Tolerating Exceptions in Workflows: a Unified Framework for Data and Processes

Alex Borgida **Takahiro Murata**
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903, USA
{borgida,murata}@cs.rutgers.edu

"It is vain to do with more what can be done with fewer." William of Occam (c1285-1349)

ABSTRACT

Practical workflow systems need to be able to tolerate deviations from the initial process model because of un-anticipated situations. They should also be able to accommodate deviations in the format of the forms and data being manipulated. We offer a framework for treating both kinds of deviations uniformly, by applying ideas from programming languages (with workflow agents as potential on-line exception handlers) to workflows that have been reified as objects in classes with special attributes. As a result, only a small number of new constructs, which can be applied orthogonally, need to be introduced. Special run-time checks are used to deal with the *consequences* of permitting deviations from the norm to persist as violations of constraints.

Keywords

Exception handling, deviations, reified process model, safety.

1 INTRODUCTION

Suppose an organization is attempting to achieve some goal by carrying out a collection of activities/tasks. In order to analyze and support by computer the work to be done, it is necessary to record a description of these organizational action patterns. The formal language used for this will be called a *process modeling/description language* (PML). A PML needs to capture the way in which the steps of a process (called tasks or activities) are coordinated, as well as the effect of the activities, which may be carried out by humans or computers.

The process model describes a schema of the actions to be carried out, while a specific execution (*enactment*) of it is an instance (*workcase*) which unfolds over time. A *process support system* (PSS), containing the *workflow engine*, ensures that the enactment conforms to the schema.

Process models have found application and have been studied in a variety of fields, including office information systems, databases (focusing on complex transaction models) and software engineering (focusing on software process management) [1].

From the beginning, a major source of problems has been the *prescriptive* nature of the workflow specifications, which does not allow for unanticipated variations. For example, in order to overcome/avoid delays or simply expedite processing, in certain circumstances it may be desired to start a task (e.g., billing) before all of its immediate predecessors are completed; or start in parallel several tasks (e.g., shipping and billing), although the schema indicate that they are to be done in order; or even exchange the order of steps in a workflow. A more extreme deviation would be performing an entirely unusual sequence of actions, such as replacing the entire billing and payment process by a barter for some product or service.

Many workflow products do in fact support deviations from the process model. For example, InConcert allows a workcase to be modified by allowing tasks or dependencies to be added/removed, roles reassigned, etc. [1]. However, we believe that most proposals do not find a middle ground between requiring exceptional cases to be anticipated, and allowing ad-hoc actions that are either too specialized (e.g., restricted to forms handling [24]) or too powerful (e.g., arbitrary editing of the schema). Furthermore, the *consequences* of continuing after exceptions/deviations are mostly ignored.

In addition, almost all activities manipulate typed object (like forms and database objects), and these type constraints are subject to unanticipated deviations just as much as the process descriptions themselves. For example, the forms may need additional annotations/fields, may require multiple values where a single one is originally allowed for, may contain values of dif-

[1] See [33] for an inter-disciplinary workshop on this topic.

ferent type than the one anticipated (e.g., French Francs instead of US dollars).

The aim of the present paper is to propose a mechanism for handling exceptional occurrences in workflows which (i) *integrates* data and process deviation handling; (ii) provides a clear, precise *definition* of the notion of exception and a *disciplined* approach to handling exceptions in context; (iii) addresses the problems of allowing *deviations from the norm to persist*; (iv) introduces only a small number of new ideas/language constructs, which can be combined with standard data manipulation to achieve the desired ends.

To summarize the paper, these goals are accomplished in part by (a) reifying actions, workcases and states as instances of classes with attributes; (b) associating all exceptions with violations of constraints; (c) extending the discipline of exception handling learned from programming languages, and applying it to persistent exceptional data and deviations [7]; (d) viewing membership in the extents of systems classes as a temporal database.

The paper first introduces the data and process model to be used, and then describes procedural exception handling mechanisms. This is adapted to accommodate persistent exceptional data in Section 4, while in Section 5, we introduce our reified activities and processes, as well as the exceptions associated with them. In Section 6 we discuss and compare a variety of prior related work.

2 OUR CONCEPTUAL MODEL

2.1 The Data Model

We begin with a TAXIS-like [29] object-centered data model, where individuals are instances of classes, and are related by attributes. The class definition specifies domain and other kinds of constraints on the attributes. The subclass hierarchy provides for the usual inheritance of attributes, but also allows for the refinement of the constraints on them. Attributes can be marked as single- or set-valued, with cardinality constraints. For example, the schema for `PERSON` might include

```
class  PERSON {
    firstName: STRING;
    lastName: STRING;
    age: INTEGER;
    mother:  PERSON [lastName = self.lastName];
    younger!: (self.age < self.mother.age - 14) }
```

This example also illustrates two constraints: `younger!` is a general boolean constraint, with `self` ranging over all instances of the class[2]; `mother:PERSON [lastName = self.lastName]` introduces a special equality constraint `self.mother.lastName = self.lastName`.

Every class has an associated *extent*, with the same identifier as the class, which is the set of currently existing individuals in that class. There are operations for *creating* and *destroying* individual objects in classes, as well as for *dynamically adding* and *removing* objects from classes. Of course, there are also operations for *retrieving* or *storing* values of attributes for individuals.

2.2 The Process Model

For specificity, we adopt in this paper the ICN (Information Control Net) language for describing the coordination aspects of a workflow [19]. Control ICN (CICN) is a sublanguage of ICN to model the flow of control as a node-labeled graph[3]. Figure 1 shows a simplified college admission procedure adapted from [27], represented in CICN. Large ovals stand for *activity (processing) steps*, while filled circles – *and-nodes* (resp. open circles – *or-nodes*), represent conjunctive/concurrent control flow (resp. disjunctive/alternative control flow). Directed edges represent precedence relations among (activity and control) nodes. For example, according to the process schema in the figure, the completion of the `Review` step *enables* the `Decision` step (which can only start after `Review` is completed), while upon the completion of the `Decision` step, the succeeding OR-fork node "fires", enabling all three of its successors, `DelayedDecision`, `FollowUpAccepted`, and `FollowUpDenied`. The AND-join node preceding `Review` requires both its predecessors to be completed before firing. Multiple levels of abstraction in modeling are supported by compound activities. A more detailed description of the process model is given in Section 5.

The presence of a supporting database is assumed throughout, in order to provide not only data storage but also communication between workflow participants and with the outside world.

3 COMPONENTS OF AN EXCEPTION MECHANISM

Exception handling is a programming language control structure that allows the normal execution of a program to be replaced or augmented by special exception handling code when certain special events or conditions occur [20]. We use a synthesis of a variety of ideas for exception handling in an object-oriented language context, described in terms of the following major steps:

1. *Exception signaling:* When a special situation is discovered, an exception object is created. This object also belongs to one or more classes and can have attributes. The class identifier allows us to distinguish different kinds of exceptions (e.g., `DEADLINE-VIOLATION`),

[2] Through quantification, such a constraint can involve objects spanning multiple classes.

[3] ICN graphs also capture data flow between repositories and activities. For this paper, we omit the data flow aspects of process models.

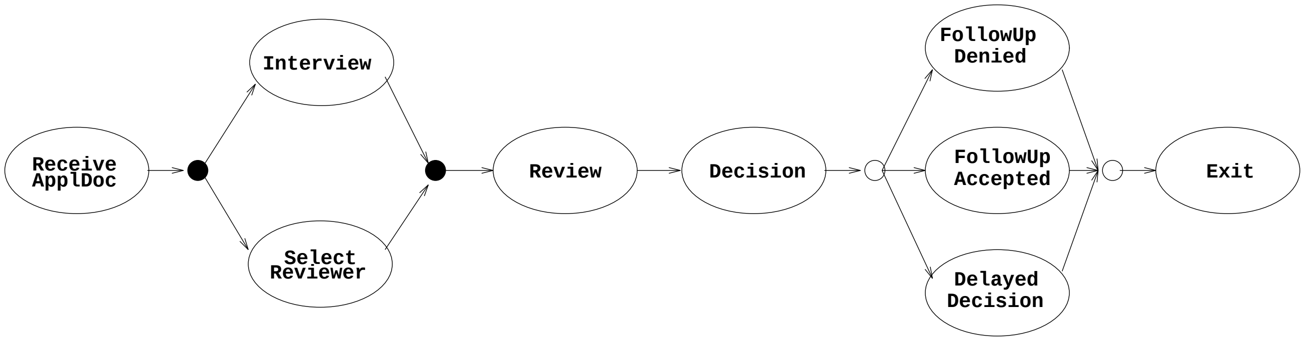


Figure 1: ICN diagram of the Admission workflow

while the attributes can pass out detailed information about the context. Exception classes can be organized into subclass hierarchies (e.g., `POSTPONABLE` **IsA** `DEADLINE-VIOLATION`).

2. *Exception handling*: The raising of an exception suspends the normal flow of control, and an attempt is made to locate an exception handler that can be invoked. One desirable feature is that exceptions be handled in a *context-dependent* manner, so that, for example, the passing of the same deadline might cause different reactions in different circumstances. This is translated into practice by the convention that when a function C_0 signals an exception of kind X, it is the *invoker*, C_1, of C_0 that is first given the opportunity to provide a handler, since the invoker knows what the signaler was supposed to have accomplished. Otherwise the exception is re-raised/propagated up the calling hierarchy, through intermediate calls to $C_2, \ldots$ till the invocation of C_n provides a handler H. An alternative is to associate (default) handlers directly to the exceptions themselves.

Note that the mechanism of subclass hierarchies allows handlers for a general exception class to be applied to exception objects raised in any of its subclasses. Conversely, specialized exception subclasses can over-ride the (default) handlers to be invoked.

3. *Resuming control flow:* While the handler H is executing (usually in the environment of the procedure C_{n+1}, which called C_n), the procedures C_k, $k \leq n$, are in a suspended state. At the end of the handler H, there are several options: Flow of control may be specified to **resume** C_k for some k, which means that all invocations of C_j, for $j < k$ are *terminated* and flow of control continues in C_k with the next statement after the one that signaled/propagated the exception X. A second alternative is to **retry** C_k, which restarts the flow of control at the beginning of C_k instead. Finally, if H does not have a resume or retry statement, then all the invocations $C_n, \ldots, C_0$, are **terminated**, and flow of control continues in C_{n+1} after the call to C_n.

A procedure that is about to be terminated may perform *clean-up actions* on the objects that persist beyond its lifetime (e.g., [21]). In the database context, this may involve aborting a transaction or performing a preprogrammed compensation action [23]. One advantage of the workflow context will be to permit *end-users* to act as exception handlers for unanticipated exceptions.

In summary, modern programming language exception mechanisms offer a *controlled, structured* way in which processes influenced by an exceptional event can participate in its resolution and either terminate or resume after appropriate repairs have been made.

4 OBJECTS AND EXCEPTIONS

A fundamental principle of our approach [7] is that *an exception occurs when some constraint is violated.* In our case, every constraint is associated with a class through an attribute, and so the constraint can be identified by the <class name, attribute name> pair. This means that there is no need to declare a specific exception object to be raised for every constraint. Instead, a single super-class

```
class  VIOLATION {
     forClass : CLASS;
     forAttrib: ATTRIBUTE-ID;
     signaler : OPERATION  }
```

suffices. Of course, one is encouraged to declare subclasses of `VIOLATION`, such as `SECURITY-CONSTRAINT-VIOLATION`, and have these be raised when any of several constraints chosen by the modeler is violated.

Constraints, such as `PERSON`s' `age`s being `INTEGER` values, are useful for catching data entry errors and for setting up efficient storage and access structure. In rare cases, we might want to store the string `"young"` as the value for some person's `age` because no precise value is known. The actual storing of the exceptional value is accomplished by **resuming** the update `store(jane,age,"young")` that was interrupted by the exception signal, and **excusing** this act. In [7], this is accomplished by creating an `EXCUSE` object that records (i) which violation (hence constraint) is being addressed,

(ii) which agent is performing the excuse and when, (iii) some text explaining the reasons, and (iv) an expiration date for the excuse. Security can be maintained by limiting who can provide excuses, and for which subclass of violations (and hence constraints).

It is important to realize that future operations on these "persistent exceptional values" will have to be monitored closely, since programs and users assume that the constraints in the schema hold. Thus, a program computing the average age of instances of PERSON might fall into a grave error if it tried to interpret the string "young" as an integer value. For this reason, jane.age needs to be marked as exceptional. We do so using an instance of the special built-in class

```
class   EXNAL-ATTRIBUTE
     {onObj : OBJECT;
      attrib : ATTRIBUTE-ID;
      value  : ANYTHING }
```

and syntax like

```
new EXNAL-ATTRIBUTE(onObj:=jane,
      attrib:='age,value:="young").
```

The data manager is then extended so that any time such an attribute is accessed, an exception (which is exactly this exceptional attribute marker) is raised to alert the program/user, thus giving them a chance to decide whether to skip the value or replace it with some numeric equivalent appropriate for this case.

As part of the object-centered approach, one can define new subclasses of EXNAL-ATTRIBUTE, to describe common categories of persistent exceptions (e.g., VALUE-UNKNOWN, WRONG-UNIT-OF-MEASURE, EXTRA-ATTRIBUTE, etc.) so that additional semantics can be captured. To support on-line exception handling, such extensions to the class hierarchy can be made at run-time. Moreover, any violation class (including an exception marker) may have attached to it a procedure performing default exception handling. This provides a way to avoid inundating users with unimportant exception signals (e.g., the handler procedure could write a warning message to some error file, and then resume with the value "null"). A variety of default handlers can be written ahead of time, and attached to existing exception classes. Then the user need only indicate the appropriate superclass for their new violation class in order to obtain the desired reaction to exceptions being encountered.

The above idea is extended to objects that are *exceptional instances of classes* by considering a built-in attribute instanceOf, which has as values the set of classes to which an object belongs.

5 ACTIVITIES AND EXCEPTIONS

Our plan is to reify activities so that they are also persistent objects in classes with attributes. This general approach began in our original work [3] on workflow-like *scripts* in Taxis, and has been independently adopted by others (e.g., [32]). Although we will not exploit this here, the above allows activity/workflow descriptions to be organized into subclass hierarchies, with the usual advantages of abbreviation, reuse and change propagation due to inheritance.

5.1 Activities

Activities have several kinds of information associated with them. First, there is the **body**, which can be an elementary activity (not modeled here) carried out by a single actor (human or electronic), or a compound activity, whose attributes will be *steps* that are partially ordered, and which themselves are activities.

Next, there is the responsibleAgent — to be filled by some actor who has authority on achieving the goal of the activity. Note that the responsibleAgent filler is the natural agent to perform on-line exception handling.

Finally, activities have associated a variety of assertions:

(1) Tests are conditions verified at some specified point during the action's execution. Two specialized kinds, **initialTests** and **finalTests**, are checked immediately at the beginning and end of the execution. Initial tests can anticipate the violation of integrity constraints or check dynamic ones (e.g., that the salary values are only increasing). Final tests are for integrity constraints that are too expensive to check ahead of time (e.g., that the average salary cannot exceed some threshold). In addition, **invariantTests** are monitored throughout the execution of the activity. For example, deadlines on the completion of an activity are invariant (temporal) assertions in this category.

(2) initialAssumptions and **finalGoals** are conditions *assumed* to hold at the beginning, respectively end, of activities. They are *proof obligations* for the correct functioning of the entire process. For workflows, the presence of appropriate data flow values (necessary inputs and outputs) are typical assertions of this kind. Of course, under normal circumstances such conditions are not evaluated at run-time because they are redundant (assuming the program is properly written), and may be expensive.

Following the paradigm for data objects, we shall make all such assertions be values of attributes of the activity class, which also has attributes for parameters and local variables. For the Admission example, we might have

```
activity class  ADMISSION
  responsibleAgent: ADMISSIONS-OFFICER
  input
      applDoc : APPLICATION-DOCUMENT
  locals
      appl   : APPLICATION-FOLDER
  initialAssumptions
      havePrelim!:  NOT (applDoc = nil)
```

```
finalGoals
    haveDecision!: (appl.decision = "admitted")
                   OR (appl.decision = "denied")
```

Violations of the user-specified conditions, which are attributes of elementary or compound activities, raise exceptions, as before. These can now be dealt with using the exception handling mechanism described in the previous two sections. Among others, if the `responsibleAgent` of one activity does not handle the exception, it is re-raised to the `responsibleAgent` of the containing/invoking compound activity. And the handler can use the excuse mechanism of `EXNAL-ATTRIBUTE` on activity instance objects to permit violations of such constraints to persist, and continue with the process.

5.2 Workflow Steps

An activity being carried out as a step in a workflow has an "enactment life-cycle". Figure 2 is our elaboration of the state transition diagram in [10].

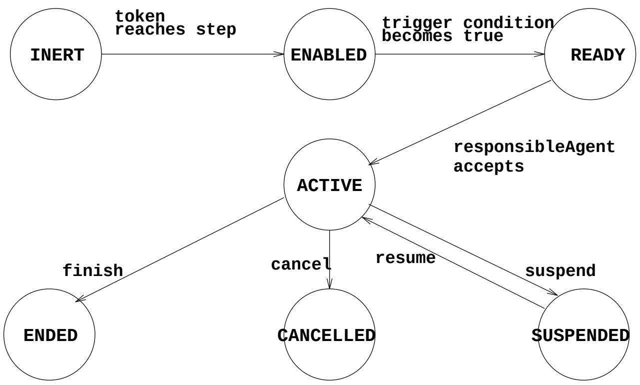


Figure 2: State diagram for workflow steps

According to the diagram, the workflow engine/PSS is in charge of moving the activity step between various stages:

- From `INERT` to `ENABLED`, when the corresponding CICN node becomes enabled (according to the formal model in [19]).
- From `ENABLED` to `READY` when the **trigger conditions** associated with the step become true. A trigger is an additional kind of condition associated with an action step, which needs to be true before an enabled action is offered to an actor for execution. Among others, this is used to implement conditional branches in workflows in conjunction with an OR-fork.
- Once ready, a step is offered to an actor for execution. If the task is automated, the transition to `ACTIVE` is immediate; otherwise the responsible person has to first signal that (s)he accepts.
- When the activity is finished (ensuring the `finalGoals`), the transition to `ENDED` is taken, and the enablement relation is recomputed in the CICN.
- The `responsibleAgent` or someone in higher authority may **suspend** the activity, thus putting it in the `SUSPENDED` state, and a **resume** puts it back into `ACTIVE`.
- A **cancel** action terminates the activity, with no assurance that the `finalGoals` are true, and without enabling its successor step(s) in the CICN.

In the spirit of object-centered reification, we make classes of activities `INERT`, `ENABLED`, `READY`, `ACTIVE`, `SUSPENDED`, `ENDED`, and `CANCELED`, and *"entering/leaving a state"* becomes synonymous with *"being added/removed from the class extent as an instance"*.

The control steps in CICNs have a similar lifecycle, except that they go from `ENABLED` to `ENDED` directly. The AND-join node requires special treatment: an additional class (`PARTIALLY-ENABLED`) is inserted between `INERT` and `ENABLED` in order to represent the case when some, but not yet all of the node's predecessors have ended.

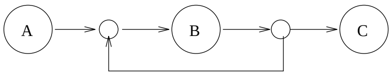
If an exception occurs during a task execution, the activity is suspended, and is in fact inserted into a subclass `SUSPENDED-BY-EXN`. For elementary activities, this means that no further updates may occur under the aegis of this task, but querying is allowed in order to investigate possible reasons for the exception. Resumption after an exception corresponds to a move of the instance to class `ACTIVE`, while termination causes a move to a subclass `CANCELED-BY-EXN`. When a compound activity instance C raises an exception, then the execution of the workcase C itself is suspended (so that the engine makes no further moves in enabling actions, etc.) but the individual steps are *not automatically* suspended (unless they raised the exceptions). They can, of course, be selectively suspended by the exception handler.

Essentially, the above approach moves towards a reflective architecture for workflow enactment, but only the data for the workflow engine is available for modification, not the engine itself. It provides a number of advantages: First, suppose we keep track of the times when an activity becomes or stops being an instance of each state class. Using an appropriate (temporal) query language, it is then possible to retrieve or constrain workcases with a wide variety of *time-related conditions*, such as how long an activity/step has been "active", "active or suspended", "enabled but not fired", "waiting to have an exception handled", etc. For example, it is possible to assert about an activity step B that *"once B is enabled, it must be started by 7/20/1998"*, by stating as an invariant that if it is in class ENABLED, then the current date must be less than 7/20/1998. This approach provides a clear advantage in flexibility and uniformity over one with a fixed, built-in set of special temporal expressions, such as "**delay**".

Second, during exception handling, the handler itself can explicitly move some activity instance from one state to another. For example, when the previous assertion is violated because today is 7/20/1998, the *handler* may decide that activity B cannot be delayed any further, and may then move the object from class ENABLED to READY. This will raise an exception since only the workflow engine is supposed to move objects between these classes, and this gives the `responsibleAgent` an opportunity to give an appropriate excuse. When exception handling is finished, the workflow engine resumes the suspended workcase and it will then start the activity instance (supposing it is automated) as part of its regular cycle. Once again, this is more parsimonious than providing special names for operations that perform all possible pairs of moves between states in the diagram.

5.3 Reifying CICN Graphs

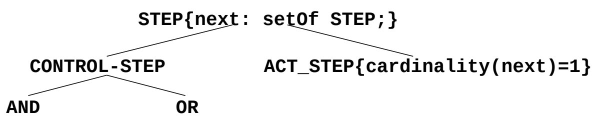
We have considered above the life-cycle of a single step of the workflow. To reify the control aspects, we make control graphs be objects with the steps as attributes. The names of the attributes can be arbitrary (though they might be chosen to describe the action or the state reached after the step completes). For example, the control graph ICN1



may have the following (preliminary) class definition:

```
class  ICN1 IsA CICN {
     step-a: A;
     step2 : OR;  step-b: B;  step4 : OR;
     step-c: C;                          }
```

As suggested by the example, in addition to activity classes A, B and C, there are built-in classes OR and AND for control nodes. To model the actual sequencing constraints (the edges in the graph), we require step values to be instances of a special class STEP, which has an attribute `next`, whose value will be the successor step to be enabled by the workflow engine once this step ends successfully. This attribute is single valued for activity steps (members of class ACT_STEP), but set-valued for control steps. Therefore we have a class hierarchy of the form



In addition, we need to mark the starting step (by convention, it will be the first one mentioned) and the end steps (by convention, the ones having `next` equal to the special step `end`). Therefore, the complete specification of ICN1 is

```
class  ICN1 IsA CICN {
   step-a: A and ACT_STEP[next=self.step2];
   step2 : OR [next=self.step-b];
   step-b: B and ACT_STEP[next=self.step4];
   step4 : OR [next={self.step-c, self.step2}];
   step-c: C and ACT_STEP[next=self.end]   }
```

PSS implementation.
When enacting a workflow containing ICN1 as a body, the PSS engine creates an instance object `wf1` of class ICN1, with `wf1.step-a` initialized to a new A object, which is also placed in classes ACT_STEP and ENABLED. The other step attributes have null values at this point. Thereafter, (conceptually) the PSS cycles through each `step` attribute of workflow `wf` that has a non-null value `sv`, performing procedure `Advance(wf,step,sv)`. The main novelty is the case when `sv` $\in$ ACTIVE and `sv` has just finished. As shown in the pseudo-code below, normally one would clear `wf.step` (though maintaining it in a temporal data base) and create an instance of the successor step in the workcase. However, the PSS behaves in a special manner, to be exploited later, if `sv.next` had been exceptionally *pre-set*.

```
wf.step := null   //but record in temporal db
if sv.next==null then {  //normal case
  step1 := from constraint [next=self.step1] in schema(wf);
  Kind1 := from constraint step1:Kind1 in schema(wf);
  wf.step1 := new Kind1 also in  ENABLED;
  sv.next := wf.step1;   //to record history
 }
if sv.next!=null  then   //sv.next exceptionally pre-set
   if sv.next ∈ INERT  then
     {move sv.next from INERT to ENABLED;  }
```

To save space, we omit the rather obvious actions for steps becoming active, suspended, etc., and for the control steps of the CICN.[4]

Finally, the PSS must also react to changes in the data values in the database (including the clock), by verifying that no tests associated with objects (including activities!) are violated, and that triggers becoming true for enabled actions cause appropriate transitions.

5.4 Exceptional Coordination

We explore next the ways in which our techniques for handling exceptional data objects (Section 4) can be applied to the above representation of workflow instances in order to deal with a variety of exceptional occurrences. Consider a simple sequential net ICN2 of the

[4] We note that nondeterminism creates technical difficulties in computing the "enabled step" relationship for CICN workflows, as formally specified in [19].

form

`A-->B-->C-->D`

with steps called `step-a,step-b,step-c,step-d`. Suppose ICN2 has a constraint specifying a deadline by which `step-b` must be enabled or activated. Let `wf2` be an enacted instance of ICN2, and consider various scenarios of the `responsibleAgent` reacting to a violation of this constraint:

(1) Suppose `step-a` is the only step currently active in `wf2`, and the `responsibleAgent` decides to start step B before step A is finished. She therefore creates an instance (say `b45`) of activity class `B` that is enabled (i.e., is also in class `ENABLED`), and assigns `b45` to `wf2.step-b`. However, notice a subtle problem: when `step-a` finishes, it would normally lead to the enablement of *another*, unwanted instance of activity B. To prevent this, assign `b45` to `wf2.step-a.next`; the workflow engine (as described above) manages the rest appropriately. Note that the assignment of `b45` to `wf2.step-b` and to `wf2.step-a.next` above raise violations of the built-in authorization constraint stating that only the PSS engine should change values of a step attribute for an ICN, or its `next` attribute. The `responsibleAgent` must therefore excuse these violations by associating appropriate `EXNAL-ATTRIBUTE` markers to the attributes, and creating an excuse that explains the situation.

(2) Suppose that now step A is ended, and step B is enabled, but the deadline for activating B has passed without the trigger becoming true. This time the `responsibleAgent` decides to skip step B altogether. This is done by moving `wf2.step-b` from class `ENABLED` to `CANCELED`, to prevent it from firing. At the same time, we need to manually enable `step-c` because it does not have any predecessor activity instance:

```
wf2.step-c := new C also in ACT_STEP,ENABLED;
```

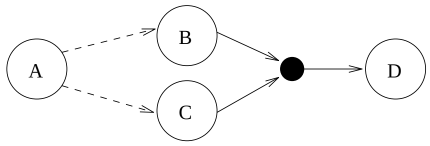
(3) If, instead, before the end of A, the `responsibleAgent` wants to exchange the order of execution of steps B and C, she can perform the following updates

```
wf2.step-c := new C also in ACT_STEP,INERT;
wf2.step-a.next := wf2.step-c;
wf2.step-c.next := wf2.step-b;
wf2.step-d := new D also in ACT_STEP,INERT;
wf2.step-b.next := wf2.step-d;
```

(4) Finally, the `responsibleAgent` decides to allow steps B and C to be carried out in parallel. First, we start `step-c` going:

```
wf2.step-c := new C also in ACT_STEP,ENABLED;
```

Then, we force an additional attribute for a conjunctive control node simulating the following graph



by performing the following updates

```
wf2.extra-and  := new AND also in INERT;
wf2.step-b.next := wf2.extra-and;
wf2.step-c.next := wf2.extra-and;
wf2.step-d     := new D also in INERT;
wf2.extra-and.next:={wf2.step-d};
```

Again, each update causes an authorization violation; in addition, the updates to `extra-and` and `step-b/c.next` also violate the original type constraints for ICN2. Appropriate exceptional-attribute markers and one excuse explaining them are needed.

Observe the way in which we have added the new AND control node as the value of `extra-and`; this illustrates how one can add arbitrary new steps as attributes to a workflow.

Note that the above description is not just an implementation view, but is in fact the conceptual view that we would like the workflow participants to have. On the other hand, it is too much to expect end-users to write program fragments such as the ones above. A partial solution is to take taxonomies of exception kinds and frequently observed reactions to them, such as those offered in [12, 15], and to pre-program these as parameterized patterns, making them available to users via a menu. A more general fallback is to use a visual formalism (e.g., APEL [16]), in which such "programming" can be achieved graphically.

Above, the exceptional occurrence was partly anticipated since there was a constraint whose violation alerted us to something being amiss. If a handler had been specified ahead of time, this would have been a pre-programmed special case. As it was, we allowed dynamic, context-sensitive handling of the violation. At the other extreme, the process specification may not have had any assertion about the (relative) time when activity B was supposed to take place, but the `responsibleAgent` of `wf2` might have noticed at run-time an abnormally long delay. For this purpose, we need to allow the `responsibleAgent` to place a workcase into `SUSPENDED-BY-EXN`, and then carry out the special handling actions. In this case, she is required to create a violation – and this gives her an opportunity to record for the first time a previously un-noted constraint.

5.5 Repercussions of Deviations.

Consider some of the effects of allowing users to modify the state of activities and their successors. Suppose that some activity instance `b` was enabled either (i) by a user, as part of exceptional actions, or (ii) by a step `p` such that `p.next` is exceptional. In these cases the normal predecessor of `b` in the workflow did not get a chance to establish its goal, and the current step might have depended on that goal. We believe that in such situations `b`'s **initialAssumptions** need to be checked explicitly, and `b.responsibleAgent` should also be notified of the deviation. Let us call this part of the *safety policy*. For example, in the first scenario above, when starting B before A is completed, the system should verify `step-b`'s **initialAssumptions**. Or, in the Admission workflow, if a deviation wants to skip over the `Review` step (because, say, the applicant already has offers from elsewhere) then according to the above policy the `responsibleAgent` of `Decision` would be alerted that the input data normally expected is not complete. (Thus, the `responsibleAgent` can review the `Decision` activity, and if it is automated, probably carry it out by hand.)

Similarly, an additional part of the safety policy is the run-time verification of the **finalGoals** of an activity when (i) it is manually moved to the `ENDED` state, (ii) when it is exceptionally resumed after a failed **initialTest** or **initialAssumption**, or (iii) when a compound activity's coordination has been exceptionally changed. Both [19] and [14] give examples where during on-line changes of step ordering, one has to make sure that neither of the steps is omitted in the end. In our case, such a condition should be expressed as a temporal assertion (involving possibly former values of the respective step attributes) for the workflow. The advantage of our approach is that even this (meta-)constraint may be violated with suitable excusing privileges.

6 RELATED WORK AND CONCLUSIONS

To begin with, there have been many papers on the *evolution* of workflow and process models, including [1, 2, 4, 11, 18, 19, 26, 32, 34]. We believe there is a distinction to be made between deviations during workflow enactment (the topic of this paper) and workflow evolution. The difference is analogous to the one between allowing exceptional individuals in a database (e.g., [7]) and schema evolution in a database (e.g., [5]). The latter task might be prompted by multiple occurrences of the former, and work on machine learning may help in such evolution [8].

Our framework does support a structured *additive* form of evolution by codifying exception handling into procedural handlers that are suitably placed, possibly as default handlers attached to the violations, as discussed in [7].

Representative of the works cited above is [19], which also applies to the CICN formalism. It proposes a technique whose core is a mapping function from the tokens in the state of the original CICN to those of the modified CICN. In our examples (see Section 5.4), this mapping is essentially simulated by the creation of activity nodes for steps placed in the `ENABLED` class. The other Petri-net like approaches to modifying workflow models (e.g., [2, 34]) bear a similar relationship to our work. In fact, in [9] we show how the present framework can be applied to the workflow coordination model in [2], which has the nice theoretical property that one can analyze when it is safe to make changes.

A second class of work, which usually deals with *anticipated exceptions*, involves so-called "advanced transaction models". From the seminal work in [17] to such recent papers as [23] and [35], this work has normally addressed issues such as failure recovery and co-ordination between multiple workflows. Following the example of the WIDE project [13], we have chosen to separate these issues from the more local exceptional occurrences studied in this paper. However, we plan to return to this problem, particularly in light of the combination of transactional notions and programming language exceptions recently explored in [22].

Several other research efforts bear a closer look.

The WIDE workflow system [10, 13], has a conceptual model resembling ours, based on CICN. It facilitates the declarative specification of constraints and their error handling by associating *condition-reaction* pairs to each activity. A taxonomy of different kinds of events that can be exceptional is given in [12]. These rules provide an essentially *static* mechanism for exception handler association, in contrast with our dynamic approach, which uses the invocation hierarchy. Also, exceptional events in WIDE have to be identified at design time, while we are interested in ad-hoc deviations, including the ability to resume.

The ADEPT [30] workflow project is concerned, like we are, with ad-hoc changes to individual workcases, and controlling them to avoid undesirable consequences. The most notable advances are made in using the *data flow* constraints expressed in the workflow to evaluate proposed changes. This nicely complements our work on enforcing assumptions and goals, which deal with control flow.

Cugola and colleagues investigate deviations during enactment of software processes [14], distinguishing assertions of invariance and triggers of state transitions; deviations are then tolerated only for the later. The remainder of [14] is devoted to analyzing, using temporal logic, the propagation of "possibly polluted information" when deviations are allowed, assuming that any deviant action produces suspect data. Shifting to an "artifact"-centered model, [15] extends the above work by (i) proposing a fixed set of reactions to invariant violations, and (ii) providing a detailed analysis and categorization of deviations and actions for handling them, collectively called "deviation policies". The taxonomy of exceptions and ways of handling them is of considerable interest to us, since it provides the basis for a library of exceptions and handlers, which would greatly facilitate the task of users faced with run-time deviations. Once again, this can be achieved in our model by refining the subclass hierarchy of violations, and associating appropriate default handlers to them.

In contrast to our more technical focus, several papers have addressed the organizational aspects of exceptions. In the early 1980's, Kunin [27] distinguished between the *main-line* of a business process descriptions, and deviations from it, and provided a language to support a methodology for using this form of abstraction. In [25], an approach is reported toward effective exception resolution that achieves "organizational integrity" based on taxonomies of exceptions, mapping them to potential diagnoses, and resolution strategies, all housed in a knowledge base. Along similar lines, and supported by a substantial empirical study, [31] suggests a meta-model for exceptional event handling based on "degree of exceptionality". Finally, in the situated work camp, [6] describes a workflow system where activities are modeled as an executable network of obligations, with flexible placement of tokens.

6.1 Contributions

We have laid out a computational framework that provides generic, flexible, and disciplined means of exception handling in workflow/process enactment, based on a precise account of deviations as violations of constraints. It is *uniform* since it deals with deviations, anticipated or unanticipated, from both data schema and process schema, each causing a violation of either a user-specified constraint or one inherent in the data/process model itself. It is *parsimonious* since it uses standard data manipulation operations on class extents and attribute values, augmented by the ability to resume after an exception, to describe deviations of both kinds.

The framework relies on three ideas: (1) Activities and workflows are *conceptually* reified as objects whose attribute values and class memberships encode the information and constraints maintained by the workflow engine for each workcase enactment (e.g., the current state, the trigger, the `next` step). As such, they resemble the data and forms manipulated by workflows. (2) The technique for handling exceptions, and especially for *permitting exceptional values to persist*, described in [7], is extended so that it can be used to support all the desired kinds of deviations from the norm in process descriptions (e.g., specifying an exceptional, alternate `next` step). (3) Responsible agents, or other authorized users, are allowed to act as on-line exception handlers, in order to exercise judgment in coping with unanticipated situations or when encountering "persistent violations" left by others. To facilitate this, a taxonomy of exception kinds, as well as pre-programmed handlers attachable to exceptions, can be made available.

To deal with the consequences of allowing deviations to persist, "persistent violations" raise exceptions when accessed later (as in [7]); in addition, our safety policies suggest expressing constraints such as **initialAssumptions** and **finalGoals**, whose run-time checking may be activated in order to protect the workcase from performing illegal or non-sensical operations after ad-hoc changes in the workcase.

To limit the set of constraints that can be violated, or the persons who can allow these violations to persist, one can apply standard authorization policies to the *creation of excuses*, which must accompany every deviation. (See [28] for a declarative language to state such authorizations.)

In addition to a prototype implementation of the PSS, our future plans include developing a logical semantics of workflows and their exceptions, a connection to higher-level goals and plans (which are the true clues of what is a permitted deviation), integration with transactional workflows, and retrieving and learning generic handlers for deviations.

ACKNOWLEDGEMENTS:

This research is supported by NSF Grant IRI-9619979.

REFERENCES

1. K.R. Abbott, S.K. Sarin. "Experiences with Workflow Management: Issues for the Next Generation", *CSCW'94*, Chapel Hill, NC, 1994.

2. A. Agostini, G. De Michelis. "Simple Workflow Models" *Proc. Workshop on Workflow Management*, June 1998, pp. 146-164.

3. J.Barron. "Dialogue and process design for Interactive Information Systems Using Taxis", *Proc. SIGOA Conf, on Office Information Systems*, June 1982

4. S. Bandinelli, A. Fuggetta, and C. Ghezzi. "Software Process Model Evolution in the SPADE Environment." *IEEE Trans. on Softw. Eng.*, Dec. 1993.

5. J. Banerjee, W. Kim, H. Kim, H.F. Korth: "Semantics and Implementation of Schema Evolution in Object-Oriented Databases." *Proc. ACM SIGMOD'87*, pp.311-322

6. D.P. Bogia, S.M. Kaplan. "Flexibility and Control for Dynamic Workflows in the wOrlds Environment", in *Proc. Conf. Organizational Computing Systems*, Milpitas, CA, November 1995.

7. A. Borgida. "Language Features for Flexible Handling of Exceptions in Information Systems", *ACM Trans. on Database Systems*, 10(4), Dec. 1985, 565-603.

8. A. Borgida, K.E. Williamson. "Accommodating Exceptions In Database, and Refining the Schema By Learning From Them", *Proc. VLDB'85*,pp.72-80.

9. A. Borgida, T. Murata. "Workflows as Persistent Objects with Persistent Exceptions", *CSCW-98 Workshop:Towards Adaptive Workflow Systems* (obtainable from http://ccs.mit.edu/klein/cscw-ws.html).

10. F. Casati, S. Ceri, B. Pernici, G. Pozzi. "Conceptual modeling of workflows",*O-O ER'95*, 341-354, Gold Coast, Australia, Springer Verlag, Dec. 1995.

11. F. Casati, S. Ceri, B. Pernici, G. Pozzi. "Workflow Evolution", *Data and Knowledge Engineering 24(3)*, 1998, pp.211-238.

12. F. Casati, S. Ceri, S. Paraboschi, G. Pozzi, "Specification and Implementation of Exceptions in Workflow Management Systems", TR 98.81, Dipt. di Elettronica e Informazione, Politecnico di Milano, August 1998.

13. S. Ceri, P.W.P.J. Grefen, and G. Sanchez. "WIDE: A distributed architecture for workflow management", in *Proc. RIDE '97*, Birmingham, UK, Apr. 1997.

14. G. Cugola, E. Di Nitto, C. Ghezzi, M. Mantione, "How To Deal With Deviations During Process Model Enactment", *Proc. ICSE'95*, Seattle, WA, May 1995.

15. G. Cugola. "Tolerating Deviations in Process Support Systems Via Flexible Enactment of Process Models", *IEEE Trans. Softw. Eng., Special issue on Managing Inconsistency in Software Development* (to appear).

16. S. Dami, J. Estublier, M. Amiour. "APEL: A Graphical Yet Executable Formalism for Process Modeling", *Automated Software Engineering 5(1)*, 1998, pp.61–96.

17. U. Dayal, M. Hsu, R. Ladin: "A Transactional Model for Long-Running Activities". *Proc. VLDB'91*: 113-122.

18. P. Dourish, J. Holmes, A. McLean, P. Marqvardsen, A. Zbyslaw. "Freeflow: Mediating between Representation and Action in Workflow Systems", *Proc. CSCW'96*, 1996, pp.190-198.

19. C. Ellis, K. Keddara, G.Rozenberg. "Dynamic Change within Workflow Systems," *Proc. Conf. on Organizational Computing Systems*, 1995.

20. C. Ghezzi, M. Jazayeri, *Programming Language Concepts*, 3rd Edition , J. Wiley and Sons, 1997.

21. J.Goslin, B.Joy, G.Steele, *The Java Language Specification*, Addison-Wesley, 1996.

22. C. Hagen, G. Alonso. "Flexible Exception Handling in the Opera Process Support System", *Proc. ICDCS'98*, Amsterdam, May 1998.

23. M. Kamath, K. Ramamritham. "Failure handling and coordinated execution of concurrent workflows", *Proc. ICDE'98*, Orlando, FL, Feb. 1998, pp.334-341.

24. B.H. Karbe, N.G. Ramsperger. "Influence of Exception Handling on the Support of Cooperative Office Work", *Multi-User Interfaces and Applications*, S. Gibbs and A.A. Verrijn-Stuart Eds., Elsevier , pp.355-370, 1990

25. M. Klein. "Exception Handling in Process Enactment Systems", *Proc. ECAI'96*, 1996.

26. M. Kradolfer, A. Geppert "Dynamic Workflow Schema Evolution based on Workflow Type Versioning and Workflow Migration", TR 98.02, Dept. of Computer Science, University of Zurich, April 1998

27. J.Kunin. "Analysis and Specification of Office Procedures", MIT/LCS/TR-275, 1982.

28. N. Minsky, V. Ungureanu. "Unified Support for Heterogeneous Security Policies in Distributed Systems", *USENIX Security Symposium*, January 1998, San Antonio, Texas.

29. J. Mylopoulos, P. A. Bernstein, H.K.T. Wong. "A Language Facility for Designing Database-Intensive Applications." *ACM Trans. Database Systems* 5(2): 185-207 (1980)

30. M. Reichert, P. Dadam. "ADEPT – supporting dynamic changes of workflows without loosing control", *J. Intelligent Information Systems*, 10(2), March 1998, pp.93–130.

31. H.T. Saastamoinen. "On the handling of exceptions", Ph.D. Thesis, University of Jyvaskyla, Jyvaskyla, 194 pages, 1995.

32. S.K. Sarin: "Object-Oriented Workflow Technology in InConcert". *Proc. COMPCON'96*, pp.446-450.

33. A.Sheth editor, *NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996, Athens, Georgia.

34. M. Voorhoeve, W. van der Aalst. "Ad-hoc Workflow: Problems and Solutions", *Proc. Workshop Databases and Expert Systems*, 1997, pp.36-41.

35. D. Worah, A. Sheth. "Transactions in Transactional Workflows" in *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds., Kluwer, 1997.