A Unified Framework for Exceptions in Workflow and Process Models – An Approach based on Persistent Objects

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

January 1998

Abstract

The need for flexibility in workflows and the problem of unanticipated exceptions are well recognized, and in fact a number of solutions have already been proposed in areas such as computer-supported collaborative work and software process modeling.

On the other hand, workflow systems frequently maintain forms and other information in databases, and there is also strong evidence for the need to consider exceptions to the constraints imposed by the schema on the data in databases.

This paper presents a framework for treating both kinds of exceptions uniformly by applying ideas from programming language exception handling (but where the responsible agents of workflows are allowed to handle exceptions on-line) to a situation where most aspects of workflows have been reified as persistent objects in special classes and with special attributes. As a result, only a small number of new constructs need to be introduced, and power is achieved through orthogonality.

Unlike most previous work, the framework pays particular attention to the *consequences* of permitting deviations from the norm to persist, in part by defining exceptions as violations of constraints and then developing a taxonomy of constraints.

Also as a departure from prior efforts, the paper takes the view that the evolution of the schema (workflow *or* data) is a different problem from that of individual exceptions, but that a limited form of incremental evolution – providing exception handlers *ahead of time* – is a useful abstraction mechanism to help organize the details of complex workflows.

Contents

1 Introduction **3**

2 Our conceptual model **5**
2.1 The data model 5
2.2 The process model 5

3 Components of an Exception Mechanism **6**

4 Objects and Exceptions **8**

5 Activities and Exceptions **10**
5.1 Elementary Activities. 10
5.2 Compound Activities 13
5.3 Exceptions in workflows 14

6 Related work and conclusions **16**
6.1 Exceptions vs. Evolution 16
6.2 Other closely related work 17
6.3 Contributions 18

1 Introduction

Suppose an organization is attempting to achieve some goal by carrying out a collection of activities/tasks. Each of these activities can be performed either automatically (by computers) or by humans, supported by the computer providing contexts, databases, forms, to-do lists, etc. In order to analyze and possibly automate portions of the work to be done, it is useful to be able to capture a description or model of these organizational action patterns. The language used for this might be called a *process modeling language*.

Some of the activities, called **elementary**, are described using a **computation language** [32, 19]. This may vary from a very detailed, procedural program for computer execution to a screen displaying an English description of the goals to be accomplished, accompanied by a menu offering a variety of ways for a person to indicate when and whether the task has been successfully carried out. More importantly, from the point of view of the present paper, there is also a description of the way in which the steps of a **compound activity** need to be inter-related, using a **co-ordination language**. This part of the process model describes ordering constraints as well as conditions under which activities can be carried out – the "control flow" aspects. Note that the model needs to be able to capture not just simple sequential task ordering, but also parallel execution and loops. The co-ordination aspect also often describes (in greater or lesser detail) the flow of information, using notions like repositories, inputs and outputs.

The process modeling language can be used to describe essentially a *schema* of the actions to be carried out, while a specific execution or *enactment* of it is an instance (sometimes called a workcase) which unfolds over time. This is much like the relationship between a procedure definition and one of its activations. A *workflow engine* ensures that the enactment conforms with the schema.

The ability to execute workflow models on a computer was intended to gain several advantages, including (i) coordinating the work and communication of people in an organization and supporting time/task management for individuals (e.g., creating to-do lists); (ii) allowing monitoring of the processes being carried out; (iii) automating some of the mundane tasks; (iv) managing the movement of (semi-completed) forms through the computer.

In addition to business computing, other areas of computer science, especially software engineering, have also been interested in process modeling and enactment [31, 16].

From the beginning, a major source of problems was the perceived *prescriptive* nature of the workflow specifications, which laid down rules according to which workcases were to be handled, without allowing for unanticipated variations, deviations, etc. This contrasted markedly with the flexibility of the human system it was meant to assist or replace.

For example, in order to overcome/avoid delays or simply expedite processing, in certain special circumstances it may be desired to

- start a task (e.g., billing) before all of its immediate predecessors, which have already begun, are completed;
- start in parallel several tasks (e.g., shipping and billing), although the schema indicated that they were to be carried in some particular order;
- exchange the order of steps in a workflow (e.g., although the subordinate was supposed

to sign off on some document before the manager did, the manager is leaving town so the order of the two steps is swapped);

According to the survey in [21], many workflow products on the market do in fact support deviations from the process model during an enactment. 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 in most cases the mechanisms considered so far suffer from a number of failings, including being either too specific/ad-hoc (e.g., restricted to forms handling [24]) or too powerful (e.g., allowing arbitrary editing of the schema), and giving insufficient consideration to the *consequences* of having exceptions.

Furthermore, the kinds of workflow or software process models we consider always operate on typed object (like forms), and we contend that these type constraints are just as much subject to unanticipated exceptional cases as the process descriptions themselves. For example, the forms typically manipulated by office workflows may need additional annotations/fields, may require multiple values where a single one is originally allowed for, etc.

The aim of the present paper is therefore to propose a mechanism for handling exceptional occurrences in workflows which

- *integrates* data and process exceptions (doing so at the instance level rather than as schema evolution);
- provides a clear, precise *definition* of the notion of exception; and provides a *disciplined* approach to handling exceptions in context;
- introduces only a small number of new ideas/language constructs, and achieves its power through the orthogonal application of these ideas together with standard data manipulation;
- supports, with one minor addition, the specification of many kinds of *time-related exceptions*, which appear to form a large and useful subclass of workflow exceptions.

To summarize the paper, these goals are accomplished by

- reifying actions and workflows as instances of classes, and reifying states of actions as classes;
- associating all exceptions with violations of constraints;
- extending the discipline of exception handling learned from programming languages, and its application to persistent exceptional data [9];
- viewing membership in some of the system-defined class extents as a temporal database.

In this paper we focus on dealing with *unanticipated* special cases, which we see as being taken care of by human role-players acting as *on-line* exception handlers. However, we foresee the eventual codification of these actions as pre-programmed exception handlers, that are invoked automatically. In fact this feature can support an important form of program abstraction: dealing with normal cases first, and relegating special cases as the handling of exceptions to "normalcy conditions."

Because much prior *database* work on workflows has concerned in some way advanced transaction models (see [2, 38] for reviews), it may be worth pointing out right from the beginning that the work presented here does not (yet) consider issues of concurrency control and recovery. We believe that our mechanism can be considered orthogonal to such aspects, although we mention some alternatives in the concluding remarks.

2 Our conceptual model

2.1 The data model

We begin with a Taxis-like [29] object-based data model, where individuals are instances of classes (which have extents), and are related by attributes. The class definition (schema) specifies range 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. Like in most semantic and object-oriented data models (but unlike Taxis), attributes can be marked as single- or set-valued, with cardinality constraints. We also add a more unusual notation, to describe *restricted attributes*: `children[MALES]` denotes a specialization of the `children` relationship to the case where the values are in the class `MALE` – a relationship we might call "sons" in English[1].

For example, the schema for PERSON might include

```
class  PERSON {
          age: INTEGER;
          mother: PERSON;
          friends: setOf  PERSON;
          friends[OLD-PERSON]: setOf  EMPLOYEE
             // older friends are employed;  }
```

We assume that 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 therefore operations for creating and destroying individual objects in classes, but also for dynamically **adding** and **removing** objects from classes (a feature regretfully not supported by most OODB). 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 a variant of the ICN (Information Control Net) language for describing the co-ordination aspects of the workflow[18, 19]. ICN has a quite powerful notation for control-flow (CICN), which is however more restrictive than arbitrary Petri-nets; it was the basis of the FlowPATH commercial product, and has been recently used by the WIDE project in Milan [11] as the basis of a prototype workflow system that is implemented using active database technology. The WIDE system in fact provides an explicit exception facility, which is related to the one in this paper (Section 6).

[1]This idea is suggested by both description logics [10] and the notation of X-SQL queries [25].

ICN is a process, rather than document or message oriented workflow modeling languages. The steps of a workflow are either activities or control steps. The main goal of ICN is to provide partial ordering of the constituent steps of a workflow via control flow in a graph-like notation augmented by data flow between a data repository and an activity.

Figure 2, at the end, is an ICN representation of a college admission procedure adapted from [27]. A large oval stands for an ordinary *activity (processing) step*. The small circles are *control steps/nodes* affecting flow of control: a filled circle (**and-node**) for conjunctive ("concurrent") control flow; an open circle (**or-node**) for disjunctive ("exclusive OR") control flow. Directed edges represent relevant *precedence* relations among (activity and control) nodes, and enactment is represented by the flow of tokens through the graph.

For example, according to the process schema in Figure 2, the termination of the Review step **enables** the Decision step, by moving a token from the former node to the edge that connects the two. When an instance of Decision is actually started the token moves onto the corresponding node. On the other hand, after the Decision step terminates, the token passes through the OR-fork and may then move along any of its three outgoing edges, thus enabling any of FinalDecision, ReceiveReply, DeniedFollow-up1. As in [19], we allow arbitrary sequencing of control steps, rather than invoking the restriction that every control node needs to be followed by an actual real-world activity.

In order to accommodate different levels of abstraction in modeling, activities can be compound, in which case they contain another workflow to describe the coordination steps at one level lower.

We assume the presence of a supporting database, which is used not only for data storage but also as means to support communication between workflow participants and with the outside world. In ICN, dashed-edges represent input and output data flow between repositories (in square boxes) and activities. For example, in Figure 2, a preliminary application is used by the VerifyPrelim step. In our example, all but the first and last steps get as input and provide as output data in an ApplicationFolder repository; we have omitted this from our diagram for simplicity (and will model it later as a parameter to the activities in question).

We will provide a more precise description of the process model in Section 5, where we make it resemble much more our data model.

3 Components of an Exception Mechanism

Exception handling mechanisms are a family of programming language *control structures* that allow the normal execution of a program to be replaced or augmented by so-called exception handling code when certain special events or conditions occur. (See [36] for a standard introduction.)

The execution of such a control structure can usually be described in terms of the following major steps: 1. signaling an exception, when a problem situation is encountered, resulting in the suspension/interruption of the normal flow of control; 2. collecting a set of *potential handler* procedures that might be used to deal with this situation; 3. choosing among these some (usually just one) *actual handler* and executing it; 4. and, finally, continuing the flow of control somewhere in the original program.

The following is a synthesis of a variety of ideas for exception handling in an object-oriented language context, which will form the basis of our own mechanism.

1. An exception is itself an object, that belongs to a class and can have attributes. The class identifier allows us to distinguish different kinds of exceptions (e.g., `UNDERFLOW` vs `OVERFLOW`), while the attribute values can be used to pass out more detailed information about the context in which the exception occured (e.g., the two arguments and the operation that lead to the underflow). Note that the exception classes can be organized into superclass hierarchies (e.g., `UNDERFLOW` **IsA** `NUMERIC-EXCEPTION` **IsA** `RECOVERABLE-EXCEPTION`).

2&3. The essential feature of modern programming language exception mechanisms is the idea that exceptions should be handled in a *context-dependent* manner. For example, when an underflow occurs, in some contexts one may be able to continue with the value 0, while in others, one has to abort the entire computation. This is operationalized by the convention that when a function S signals an exception of kind X, it is the *invoker*, C_1, of S that is first given the opportunity to provide a handler, since the invoker knows what the signaler was supposed to have accomplished. Usually, if the invoker does not provide a handler then the exception is re-raised/propagated up the calling hierarchy, through intermediate calls to C_2,... till the invocation of C_n provides a handler H, which we write schematically as

`call` C_n `[X ==> H]`

In some languages supporting classes, all users of a particular object (e.g., a memory manager), as well as the specification of the object's class, may offer handlers for consideration. And in languages supporting parallelism/concurrency, threads "collaborating" with the one in which the exception was signaled may also provide handlers (e.g., [33, 22]). Finally, default handlers may be attached to the exception class definition itself. Every language must then provide some specific policy for choosing the actual handler(s) to be invoked.

Note that a single invocation `call C` might need to offer handlers for each kind of exception that C might signal, but the mechanism of subclass hierarchies allows one to provide a handler for a general class, which is then applicable to exception objects raised in any of its subclasses.

4. 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[2]. At the end of the handler H, it has the opportunity to **resume** C_k for some k, which means that (a) all invocations of C_j, for $j < k$ are *terminated*, (b) in C_k, flow of control continues 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. If H does not have a resume or retry statement, then all the invocations C_n, ..., C_0, are terminated, and flow of control continues in C_{n+1} after the call to C_n.

Initially, for simplicity of implementation, most languages took "termination" to mean that the procedure activation record is simply cleared off the run-time stack. More recently, there has been a recognition that a procedure that is about to be terminated should be offered the opportunity of performing **clean-up actions** on the objects that persist beyond its lifetime.

In the programming language context, all the above aspects of exception handling must be pre-programmed in the original code. One of the advantages of the workflow context

[2] For convenience, let us rename the original signaler S to be C_0

will be to permit *end users* to act as exception handlers for unanticipated exceptions. A second advantage, will be the presence of database notions. So for example, the clean-up actions can be greatly facilitated by the presence of "backward-recovery" transaction-like operations such as `checkpoint`, `commit` and `abort`. The clean-up section can also be used for "forward-recovery" compensating actions (as in sagas [20]), which may be preprogrammed or once again directed on-line by users (e.g., sending follow-up e-mail messages).

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

4 Objects and Exceptions

The fundamental motto of our approach [9] is

An exception occurs when some constraint is violated

In our case, every constraint will be associated with a class through an attribute, and so the constraint can be identified by the <class name, attribute name> pair. This means that in applying the exception mechanism detailed in the previous section, there is no longer a need to declare a specific exception object to be raised for every constraint. Instead, a single super-class

```
class EXCEPTION {
        forClass : CLASS
        forAttrib: ATTRIBUTE-ID }
```

suffices. Of course, it is acceptable to declare a subclass of `EXCEPTION`, such as `POLICY-CONSTRAINT-VIOLATION`, and have this be raised when any of several constraints chosen by the modeler is violated. By associating the handler with the exception raised, rather than the constraint, we save repeating redundant information.

In any data model there are two kinds of constraints: those inherent in the data model itself (e.g., an individual cannot have a value for an attribute that is not part of the definition of some class of which the individual is a member), and those explicitly asserted by designers as part of the domain model captured by the schema (e.g., `age` values are integers). In fact, as in Taxis, we can associate with classes arbitrary integrity constraints as values of attributes:

```
PERSON::younger!:  (self.age < self.mother.age - 14)
```

Constraints are of course useful for catching data entry errors and for setting up efficient storage and access structure. Thus, if `b` is an instance of class `PERSON` only, and we try to perform the update `store(b,mather,e)`, then we would get an exception since `mather` does not appear in the definition of `PERSON`, nor of any of its super classes. This exception likely alerts us of a mistake and might have the handler repairing the update by changing `mather` to `mother`. On the other hand, an update like `store(b,birthMother,f)` may turn out to be a case of an exceptional fact not anticipated in the schema (most people have the same mother as birth-mother, or do not need to have both recorded in the database).

In this case we encounter the need to have **persistent exceptions** in the database. Similarly, if we have only partial information about the age of person `jane`, in rare cases we might want to store the string `"young"` as the value for `age`, since no precise value is known. The actual storing of the exceptional value shall be accomplished by **resuming** the update `store(jane,age,"young")` that was interrupted by the exception signal, and **excusing** this act[3]. However, it is important to realize that future operations on these "persistent exceptional values" will have to be monitored closely, since programs 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 }
```

using syntax like

```
new EXNAL-ATTRIBUTE(onObj=jane,attrib='age).
```

(Balzer [3] has felicitously called such an object a *pollution marker.*) There is then a built-in constraint that any time such an attribute is being accessed or modified, an exception (which is exactly this pollution marker) is raised to alert the program or 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 specialized subclasses of `EXNAL-ATTRIBUTE`, to capture common categories of persistent exceptions (e.g., `VALUE-UNKNOWN`, `WRONG-UNIT-OF-MEASURE`, `EXTRA-ATTRIBUTE`, etc.) so that some additional semantics can be captured. To support on-line exception handling, such extensions to the class hierarchy can be made at run-time.

Note that in certain cases, such as when the assertion `PERSON::younger!` is violated by the combination of values `jane.age=20`, `jane.mother=diane`, and `diane.age=30`, it is not clear which value is "exceptional" (e.g., it could be that the mother relationship between jane and diane is exceptional) so we need human intervention to *blame* the appropriate fact(s) by marking them polluted. (Recall that the ability to have human agents act as exception handlers is one of the most important additions made to the programming language approach, in order to deal with *unanticipated exceptions*, which are the focus of our paper.)

We extend the above idea to objects being *exceptional instances of classes* by considering a built-in attribute `instanceOf`, which has as values the set of classes to which an object belongs. One can then mark the restricted attribute `instanceOf[{PERSON}]` exceptional for some object, say `Spock` of Star Trek fame; as a result, for example, any iteration over the extent of class `PERSON` will stop and raise an exception when reaching `Spock`, but can then be resumed or skipped in that loop, using the usual exception handling mechanism.

The main advantages of the mechanism described in this section is that we do not have to weaken constraints (and thus weaken the ability to detect errors or to find efficient storage

[3] Of course, it will be important to have ways of placing administrative controls over such actions; in [9] this is accomplished by creating an `EXCUSE` object that records who, when and why allowed the exceptional property in the database, as well as expiration date for the excuse.

structures) nor do we have to attempt the impossible, which is to anticipate all the kinds of facts that might need to be stored in the database: the on-line handling of exceptions provides flexibility to the system.

5 Activities and Exceptions

As mentioned in the beginning, our plan is to reify activities so that they are also persistent objects in classes with attributes. This general approach was implicit in our original work on workflow-like *scripts* in Taxis [6] and explicit in our requirements modeling languages RML [23] and Telos[30]. It has been independently adopted by the InConcert[35] workflow product. Although we will not use this point in the present paper, this means that activity/workflow descriptions can also be organized into subclass hierarchies, with the usual concomitant advantages of abbreviation, reuse and change propagation due to inheritance (see [8]).

5.1 Elementary Activities.

For our purposes, the most important attribute of an activity is `responsibleAgent` – to be filled by some agent who has authority on achieving the goal of the activity. The reason for this is that the `responsibleAgent`, proposed in many process models (e.g.,[16]) is the natural agent to perform on-line exception handling.

In addition, activities have as attributes parameters, inputs and outputs, as well as local variables. Many process models also support *role* attributes, whose range restrictions specify requirements on agents who can carry out the activity. (This allows more flexibility in work assignment to humans.) Since these are data-valued attributes, the previous section already deals with exceptions to constraints involving them.

We will not model explicitly the body of an elementary activity in this paper, but since we plan to continue with the view that exceptions correspond to violations of assertions, let us consider a taxonomy of assertions that can be associated with an activity (elementary or compound)[4]:

trigger: a condition that needs to be true before an enabled action is offered to an agent for execution; among others, this is used to implement conditional branches in workflows in conjunction with an OR-fork.

checkTest: a condition that is verified at run-time at some specified point during the action's execution; two specialized versions of it, **initialTests** and **finalTests**, are checked immediately after the start of the execution and immediately before its end. Initial tests can anticipate the violation of integrity constraints or check dynamic ones (e.g., that the age 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, after some global raise affecting most employees).

[4] This is a considerable refinement of the assertion mechanism offered in several process models as well as the Eiffel language

invariantTest: a condition that needs to be monitored throughout the execution of the activity. For example, in workflows, absolute deadlines on the termination of an activity are frequent invariant tests.

initialAssumption,**finalGoal**: these assertions describe conditions that are *assumed* to hold at the beginning, respectively end, of activities, and are essentially *proof obligations* for the correct functioning of the entire process. For example, an initial assumption of a B-tree search would be that the tree was properly constructed, while a final goal of a bubble sort would be that the array values are in increasing order. 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, and as before, violations of assertions/tests will lead to exceptions being raised.

It is usual to describe the "enactment life-cycle" of an activity by providing a state transition diagram. Figure 1 is our elaboration of the diagram in [11][5]. As indicated in the

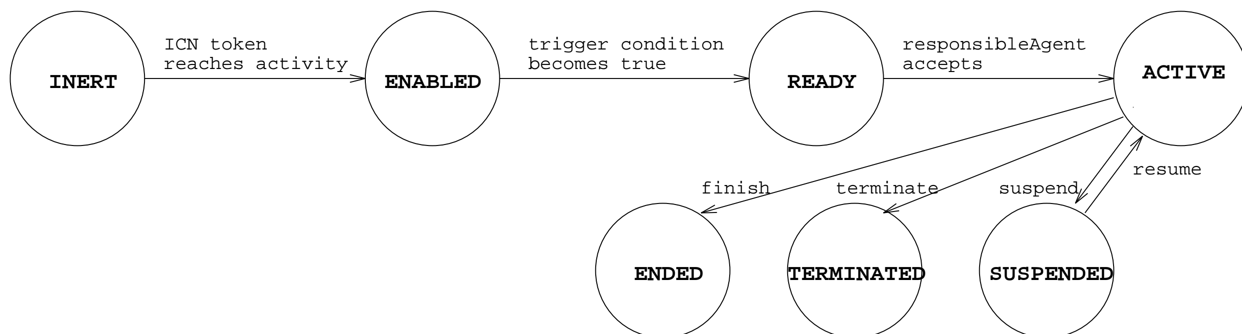


Figure 1: State diagram for activities

diagram, the workflow engine is normally in charge of moving the workflow instance from `INERT` to `ENABLED` to `READY` under the appropriate conditions. If the task is automated, the transition to `ACTIVE` is immediate; otherwise the responsible person has to first signal that (s)he accepts. (Rejection is modeled by raising an exception.) When the activity is finished, the transition to `ENDED` is taken. 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`. Finally, a **terminate** action cancels the activity, the main difference between activities in states `ENDED` and `TERMINATED` being that the former are guaranteed to have established their `finalGoals`, while the latter do not provide such a guarantee.

In the spirit of object-centered reification, we make each of the above states be a class, and "entering/leaving a state" is synonymous with "being added/removed from the class extent as an instance."

[5] In the interest of simplicity and uniformity, we have made the diagram be identical for elementary and compound activities.

If an exception occurs during a task execution, the activity is suspended, and is in fact inserted into a subclass SUSPENDED-BY-EXN:

```
class  SUSPENDED-BY-EXN   IsA SUSPENDED {
            cause : EXCEPTION;            }
```

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 TERMINATED-BY-EXN:

```
class  TERMINATED-BY-EXN   IsA TERMINATED {
             cause : EXCEPTION;             }
```

Essentially, the above approach (which is also applied to compound activities) is a partial step towards a reflective architecture for workflow enactment — only the data for the workflow engine is available for modification, not the engine itself. It provides a number of advantages:

- As also pointed out by [35], having the extents of the classes of activities in various states available in the database makes it possible to query them *using the standard query facilities* of the database (rather than requiring specialized report generators), and obtain all sorts of ad-hoc information about the status of individual or collections of workcases. For example, in the Admission example, we can ask how many FinalDecision actions are currently going on with some specific person being the responsibleAgent.

- Suppose we keep track in the database of the times when an activity becomes or stops being an instance of each state class. Using an appropriate (possibly 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 (i) active, (ii) active or suspended, (iii) enabled but not fired, (iv) waiting to have an exception handled, etc. For example, it is possible to assert about activity B that B must be ready (i.e., enabled and with true trigger) by 1/20/1998, by stating as an invariant that if it is in class INERT or ENABLED, then the current date ($TODAY$) must be less than 1/20/1998.

  Once again, 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**".

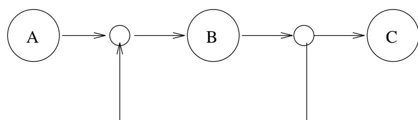
- During exception handling, the handler itself can explicitly move some activity instance from one state to another. For example, when the previous assertion is violated because today is 1/20/1998, the *handler* may decide that activity B cannot be delayed, and may then move the object from class INERT or ENABLED to READY, giving an appropriate excuse of course. When the workflow engine resumes, 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. In fact, we can do away with user-issued "suspend" and "resume", and have these be executed through the exception handling mechanism.

5.2 Compound Activities

Compound activities, like all activities, have attributes for parameters, `responsibleAgent`, local variables, and various kinds of assertions. For the `Admission` example, we might have

```
workflow class ADMISSION
   resonsibleAgent:  ADMISSIONS-OFFICER
   input
         prelim : PRELIMINARY-APPLICATION
   locals
         appl   : APPLICATION-FOLDER
   initialAssumptions
      havePrelim!:   NOT (prelim = nil)
   finalGoals
      haveDecision!:  (appl.decision = "admitted") OR
                      (appl.decsion = "denied")
```

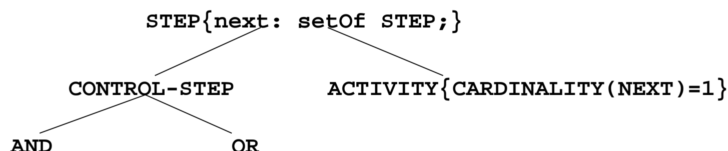
Workcases are also members of the classes representing stages of enactment introduced in Figure 1. The main difference is that compound activities have as body an ICN network of activity and control steps. To model the steps, we make them be explicit attributes of a compound activity class. The names of the attributes can be arbitrary (though they should probably be chosen to describe the state of the world reached after the step completes) and the range of values for each step attribute is specified as usual by giving a range class. For example, the ICN1 control graph



may have the following (preliminary) class definition:

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

As suggested by the example, there are built-in classes `OR` and `AND` for control nodes. To model the actual sequencing constraints (the edges), we require step values to be instances of a special class `STEP`, which has an attribute `next`, whose value is the *name* of the successor step to be used by the workflow engine in figuring out where the token should flow. (This attribute is single valued for activity steps, 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`='end). Therefore, a

more complete specification of ICN1 is

```
class ICN1 IsA ICN  {
        step-a: A{next='step2};
        step2: OR{next={'step-b}};
        step-b: B{next='step4};
        step4: OR{next={'step-c,'step2}};
        step-c: C{next='end};            }
```

If asked to enact this workflow, the engine now creates an instance `wf1` of ICN1, initializes all steps with `INERT` instances of the corresponding activity classes, and moves `wf1.step-a` into `ENABLED`. Thereafter it cycles through all activity instances (which are mostly values of steps) and checks whether appropriate transitions from one class to another can be carried out (e.g., from `ENABLED` to `READY` if the trigger is true). When an activity step finishes, it helps enable the successor step indicated by the value of the `next` attribute[6]. There is however detected a technical difficulty in computing the "enabled step" relationship for ICN workflows, as formally specified in [19], and we offer an approach in [28] which deals with the inherent nondeterminism in an relatively efficient manner in many practical cases.

Note that according to ICN1, `wf1.step-b` is expected to have at most a single `ACTIVE` value at any moment of time – the then executing instance of that step for `wf1`. However, in general there might be a set of such activity instances as values for a step, representing concurrently executing activities. (This might happen even in the case of `step-b`, as a result of an exception, which would artificially enable `step2` and thus start a second execution of `step-b` before the first one is completed.)

5.3 Exceptions in workflows

We are now ready to explore the ways in which our techniques for handling exceptional data objects (Section 4) can be used together with the above representation of the workflow instances in order to deal with various exceptional occurrences.

Consider first dealing with desired deviations from the pre-specified control flow. Take a net ICN2 of the form

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

with steps called `step-a,step-b,step-c,step-d` for simplicity. Let `wf2` be an instance of ICN2 in which `step-a` is currently active. Suppose that some exception is raised signaling an unmet deadline, and the `responsibleAgent` wants to react to it. Here is how (s)he could perform any of the three exceptional alterations in the running of the workflow mentioned in Section 1:

- To exchange the order of execution of B and C, the handler would simply perform the updates

  ```
  wf2.step-a.next := 'step-c;
  wf2.step-c.next := 'step-b;
  wf2.step-b.next := 'step-d;
  ```

[6] Note that `next` does not model the *specific* individual activity that follows one that has just terminated – only the step in which it must belong. If we want to have a trace of the causal connections between the activities in a workflow instance, it would be best to add a system-maintained attribute called `successorOf`, whose value is a set of activity instances – those that led to the enablement of the current activity.

Note that if some of these sets had more than one instance, the `responsibleAgent` would have to choose the one whose successor needs to be modified. For example, if there were several actions in `step-a`, but only one was active then we could use

```
wf2.step-a[ACTIVE].next:= 'step-c; .
```

- To start step B before step A is finished we need to do two things: first, make the instance of B be enabled, by moving `wf2.step-b` into `ENABLED`; second, we need to take care of a subtle problem: when `step-a` finishes, it would normally lead to the instantiation of *another* B activity; if the previously enabled `step-b` had already finished, there would be no obvious way to see that this was undesirable. To avoid this, we mark the attribute `wf2.step-b` to be exceptional, using an instance of `EXNAL-ATTRIBUTE`. Then, whenever the workflow engine accesses or tries to modify this attribute, we get an exception from the access/modification function. In response to this exception, we can terminate the attempt to create a new instance of B for `step-b` when this is due to the completion of `step-a`. (Note that this mechanism would continue to work even in the case of nodes with multiple predecessors.)
- To make B and C perform in parallel, we would add additional attributes for the 2 conjunctive control nodes and newly named steps for B and C:

```
alt-step1 : AND{next={'alt-b,'alt-c}};
alt-b     : B{next='alt-step2};
alt-c     : C{next='alt-step2};
alt-step2 : AND{next={'step-d}}
```

  and again change the successor `next` for the running instance of `step-a` to be `alt-step1`.

Consider next violations of the user-specified conditions, which are attributes of elementary or compound activities. These can now be handled using the exception handling mechanism described in Section 4, applied to the activity object. In particular, if the `responsibleAgent` of one activity does not handle the exception, it is re-raised to the responsible of the containing/invoking compound activity. 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 be selectively suspended by the exception handler for C of course.

The handler can, among others, use the excuse mechanism of `EXNAL-ATTRIBUTE` on activity instance objects to permit violations of such constraints to "persist" and continue with the process.

Consider now conditions in the category **initialAssumptions**. We propose that these need to be checked for an activity instance `b`, if `b` was enabled by a step `p` such that either `p.next` is exceptional or `p` is an instance of `TERMINATED`, rather than `ENDED`. The reason for this is that the predecessor of the `b` did not get a chance to establish its goal, and the current step might have depended on that goal. For example, when starting step B before step A is completed (in the second scenario above) `step-b`'s **initialAssumptions** should be verified. In the more specific context of the `Admission` workflow, if in an exceptional case one wants

to skip over the `Review` step (because, say, the applicant already has offers from elsewhere) then this check would result in the `responsibleAgent` of `Decision` being alerted that the input data normally expected is not complete, so if `Decision` is automated it is likely to fail, and hence should be carried out by hand.

Similarly, run-time verification of the **finalGoals** should be performed when the activity is manually moved to the `ENDED` state, when it is exceptionally resumed after a failed **initialTest** or **initialAssumption**, or when a compound activity's coordination has been exceptionally changed. For example, 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. Such a condition should be expressed as a **finalGoal** for the corresponding workflow.

Of course, the violation of even such assumptions and goals may be excused, but at least the `responsibleAgent` has been made aware of the potential problems.

6 Related work and conclusions

6.1 Exceptions vs. Evolution

There have been a number of papers on the evolution of process models, including [12, 19, 4], and some techniques for accommodating run-time exceptions are based on the use of schema-editors applied to running workflow instances (e.g., [35]). However, we believe there is an important distinction to be made between exceptional occurrences 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., [9]) and schema evolution in a database (e.g., [5]). The latter task might be prompted by multiple occurrences of the former, but it is a different process, more like schema development. There are relatively few constraints one can impose on the changes that can be made – it is mostly a matter of cataloging the operations needed to achieve *any* goal.

One important issue in schema evolution is what to do with existing individuals/on-going enactments. This issue has been addressed [19, 4] by finding invariants that should not be violated even during schema evolution. (We have argued earlier that these can be captured by our **finalGoals** for workflows.)

On the other hand, as we have remarked in the introduction, our framework supports a limited *additive* form of evolution by codifying exception handling into procedures and invoking these as in programming languages. In fact, this coincides with a form of abstraction of details: in the first pass, designers consider only the normal cases, and in fact are encouraged to make assertions about the normal case holding (including temporal constraints on the duration of various steps); in the second pass, they name exceptions raised by *some* of the constraints, and provide handlers for them, thus making the workflow more robust. The remaining constraints require run-time handling of exceptions.

This distinction between a *main-line* and deviations from it has also been studied and supported in a language for organizational description developed at MIT in the early 80's [27].

6.2 Other closely related work

As mentioned earlier, WIDE is a workflow system implemented using active database technology [11], whose conceptual model is in some ways similar to ours (it is also based on the ICN framework, and has similar state transition diagram to ours). It facilitates the declarative specification of constraints and their error handling by associating *condition-reaction* pairs to each action. This essentially *static* mechanism for handler association contrasts with our more dynamic approach, which uses the invocation hierarchy. Of course, our approach has an entire additional dimension, dealing with the consequences of resuming after an exception, and having *persistent* violations.

Exceptions in transactional workflows have of course been present almost from the beginning, including [17].

From the software-engineering process modeling camp, most relevant is the work on deviations during enactment [14]. This work distinguishes "tolerable" deviations (which are state transitions imposed in cases when the trigger constraint is still false) from intolerable ones (where certain specially asserted invariants are violated). We have already discussed how our mechanism can handle, as special cases, these situations. 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[7]. As future work we will be considering such a "pollution analysis" tool as support for users who are asked to choose which values are to be *blamed* for an exception.

In what [32] calls the situated work camp, [7] is a report of a workflow system where activities are modeled as an executable network of obligations, a request from one person to other agents. An obligation network is constructed based on a overhead transparency metaphor, and "sheets" are categorized into local modifications and general specifications, which enables flexible control over the extent of changes in schema. Means of manipulating a network, such as altering the network and adding new tokens, are provided. Although the notion of "errors" exists together with built-in detection mechanism thereof to issue warnings, there is no explicit notion of exceptions, especially user-defined ones, as well as their handling.

In contrast to our more technical focus, several papers have addressed the organizational aspects of exceptions [37]. In [26], 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, [34] suggests, supported by a substantial empirical study, an interesting dimension for a taxonomy: "exceptionality", the degree of availability of applicable rules and exception handling routines in the organization given to address a deviation from a *main line*. He goes on to show a meta-model for general event handling in which each classified exception is handled by situation analysis and creating a new event handling rule and/or updating the rule base either at the instance level or at the type level of a procedure.

[7] This idea is based on Balzer's exploration of the utility of exceptions in the software engineering context[3], which in turn uses some of the technical notions in [9].

6.3 Contributions

Our primary objective has been to lay out a computational framework for providing generic, flexible, and disciplined means of exception handling in workflow/process enactment.

The main contribution of this paper is a proposal for dealing with unanticipated exceptions to both data schema and process schema constraints, which is *uniform* across the two areas and which deals with violations of both user-specified constraints and those inherent in the data/process model itself.

This is accomplished primarily by the conjunction of three techniques: 1) Activities and workflows are reified as persistent objects whose attribute values and class memberships encode the information maintained by the workflow engine for each workcase enactment (e.g., the current state, the `next` step), and are kept in the same database as the data needed by workflows. 2) The technique for handling exceptions, and especially for *permitting exceptional values to persist*, described in [9], is extended so that it can be used to permit all the desired kinds of deviations from the norm both in data and process descriptions (e.g., specifying an exceptional, alternate `next` step and flow of control). 3) Responsible agents, or other users, are allowed to act as on-line exception handlers, in order to exercise judgement in dealing with unanticipated exceptions or when encountering "persistent exceptions."

The resulting system avoids vagueness by correlating exceptions with violations of constraints. As part of our disciplined approach to exceptions, we have paid particular attention to the *consequences* of permitting exceptions to persist. This includes our original notion of exceptional attributes (recorded by pollution markers), which raise exceptions when they are touched later, in order to alert the user that the value being used in the computation may need to be interpreted differently than in the ordinary case. It also includes our use of **initialAssumptions** and **finalGoals** to protect as much as possible the workcase from performing illegal or non-sensical operations when the control flow of the process has been changed on the fly, with possibly insufficient consideration (e.g., invoking an activity without having its normal predecessor completed).

By representing the states of a workflow instance as classes with extents, for which the time when membership begins and ends is recorded, we have permitted the application of the standard query and update language for the DBMS to take the place of a large number of special operations on workflows such as **suspend**, report generation on the status of large numbers of workcases, and especially stating temporal deadlines for the normal termination or beginning of workcase steps. Furthermore, a number of intrinsic constraints for the process model, such as only allowing the activity to start when its **trigger** has value true, can once again be superseded in exceptional cases by *explicitly* moving the individual activity ¿from one class to another (e.g., `ENABLED` to `READY`), and excusing the falseness of the appropriate constraint (e.g., **trigger** in this case).

We note that although our presentation was couched in object-centered terminilogy, there is no obstacle to representing this information in ordinary relational tables, and using active rules, for example, to implement workflow enactment, as in [13].

There is considerable work that remains to be done in completing our enterprise. Among others, we will need to take a closer look at the traditional issues of concurrency control and error recovery. As argued in [2, 38], it might be the case that further research will yield better *workflow-specific* solutions to these issues than the standard Advanced Transaction Models.

In particular, we find quite interesting the proposal for a variety of ways of sharing documents in the APEL graphical process language [16], which is based on long practical experience of its authors, including commercial workflow products. And in the spirit of [2], we plan to examine the use of the exception mechanism (together with primitive transaction facilities like begin/commit/abort) as a way of providing programmable yet declarative simulations of ATM mechanisms. A logical formal semantics for workflows with exceptions, based on AI theories of action, is also on our agenda, and this would likely lead to a technique for generating a Prolog prototype system for workflows.

Acknowledgements:

This research was supported in part by NSF Grant IRI 9619979.

References

[1] K.R. Abbott, S.K. Sarin, "Experiences with Workflow Management: Issues for the Next Generation", in the Proceedings of ACM Conference on Computer Supported Collaborative Work (CSCW94), Chapel Hill, NC, 1994.

[2] Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Guenthoer, R., Mohan, C. "Advanced Transaction Models in Workflow Contexts", *Proc. 12th International Conference on Data Engineering*, New Orleans, February 1996.

[3] R. Balzer Tolerating Inconsistency in Software Development, *Proceedings of the 13th International Conference on Software Engineering*, pp. 158-165, May 1991.

[4] S. Bandinelli, A. Fuggetta, and C. Ghezzi. "Software Process Model Evolution in the SPADE Environment." *IEEE Transactions on Software Engineering*, December 1993.

[5] Jay Banerjee, Won Kim, Hyoung-Joo Kim, Henry F. Korth: "Semantics and Implementation of Schema Evolution in Object-Oriented Databases." *Proc. ACM SIGMOD'87*: 311-322

[6] J. Barron, "Dialogue and Process Design for Interactive Information Systems Using Taxis", *ACM SIGOA Conference*, 1982.

[7] D.P. Bogia, S.M. Kaplan, "Flexibility and Control for Dynamic Workflows in the wOrlds Environment", in *Proceedings of the Conference on Organizational Computing Systems*, Milpitas, CA, November 1995.

[8] A.Borgida, J.Mylopoulos, H.K.T.Wong "Generalization as a basis for software specification", M. Brodie, J. Mylopoulos, and J. Schmidt Editors, *On Conceptual Modeling*, pp.87–114, Springer Verlag, 1984.

[9] A. Borgida, "Language Features for Flexible Handling of Exceptions in Information Systems", *ACM Transactions on Database Systems*, Vol 10, No.4, December 1985, pp565-603.

[10] A. Borgida, "Description Logics in Data Management", *IEEE Transactions on Knowledge and Data Engineering*, October 1995.

[11] F. Casati, S. Ceri, B. Pernici, G. Pozzi. "Conceptual modeling of workflows",*O-O ER'95*, Gold Coast, Australia, Springer Verlag, Dec. 12-15, 1995.

[12] F. Casati, S. Ceri, B. Pernici, G. Pozzi. "Workflow Evolution". *ER '96* (Cottbus, Germany).438-455

[13] Fabio Casati, Stefano Ceri, Barbara Pernici, Giuseppe Pozzi: "Deriving Active Rules for Workflow Enactment". *Proc. DEXA 1996*: 94-115

[14] G. Cugola, E. Di Nitto, C. Ghezzi, M. Mantione, "How To Deal With Deviations During Process Model Enactment", *Proceedings 17th International Conference on Software Engineering,* Seattle, WA, May 1995.

[15] G.P. Cugola, E. Di Nitto, A. Fuggetta, C. Ghezzi, "A Framework for Formalizing Inconsistencies in Human-Centered Systems", *ACM Transactions on Software Engineering and Methodology,* September 1996.

[16] S. Dami, J. Estublier, M. Amiour, "APEL: A Graphical Yet Executable Formalism for Process Modeling", Automated Software Engineering (ASE) to appear (obtainable at `ftp://ftp.imag.fr/pub/ADELE/ASE.ps`).

[17] Umeshwar Dayal, Meichun Hsu, Rivka Ladin, "Organizing Long-Running Activities with Triggers and Transactions". *Proc. ACM SIGMOD Conf.* 1990, pp. 204-214

[18] C. Ellis and G. Nutt, "Modeling and Enactment of Workflow Systems", in *Application and Theory of Petri Nets*, M. Ajmone Marsan Ed., *LNCS 691*, Springer Verlag, 1993, pp.1-16.

[19] C. Ellis and K. Keddara, "Dynamic Change within Workflow Systems," draft of a paper with the same title published in the Proceedings of the Conference on Organizational Computing Systems, 1995.

[20] Hector Garcia-Molina, Kenneth Salem: "Sagas", *Proc SIGMOD'87*: 249-259.

[21] Dimitrios Georgakopoulos, Mark F. Hornick, Amit P. Sheth: "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure." *Distributed and Parallel Databases* 3(2): 119-153 (1995)

[22] J.Goslin, B.Joy, G.Steele, "The Java Language Specification", Addison-Wesley, 1996.

[23] S.Greenspan, J.Mylopoulos, A.Borgida. "Capturing more world knowledge in the requirements specification", *Proc. 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982, pp.225-234.

[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 Science Publishers, pp.355-370, 1990

[25] M. Kifer, Won Kim, Y. Sagiv: "Querying Object-Oriented Databases." *Proc. SIGMOD'92*, pp.393-402.

[26] M. Klein, "Exception Handling in Process Enactment Systems",*ECAI96, 12th European Conference on Artificial Intelligence,* W. Wahlster Ed., John Wiley & Sons, 1996.

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

[28] T. Murata, A. Borgida, "Towards an On-line Enablement Algorithm for ICN Control Nets", forthmcoming Technical Note, available by e-mail from authors.

[29] John Mylopoulos, Philip A. Bernstein, Harry K. T. Wong, "A Language Facility for Designing Database-Intensive Applications." *ACM TODS* 5(2): 185-207 (1980)

[30] John Mylopoulos, Alexander Borgida, Matthias Jarke, Manolis Koubarakis, "Telos: Representing Knowledge About Information Systems." *ACM TOIS* 8(4): 325-362 (1990)

[31] *NSF Workshop on Workflow and Process Automation in Information Systems*, A.Sheth editor, May 8-10, 1996, Athens Georgia, `http://lsdis.cs.uga.edu/activities/NSF-workflow`.

[32] G. Nutt, "The evolution toward flexible workflow systems", *Distributed Systems Engineering*, Dec. 1996.

[33] A. Romanovsky , J. Xu, B. Randell "Exception Handling and Resolution in Distributed Object Oriented Systems", *ICDCS '96; Proceedings of the 16th International Conference on Distributed Computing Systems*, May 27-30 1996, Hong Kong,, pp. 545-553

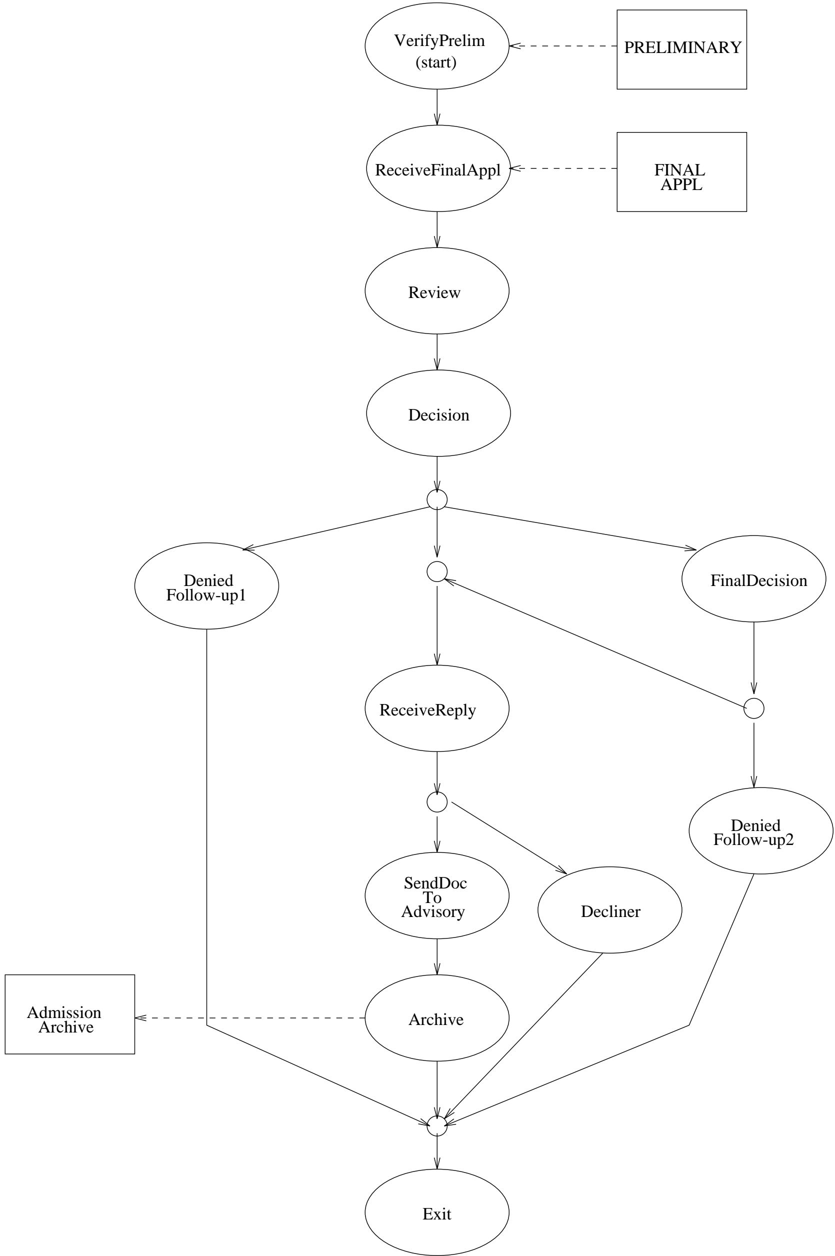
[34] H.T. Saastamoinen, "On the handling of exceptions", Ph.D. Thesis, University of Jyvaskyla, Jyvaskyla, 194 pages, 1995.

[35] Sunil K. Sarin: "Object-Oriented Workflow Technology in InConcert". *COMPCON 1996*: 446-450

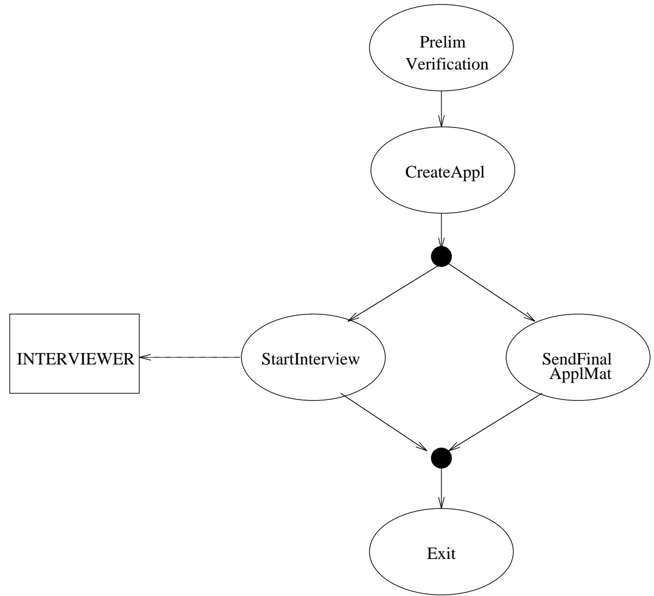
[36] R. Sebesta, *Concepts of Progragramming Languages*, 3rd edition, Addison Wesley, 1996.

[37] Diane M. Strong, Steven M. Miller: "Exceptions and Exception Handling in Computerized Information Processes." *ACM TOIS* 13(2): 206-233 (1995)

[38] D. Worah and Amit Sheth. "Transactions in Transactional Workflows" in *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds., Kluwer Academic Publishers, 1997 (to appear).



Top-level



VerifyPrelim expanded

Figure 2: ICN diagrams for portions of the Admission workflow