Managing Overlapping Transactional Workflows *

Juha Puustjärvi[1], Henry Tirri[1] and Jari Veijalainen[2]

[1] Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 UNIVERSITY OF HELSINKI
Finland
e-mail: {puustjar, tirri}@cs.helsinki.fi
[2] VTT Information Technology
Multimedia Systems
P.O. Box 1203
FIN-02044 VTT
Finland
e-mail: Jari.Veijalainen@vtt.fi

Abstract. Workflow management techniques have become an intensive area of research in information systems. In large scale workflow systems modularity and reusability of existing task structures with context dependent (parametrized) task execution are essential components of a successful application. In this paper we study the issues related to management of overlapping transactional workflows, i.e., workflows that share component tasks and thus avoid redundancy in design. The notion of parametrized transactional properties of workflow tasks is introduced and analyzed, and the underlying implementation mechanism based on Event/Condition/Action (ECA) rules is discussed.

1 Introduction

Workflow management techniques have become one of the most exciting areas of research in information systems. The underlying concepts have been around in various forms for a long time, however, only recently the know how to implement commercial systems has been available.

Modern business applications are usually composed of independently designed components which are accessed concurrently by a large set of users. Workflow management can be seen as central techniques to coordinate and streamline *business processes*, which themselves are represented as *workflows*. On the other hand current organizational investments into data processing equipment and software are significant, and there is a growing interest to benefit as long as possible from the existing "legacy systems". In large scale workflow systems with 10,000 users linking together several thousand geographically distributed sites, modularity

* This work was done in ESPRIT LTR project TransCoop (EP8012), which is partially funded by the European Commission. The partners of TransCoop are GMD (Germany), University of Twente (The Netherlands), and VTT (Finland).

of design, reusability of existing task structures and context dependent (parametrized) task execution are essential components of a successful application. It is within this framework of efficient business process re-engineering that our research is inscribed.

Mohan et el [21] listed several key aspects in large scale workflow management design: failure handling, availability, navigational flexibility, replication and distributed coordination. Such aspects are clearly in the realm of *transactional workflows* where the application execution has to conform to a set of correctness constraints derived from the application domain. Since workflow management systems have become the top layer in the corporate information system architecture, we would like to add to this list of issues *modularity* and *task reusability* and focus here on mechanisms achieving these goals.

Workflow task reusability is an essential feature of real life workflow systems, and should be taken into account also in the development of such systems. Reusability allows one to avoid redundant design, which is an important aspect as large workflow applications are typically defined by combining existing applications. Consequently management of *overlapping workflows*, i.e., workflows that share one or more tasks, is an important topic to be addressed in transactional workflows.

Workflow management research is clearly related to the work done in developing advanced transaction models, especially for cooperative environments. Cooperative transaction models have been discussed in [18, 19, 26, 24, 15, 23] in the centralized case, and in order to better match the requirements of various modern database applications, more general transaction frameworks are developed in [9, 22, 33]. The ASSET system [3] provides transaction primitives for the specification of extended and cooperative transaction models. Some of the recent work also makes direct connections between workflow management systems and advanced transaction processing: workflows are treated as extended transactions in DOM systems [4, 13], and the Contract model [31, 32] provides task reusability and application specific concurrency control.

In this paper we will study the design of workflow management systems that support modular, overlapping workflows. In order to achieve this goal of increased task reusability we introduce *task execution modes* and the notion of *task integration*, i.e., we rely on *context-dependent, parametrized transactional properties* of the workflow tasks. By using different execution modes it is possible to combine tasks located at different sites to compose appropriate transactional units. In addition, by task integration we can bind together tasks executed at the same site to comprise appropriate units of atomicity or isolation. In this way we incorporate workflow notions similar to multilevel atomicity [20], and increase task reusability and workflow system throughput. As far as we are aware of, this notion of parametrized transactional properties to increase reusability of tasks has not been addressed before. The mechanisms to implement such properties are based on Event/Condition/Action rules (ECA rules) [5] which are defined on the global workflow schema (i.e., task parameters). The ECA rules are used to specify which synchroniztion operations are needed. Such rules are given as

part of the workflow specification, and used by the global level workflow manager module during the execution phase.

The remainder of this paper is organized as follows. In Section 2 we will introduce the components of workflow specification, and describe the underlying workflow system architecture. In Section 3 we illustrate our concepts through an extensive example by considering five workflows in a banking domain. These workflows are used through the paper as an example to demostrate our approach. Section 4 discusses the management of concurrent execution of interfering workflow tasks with ECA rules. The use of parametrized transactional properties for task execution modes and task integration issues are introduced in Section 5. Section 6 concludes the discussion.

2 The TransCoop Workflow architecture

2.1 Workflow specification and task variations

Full specification of a workflow (scenario) is a complex task.For our purposes only the following workflow specification components need to be described:

- The tasks T_i involved in a particular workflow.
- The transactional requirements of the workflow.
- The data flow between the workflow tasks.
- The execution structure of the workflow.

In this paper we are focusing on the first two aspects since they are directly related to task reusability. Data flow between tasks is determined by specifying the input and output parameters of the tasks, and the execution structure specifies the ordering constraints for the executions of the tasks, e.g., a task may not begin before a particular previously started task commits. Such constraints can be specified e.g., by triggers [7, 6] of which the ECA rules are a special case, by Petri nets [16] or by finite state automaton [1]. In the following, we will make no assumptions of the method the execution structure is specified with. For a detailed discussion on inter-task dependency enforcement in workflow context see [27].

In our terminology a *task* defines some unit of work to be done, which can also be shared by several different workflows. A task may be specified in a number of ways. However, we model only the aspects which are relevant from task's reusability point of view. In order to maximize task reusability a task is allowed to have several *task variations*. Task reusability and task variations can be illustrated by the component structure given in Figure 1.

The component structure consists of the two levels: *workflow level* and *component level*. The overlapping tasks in the *component set* represent the variations of tasks. The tasks of a workflow comprise a subset of the overall *component set*.

We make the distinction between a *basic task* and *task variations*. Basic tasks can be used by the workflow designer to integrate tasks in order to compose appropriate transactional units (e.g., for atomicity or isolation). This issue will

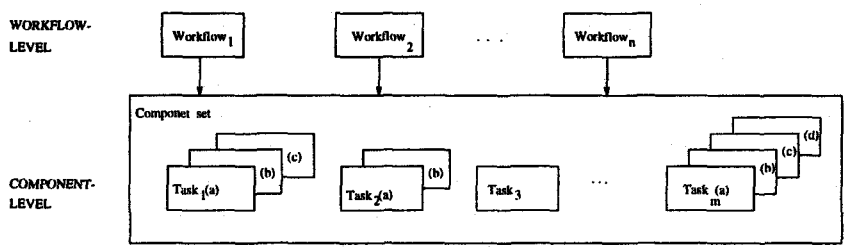

Fig. 1. Workflow component structure.

be discussed more detail in Section 5.1. *A task variation* is a modification of a task which matches better the requirements of a particular workflow. Task variation can be seen as a logical concept in the sense that the variation is specified as a parameter in the task call itself. In principle a task variation can be understood as workflow's view of the basic task, and it is analogous to view definitions in a traditional database setting. In addition task variations can also be used to compose appropriate units of atomicity or isolation. As an example consider a case where two tasks located at different sites will be integrated to compose a unit of atomicity. In such situation one uses task variations that provide the prepared state as an external state. We will return to these execution mode issues in Section 5.2.

2.2 Architectural considerations

Our architectural views for the workflow environment are influenced by the general reference architecture under development in the TransCoop project [8]. The TransCoop architecture is intended to provide a platform for transaction management support for cooperative applications in general. The architecture consists of both a specification environment (TSE) and a run-time environment (TRE) which use the services of an object server (TOS). Since we are concerned with a particular application domain, workflow management, the architecture presented is more specific than the original generic architecture. The overall modified architecture is illustrated in Figure 2.

We assume that the workflow environment consists of a global and a local level. The global level supports the overlapping workflows whereas the local level supports existing legacy applications and databases. Here we focus on the global level workflows only, and discuss briefly those architectural issues that are related to the topic of the paper, specification and execution of concurrent overlapping workflows. For more detailed discussion on architectural questions see [8].

A workflow designer specifies system supported workflows as well as the consistency constraints of the system. The *Workflow Specification Environment (WSE)* analyzes the specifications to determine whether such a workflow can be implemented. For example, if the specification includes a reusable task but such

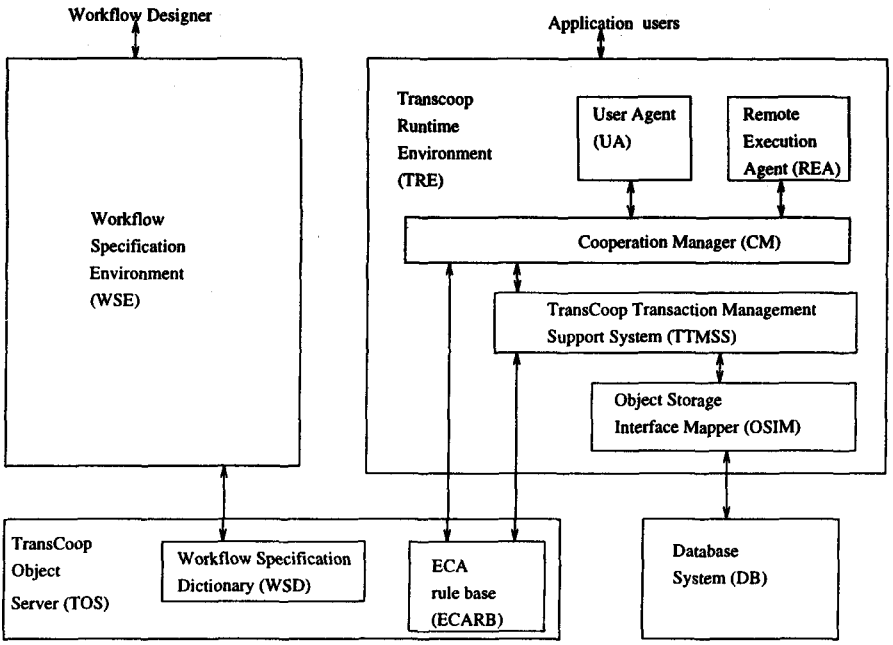

Fig. 2. Workflow system architecture.

a task is not found from the *Workflow specification dictionary* stored in the TOS object server, then the mismatch is reported to the workflow designer. If there are no mismatches the specification is stored in the *Workflow Specification Dictionary*. In addition to the individual workflow specifications a workflow designer can specify consistency constraints of the system. To enforce such a constraint WSE compiles the necessary set of Event/Condition/Action (ECA) rules which are stored in the *ECA rule base* in the object server. In a distributed case the ECA rules (and the possible locking data structures) are stored at the sites where the tasks in question are executed.

The *User Agent (UA)* module provides the interface between the users and the system. A user may be a human, an application program or a remote workflow or task request. When it receives a workflow call, it generates a *workflow specification* based on the *Workflow Specification Dictionary*. Then it sends non-local task requests to the UA's of the appropriate sites via the *Remote Execution Agent (REA)*. Local workflow specifications are passed to the *Cooperation Manager (CM)*, which is responsible that the execution is compatible with its specification. WM uses the services of the *TransCoop Transaction Management Support System (TTMSS)*. It can enforce correct sequencing of the tasks.

To ensure correct interleaving of concurrent workflows and that predefined constraints are not violated, the Cooperation Manager uses the ECA-rule base generated in the specification phase. If CM will not delay the operation or a task the necessary operations will be performed by the TTMSS.

3 An example of a set of overlapping workflows

As an illustrative example we will consider the following five workflows in a banking environment: *Client Credibility, Credit Account Request, Credit Card Request, Loan Request* and *Bill request.* In the figures describing these workflows we have denoted subworkflows (i.e., the tasks which may also be executed as an independent workflow) by double outlines, and shared tasks by rounded rectangles (i.e., tasks that can be included as components in more than one workflow).

The workflow *Client Credibility* can be executed as an independent workflow or as a subworkflow in other workflows. Its modular task structure and task execution ordering constraints are presented in Figure 3.

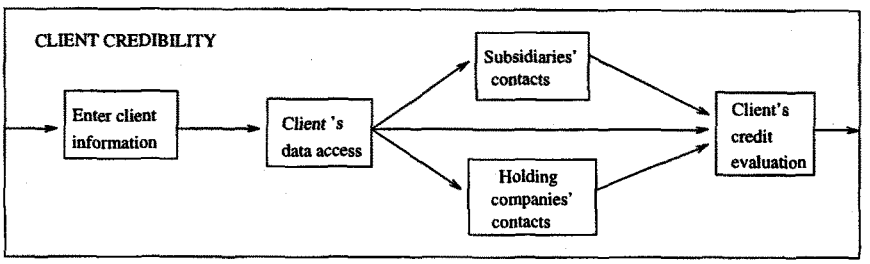

Fig. 3. Execution precedence graph for the workflow *Client Credibility.*

The task *Enter client information* accepts the information of the client. Based on this information the task *Client's data access* retrieves from the bank's database the relevant client information. This information includes the amount of the loan the client has in the bank, the names of the subsidiaries and the holding companies where the client has liabilities. If such liabilities exist the subsidiaries or holding companies are contacted to get their assessment of the client. These are performed by the tasks *Subsidiaries contacts* and *Holding companies' contacts.* The task *Client's credit evaluation* decides the credibility of the client.

The workflows *Credit account request* in Figure 4 and *Credit card request* in Figure 5 have the subworkflow *Client Credibility* as a second task, and share the last task *Client data update* which updates client's data by the given limit if the request has been accepted. In practice, however, these workflows are quite different in nature as the task *Enter credit account decision* is processed by the bank while the task *Enter credit card decision* is processed by the credit company. Hence, the former is typically a short duration activity, while the latter tends to become a long lasting workflow.

The workflow *Loan request processing* is presented in Figure 6. The first task *Enter loan request* accepts the amount and the information of the client. Based on the client information the subworkflow *Client Credibility* is processed which gives input for the task *Risk evaluation.* The risk evaluation task computes the interest

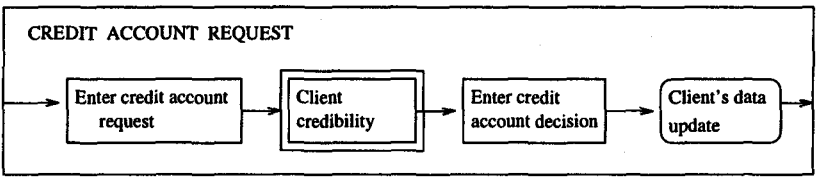

Fig. 4. Execution precedence graph for the workflow *Credit account request.*

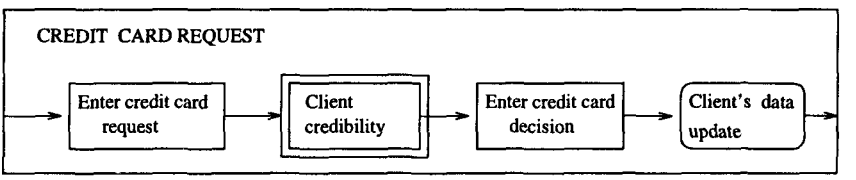

Fig. 5. Execution precedence graph for the workflow *Credit card request.*

for the loan request based on the amount of the loan and client credibility. The task *Enter loan decision* makes the decision to either grant or refuse the loan request. Before the loan can be granted the task *Bank's liability update* checks that the bank does not exceed its liability limit, and if the limit will not be exceeded, bank's total liability will be increased by the amount of the loan. If the loan is granted the task *Client's data update* adds the information of the granted loan to client's data, and if the loan request is refused the task *Bank's liability decrement* compensates the increment of bank's total liability.

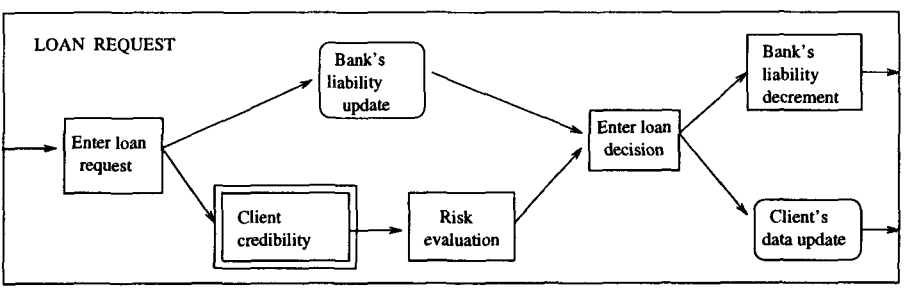

Fig. 6. Execution precedence graph for the workflow *Loan request processing.*

A bill is a special type of loan which differs from other loans in that the capital involved is usually smaller, and no pledges are used. The task *Bank's liability update* is shared by both the workflow *Bill request* (Figure 7) and the workflow *Loan request.* In these workflows, however, the task has different execution requirements: in the former it is executed as a traditional ACID transaction and

in the latter as a semantically atomic transaction, i.e., it can be compensated by the task *Bank's liability decrement* if necessary.

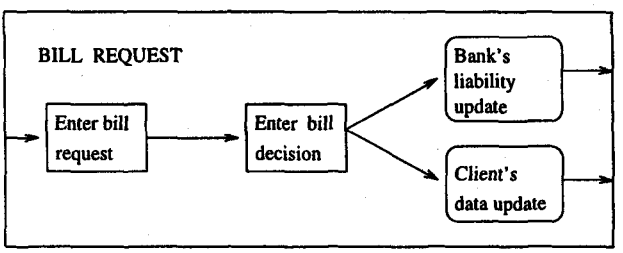

Fig. 7. Execution precedence graph for the workflow *Bill request processing.*

The simplified example above was chosen only for illustrative purposes. In our analysis of real workflow processes, one example being Telecom Finland [28], we have encountered numerous examples of overlapping workflows. From these studies it is evident that there are tasks which can be used as part of different workflows. For example the customer credibility check can be used in several different service workflows, whenever a new customer is added into the set of serviced customers (a new mobile unit, a new "101-service" etc.). In principle the same task can be used, but the credibility requirements might differ due to the differences in tariffs. The same holds for the customer billing base creation. In such a case different type of tasks or task variation is needed, because mobile customer records and practices differ from those of a 101 customer. Otherwise, the overall process specifications of customer insertion into these service processes are different and non-overlapping (in fact they are handled by separate units).

The reusability of the tasks is naturally closely linked to their definition. It is evident that the processes supported by workflows are changing. Consequently the tasks and their signatures change according to the changing functions of the organization. Methodically, whenever a new workflow specification is being developed, the existing tasks are checked and compared against the emerging needs, and the new workflow is based on a modification of existing ones, assuming that close enough match can be found. This way the best fitting set of tasks for the organizational needs will be adopted.

4 Managing overlapping concurrent workflows

4.1 Event/Condition/Action rules and markers

When several related workflows are executed concurrently, the interference of their tasks has to be controlled. The problem is naturally related to traditional transaction concurrency control [2]. A trivial approach to this problem (as well as to all concurrency control problems) would be to treat each workflow as a single monolithic transaction and require the execution of workflows to be serializable. The problem is that the workflows are prime examples of long lasting

activities which cannot be handled using traditional concurrency control methods without heavy penalties to the overall system performance. For example, if straight-forward locking is used for concurrency control, the task *Client credibility* would lock client's data until the end of the workflow. Such a lock would then hold for many days preventing client's data to be updated, which clearly is not an acceptable alternative. In the other extreme are the s-transaction model [30] and the Sagas [12], where the local subtransactions (including the compensation transactions) are run in a serializable manner together with other subtransactions and local transactions, without global level restrictions on the order of the subtransactions (for more discussion on the topic see [29]).

We use *ECA rules* [17] and *markers* as mechanisms to control the interference caused by several tasks (from different workflows) executing concurrently. ECA rules with the related marker conditions are defined at the global schema level, since in the general case it is not possible to control the data modification at the underlying "legacy system level". In this respect our ECA rule mechanism is similar to, but more general than predicate locking [10] or escrow locking [25]. Unlike these locking methods our ECA rules are based on a predefined set of markers. In principle, the more marker types we have, the more application semantics we can utilize in our workflow concurrency control. An additional benefit in using ECA rules is that they conform nicely to the work done in active database area, and can reuse (at least part of) the implementation mechanisms available [5].

In general, an ECA rule is of the following form:

```
DEFINE RULE <rule_name>
    CALLED BY <ctask_id>
    IF <marker_condition> THEN
    EXECUTE <control_operation>
ACTIVATED FOR <workflow_id,task_id>
```

For our purposes the ECA rules can be viewed as predicates that should be satisfied by the system in order to guarantee that the workflows work correctly. It should be pointed out that ECA rules (unlike 2-phase locking) do not aim at ensuring that the execution history will be serializable. In managing concurrent workflows ECA rules are activated by tasks. Therefore the workflow designer determines the constraints that the tasks have to follow in order to ensure that the workflow behaves correctly.

In our example bank application case the following ECA rules are introduced:

```
DEFINE RULE upperlimit(x,1)
    CALLED BY <ctask_id>
    IF value(x) > 1  THEN
    EXECUTE suspend(ctask_id)
ACTIVATED FOR <workflow_id,task_id>
```

```
DEFINE RULE bottomlimit(x,l)
    CALLED BY <ctask_id>
    IF value(x) < l  THEN
    EXECUTE suspend(ctask_id)
ACTIVATED FOR <workflow_id,task_id>

DEFINE RULE value(x,S)
    CALLED BY <ctask_id>
    IF value(x) NOT IN S  THEN
    EXECUTE suspend(ctask_id)
ACTIVATED FOR <workflow_id,task_id>

DEFINE RULE variation(x,S)
    CALLED BY <ctask_id>
    IF value(x) NOT IN [x-s,x+s], s IN S  THEN
    EXECUTE suspend(ctask_id)
ACTIVATED FOR <workflow_id,task_id>
```

The ECA rule `upperlimit(x,l)` prevents other workflows from updating a data object `x` such that the new value would be greater than `l`. Respectively `bottomlimit(x,l)` prevents other workflows of updating `x` in such a way that the new value would be smaller than `l`.

A `value(x,s)` prevents other workflows from updating the data object `x` in such a way that the new value would not be included in the value set `S`. The value set may be an interval or a finite set of values. Setting a value lock does not require that the current value of `x` is included in the value set. This is necessary if value rules are used as dynamic consistency constraints. A related rule type is `variation(x,S)`, which allows other workflows to update `x` only if the new value of `x` deviates from its old value by the value which is an item of the set `S`. In all these ECA rules the execution condition "suspend" implies that the task attempting to violate the constraint is suspended to wait the removal of the activation of the rule. This corresponds to traditional locking. ECA rules offer also other possibilities, for example the execution of the calling task can be continued with a notification that the attempted modification of the data object was not allowed.

As an example of the use of an ECA rule consider the workflow *Client credibility* of Figure 3. The correctness of the workflow requires that data on which the evaluation is based on should be valid at the end of the execution of the workflow. For example, the loans the client has should not have essential increments during the process. Minor increments could be allowed as they may be resulted from interest additions. Such a constraint can be forced by activating in the task *Client's data access* the rule `upperlimit(client_loan,l)`, where the value of `l` is e.g., five percent greater than the value of client's current loan. The rule may be deactivated at the end of the workflow or at the end of the workflow which

called *Client credibility* as a subworkflow. In general, the moment when the rules are deactivated may be a parameter of the task or a workflow call.

4.2 Managing static consistency constraints

As an example of the use of the `upperlimit` ECA rule to maintain a consistency constraint assume that there is a consistency constraint stating that a fixed limit `maxliability` exists for the bank's liability. To ensure this consistency constraint the workflow designer defines the ECA rule `upperlimit(bank's liability, maxliability )`, after which workflows can be processed concurrently as long as the validity of the constraint is not challenged. Evidently this is a more liberal approach than testing the validity of the consistency constraint in each transaction program updating bank's liability and setting a traditional "data lock" on bank's liability. It should be observed that activating an upperlimit rule does not require reading the state of the object `x` itself. This is important as the workflow management system is a global system constructed on top of possibly autonomous applications, and thus has no way of controlling the modification of the data objects used by the applications.

4.3 Managing dynamic consistency constraints

By activating more than one ECA rule on a single data object it is possible to utilize more application semantics in the synchronization. For example, by activating rules `upperlimit(x,10)` and `variation(x,1)` we can enforce that data item `x` can only grow in increments of one from it's current value up to value 10.

By allowing ECA rules to invoke other ECA rules we can also support dynamic consistency constraints. To illustrate this assume that the workflow system supports the states `(hand-in, in-processing, granted, rejected)` of the"request" workflows such that the state transitions have dynamic consistency constraints (Figure 8).

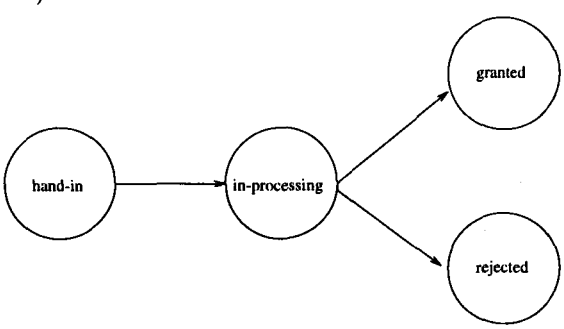

Fig. 8. A graphical presentation of a dynamic consistency constraint.

For example assume that the marker object is `loan-request-status` having the value `hand-in`. Then the ECA rule `value` with the value set `S = {in-processing}`

would only allow the modification `hand-in` → `in-processing`. If such a modification activates a value rule with the value set consisting of the items `granted` and `rejected`, the dynamic consistency constraint stating the intended legal state transitions is enforced.

4.4 Ensuring compensations

The use of the traditional transaction models would require that a whole workflow or a workflow task constitutes an atomic unit. In fact we argue that atomicity, i.e., a workflow being always run into a successful or a non-successful completion, is in many cases a useful transactional property for users. Unfortunately, if "too much" autonomy is left for the local legacy systems, atomicity cannot be fully guaranteed. Some tasks do not, even in principle, have a compensating task, and even if they had, the compensating task cannot always be completed without violating the local database consistency. This can easily be illustrated by a simple example. Let us assume that a task transferring an amount consists of an action which performs a withdraw on bank account `account_9` and an action which deposits into account `account_1` the same amount. Assume now that the withdrawal fails and the deposit succeeds, and immediately after the successful deposit an action succeeds to withdraw the whole account `account_1`. As a result the compensation, i.e., the withdrawal from account `account_1` will not succeed anymore.

To prevent the above described situations from happening, we can use previously described ECA rules. For example, one only needs to activate a rule which prevents withdrawals in the above example case. However, this is overly conservative as withdrawals have to be prevented only if the balance of the account is less than the amount to be withdrawn. This can be enforced by activating a rule `bottomlimit(account_1,amount)` in the withdrawing task.

5 Task reusability

We have already argued above that workflow task reusability is an essential feature of real life workflow systems. However, task reusability introduces the problem that the task variations appearing in different workflows may require different transactional properties (e.g., isolation or atomicity requirements). This leads us to consider parametrized transactional properties which reflect the application dependent variations in the workflow tasks. We will address the reuse issues by introducing the concepts of *task integration* and *task execution modes*, implemented by the ECA rule mechanism.

5.1 Task integration

ECA rule based mechanisms are very suitable for providing less restrictive properties than the traditional ACID-properties. For example, the traditionally used *consistency degrees* [14] from degree 0 to degree 3 provide various atomicity and

isolation levels, and for each of the levels there is a corresponding ECA rule[3]. Such rules differ in their execution policies described in their action parts. In particular the "degree 3 consistency" provides traditional serializable and recoverable executions, while other levels weaken these criteria.

It should be noted, that theoretically task integration with ECA rules is akin to multilevel atomicity [20], which weakens the usual notion of serializability by permitting controlled interleaving among transactions. Multilevel atomicity in the general transactional context, analogously to our workflow task integration, is based on the observation that there are different purposes for grouping the steps into transactions (transaction forms a unit of atomicity, or a unit of serializability).

A simple example illustrates the notion of task integration. Let us assume that the basic tasks T_1 and T_2 are units of serializability, while their compound execution $T' = T_1T_2$ should only be a unit of atomicity. In our example domain, banking environment, transfer of money (i.e., withdrawing an account followed by the deposit of another account) can be seen as a compound transaction T'.

If the workflow tasks T_1 and T_2 are units of serializability and they are to be integrated into one unit of serializability, we have to address three things:

- First, as the integrated task T' is a unit of serializability the operations of the basic tasks T_1 and T_2 should not cause any conflicts.
- Second, there must be a moment during the execution of T' when both T_1 and T_2 have an exclusive access to all the data objects they need. Such a moment corresponds to the traditional serialization point [2] and thus guarantees serializable behavior.
- Third, either both T_1 and T_2 commit or neither of them will.

The first of the above aspects can be enforced by creating an equivalence class for the task identifiers `task_id`(T_1) and `task_id`(T_2) used in the "`ACTIVATED FOR`" component in the ECA rule. For ECA rules it is assumed that the rule conditions are not checked for equivalent `task_id` identifiers. The second aspect can be enforced by using markers `rel_an_object`(T_1) and `rel_an_object`(T_2) which test the signal that the corresponding basic task has acquired access to all necessary data objects, and is ready to share an object value with others. These markers are tested in a single ECA-rule `all_accesses_acquired`(T') and the access decision in the action part for other tasks will then depend on the result of this conjunctive condition. The third condition, commit is handled analogously.

The three most straight-forward task integration choices can be specified e.g., by the following specification primitives, which the specification environment then tranlates into a set of ECA rules:

[3] In the most degenerate case ECA rules can be used to implement traditional locking policies to achieve the various degrees of consistency defined by Gray. However, in the general case the flexibility of the action part in an ECA rule allows us a more fine-grained scale of consistency degrees.

- **DEFINE task-atomicity**($T_1, ..., T_n$)
- **DEFINE task-serializability**($T_1, ..., T_n$)
- **DEFINE task-transaction**($T_1, ..., T_n$) (corresponding to degree 3 consistency)

Naturally this set of task integration primitives is open in the sense that new primitives can be introduced if needed, e.g., the integration primitives corresponding degree 1 and degree 2 consistency could be introduced.

5.2 Task execution modes

Coming back to our example, we have seen that the task *Bank's liability update* is included in the workflows *Loan request* and *Bill request*. In the former it can be compensated by the task *Bank's liability decrement* if necessary. In the latter it is not executed until the bill is granted, and thus its execution requirement matches more to that of the traditional transaction without any requirement for compensation. This example suggests that analogously to the different isolation modes discussed in the context of basic task integration, a variation task may also require different modes for atomicity. This situation is analogous with nested transactions [22, 11, 12] where typically subtransactions have to satisfy ACID properties when executed alone, but the nested transaction does not necessary have to be fully isolated.

To capture the facility of varying execution requirements of a task we introduce the following execution modes:

- **DEFINE serialization-mode**(T)
- **DEFINE prepare-mode**(T)
- **DEFINE compensation-mode**(T)
- **DEFINE independent-mode**(T)

The **serialization mode** is the strictest way to execute a task. A task is in a serialization mode if it is in a prepared mode [14] and it has not released access to any data object it requires. If several tasks $T_1, T_2, ...T_n$ are executed in serialization mode then the commit protocol (e.g., the 2PC-protocol [2]) implies that this set of tasks can be combined into one transaction. Executing the transactions $T_1, T_2, ...T_n$ in the **prepare mode** ensures only the atomicity of the combined task. In many workflow application domains even prepare mode is too strict or cannot be used as no commit protocol is available (e.g., due to local autonomy). In such cases a more liberal way to execute a task is to use application semantics and execute a task in **compensation mode** , i.e., to enforce semantic atomicity. Compensation mode is particularly useful in the execution of distributed workflows as it allows one to preserve the autonomy of local systems. Executing a task in a **independent mode** means that no assurance of ACID properties of a task execution is needed. In such cases it is assumed that the workflow itself restarts the failed task.

6 Conclusions

Modern business applications are usually composed of independently designed components which are accessed concurrently by a large set of users. Workflow techniques can be seen as techniques to coordinate and streamline such business processes. We have discussed here several key aspects in the development of large scale workflow management systems based on the notion of transactional workflows, where the application execution has to conform to a set of correctness constraints derived from the application domain.

One salient feature of a real workflow system is its dynamicity. New workflow definitions and the modifications of existing workflow specifications occur frequently. We argue that to avoid redundant design and maintenance it should be possible to produce new specifications by using existing specifications, i.e., workflow task reusability should be possible. Consequently management of overlapping workflows, i.e., workflows that share one or more tasks, is an important issue in transactional workflows. Our work presented relies on the observation that there is no reason to bind the specification of a task to any particular set of transactional properties (atomicity and isolation requirements) since these requirements may vary in different workflows. Consequently in our approach transactional properties are not fixed until a specification is linked to a workflow.

Our goal has been to address issues related to concurrent execution of overlapping workflows and to provide concepts for managing tasks with transactional properties in such contexts. In particular we have introduced the novel idea of parametrized transactional properties of task specifications with the related notions of task integration and task execution modes. Our approach is based on the use of the ECA rule mechanism at the global schema level in the TransCoop architectural framework described in Section 2. Evident topics for future work are twofold: formalization of the parametrization concepts introduced, and application of the techniques in the real life cases available in the TransCoop project.

References

1. M. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *The 19th International Conference on VLDB*, 1993.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
3. A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. Asset: A system for supporting extended transactions. *SIGMOD Record*, 23(2), June 1994.
4. Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *Sigmod Record*, 22(3), September 1993.
5. C. Bussler and S. Jablonski. Implementing agent coordination for workflow management systems using active database systems. In S. Chakravarthy and S. Urban, editors, *IEEE Proceedings Research Interests in Data Engineering: Active Database Systems (RIDE'94)*, 1994.

6. U. Dayal, M. Hsu, and R. Ladin. A transaction model for long-running activities. In *The 17th International Conference on VLDB*, 1991.
7. U. Dayl, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD International Conference on Management of Data*, 1990.
8. R. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen, and J. Wäsch. A reference architecture for cooperative transaction processing systems. Technical Report 1694, Technical Research Centre of Finland, 1995.
9. A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multibase transaction model for interbase. In *The 16th International International Conference on VLDB*, 1990.
10. K.P. Eswaran, J.N. Gray, P.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), November 1976.
11. H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested sagas. *IEEE Data Engineering Bulletin*, 14(1), March 1991.
12. H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD International Conference on Management of Data*, 1987.
13. D. Georgakopoulos and M. Hornick. A framework for enforceable specification of extended transaction models and transactional workflows. *Journal of Intelligent and Cooperative Information Systems*, September 1994.
14. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
15. W. Harrison, H. Ossher, and P. Sweeney. Coordinating concurrent development. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, 1990.
16. K. Jensen. *Colored Petri Nets*. Springer-Verlag, 1991.
17. S. Joosten. Trigger modeling for workflow analysis. In *Proceedings of CON'94: Workflow Management, Challenges, Paradigms and Products*, 1994.
18. G. Kaiser. Flexible transaction model for software engineering. In *Proceedings of Sixth International Conference on Data Engineering*, 1990.
19. G. Kaiser and C. Pu. Dynamic restructuring of transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 8. Morgan Kaufmann Publishers, 1992.
20. N.A. Lynch. Multilevel atomicity - a new correctness criteria for database concurrency control. *ACM Transactions on Database Systems*, 8(4), December 1983.
21. C. Mohan, G. Alonso, R. Günthör, and M. Kamath. Exotica: A research perspective on workflow management systems. *Bulletin of the IEEE Technical Committee on Data Engineering*, 18(1), March 1995.
22. J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1985.
23. K. Narayanaswamy and K. Goldman. "Lazy" consistency: A basis for cooperative software development. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, 1992.
24. M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 3. Morgan Kaufmann Publishers, 1992.
25. P.E. O'neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4), December 1986.

26. C. Pu and N. Hutchinson. Split transactions for open ended activities. In *The 14th International Conference on VLDB*, 1988.
27. J. Tang and J. Veijalainen. Enforcing inter-task dependencies in transactional workflows. In *Proceedings of the the Third International Conference on Cooperative Information Systems (CoopIS-95)*, 1995.
28. T. Tesch and P. Verkoulen. Transcoop deliverable ii.2. Technical Report TC/REP/GMD/D2-2/207, ESPRIT Basic Research Action 8012, 1995.
29. J. Veijalainen. Heterogeneous multilevel transaction management with multiple subtransactions. In *Proceedings of the DEXA'93*, 1993.
30. J. Veijalainen, F. Eliassen, and B. Holtkamp. The s-transaction model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 12. Morgan Kaufmann Publishers, 1992.
31. W. Wächter. Contracts: A means for improving reliability in distributed computing. In *IEEE COMPCON*, 1991.
32. H. Wähter and A. Reuter. The contract model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7. Morgan Kaufmann Publishers, 1992.
33. G. Weikum and H. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, 1992.