WERDE: a pattern-based tool for exception design in workflows *

F. Casati †, S. Castano †, M.G. Fugini †, I. Mirbel †, B. Pernici †
{casati,castano,fugini,mirbel,pernici}@elet.polimi.it

† Politecnico di Milano
Piazza L. Da Vinci, 32 - I-20133 Milano, Italy

‡ University of Milan
Via Comelico, 39 - I-20135 Milano, Italy

Abstract. When designing a WorkFlow schema, often exceptional situations, such as abnormal termination or suspension of task execution, have to be dealt with by the designer explicitly, as well as operations which typically occur at the beginning and termination of tasks. This paper describes WERDE, a tool for designing exceptional behavior within a WF schema by reusing pre-configured exceptions skeletons.

1 Introduction

WorkFlow (WF) systems are currently becoming a more and more applied technology in the direction of providing a common framework for supporting activities within office information system.

WFs in the organizations are modeled to capture the possible sequences of activities needed to reach a given goal, and to provide tools and links to the existing Information Systems of the organization. Many aspects have to be considered in conjunction with the pure activity flow. In particular, the documents and information associated with the activities, the performers of the activities and their role in the organization, and the possible exceptions to the normal flow of activities. In other related fields, such as in Software Engineering and Information Systems, analogous techniques and tools are used combining data and function modeling to represent the application functionalities and to represent both normal and abnormal situations and exceptions[12].

Currently, few techniques and tools exist for WF design. Some of them [15] are based on rules and catalogs of predefined WF portions for WF management. In general, the designer of a WF schema has to provide many details in order to obtain a schema which is executable by a WorkFlow Management System (WFMS). This description requires the specification of both the normal flow and of the possible variations due to exceptional situations that can be anticipated and monitored.

* Research presented in this paper is sponsored by the WIDE Esprit Project N. 20280 and by the CNR-CINI Project “Informatica per la Pubblica Amministrazione”.

‡ This work is being published by Springer-Verlag LNCS n° 1413 (Proceedings of CAISE’98 Conference, Pisa, Italy, June 1998).
This paper addresses the issue of providing support in designing exceptions in a WF focusing on exceptions which can be statically foreseen. It describes an WERDE (Workflow Exceptions Reuse and Design Environment), an environment of tools for the specification of WF schemas. Exceptions can be included in a schema by reusing (i.e., selecting and customizing) a set of pre-defined generic exceptions. The term “exception” is used in a broader sense than simply for exceptional situations. In fact, exceptions model also conditions regarding the starting, termination, or repetitions of WF instances (called cases) or regarding the periodical occurrence of operations in the WF execution.

The support environment, called WERDE, consists of a catalog of pre-designed WF schema portions, and of tools for accessing the catalog. We are mainly concerned with the specification of exceptional situations in WFs, that is, situations where events occurring in a WF schema may cause a task to be canceled, or a WF case to be terminated. Such exceptional situations may be designed in advance and inserted in the WF schema specification by including exception handlers for the most likely occurring events or by designing typical termination handling strategies. Another situation is the WF start phase which can be subject to typical situations. These situations may be included as a pre-set part of the schema specification.

The catalog of WERDE stores WF schema portions, mainly dealing with exceptions, to be reused during the design of a new schema. The contents of the catalog is a set of typical patterns[14] enabling the designer to deal with frequent occurrences of similar situations in WF design in given application domains, and with generic situations that may occur in any application domain. The catalog is based on:

- rules: rules model exceptions, starting and termination situations, and WF schema evolution due, for instance, to changed requirements deriving from new types of exceptional events;
- patterns: typical rules may be used in several designs. Patterns described in the paper are rules, or sets of related rules, to control a given typical situation. Patterns are a way for the designer to specify the WF schema by reusing previous design experience.

Rules in WF modeling have been considered, among others, by [10]. However, they are mainly used to represent the complete flow. In other approaches to WF modeling ([11, 13]), a strong emphasis is put on providing a simple and graphic interface to WF design. However, if such interfaces are to capture all the details of anomalous flows, readability is penalized, such as in Petri Net based approaches ([2]).

The approach of this paper based on pattern reuse aims at improving the speed and quality of WF schema design. A pattern is a generalized description of a rule or sets of rules that can be associated to a WF schema, either to model exceptions, or to model starting and termination situations, or situations to be controlled due to schema evolution. Reuse, also based on patterns, is being successfully applied to software development ([5, 16, 17]).
The paper focuses on the functionalities of the WERDE environment which allow the designer to reuse patterns stored in a catalog for dealing with exceptions in WF schema design. Functionalities for pattern design are also presented.

The paper is organized as follows. In Section 2, we illustrate the catalog of patterns, and describe the use of rules in WF design by presenting exceptions and patterns. In Section 3, we present the functionalities of WERDE for access and management of the catalog and show the tools usage for exception design in WF schemas and for exception analysis. Finally, in Section 4, we comment and give some hints about future work.

2 A Pattern Catalog

This section describes the pattern catalog of the WERDE environment where a set of pre-defined WF patterns is stored at different levels of abstraction and genericity in a layered structure.

Patterns model situations that can occur in WF design in an application-independent way, at different levels of genericity (e.g., integrity constraints holding for schemas of any application domain, or schema fragments describing frequently occurring combinations of tasks in a given application domain). They can be described either using a language for specifying exceptions and/or through the WF process description language WPDL [6].

In Figure 1 the categories of patterns stored in the catalog are depicted. Patterns are classified at three different layers.

- **Built-in patterns.** Patterns at this level deal with reaction to events and time alarms (Basic patterns), integrity and security constraints over WF data (Authorization, Integrity, and Count patterns), WF management (Starting, Interaction, and Continuation patterns), and with changes in the organization structure (Organization and Schema evolution patterns). They describe context-dependent and time-dependent knowledge related to basic exceptional situations independent of the semantics of all possible applications.

- **Generic tasks.** Patterns at the generic tasks layer allow the description of frequently occurring activities in WF design in the form of tasks, independent of the documents involved in their execution (e.g., a task for registering information in a document).

- **Generic application-oriented structures.** This layer of the catalog contains WF fragments and associated exceptions needed to model frequently occurring situations of one or more application domains. These are frequently occurring combinations of tasks (i.e., WF fragments) generally performed together to achieve an organization goal in a given application domain (e.g., a generic structure describing the concurrent review of a document by several people).

In the following of the paper, we describe exceptions and patterns. In particular, we show how exceptions are specified and used, and how exceptions are then abstracted in patterns. For details about the patterns contained in the various categories of the catalog, the interested reader can refer to [9].
2.1 Exceptions

A short introduction to the exception language is given (see [7] for a detailed description). The Chimera-Exc exception language is based on the Chimera language for active object-oriented databases. An exception is specified by an event-condition-action (ECA) rule (also called trigger in the following). Events denote the occurrence of a possibly exceptional situation; conditions determine if the occurred event actually corresponds to an exceptional situation to be managed, while the action part defines the operations to be performed in order to manage the exception. ECA rules may refer in their event, condition, or action part, to objects and classes of the WFMS database. These include static and dynamic information on WFs and tasks (stored in objects of classes case and task respectively), data relating WF participants (objects of class agent), plus a WF-specific class, bearing the name of the WF, whose objects store process-specific variables local to each single case. For instance, variables needed for the execution of cases of a hotel roomReservation WF, such as the room number or the customer name, are stored as attributes of an object of the roomReservation class.

An exception can be sensitive to different kinds of events: data events are raised upon modification of the content of the WFMS database. Data events include the creation of a new object (create), the modification of an object's attribute (modify), and the deletion of an object (delete). For instance, event
modify(task) is raised when an attribute of an object of the task class is modified. Workflow events are raised when a case or task is started or completed. WF events include caseStart, caseEnd, taskStart, and taskEnd. For instance, event taskStart(myTask) is raised when an instance of task myTask is started. Temporal events are raised upon the occurrence of a defined temporal instant. They can be expressed as deadlines, temporal periods, or interval elapsed since a specific date or time. Finally, external events, qualified by their name, are notified by external applications: these may correspond to applicative situations, such as a phone call by a customer cancelling a reserved room. An example of external event is raise(cancelReservation).

The condition part consists of a declarative query, expressed by a formula evaluated over the WFMS database state. The formula is a conjunction of predicates, and includes a set of variables to which bindings are associated as a result of the formula evaluation: if the result of the query is empty (i.e., if no bindings are produced), then the condition is not satisfied, and the action part is not executed. Bindings resulting from the formula evaluation are passed to the action part in order to perform the reaction over the appropriate objects. The condition includes class formulas, for declaring variables ranging over the objects of a specific class (e.g., task(T)), type formulas for declaring variables of a given type (e.g., real(R)), and formulas expressing comparisons between expressions (e.g., T.executor="Lisa"). Objects affected by events are captured in the condition part by the occurred predicate. Thus, for instance, in an exception triggered by the caseStart event, the binding with the started instance is obtained by means of the predicates case(C), occurred(caseStart,C).

The action part includes data manipulation actions, updating the content of the WFMS database, and operations executed through calls to the WF engine. The first category includes the primitives create, delete, and modify. For instance, delete(Log,L) deletes all objects of class Log to which L is bound after the condition evaluation. The second category includes primitives to start, rollback, or suspend the execution of cases and tasks, assign or delegate the execution of tasks, and send messages to agents. Examples are delegateTask(T,A), startCase(mySchema, startingParameter), and notify(T.executor, "deadline is approaching").

A sample trigger activated at the end of cases of the roomReservation WF is as follows. The condition determines which is the case that has caused the triggering (predicates case(C), occurred(caseEnd,C)), retrieves the object storing the variables of the just completed case (roomReservation(R), R.caseId=C) and then checks if the customer has not paid (R.paymentDone=FALSE). If the condition is verified (i.e., if bindings are produced), then the action part is executed, causing the start of an instance of the cancelBooking WF and providing the customerID as input parameter.
define trigger missingPayment
events caseEnd
condition case(C), occurred(caseEnd,C), roomReservation(R),
R.caseId=C, R.paymentDone=FALSE
actions startCase("cancelBooking",R.customerId)
end

2.2 Patterns

We introduce patterns as generalized descriptions of triggers and exceptional situations that can frequently arise in WF modeling. Patterns predefine typical rules or set of rules that capture the knowledge about the occurrence of an exceptional situation and the actions that can be performed to deal with it. Patterns consist of predefined parts, parameterized parts, and optional parts. Parameterization and optionality support pattern re-usability and adaptation in WF design for the aspects related to exception modeling and handling.

Pattern Description: The pattern are described by a specification part and a sample usage as follows.

Specification part: The specification part is the description of an exception or of an application, including a set of textual fields and a WPDL and/or Chimera-Exc portion implementing the pattern. In particular, the pattern specification is structured in the following fields:

- name, uniquely identifying the pattern in the catalog of all patterns;
- intent, a textual part describing the purpose of the pattern (intended scope and known uses);
- classification, according to the categories and sub-categories of the catalog;
- template, containing the core specification of the pattern. It can be provided in two forms:
  - For patterns representing exceptions, the template contains the specification in terms of events, conditions, and actions. The action part provides only suggestions, because exception patterns focus on how to capture exceptions, more than on how to fix reactions, which are application dependent: in a given application, a warning message can be a satisfying reaction, whereas a cancelation of the case can be more suitable in another application of the same pattern.
  - For patterns representing application structures, the template contains the diagrammatic WF schema skeleton describing the application fragment.

The template usually contains parametric fields to be filled in with specific values provided by the user; mandatory and optional parts can also be specified.

- keywords, which are a set of user-selected terms that can be used to refer (select, search, etc.) to the available patterns in the catalog;
- related to, establishing links among patterns to combine them in different structures;
- guidelines, providing suggestions to the user about possible usage and personalization of patterns.

In Figure 2, an example of pattern specification is given. It is a pattern to be used to start a new case upon termination of another one.

<table>
<thead>
<tr>
<th>Pattern Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name:</strong> Termination</td>
</tr>
<tr>
<td><strong>Intent:</strong> This pattern allows the definition of the initiation of a case upon termination of another case, considering its status (in terms of WF variables).</td>
</tr>
<tr>
<td><strong>Classification:</strong> Continuation pattern</td>
</tr>
<tr>
<td><strong>Template:</strong></td>
</tr>
<tr>
<td><strong>Keywords:</strong> case termination</td>
</tr>
<tr>
<td><strong>Related to:</strong> Related to</td>
</tr>
<tr>
<td><strong>Guideline:</strong> The condition may contain clauses related to values of one or more variables in the terminated case.</td>
</tr>
</tbody>
</table>

The complete description of the pattern-specification language can be found in [8].

Sample usage part: Since the user of the catalog is intended to be an expert of the application under design and is not required to have a detailed knowledge of the Chimera-Exc syntax, the pattern model is endowed with the user oriented sample usage pattern. This is a set of instantiations of patterns on specific examples. They show how patterns can be personalized in different contexts and applications by illustrating how parameters of patterns can be supplied by the designer to produce a concrete WF. The sample usage part of the pattern description is a set of WF-specific instantiations of patterns related to an application domain. It is an instantiation of the variables/parameters appearing in the template field of the pattern specification according to the domain selected to construct the example. The missingPayment trigger given in Sec. 2.1 is a sample usage of the pattern termination presented in Figure 2.
**Pattern Reuse:** Two basic mechanisms are provided for reusing patterns in WF design: *specialization* and *instantiation*.

*Pattern specialization:* When designing triggers, one may want to specialize a trigger previously defined. The specialization focuses on the event and condition parts. Three ways are given to specialize a trigger. They can be combined.

- The first way consists in rewriting the trigger using more specific generic-terms. Consider a pattern composed of a trigger with an event part containing the generic term `<referenceEvent>`. If one wants to write another pattern, with the same structure as the previous one, but focused on external events only, one can specialize the exception part of the pattern by replacing `<referenceEvent>` with the more specialized generic-term `<externalEvent>`.
- The second way consists in instantiating part of the trigger. This is done by replacing a generic term by a non-generic one. For example, `<class>` in a given trigger of a pattern can be specialized into *task* in a more specific pattern dealing only with tasks.
- The third way consists in adding conditions and actions to a given trigger in order to obtain a more specialized one. These additions can be done only in the condition and actions parts. Adding events is not allowed since this would change the invocation of the trigger.

*Pattern instantiation:* When designing a new WF, after choosing the suitable pattern, this has to be instantiated. This can be done by replacing the generic-term with a value, for example `<referenceEvent>` becomes *documentArrival*. A more specific name must also be assigned to the pattern and the triggers. If there are optional parts, the user has to decide if he/she takes the optional parts or not. If several possible actions are defined, it is also necessary to choose one or several of these actions.

A more complete and formal description of the instantiation and specialization mechanisms can be found in [8].

### 3 Specification of the Exception Design Environment

In this section, we present the environment supporting the pattern-based approach illustrated in Section 2. The tool is mainly given in terms of specification of its functionalities which deal with *pattern management* in the catalog and with the *design and analysis of exceptions*. The current prototype environment implementing most of the described functionalities is finally described.

#### 3.1 Management of Catalog

Management of the pattern catalog deals mainly with the *insertion, modification* and *deletion* of patterns. However, it also takes into account the management of the *categories* useful to classify and retrieve patterns.
**Category management:** As presented in Section 2, the catalog is divided in categories. Each is divided in sub-categories. *Termination*, for example, is a sub-category of the *built-in patterns* category. Since the catalog evolves during time, it is possible to add new categories and sub-categories, and to remove some of them. For removal, a check is performed to ensure that no pattern of the catalog refers to the category or sub-category being deleted.

**Pattern management:** This functionality deals with *creation, modification* and *deletion* of patterns.

When *creating* a pattern, its parts have to be filled in by the user: name, intent, classification, template, keywords, related-to patterns and guidelines. The pattern name must be unique. The classification conforms to the categories and sub-categories of the catalog. Patterns keywords can be selected from a tool-proposed list of keywords and belong to a thesaurus automatically updated whenever new keywords are specified for newly defined patterns. Guidelines can be textual information to be compiled off-line by the pattern designer. Links between templates ("related-to" fields) can be defined by activating an "add link" and starting a corresponding dialog window. Sample usages can also be written to facilitate pattern reuse. If the specialization mechanism is used to create a new pattern, an existing pattern $P$ can be selected, and the designer can select the parts of $P$ that must be modified. The pattern has to share at least one keyword of its generic pattern (the pattern from which it is derived) and has to be placed in the same category/sub-category. Links are maintained in the catalog between patterns belonging to a specialization chain.

When *modifying* patterns, in addition to the modification of its features (intent, template, keywords, related-to fields and guidelines), new sample usages can be added to an existing template, by defining an “executable example” and by selecting the template of interest from the catalog. It is also possible to change the category and/or sub-category of a pattern. This means changing the location of the pattern in the repository, but also changing the location of the patterns linked to it through specialization or generalization, in order to maintain the catalog consistency.

*Deletion* of patterns is performed by selecting the pattern of interest. The tool then performs a set of basic integrity checks:

- if another pattern uses the current one, the link is also removed;
- if the pattern is involved in an inheritance hierarchy (is inherited by at least a pattern $p_2$, and possibly inherits also from at least pattern $p_1$), the user has to choose (i) to delete also $p_2$, or (ii) to build a new inheritance link between $p_1$ and $p_2$.

### 3.2 Design of Exceptions

The environment support the pattern-based approach to the design of exceptions in WF schemas limited to the design of its exceptions. In the following, the word *exception schema* denotes the set of exceptions related to a given WF schema.
Exceptions can be designed following a bottom-up process, based as much as possible on reusing the patterns available in the catalog through the instantiation mechanism. The tool guides the designer during the process of exception specification by providing both classification information, which facilitates the search in the catalog, and guidelines to enter values for pattern parameters and to compose patterns into more complex structures.

When creating an exception schema, exceptions can be written (created) from retrieved patterns. These patterns are instantiated/personalized for the WF schema at hand. New exceptions can also be defined from scratch, if the available patterns do not fit the WF requirements.

Two search modalities are provided, namely, keyword-based and classification-based search. The first relies on a pre-defined thesaurus of terms loaded at pattern specification time to characterize the pattern within the catalog. The second modality allows for the retrieval of patterns based on two orthogonal views of the catalog:

- typology of pattern, with respect to event types (e.g., temporal, external, data);
- abstraction level of patterns in the catalog (built-in, generic, application-oriented).

The instantiation consists in binding the pattern parameters on the basis of the semantics of the WF schema, by exploiting the syntax-driven capabilities of the tool and its interface facilities. A set of dialog windows drives the designer in entering the patterns fields, possibly enabling the visualization of the guidelines and usage examples associated with the template under manipulation. New conditions and actions can be added to the exceptions derived from the patterns. The tool dialog facilities support also the definition of new exceptions from scratch. Triggers within a given exception schema can be modified and/or deleted as needed by the designer. No special checks have to be performed, because the triggers, in this part of the tool, are not linked to the patterns of the catalog.

Exception schemas can be modified, with respect to their exceptions. These modifications include insertion of triggers, by searching in the catalog or by writing them from scratch, modification of the existing triggers, and deletion of some of them if they are no longer needed.

3.3 Schema Analysis

Exceptions in WFs are typically designed individually, by identifying the exceptional situation, selecting the appropriate pattern from the catalog, and instantiating it in order to obtain a Chimera-Exc exception. However, although an exception considered individually may be correctly specified, mutual interactions among exceptions may lead to unforeseen and undesired behaviors. For instance, a set of exceptions may trigger each other, or two exceptions might operate on the same data; if they are concurrently triggered, the result of their execution depends on their non-deterministic execution order.
Exceptions analysis aims at statically verifying that such undesired interactions do not occur. In particular, it aims at verifying two significant properties for a set of exceptions: termination and confluence [1, 3, 4, 18]. Termination holds if exceptions do not trigger each other indefinitely, so that rule processing eventually terminates. Termination analysis is typically based on building graphs depicting triggerings and activations among exceptions, and on detecting cycles in these graphs (see [4, 7] for details).

Confluence requires the final effect of the rule processing (represented by direct or indirect changes to the WFMS database) to be independent on the (non-deterministic) scheduling of triggers having the same priority [8]. Sufficient conditions for confluence have been proposed in [1, 3, 8]. Unfortunately, the above properties are undecidable in the general case.

A fully automated analysis of interactions between exceptions is generally difficult to achieve, since it requires to understand the semantics of conditions and actions. A simple, conservative solution consists in checking that exceptions query and operate on different classes. More sophisticated and less conservative techniques for rule analysis, which consider the semantics of conditions and actions, have been proposed in [3].

3.4 WERDE Architecture

This section presents the architecture and the functionalities provided by the WERDE tool and gives a usage example. The architecture is depicted in Figure 3.

The definition of an exception for a WF schema is performed through the Schema Editor module, based on reusing the available patterns. From within the Schema Editor, the designer can interactively search the catalog for a suitable generic pattern using the Exception Retriever module. Retrieved patterns are instantiated/personalized for the WF schema at hand, through the Exception Writer module, which can also be used to define new patterns or exceptions from scratch, if the available ones do not fit the WF requirements. Exceptions defined for a schema can be analyzed by exploiting the Schema Analyser module of WERDE. The Catalog Manager performs the catalog management functions that is, insertion, modification and deletion of patterns.

A Visual C++ Windows prototype of WERDE is available, operating in a Client/Server environment, in accordance with the design guidelines of the other wide tools [6].

Managing the Catalog: The pattern catalog is extensible by defining new patterns and/or specializing the existing ones. To this purpose, WERDE provides maintenance functionalities to manage the catalog (Catalog Manager module) by allowing insertion of new patterns and modification or deletion of existing ones. It also allows one to add/remove pattern categories and sub-categories.
Pattern management:

- **Creation:** Newly defined patterns are stored in the catalog, under the proper category/sub-category. Fields of the pattern specification are interactively filled in. The pattern template is specified by invoking the *Exception Writer* editor. Patterns keywords can be selected from a proposed list. Keywords in the thesaurus are automatically updated whenever new keywords are specified for newly defined patterns. Guidelines can be inserted as textual information to be compiled off-line by the pattern designer.

- **Modification:** An existing pattern can be modified, after being selected using the *Exception Retriever*. The designer can then select the parts of the pattern that must be modified, and operate on them through dialog windows. In Werde it is possible to add new sample usages to an existing pattern template, by defining an “executable example” and by selecting the template of interest from the catalog.

- **Deletion:** it is performed by selecting the pattern of interest, and the tool performs the basic integrity checks.

An example of work session, dealing with the specification of the *termination* pattern shown in Figure 2, is presented in Figure 4.

*Insertion/removal of categories:* Categories and sub-categories can be added to the catalog at each level. Removal of categories implies the deletion of all sub-
categories, provided that the designer has already cancelled all the corresponding patterns.

Using the Catalog: Werde guides the designer during exception specification by providing both classification information, which facilitates the search in the catalog, and guidelines to enter values for pattern parameters and to compose patterns into more complex structures.

A typical exception-design session consists in activating the Schema Editor module. The main functionalities that can be invoked from within this module deal with the creation, modification and deletion of exception schemas. The functionalities provided through the creation and modification of exception schemas are the following:

- Retrieval, to access the catalog and search for suitable patterns. The two search modalities presented above are implemented (keyword-based and classification-based search).
- Insertion, to support the interactive definition of exceptions starting from a retrieved pattern or from scratch. The instantiation consists in binding
the pattern parameters on the basis of the semantics of the WF schema, by exploiting the syntax-driven capabilities of the Exception Writer module and the template interface facility of WERDE. This facility consists of a set of dialog windows which drives the designer in entering the patterns fields, possibly enabling the visualization of the guidelines and usage examples associated with the template under manipulation. The dialog facilities of the Exception Writer support also the definition of new exceptions from scratch. The instantiation of the termination pattern works as in the sample work session of Figure 5, where the pattern is instantiated to obtain the trigger missingPayment of section 2.1. The instantiation steps consist of activating the exception writer module, selecting the item to be instantiated (<wfName>W in the example of Figure 5), and overriding it with the desired Chimera-Exc condition (roomReservation(R)).

Fig. 5. Instantiation of the termination pattern

- Modification, to modify exceptions inside exception schemas.
- Deletion, to remove exceptions from exception schemas.
- Analyze, to analyze the exceptions written for the current schema, using
the Schema Analyser module. WERDE currently provides a basic set of the static analysis functionalities, by focusing on interactions within each pair of exceptions. In WERDE, two triggers are considered as conflicting if they are active simultaneously and contain incompatible actions; actions are incompatible if the order of their execution leads to different states of the WF management database [6]. A table of incompatible actions is maintained to support the analysis, which can be performed at three levels:

- task level, to check exceptions defined for a given task;
- task set level, to select a number of tasks for which the exceptions must be analyzed;
- schema level, to check all the exceptions defined in the schema.

Anomalous situations detected by the Analyser are signaled to the designer who can activate the Exception Writer for editing the involved exceptions.

4 Concluding Remarks

We have presented the WERDE environment supporting the design of exceptions in WF schemas. The environment is based on a catalog of WF patterns which are a generalized description of triggers which can be reused when designing a new WF schema. Exceptions model both abnormal situations and typical start/end/repetitions of operations in a WF. We have presented the structure of the pattern catalog and the mechanisms for pattern reuse. We have also described the related functionalities of the WERDE tools for exception design and catalog management.

Currently, the catalog is populated with a set of patterns which have been constructed by analyzing a set of real cases provided in the WIDE project testbed. A set of primitives for pattern reuse has been fully specified in [8] and is being tested to check the quality of these patterns in terms of significance and accessibility for reuse. Particularly the sample usages of patterns stored in the catalog are proving to be an effective means for guiding the designer in selecting the appropriate patterns, in instantiating them appropriately, and in inserting the exceptions correctly in the WF schema.

The implementation of WERDE has currently completed most of the functionalities described in the paper. Some functionalities, such as pattern design and schema analysis, are being extended in order to achieve a more complete and effective suite of functions. For example, for schema analysis, we plan to consider exception termination analysis and confluence analysis on rule sets, and to consider in more detail the semantics of actions and conditions. For pattern design, we plan to have functionalities which help the catalog administrator in abstracting exception skeletons from WF schemas in a guided way and storing new patterns in the appropriate categories with suitable links to the existing repository. In fact, the catalog is regarded as an extensible tool to be populated gradually with new categories of patterns to be defined by examining further real WF applications. Further points of research regard the use of exceptions as a basis for WF dynamic evolution and the design of a user feedback mechanism.
for validation of the quality and effectiveness of a given WF, using adaptive techniques.

References


This article was processed using the LaTeX macro package with LLNCS style