events occur, not "always" Finally, it enables flexible execution the condition can be evaluated at a different point in a transaction from the point at which the event occurs, or even in a different transaction

The separation of condition and action parts enables flexible execution and facilitates optimization by packaging the pure query part of the rule

3.2 Coupling Modes and Transactions

Most other rule or trigger models do not specify whether the triggered actions execute in the same transaction as the triggering transaction or in a separate transaction (In Postgres [Ston86], the choice is made by the system) Our model gives the rule definer fine control over where the condition and action of a rule are executed relative to the triggering transaction The rule definer specifies a coupling mode between the event and condition, and another (possibly different) coupling mode between the condition and action, these determine where transaction boundaries should occur Three coupling modes are supported *immediate*, *deferred*, and *decoupled*

Suppose that transaction T executes an operation that signals event E for rule R, which has condition C and action A If the E-C coupling mode is *immediate*, then C is evaluated within T immediately when E is detected, preempting the execution of the remaining steps of T If the mode is *deferred*, then C is evaluated within T, but after the last operation in T and before T commits If the mode is *decoupled*, then C is evaluated in a separate transaction The same options are available for the C-A coupling The deferred mode is especially useful for checking integrity constraints The decoupled mode is useful for breaking up a long cascading sequence of triggers into short transactions

When the condition or action is decoupled, the *triggered* transaction (call it T') can execute concurrently with the *triggering* transaction T If serializability is our only correctness criterion, then T' might be serialized before T This, however, may violate *causality* T may see the results of T' Also, T may abort after T' committed For applications where causality is important, the model allows the rule definer to specify that the decoupled transaction is *causally dependent*, i e , T' must be serialized after T, and T' can commit only if T commits As we shall see, decoupled, causally dependent transactions are the basis for workflow control (chaining related transactions) in a long-running activity Note that sometimes causally independent transactions are desirable For example, suppose we want to trigger a rule that writes a record in the security log whenever a user accesses some data object, we want the security log record to be written irrespective of whether the original transaction that accessed the object commits or aborts In this case, we can specify the coupling mode to be decoupled, causally independent

3.3 An Extended Nested Transaction Model

We use an extension of the nested transaction model described in [Lisk85] and [Moss81] to govern the execution of rules under the various coupling modes A *nested transaction* is a transaction that is started from inside another transaction (the *parent transaction*) In the standard nested transaction model, nested transactions are created explicitly by the parent transaction Also, the parent transaction is suspended until the nested transaction terminates (commits or aborts) We will call such nested transactions *subtransactions*, and we will introduce three additional kinds of nested transactions

We use nested transactions to model rule execution as follows If the triggering event occurs inside a transaction (the *triggering transaction*), the system scheduler, or the transaction manager, creates a nested transaction (the *triggered transaction*) of the triggering transaction to execute the rule In general, we also allow triggering events to be signalled by processes or humans outside transactions, in this case, a top transaction is started to execute the rule

The triggered transaction T' does the following first, it creates a subtransaction C that evaluates the condition part of the rule, then, if C commits and the condition is satisfied, it starts another nested transaction A that executes the action part of the rule If the E-C coupling mode is immediate, then T' is a subtransaction of the triggering transaction T (and C is a subtransaction of T') If the C-A coupling mode is also immediate, then A is also a subtransaction of T'

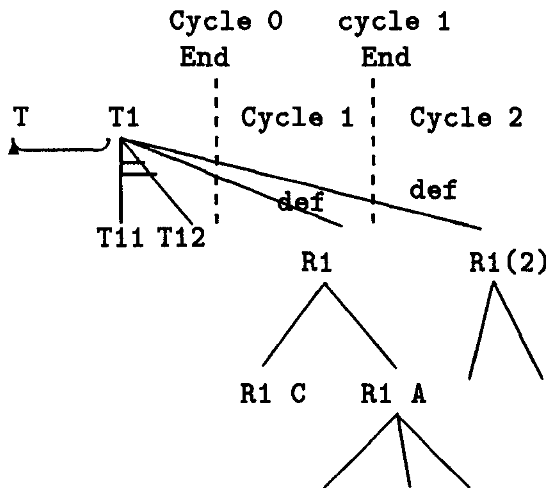

Figure 1 A Transaction Tree

To accommodate the deferred and decoupled modes we extend the nested transactions model to allow three more types nested transactions

First, we allow the creation of *deferred subtransactions* whose execution is explicitly delayed until the end of the user's top transaction T and before any deferred subtransaction is executed, a point we shall refer to as the *cycle-0 end* When T reaches its cycle-0 end, a deferred subtransaction is started, and runs as a proper subtransaction of T If several deferred subtransactions are created before T reaches its cycle-0 end, then all these subtransactions are started as concurrent subtransactions in *cycle-1* at cycle-0 end If the processing of subtransactions in cycle-1 causes more deferred transactions to be created, the latter are started when all subtransactions in cycle-1 have finished, and are started as concurrent subtransactions of T in Cycle-2 The cycles of execution of T continue until the last cycle finishes in which no more deferred subtransactions are created Like a regular subtransaction, the commit of a deferred subtransaction is conditional on its triggering transaction committing through the top Thus, if the E-C coupling mode is deferred, then the triggered transaction T' that executes the rule is started as a deferred subtransaction Similarly, if the C-A coupling mode is deferred, then the action A is executed in a deferred subtransaction For example, the place of $R1$ in the transaction tree is shown in Figure 1, where $R1$ is a deferred subtransaction triggered from within subtransaction $T12$

Second, we allow top transactions to be started from another transaction Such a *nested top transaction*, unlike a subtransaction, has no special privileges relative to its parent, for example, it can't read an object modified by its parent Furthermore, the commit of a nested top transaction is not relative to its parent but rather independent Note that we don't constrain the serialization order of a nested top transaction relative to its parent A nested top transaction will be represented by its own tree When the E-C coupling mode is decoupled, then the triggered transaction is executed as a nested top transaction (similarly for the C-A coupling mode)

Finally, we allow a *causally-dependent-top-transaction*, *CDtop* for short, to be spawned from inside another transaction This type of "nesting" imposes the causality constraint Let T be the user's top transaction and T' be the causally-dependent transaction created either by T or by one of its descendants Then T' is serialized after the execution of T A nested CDtop transaction, unlike a nested top transaction, is *commit – dependent* on the committing of its parent Thus, T' commits only if its parent is committed through the top Nested CDtop transactions will be represented by their own tree with commit dependency link depicted by a dotted line Note however, that aborting a CDtop transaction has no effect on its parent Figure 1 depicts the execution of $T1$ as a nested causally dependent top transaction of T It is important to note that CDtop transactions whose triggering transactions have committed must be scheduled for execution Also, since users are not aware of the triggered transactions, they cannot be expected to restart them Therefore, CDtop transactions that are interrupted by a system failure should be automatically restarted as part of recovery Algorithms for implementing this extended nested transaction model are discussed in [Chak89]

Most rule models do not clearly describe what happens when the execution of a rule fails (i e , terminates abnormally) Our model supports two options either the execution of that rule alone is aborted and the triggering transaction continues from the point where the rule was triggered, or the whole transaction in which the rule execution occurs is aborted Note that in the explicit subtransaction model, the failure of a subtransaction results only in aborting the failed subtransaction The parent of the aborted transaction is waiting for its child's termination at a point at which it could take appropriate measures In our model, however, rules may be executed as subtransactions anywhere in the program without the parent explicitly starting them or even being aware of their execution As a result, the normal failure semantics must be extended to allow a subtransaction to request that the top transaction and all its descendants (excluding nested top transactions) be aborted

The nested transaction model is appropriate for modelling the execution of a user transaction coupled with the execution of rules for three reasons First, its structure accommodates nicely the hierarchical relation between the triggering transaction and the triggered activities, second, it allows a collection of rules to run concurrently within a transaction, and finally it supports the modularity property of rule execution

3.4 Ordering Constraints and Context Switching

An event may trigger more than one rule In our model, these rules are executed as concurrent (immediate, deferred, or top) nested transactions of the triggering trans-

action As concurrently running siblings, their execution is serializable This is in contrast to other rule models in which rules are executed sequentially Production rule systems [Forg82] use conflict resolution to select one rule out of a set of candidate firable rules in each inference cycle If the condition parts of the other rules are still satisfied at the start of the next inference cycle, then these remain candidates for firing Postgres [Ston86], on the other hand, assigns priorities to rules and executes only the highest priority rule that is firable, disabling the rest

To control the serialization order of concurrently executed rules (i e , to simulate conflict resolution), our model supports three mechanisms

First, priorities can be assigned to rules Instead of numerical priorities (which we believe compromise modularity and are difficult to use), we support priority categories or *urgencies* high, medium, low The system spawns the triggered transactions in priority order, starting with the highest priority rule All rules with the same priority are spawned concurrently

Second, for deferred subtransactions, we have described the cyling mechanism This mechanism is especially useful for integrity rules whose action parts may themselves perform operations (e g , to patch the database state) that trigger the integrity rules again Cycling ensures that the first "generation" of actions is complete before the next generation starts, this assumption might make it easier to write the rules The cycling mechanism interacts with the priority mechanism within each cycle, the subtransactions are executed in priority order [2]

Third, for decoupled rules, we support a *pipelining* mechanism We say that a decoupled transaction T' triggered by transaction T satisfies the pipelining property if for all transactions Ti that are serialized before (after) T, any decoupled transaction Ti' triggered by Ti is serialized before (respectively, after) T' Thus, suppose a decoupled rule is used to display a moving target's position on a screen every time the position is updated If many update transactions occur in a short period, several decoupled display actions may be queued For the display to reflect the correct sequence of updates, the display actions must be pipelined This pipelining property is probably stronger than what is required in some applications It can be weakened by associating the pipelining constraint with rule classes, and pipelining only within a class By default, each rule forms a pipeline class of its own

To select some rules to fire while preventing others from firing, we provide context switching mechanisms The most straightforward context mechanism (provided in any rule model) is to set "flags" in the database when one phase of processing (viz , one context) is complete (e g , Initial_ Diagnosis_Completed = True), the condition parts of rules are then augmented to test the flags However, this makes condition evaluation more expensive An alternative is to signal special events (e g ," Initial_Diagnosis_Completed"), the event parts of rules are then augmented with these special context events, so that the rules for the next context are triggered only when the previous-context event occurs This makes event detection more expensive A third alternative provided in our model is to use explicit *enable* or *disable* operations, which can be invoked from action parts of rules or from application transactions Instead of referring to rules by number or some other identifier, they can be typed (i e , organized in classes) and can also have attributes An enable or disable command can then specify all rules belonging to a class or all rules that satisfy some predicate on their attribute values (e g , enable all high priority integrity rules where Purpose = "Check_Allergy_Conflicts")

3.5 Discussion

In the model described above, we can consider the action tree of a top transaction as a *unit of control* The action tree consists of a top transaction, and all its committed and on-going nested subtransactions and nested CDtop transactions (We do not consider the nested *causally independent* top transactions spawned by a top transaction as part of the action tree)

For the purpose of this discussion, we assume that a program that creates a top transaction is given a *handle* for the transaction After a top transaction is created, the program may *query* the status of the top transaction by presenting its handle to the system The program may also ask the system to *cancel* the top transaction

The status of a top transaction is represented by the action tree spawned by the top transaction By preserving the action tree, the system can allow the users to query and display the status of the different steps of the activity

Cancelling a top transaction is more complicated If the cancel request is issued before the top transaction commits, then the required effect can be achieved by aborting the top transaction that causes the entire action tree to be aborted However, if the top transaction has already committed (although parts of the action tree spawned from nested CDtop transactions may still be in progress), then the system will have to reject the cancel request, because the committed transactions cannot be aborted To support cancellation after the top transaction has committed requires an additional system facility that invokes compensation transactions

To guarantee the recovery of an active action tree after system failures, the system must recover the events signalled by transactions that spawned uncommitted nested CDtop transactions Therefore a transaction commits only after its database updates, *and* the events signaled by it, are stably captured With these signals recovered, the system can restart the activities of the action tree, and ensure the completion of the tree's execution We distinguish between *recoverable* and *non-recoverable* events All events triggered by database updates are recoverable

[2] We could have chosen the opposite semantics allow transactions from later cycles to join the current concurrently executing set if their priority is the same as or higher than that of the current set In the absence of "real-life" examples that favour the latter policy, we chose the one with simpler semantics

Temporal events, on the other hand, may be recoverable or not. Upon recovery, events that are signalled by committed transactions and for which the necessary action was taken before the failure, are recovered independently of whether they are recoverable events or not. Events that are signalled by committed transactions, and whose processing had not completed before the failure, are signalled only if they are recoverable events.

4 Language Constructs

Specifying a rule consists of specifying its event, condition, and action parts, the E-C and C-A coupling modes and any causality constraints, priority, optional rule class(es), and any other attributes. We describe key constructs of a language for programming with rules. A detailed specification of such a language is beyond the scope of this paper. However, the examples in the next section illustrate the use of some of the constructs.

First, the primitive event types of interest and their associated signal parameters are defined. Denote event E with signal parameters X as E[X]. For any type of object in the database, any legal operation on the object can be specified as an event. Thus, for relations, the primitive events are retrieve, insert, delete, and modify. The signal parameters for these events are the sets of tuples operated upon: for retrieve, the set *Retrieved_Tuples*, for insert, the set *Inserted_Tuples*, for delete, the set *Deleted_Tuples*, and for modify, the two sets *Deleted_Tuples* and *Inserted_Tuples* that give the net effect on the relation. For operations on abstract data types, the signal parameters are the parameters of the operations. Since operations generally are not instantaneous, we can specify whether the event of interest is the beginning of the operation or the end (the default is the end). Thus, it is possible to trigger rules just when an operation starts or when it finishes (in some rule languages [Eswa78], this distinction is made by labelling the event BEFORE or AFTER). Transaction events such as *BOT* (begin top-level transaction), *EOT* (end top-level transaction), *Commit*, *Abort*, *Prepare*, may also be specified. For example, EOT is signalled by the transaction manager whenever execution returns to the root of the transaction tree and the top transaction itself has no further operations to execute. The signal parameters for these events consist of the transaction identifier. Other (*abstract*) events may be defined together with their signal parameters, these typically are explicitly signalled from applications or actions.

Composite events may be programmed from the primitive ones. The language includes the following constructors: (a) *disjunction*: the event (E1 | E2) is signalled when either E1 or E2 is signalled, (b) *sequence*: the event (E1 , E2) is signalled when an occurrence of E1 is followed by an occurrence of E2, and (c) *closure*: the event (E^+) is signalled when E has occurred a non-zero number of times. For definiteness, (E^+) must be followed a termination event. The signal parameters for the composite events are derived from those of their constituent events. For expressing their derivations, it is convenient to think of the signal parameters as sets of tuples (relations), then, the derivations can be expressed in relational algebra. For disjunction and sequence, the signal parameters typically are the outerunion of the constituent events' signal parameters. For closure, we want to aggregate the signal parameters of each occurrence of the constituent event. For example, for (Modify*), we want the *net Inserted_Tuples* and *net Deleted_Tuples*, not just the union of the sets for each Modify operation.

Conditions and actions are programmed in a typical data manipulation language, embellished with commands for spawning (sub)transactions (*begin_transaction*, *end_transaction*), spawning top level transactions (*begin_CDtop*, *end_CDtop* for causally dependent, and *begin_top*, *end_top* for causally independent), signalling events (*signal E[X]*), and enabling and disabling rules (*enable rule_expression*, *disable rule_expression*, where rule_expression is of the form {*r in rule_class where predicate*}). Also, with each of these begin commands, we can associate a priority category (the default is "medium"), and with *begin_transaction* we can associate the action to be taken on failure (*abort on fail* or *abort_top on fail*, the default is *abort on fail*).

Since we treat *EOT* as an event, and have language constructs for spawning nested transactions, it is unnecessary to explicitly specify the coupling modes. The immediate mode is expressed by enclosing the condition (and/or action) within *begin_transaction* and *end_transaction* brackets. The deferred mode is expressed by including *EOT* in the event part of the rule. The decoupled mode is expressed by enclosing the condition (and/or action) within the *begin_top*, *end_top*, or *begin_CDtop*, *end_CDtop* brackets. Furthermore, for programming the rules, we also do away with the separation between the condition and action parts. To express that they are decoupled, we can (as above) use the appropriate *begin*, *end* brackets. When the C-A coupling is immediate, it is unnecessary to start a new subtransaction for the action part, a smart optimizer can then consider them together.

The structure of a rule then is

```
RULE identifier
CLASS
<other attributes>
ON event
DO program
```

To spawn concurrent subtransactions, we need a parallel construct. The following program will generate as many concurrent subtransactions as there are members of X.

```
parfor x in X
      begin_transaction

      end_transaction
endparfor
```

The *parfor* loop will block until all spawned subtransactions terminate

To spawn concurrent top transactions, we do not need the *parfor* construct, because the spawner does not wait for the spawned top transactions to terminate (In fact, if they are CDtop, then having the spawner wait would result in a cycle of commit dependencies!)

These DML extensions are available not just for rule programming, but also for programming the applications

5 An Example — Hospital Patient Information System

This section outlines a patient information system for a hospital It models a long running activity that starts when a patient arrives at the hospital, continues through various stages of examination, diagnosis, and treatment until the patient is discharged from the hospital We omit many details However, our intent is to illustrate the following points

- the flexibility in modelling gained through the use of the immediate, deferred, and decoupled modes of the model,
- the use of decoupled top level transactions to break a long activity into short transactions that are chained together through causality,
- the use of event signals and shared state information (recorded in a database) to trigger transactions, rather than executing the transactions in some fixed, predefined order (as is done in other models of long activities),
- the use of event- and state-dependent triggers to provide flexible response to exceptional situations (and thus dynamically alter the workflow), rather than some fixed compensation policy

The long-running activity consists of several constituent sub-activities that may be performed by different organizations

- a patient arrives at the hospital and is admitted,
- the patient is examined by an attending physician,
- the physician may prescribe a series of procedures (laboratory tests, radiology, consultation by specialists, surgery, etc), may prescribe medication, and may require that the patient be hospitalized, the physician may specify dates by which these procedures have to be performed,
- the laboratory facilities and personnel required for executing the various procedures prescribed by the physician have to be scheduled,
- the procedures have to be executed within the deadlines specified,
- if the patient is hospitalized, then a room has to be assigned to him/her, meals have to be planned and ordered, a daily regimen of medication and therapy has to be followed, etc
- as the results of the procedures become available, the physician can look at them, reexamine the patient, and prescribe further procedures; this might go on for some time,
- at any time, the physician may decide that the patient has recovered and discharge the patient,
- every procedure has to be accurately recorded for legal and administrative reasons and for purposes of billing

We assume that all pertinent information about the activity is recorded in a database that is shared by all the organizations involved Treating the entire long activity as a single, atomic transaction has severe performance and organizational implications Instead, we model each sub-activity by one or more transactions that read or update the shared database Clearly, there are ordering and synchronization constraints among the transactions (e g , a laboratory test cannot be performed until the laboratory equipment and personnel have been scheduled) The transactions communicate by signalling (raising) events (e g , by updating the database), which are monitored by the active DBMS and which may cause nested subtransactions or other top-level transactions to be triggered Also, many of these transactions are interactive, i e , require communication with a human We assume that there is an interface (perhaps mail- and forms-oriented) via which the user is notified of events of interest, and via which he/she can invoke operations or supply parameters for the interactive transactions

Note that we do not expect the entire application to be written as rules Rather, the application still consists of procedures and transactions (In a technologically advanced hospital, some of the procedures may themselves be written as rule-base expert systems, however, that is orthogonal to the functionality we are illustrating here) It is the control flow between them that we express by rules Also, rules are used to check integrity constraints or exception conditions and to invoke countermeasures

Information about a patient (e g , social security number, insurance policy, medical history, allergies) is recorded in a *folder* object in the database When a patient enters the hospital, an assistant invokes the Register_Patient procedure, which creates a new *admittance record* in the patient's folder For a new patient (i e , one for whom a folder does not exist), a folder is first created by eliciting information from the patient (perhaps on a form) The interaction with the user is executed in a separate, causally dependent top transaction to allow the transaction that checks whether this is a new patient or not to commit, this frees up the folder database so that other patients can be concurrently processed

After the patient is admitted, an interactive *examine* transaction is started, which presumably notifies the attending physician and waits for input from him/her This transaction also is spawned as a decoupled, causally dependent transaction (which may execute on a different node), so that the admittance activity can now be committed The causality constraints are necessary because

if the patient decides to "walk out the door" (i e, the transaction in which the Register_Patient procedure executes is aborted), the dependent transactions should also be aborted

The physician examines the patient, makes an initial diagnosis, and may then insert procedure_request records and medication records into the patient's folder, and also may signal a "hospitalize" event We now want the active DBMS to invoke a function that checks if any of the prescribed procedures or medication "conflicts with" the patient's allergies This check essentially evaluates an integrity constraint and is deferred to the end of the examine transaction, because individual procedures or medications might not conflict, but some combination of them might The constraint is expressed in the following rule Note that the event part of the rule is signalled by a composite event that consists of one or more updates of procedure_request and medication records, followed by the EOT event We assume that the signal for this composite event is accumulated in a prescription object whose type is a set of procedure_request or medication records

```
Rule 1 (Check conflicting prescription)
ON  {insert Procedure_Request (S  SSNo,
        R  Admittance_Record, D· Physician,
        Procedure, L  Lab, Needed_By  Date) |
    insert Medication (S  SSNo, D  Physician,
        R  Admittance_Record, M  Medicine) }* ,
    EOT
    [Prescription]
DO
  begin_transaction
  for F in Folder where S = SSNO(F)
    return Conflicts (Allergies(F), Prescription)
  endfor
  end_transaction
  begin_transaction
    notify (D, Conflicts),
    repeat
      get_input (D, op),
      case op
        ''end''  end_transaction
        ''Abort''  abort_top
        else  execute op,
      endrepeat
  end_transaction
```

The physician can decide to abort the transaction (in which case the entire transaction tree is aborted), or to perform some modifications to the original prescription These updates are executed within the action part of the rule At some point, the physician will end the subtransaction At this point, this execution of the rule is complete, and control returns to the top of the tree At the top, the integrity constraint rule is again triggered if there were additional insertions of procedure_request or medication records into the patient's folder

Figure 1 depicts the above activity for an "old" patient In this figure, T is the transaction that executes the Register_Patient procedure, and $T1$ is an examine transaction

When the examination is complete and there are no further conflicts, we assume that the "End_Examination" event is signalled (we omit the rule for this) Now, if any laboratory procedures were requested during the examination, then they must be scheduled The scheduling transactions may themselves be long running activities that are interactive and are probably executed at different nodes from where the physician's examine transaction was executed They are, therefore, spun off as causally dependent, top-level actions that execute concurrently with one another Also, timer events have to be set to go off on the date by which the procedure results are needed, these events are set by signalling requests to a timer process These semantics are captured in the following three rules In Rules 2 and 3, the composite event consists of an arbitrary number of insert Procedure_Request events followed by the End_Examination event; and the signal for this composite event accumulates the Procedure_Request records that were inserted (this set is named Requests) While each procedure must be separately scheduled, we want only one timer event to be set for each unique date by which one or more results are needed Hence, in Rule 2, these unique dates are computed by the "projection" of the set Requests on the attribute Needed_By, which is denoted Needed_By (Requests) Also, to allow the original examine transaction to commit after it raises the Timer_Request signal, the condition part of Rule 4 is evaluated in a separate transaction

```
Rule 2 (Request timer events)
ON {insert Procedure_Request (S  SSNo,
       R  Admittance_Record, D  Physician,
       P  Procedure, L  Lab, Needed_By  Date)}* ,
    End_Examination
    [Requests]
DO
  parfor N in Needed_By (Requests)
    begin_transaction
      Signal Timer_Request(S,R,D,N)
    end_transaction
  endparfor

Rule 3 (Trigger scheduling actions)
ON   Event as in Rule 2
DO
  for Q in Requests
    begin_CDtop
      Schedule_Procedure (S,R,D,P,L,Needed_by)
    end_CDtop
  endfor

Rule 4 (Executed by the timer process)
ON   Timer_Request (S SSNo, D  Physician,
         R  Admittance_Record, T  Date)
DO
  begin_CDtop
    if Today = T
      then Signal Timer_Event (S,R,D,T)
  end_CDtop
```

The Schedule_Procedure activities may update the database (e g, they insert Procedure_Scheduled records that tell the date, personnel, equipment, etc assigned to

the procedure), set timer events, notify lab personnel, etc

We might want to pipeline the Schedule_Procedure activities Suppose the physician requests two procedures for the same patient in two different examine transactions These trigger two decoupled Schedule_Procedure transactions, which might go to different servers It might be important to ensure that the these triggered transactions execute in the same order as the examine transactions that spawned them, so that the first procedure requested is also the first scheduled One way of accomplishing this ordering is by passing context ("check if procedure X has been scheduled") In a distributed environment, pipelining achieves the same effect with less cost

Whether pipelined or not, the Schedule_Procedure activities eventually trigger Execute_Procedure activities Communication between these activities also happens through rules, which we omit The Execute_Procedure activities update the patient's folder with Procedure_Result records As the results are written into the database, rules are triggered to notify the attending physician

```
Rule 5 (notification of procedure results)
ON  update Procedure_Result (PQ  Procedure_Request,
              Res  Result, Performed_On  Date)
DO
  begin_CDtop
    notify (Physician(PQ), Procedure_Result)
  end_CDtop
```

Actually, we might have a second rule that checks if the results indicate a life-threatening situation for the patient, then the physician is immediately notified This second rule will have to be of a very high priority, also, the action part of this rule should disable Rule 5 for this procedure request, since the physician need not be notified twice Since we allow Rule 5 to be disabled, we must make sure that the Execute_Procedure activity enables it

```
Rule 6 (high priority notification of results)
ON  Event as for Rule 5
DO
  begin_CDtop
    if Dangerous(Res)
    then
      begin_subtransaction (priority high)
        notify (Physician(PQ), Procedure_Result),
        disable Rule 5 for PQ
      end_subtransaction
  end_CDtop
```

If the procedure's results are not available by the date they were needed, then too, the physician must be notified by a high priority rule (which we omit)

Finally, we illustrate flexibility in "undoing" activities Suppose that, during one of his examine transactions, the physician signals the "hospitalize" event This triggers several (decoupled, causally dependent) concurrent activities (e g , assign and prepare a room, plan, order, and serve meals, schedule and carry out a daily regimen of medication and therapy) Subsequently, the physician may want to discharge the patient This may be thought of as aborting the hospitalization activity, and may require rolling back the triggered activities However, some of the activities may already have committed and cannot be rolled back, instead, they have to be compensated for by other activities Which compensations to execute may be state-dependent the room assignment and meal orders, which are made weekly (say), have to be cancelled for the rest of the week, medication and therapy may need to continue (on an out-patient basis), some procedures that had been requested or scheduled for dates after the discharge date have to be cancelled, other procedures may still need to be performed We cannot simply use a fixed compensation policy (e g , execute predefined compensating activities in the reverse of the sequence in which the "forward" activities committed) Instead, in our model, the physician signals the "discharge" event (perhaps during one of his examination transactions) This event triggers the desired "compensating rules" that evaluate conditions on the patient's history and status, and trigger the appropriate compensating activities (we omit the rules) This rule-based approach gives the needed flexibility

6 Summary

The problem of reliable control flow management for long-running activities has not been satisfactorily addressed so far Traditionally, the control flow is embedded in application programs Extended transaction models, such as sagas and migrating transactions, support sharing, persistence, and failure handling for long-running activities, but provide rigid control flow structures

In this paper, we propose to use triggers for organizing long-running activities Triggers offer data- or event-driven specification of control flow, and thus provide a flexible and modular framework with which the control structures of these activities can be extended or modified

We propose a model based on event-condition-action rules, coupling modes, and nested transactions to govern the execution of long-running activities In particular, we make heavy use of the extension that allows different steps of the activity to execute as decoupled, causally dependent transactions Additional extensions for restricting the possible serialization orders of concurrent transactions have been defined These include priorities and pipelining

We illustrate the use of the various features of our rule model for specifying long-running activities in a Hospital Patient System We show how event-driven spawning of decoupled transactions essentially allows the long-running activity to be carried out through a collection of atomic transactions connected via an acyclic graph We have not focused on automatic compensation, although our paradigm does not exclude it We have shown how rules can be used to specify compensation procedures that are situation-driven, rather than following a fixed compensation path

More work is required to develop a methodology for constructing long-running activities using our model Ad-

ditional applications must be examined to gain experience in this approach and to compare it with other proposed methods Future work should investigate abstractions for cooperative work, for which transaction semantics may be too strong

References

[Barr82] Barron, J "Dialogue and Process Diesign for Interactive Information Systems Using Taxis " *Proc SIGOA Conference on Office Information Systems, SIGOA Newsletter, Vol 3, Nos 1 and 2*, 1982

[BP83] Bracchi, G , and B Pernici, "SOS A Conceptual Model for Office Information Systems " *Proc Workshop on Databases for Business and Office Applications, ACM-SIGMOD Database Week*, 1983

[Chak89] Chakravarthy, S , et al , "HiPAC A Research Project in Active Time-Constrained Database Management Final Technical Report " Xerox Advanced Information Technology, Cambridge, Mass , July 1989

[CC82] Chang, J -M , and S -K Chang, "Database Alerting Techniques for Office Activities Management " *IEEE Transactions on Communications, Vol COM-30, No 1*, January 1982

[Cohe86] Cohen, D "Automatic Compilation of Logical Specifications into Efficient Programs " *Proc 5th International Conference on Artificial Intelligence*, August 1986

[Daya88a] Dayal, U , et al "The HiPAC Project Combining Active Databases and Timing Constraints," *Special Issue of Real Time Data Base Systems, SIGMOD Record*, 17, 1, March 1988

[Daya88b] Dayal, U "Active Database Systems " *Proc 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988

[DBM88] Dayal, U , A Buchmann, D McCarthy "Rules are Objects Too A Knowledge Model for an Active, Object-Oriented Database Management System", *Proc 2nd International Workshop on Object-Oriented Database Systems*, West Germany, September 1988

[DZ81] De Antonellis, V , and B Zonta, "Modelling Events in Data Base Applications Design " *Proc Intl VLDB Conf*, 1981

[Eswa76] Eswaran, K P "Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System " IBM Research Report RJ1820 August 1976

[Forg82] Forgy, C L "Rete A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem " *Artificial Intelligence*, 19, 1982 pp 17-37

[FZ89] Fernandes, M F and S B Zdonik, "Transaction Groups A Model for Controlling Cooperative Transactions," *Proc Workshop on Persistent Object Systems*, Newcastle, Australia, Jan 1989

[GS87] Garcia-Molina, H and K Salem, "Sagas," *Proc ACM SIGMOD Conf*, May 1987

[HC88] Hsu, M and T E Cheatham, "Rule Execution in CPLEX", *Proc 2nd Internationl Workshop on Object Oriented Database Systems*, West Germany, September 1988

[HLM88] Hsu, M , R Ladin, and D McCarthy, "An Execution Model for Active Database Management System," *Proc 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988

[KDM88] Kotz, A M , K R Dittrich, and J A Mueller, "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism," *Proc Conf on Extending Data Base Technology*, Venice, 1988

[KR88] Klein, J and A Reuter, "Migrating Transactions," *Future Trends in Distributed Computer Systems in the '90s*, Hong Kong, 1988

[KS90] Korth, H and G Speegle, "Long-Duration Transactions in Software Design Projects," *Proc 6th International Conference on Data Engineering*, Los Angeles, CA, February 1990

[LR83] Lin, W K , D R Ries, B T Blaustein, and R M Chilenskas, "Office Procedures as a distributed Database Application " *Proc Workshop on Databases for Business and Office Applications, ACM-SIGMOD Database Week*, 1983

[Lisk85] B H Liskov "The Argus Language and System " *Distributed Systems Methods and Tools for Specification* pp 343-430 Springer-Verlag, Berlin 1985

[McGe77] McGee, W C , "The Information Management System IMS/VS Part V Transaction Processing Facilities," *IBM Sys Journal*, Vol 16, No 2 , 1977, pp 148-169

[MD89] McCarthy, D , U Dayal, "The Architecture of An Active, Object-Oriented Database System," *Proc ACM SIGMOD*, Portland, Oregon, 1989

[Moss81] J Moss "Nested Transactions An Approach To Reliable Distributed Computing " MIT Laboratory for Computer Science, MIT/LCS/TR-260 1981

[PKH88] Pu, C , G Kaiser and N Hutchinson, "Split-Transactions for Open-Ended Activities," *Proc 14th International Conference on Very Large Databases*, Los Angeles, CA, August 1988

[Reut89] Reuter, A , "Contracts A Means for Extending Control Beyond Transaction Boundaries," Presentation at 3rd Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1989

[RCBB89] Rosenthal, A , U S Chakravarthy, B Blaustein, J Blakeley, "Situation Monitoring in Active Databases," *Proc 15th Int'l Conf on Very Large Databases* Amsterdam, August, 1989

[Sell89] Sellis, T K (editor), *ACM SIGMOD Record, Vol 18, No 3 Special Issue on Rule Management and Processing in Expert Database Systems*, September 1989

[Ston86] Stonebraker, M et al "A Rule Manager For Relational Database Systems " *The POSTGRES Papers* Univ of California, Berkeley, Ca Electronics Research Lab, Memo No UCB/ERL M86/85, 1986

[Syba87] Sybase, Inc *Transact-SQL User's Guide i/* 1987

[WL86] Woo, C C , and F H Lochovsky, "Supporting Distributed Office Problem Solving in Organizations " *ACM Transactions on Office Information Systems, Vol 4, No 3*, July 1986

[Zism78] Zisman, M D , "Use of Production Systems for Modelling Asynchronous, Concurrent Processes," in *Pattern Directed Inference Systems*, Waterman and Hayes-Roth (eds), Academic Press, 1978

[Zloo82] Zloof, M M , "Office-By-Example A Business Language that Unifies Data and Word Processing and Electronic Mail," *IBM Systems Journal, Vol 21, No 3*, 1982