Exception Handling Patterns for Processes

Barbara Staudt Lerner
Mount Holyoke College
Department of Computer Science
South Hadley, MA 01075
+1 413-538-3250
blerner@mtholyoke.edu

Stefan Christov, Alexander Wise,
Leon J. Osterweil
University of Massachusetts, Amherst
Computer Science Department
Amherst, MA 01003
+1 413-545-2186
{christov, wise, ljo}@cs.umass.edu

ABSTRACT
Using exception handling patterns in process models can raise the abstraction level of the models, facilitating both their writing and understanding. In this paper, we identify several useful, general-purpose exception handling patterns and demonstrate their applicability in business process and software development models.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *control structures, patterns.*

General Terms
Languages.

Keywords
Exception handling, pattern, process modeling.

INTRODUCTION
A process model describes the activities and interactions of multiple agents working together to complete a task. Process models have been used in many application domains, such as software engineering, business processing, healthcare provision, and conflict resolution.

As processes typically involve coordination of multiple people and machines, there are many opportunities for problems to arise. People might be unavailable when they are needed, the actions they take might be incorrect or inappropriate, deadlines might not be met, or needed resources might be unavailable. In each of these cases, an exceptional condition has arisen and appropriate action should be taken to address that exceptional condition. As processes grow larger and more complex, it is of increasing importance to not only specify precisely the normal execution of the process, but also to provide a precise definition of how exceptional situations should be handled. Specifying exceptional behavior requires identifying the task in which the exception occurred, exactly what the exception is, what tasks are needed to remedy the exception, and how to proceed once the exception has been handled.

Process specifications that neglect addressing the above questions carefully and precisely are incomplete and inadequate. One approach to dealing with exceptions is to allow a process to be modified dynamically when an exception occurs [2]. This may be acceptable in some situations, but especially in the case of processes used in critical situations exception handling must be defined as precisely and completely as possible both to provide essential guidance and to facilitate analysis of the correctness of the exception handling.

Incomplete processes often result in misunderstanding, which in turn can lead to errors with serious consequences. In the medical domain, imprecise or missing specification of how a process should deal with exceptional situations can lead different people to handle the same situation differently, based on personal style, level of experience, and the actions of other people [6]. Yet, Henneman et. al. [15] observe that descriptions of medical processes often capture only the standard process and leave out the handling of exceptions. This results in inconsistent handling of exceptions, which creates the potential for errors due to misunderstanding. It also makes it impossible to analyze whether or not the handling of exceptions preserves process properties that are required and desirable.

Exception handling support within an appropriately articulate process language facilitates the desired clear separation of exceptional behaviors from more normal behaviors and can serve as a vehicle to keep large and complex process definitions under intellectual control. Indeed, our experience suggests that support for the explicit specification and handling of exceptions in application programming languages such as Java makes programs written in these languages clearer and more amenable to effective intellectual control. Osterweil [18] suggests that this is no less important in a process model and process language than in application software and programming languages.

Through our experience in defining processes in a variety of domains, we have realized that certain behaviors recur frequently and thus seem to comprise specifiable patterns. The identification and the subsequent use of such patterns has facilitated writing and reasoning about processes that employ these patterns. Some of these patterns deal specifically with exceptions and their handling. We believe that recognition of exception handling patterns and use of standard idioms to encode them can lead to improved readability and understandability of process definitions.

In this paper, we briefly introduce patterns that specify particularly effective use of exception handling. As in the field of design patterns, we have found that thinking in terms of patterns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WEH '08, November 14, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-229-0 ...$5.00.

helps to raise the level of abstraction associated with process definitions, making it easier both to create and to understand processes using them.

The exception handling patterns we have observed seem to be used to meet the following broad needs:

- Presenting alternative means to perform the same task.
- Inserting additional tasks before returning to normal processing.
- Aborting the current processing.

In the remainder of this paper, we first describe the exception handling mechanism of the Little-JIL process language and then discuss the exception handling patterns that we have found, using Little-JIL to elucidate the discussion and present examples. We chose to use Little-JIL mainly because its powerful support for specifying exception handling seems to reduce the size and complexity of the patterns we will present, thereby enabling us to focus more sharply on the nature of the patterns themselves. A more complete catalog of exception handling patterns, including more examples written both in Little-JIL and as UML 2 Activity Diagrams, can be found online at http://www.mtholyoke.edu/~blerner/process/patterns/ExceptionHandling/.

2. LITTLE-JIL

Little-JIL [20] is a hierarchically-scoped process language with a graphical syntax, semantics that are precisely defined by finite state machines [16], and a runtime environment that allows execution on a distributed platform [21]. The basic unit of Little-JIL processes is the step, represented graphically by an iconic black bar as is shown in Figure 1. It is useful and reasonable to think of a Little-JIL step as a procedure. Thus, in particular, a step declares a scope and includes an interface, represented by the circle at the top of the figure, which specifies the artifacts that are required and produced by the step. These artifacts are the arguments to the step, and the resources needed in order to support step execution. Pre- and post-requisites, represented by the triangles to the left and right sides of the step name, may be used to specify, respectively, processes that are responsible for checking that the step can be performed, and that the step was performed correctly.

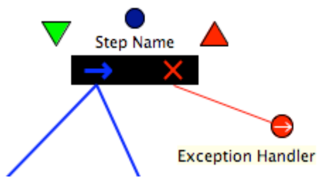

Figure 1: Little-JIL Syntax

2.1. Substeps

Little-JIL substep decomposition is represented by having substep icons connected to the left side of the parent step icon by edges. The edges are annotated with specifications of the artifacts that are passed as parameters between the parent and child steps. Edges may also carry annotations specifying that the child step may be instantiated more than once, as defined by a logical expression, a Kleene * or +, or by an integer or integer range. Each parent step specifies the execution order of its substeps using one of the four sequencing icons, shown in Figure 2, which appears in the step bar above the point where the substep edges are attached. There are four different sequencing icons: **sequential**, which indicates that the substeps are executed in order from left to right; **parallel**, which indicates that the substeps can be executed in any (possibly interleaved) order; **choice**, which allows any one of the substeps to be executed; and **try**, which indicates that the substeps are executed left to right until one succeeds. The choice and try sequencers both offer an opportunity for the process modeler to represent that there may be multiple ways of accomplishing an activity. The key difference is that in the case of the choice sequencer, all of the alternatives are presented to the process performer, often a human, who can decide which of the choice substeps to perform. In contrast, the try sequencer defines an order in which the alternatives should be attempted. These two sequencers play key roles in the two patterns presented in detail in Section 3.1.

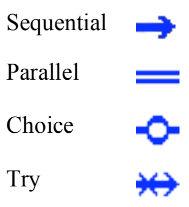

Figure 2: Little-JIL Sequencing Icons

2.2. Exception Handling Mechanisms

A parent step may offer exception handling facilities to its descendant steps. These facilities are defined by exception handlers connected to the parent by edges attached to the right side of the parent's step bar immediately below an 'X'. Each exception edge is annotated to identify the type of exception that it handles. Exception handling in Little-JIL is divided into three parts: signaling that an exceptional condition has occurred, determining what steps are invoked to handle the exceptional condition and then executing those steps, and finally determining how the process should proceed after the specified steps have been completed.

Copying programming languages such as Java, a Little-JIL step signals an exceptional condition by throwing an exception object. Unlike such languages however, Little-JIL steps are guarded by pre- and post-requisites, which function much like assert statements, and signal their failure by throwing exceptions as well. Similar to pre- and post-conditions in some traditional programming languages, the bundling of a step together with its requisites creates a scope that cleanly separates the task from its checking, but ensures that the step can only be called in the proper context, and specifies the guarantees that the step can make to its callers. As in a traditional programming language, once an exception has been thrown, Little-JIL determines how the exception should be handled by searching up the stack of invoking ancestor steps. Once a handler for the exception has been located and executed, the process specification is consulted to determine how execution should proceed. Unlike most contemporary languages, which generally only permit the handling scope to complete successfully, or throw an exception, Little-JIL offers four different exception continuations, shown in Figure 3:

- **Completion**, represented by a "check mark" icon on the edge connecting the handler to its parent step, corresponds to the usual semantics from traditional programming languages. The step to which the exception handler is attached is finished, and execution continues as specified by its parent.
- **Continuation**, represented

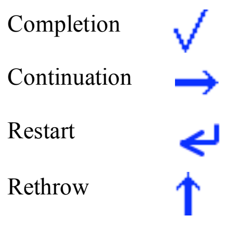

Figure 3: Little-JIL Exception Continuation Icons

by a right arrow icon, indicates that the step to which the exception handler is attached should proceed with its execution as though the substep that threw the exception had succeeded. It is important to note that this is not resumption – if several levels of scopes had to be searched before finding a matching handler, those scopes have still been exited.

- **Restart**, represented by a backwards pointing arrow, restarts the step to which the handler is attached.
- **Rethrow**, represented by an up-arrow, allows the handler to propagate the triggering exception up to an enclosing scope as in a usual programming language.

The continuation icon is placed on the edge connecting the exception handler to its parent. In case the exception handler has no steps associated with it, the continuation icon is embedded in a circle at the end of the continuation handler's edge (as in Figure 1).

Exception handling mechanisms have been present in programming languages for many years, going back at least as far as to CLU [17] and early work on exception handling mechanisms by Goodenough [12] and Yemini [22]. The focus of this paper is not on Little-JIL's exception handling mechanisms, but rather on frequently observed higher-level exception handling ***patterns*** that can be cleanly specified by utilizing such mechanisms.

3. EXCEPTION HANDLING PATTERNS

In software engineering, patterns are best known in the context of object-oriented design. Object-oriented design patterns [10] present interesting ways to combine classes and define methods to address common design problems, allowing designers to reuse high-level solutions to problems rather than reinventing solutions for each new design problem. Similarly, we have found that there are interesting ways to define higher-level exception handling patterns that address common exception handling problems. These patterns arise through particular combinations of the location where an exception is thrown, where the exception is caught, and where control flows after the exception is caught. Thus, it is not just the exception handling mechanism that is of interest, but how that mechanism is used within the context of reaching a particular process objective. The end result is to allow process designers to think in terms of these patterns and to be able to recognize when these patterns are useful within a process. By reusing a pattern, the process designer is relieved of the burden of designing every detail of every process from first principles and can instead use the patterns to guide the designer in the appropriate use of the mechanisms offered by the language.

In this section we briefly introduce the exception handling patterns that we have identified, providing more detail on two of the patterns to give a better understanding of the work. Following the style introduced in the classic Design Patterns book [10], we present our patterns as a catalog. For each pattern, we provide:

- Its name
- Its intent – what recurring behavior the pattern captures
- Its applicability – in what situations the pattern should be used
- Its structure – the general structure of the pattern expressed in Little-JIL
- One or more examples of process fragments that use the pattern

We organize the patterns into a set of categories. We describe the nature of each category, and then present the specific patterns that it contains. Our examples are drawn from different domains to suggest the generality of the patterns.

3.1.Trying Other Alternatives

One common category of exception handling patterns describes how to deal with decisions about which of several alternative courses of action to pursue. In some cases, such decisions are based upon conditions that can be encoded directly in the process, essentially using an if-statement to make the choice. In other cases, however, it may be difficult to capture a priori all conditions for which each course of action is best suited. In those cases, it is often most effective to just present the process performer with alternatives to try. If the alternative that is tried fails, another alternative is to be tried in its place.

In such cases it is often desirable to simply enumerate a set of alternatives without specifying completely the exact conditions under which each alternative is to be taken, but rather using exception handling to move on to untried alternatives. In this category we have identified two different exception handling patterns: ordered alternatives and unordered alternatives

3.1.1 Pattern Name: Ordered Alternatives

Intent: There are multiple ways to accomplish a task and there is a fixed order in which the alternatives should be tried.

Applicability: This pattern is applicable when there is a preferred order among the alternatives that should be tried in order to execute a task.

Structure: The Little-JIL diagram in Figure 4 depicts the structure of the Ordered Alternatives pattern. The alternatives are tried in order from left to right. If an alternative succeeds, the task is completed and no more alternatives are offered. If execution of an alternative throws an exception, it is handled by trying another alternative. This continues until one of the alternatives succeeds.

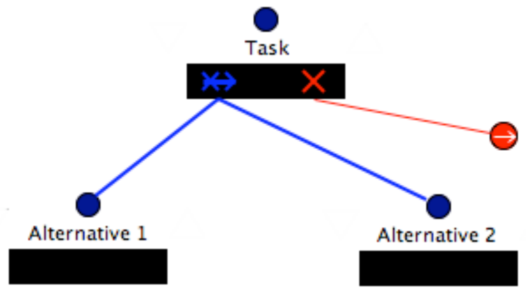

Figure 4: Structure of the Ordered Alternatives Pattern

Sample Code and Usage: Figure 5 shows the use of the Ordered Alternatives pattern in planning travel to attend a conference. This pattern can be seen in the **Book hotel** step. Here, the process requires first trying to get a reservation at the conference hotel before considering other hotels. If the conference hotel is full, the **HotelFull** exception is thrown. This is handled by causing the **Book other hotel** step to be attempted next.

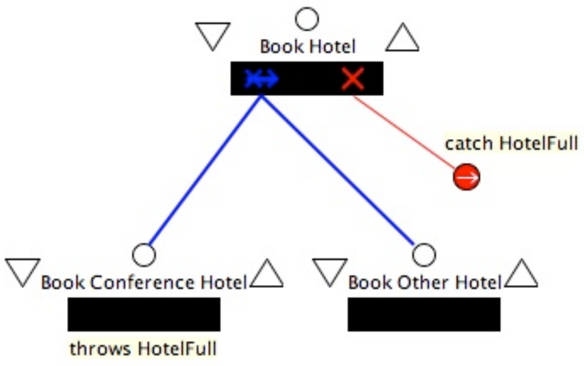

Figure 5: Using the Ordered Alternative Pattern to Select a Hotel

3.1.2 Pattern Name: Unordered Alternatives

Intent: There may be multiple ways of accomplishing a task and there is no fixed order in which the alternatives should be tried. If an exception occurs while trying one way, an alternative is to be tried instead.

Applicability: This pattern applies when there are multiple ways to accomplish a task and it is not known a priori which is most appropriate. In this case, process performers decide which steps to attempt in which order. If an attempted step fails, there is another attempt to complete the task by choosing a different step.

Structure: The Little-JIL diagram shown in Figure 6 indicates the structure of this pattern. In this case, there are two alternatives to choose from. One is chosen to execute and if it is successful, the task is complete. If it is not successful, then an exception is thrown and the alternative is considered.

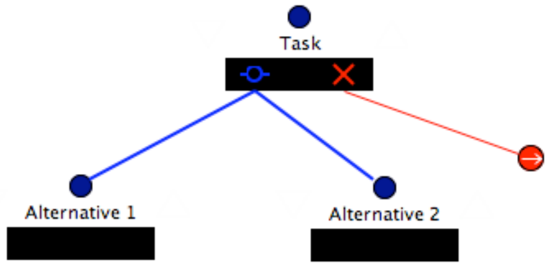

Figure 6: Structure of the Unordered Alternatives Pattern

Sample Code and Usage: Figure 7 shows the shipping task in an order management process. Here, the shipper chooses a delivery method by selecting either the **Ship with UPS** or the **Ship with Federal Express** step. If the chosen shipper does not provide the necessary service for this package, an exception is thrown. The exception handler might make a note about which shipper failed and then retry the alternatives with this knowledge.

Combining the Ordered Alternatives and Unordered Alternatives Patterns. Figure 8 depicts examples of both the Unordered Alternatives and the Ordered Alternatives patterns in defining the highest level of a process for developing software. In this example, a software company's policy may be to always try to reuse existing code, if possible, in order to reduce development costs. However, if the reuse of existing code modules is impossible under the given circumstances, then it is necessary to do a custom implementation. These alternatives are represented by the use of the Ordered Alternative pattern in defining the **Implement** activity in Figure 8 where the developers attempt to employ a reuse approach prior to doing a custom implementation.

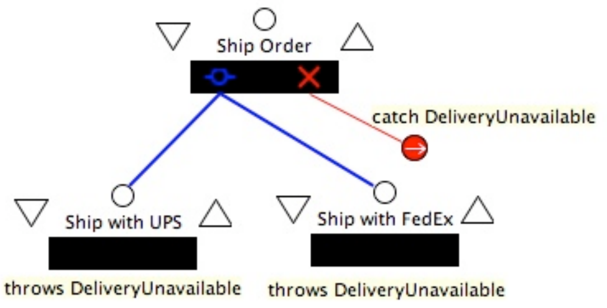
Ship Order
catch DeliveryUnavailable
Ship with UPS
throws DeliveryUnavailable
Ship with FedEx
throws DeliveryUnavailable

Figure 7: Use of the Unordered Alternatives Pattern to Ship an Order

There are several possible approaches in trying to reuse existing code. Some examples are employing inheritance, using delegation, and instantiating a parameterized class. Knowing which alternative to try first might be left to the judgment of the developers. Figure 8 expresses these alternatives by using the Unordered Alternatives pattern in defining the **Reuse existing modules** activity. If the developer's first choice does not work out, this pattern specifies that the developer can then choose one of the remaining alternatives.

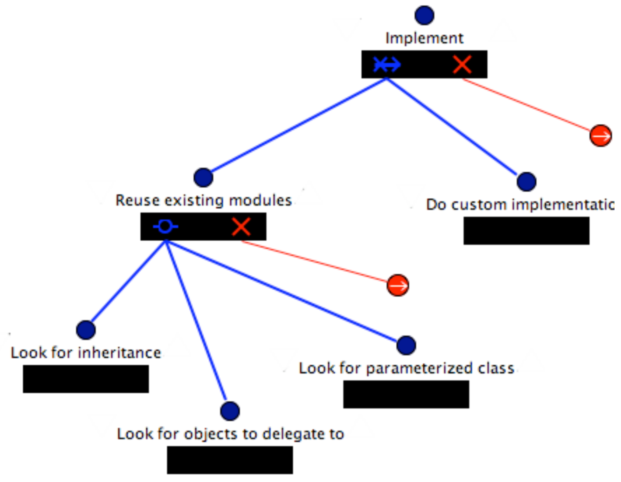

Figure 8: An Implementation Process Combining the Ordered Alternatives and Unordered Alternatives Patterns

3.2.Inserting Behavior

Because of space limitations we only sketch out the intent of each of the remaining patterns,. Further details can be found at http://www.mtholyoke.edu/~blerner/process/patterns/ExceptionHandling/.

Another commonly occurring process behavior is inserting additional actions that are needed in order to fix problems that have been identified during execution of some task. Two patterns describe common approaches to doing this.

3.2.1 *Pattern Name: Immediate Fixing*

Intent: When an exception occurs, some action is taken to fix the problem that caused the exception before continuing with the remainder of the process.

Applicability: This pattern allows the insertion of extra behavior to handle expected, but unusual, situations. It is useful in situations where an expected problem is likely to occur, where a simple procedure exists to fix the problem, and once fixed, the process can continue as if the problem had not been encountered.

Example: In software development, if an error occurs during compilation the error is immediately fixed before coding continues.

3.2.2 *Pattern Name: Deferred Fixing*

Intent: When an exception occurs, action must be taken to record the error and possibly provide partial fixing. Full fixing is either not possible or not necessary immediately. Later in the process, an additional action needs to be executed to complete the recovery from the condition that resulted in throwing the exception.

Applicability: This pattern is useful in allowing the insertion of additional behavior to prevent process execution from coming to a halt. The pattern specifies partial handling of situations that are unusual, yet predictable. This is useful in situations where complete fixing of the exceptional condition is not immediately possible or not desirable (for example, because it would be too time consuming or disruptive).

Example: When a failure occurs during execution of a test program, the bug is not fixed immediately, but rather a notation is made in a test case log so that the bug can be fixed later. After all testing is complete, the test case log is reviewed and the code is fixed at that time.

3.2.3 *Related Pattern: Rework*

While the fixes that can be inserted in response to an exception are as varied as the steps and the exceptions themselves, many fixes entail the need to go back and revisit the results of some earlier step. Cass et al. [5] argued that doing so constitutes what is commonly known as rework, which can itself be modeled as a pattern entailing re-invocation of a step, but in a different context. This characterization permits use of the above patterns to define rework as a pattern involving response to an exception.

Example: Many phases of software engineering benefit from the Rework pattern. If during requirements definition, the creation of a requirements element creates an incompatibility with a requirements element that had been created previously, it then becomes necessary to rework the previously generated requirements, but now benefiting from knowledge of all of the requirements elements created up to this point (notably the requirements element whose recent creation resulted in the observed incompatibility).

3.3.Canceling Behavior

A final category of exception handling patterns is one in which an action being contemplated must not be allowed for some reason.

3.3.1 *Pattern Name: Reject*

Intent: It sometimes becomes apparent that an action being contemplated should not be allowed. The agent contemplating the action must be notified, and allowed to make adjustments or changes and try again, if so desired.

Applicability: This pattern creates an entry barrier to a part of a process.

Example: Many processes have conditions to be satisfied if a portion or the entirety of the process is to continue. For example, an order is rejected if either the customer has bad credit or the supplier cannot fill the order.

4. RELATED WORK

Exceptional situations commonly arise during the execution of processes. In recognition of this, many process and workflow languages include constructs to allow for the definition of exception handlers (for example, Little-JIL [20], WIDE [4], OPERA [13]). While researchers continue to study how best to provide exception handling mechanisms within process languages, exception handling has become more mainstream with its inclusion in languages like WS-BPEL [1], BPEL4WS [8] and products like IBM's WebSphere [9].

Hagen and Alonso [14] identify workflow tasks as being retriable, compensatable, both or neither. In their model, exception handlers may undo the actions of compensatable tasks and attempt retriable tasks, perhaps in a different fashion. Our Ordered Alternatives, Unordered Alternatives and Rework patterns are three patterns of exception handling that capture the notion of retrying tasks. In contrast, the Immediate and Deferred Fixing Patterns primarily compensate for failed tasks. We find the distinction to not be entirely clear, however, as retrying a task may also require compensating for the alternatives already attempted, while fixing a problem caused by a task may also involve performing the original task in an alternative fashion that is only appropriate in the exception handling context.

Golani and Gal [11] express concepts that are similar to compensation and retry as rollback and stepping forward. In their model, an exception handler is expected to perform first its rollback tasks and then its stepping forward tasks, although they do note that either or both may be empty. Our work focuses more on the composition of the exception handling tasks with the normal process tasks to identify higher-level patterns. Within the exception handlers themselves, we expect there to be tasks involved in rollback and stepping forward, although our patterns do not delineate the responsibilities of the exception handling tasks in this way.

In more closely related work, Russell, van der Aalst and ter Hofstede [19] have begun to investigate the occurrence of patterns within workflow. They categorize patterns in four workflow definition semantic domains: control flow, data flow, resources, and exception handling. They approach exception handling patterns by identifying four dimensions associated with exception handling mechanisms: the nature of the exception, if and how the work item that encounters the exception continues, whether other work items are cancelled as a result of the exception, and whether there is any rollback or compensation performed. Based on this analysis, they consider combinations arising from these four dimensions to derive a universe of possible patterns in a bottom-up fashion, without regard to whether these combinations are commonly used in practice and without providing a description of the workflow problems that the pattern might be suitable for addressing. Thus, it is still left to the workflow designer to understand the mechanisms at their most basic level. Identifying those combinations may be useful as a benchmark to determine the exception handling capabilities of a process language. At the same time, these combinations do little to aid process designers in identifying and reusing existing high-level solutions since no general higher-level purpose for a particular combination is provided to guide the designer in choosing a pattern to use. The patterns (combinations) that Russell et. al. identify even lack names that might suggest their usefulness. Instead, they name them based on acronyms derived from the attributes they take on in the four dimensions. For example, they identify 30 patterns associated with expired deadlines alone, two of which are called OCO-CWC-NIL and ORO-CWC-NIL.

Our approach differs from the approach of van der Aalst et al. in that it is driven by recognition of patterns that we have seen occur in processes from multiple domains. We thus approach the identification of patterns in a top-down manner, analyzing uses of exception handling to generalize and extract patterns that we believe to be useful beyond the specific processes in which we have found them.

The concept of process patterns has been explored by Coplien [7] and later by Ambler [3]. However, these patterns differ from our approach in that they focus on the domain of software development. In contrast, our patterns can be applied in many process domains, as the examples in this paper have shown. Further, in our work, we treat processes as software, expressible in well-defined process languages. As a result, our patterns are concerned with how to express recurring behaviors in process languages. The patterns presented by Coplien and Ambler describe the activities within a particular software development activity, like software release, but don't provide a guidance how to express those activities in process languages.

5. CONCLUSIONS AND FUTURE WORK

We have found the exception handling patterns described here to be useful in raising the abstraction level of process models. They provide a way of approaching exception handling by providing a framework of questions we can ask. Can we fix the problem immediately? Is there another alternative the process should offer? Should we reject this input entirely?

Just as there are many uses of classes that do not play roles in patterns, we expect there are needs for exception handling that cannot be met by any of the patterns we define here. Thus, in this work we do not consider all legal ways of combining exception handling mechanisms. Rather we have focused on combinations that we have encountered frequently in our work in defining processes in such diverse domains as software engineering, business, negotiation, and healthcare. While we believe that the diversity of these domains confirms our claim that the patterns are general purpose in nature, we certainly do not believe that this catalog is complete and expect it will grow over time.

We are investigating the role of a process language in expressing exception handling patterns. We have found that exception handling constructs of Little-JIL are particularly good at succinctly capturing some of the patterns presented here, like the Ordered and Unordered Alternatives. We continue to examine other languages to identify the exception handling patterns at which they excel and also to consider new language constructs to facilitate the expression of exception handling patterns.

We are also interested in investigating the exception handling patterns that have arisen in programming languages. Java, in particular, has an active community identifying both useful patterns and anti-patterns. It is possible that some of these general-purpose exception handling patterns have useful analogies in the context of process programming. We are also interested in investigating how well fault tolerance techniques might work in the context of process modeling.

ACKNOWLEDGMENTS

The authors wish to express their gratitude to numerous individuals who have contributed examples and insights supporting the points made in this paper. In particular, we wish to thank Lori A. Clarke, George Avrunin, Beth Henneman, Phil Henneman, Reda Bendraou, Ethan Katsh, Dan Rainey, Norm Sondheimer, Mohammed S. Raunak, Rachel Cobleigh, Bin Chen, and Matt Marzilli for conversations, examples, and constructive comments all of which have contributed to this work.

This material is based upon work supported by the US National Science Foundation under Award Nos. CCR-0427071, CCR-0204321 and CCR-0205575. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of The National Science Foundation, or the U.S. Government.

REFERENCES

[1] Web Services Business Process Execution Language Version 2.0 Primer. 2007. http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf

[2] Adams, M., ter Hofstede, A.H.M., Edmond, D. and van der Aalst, W.M.P. 2007. Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation. BPM Center Report

[3] Ambler, S.W. Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press, 1998.

[4] Casati, F., Ceri, S., Paraboschi, S. and Pozzi, G. 1999. Specification and Implementaiton of Exceptions in Workflow Management Systems. ACM Transactions on Database Systems.

[5] Cass, A.G., Sutton, S.M. and Osterweil, L.J., 2003. Formalizing Rework in Software Processes. in Ninth European Workshop on Software Process Technology, (Helsinki, Finland, 2003), Springer-Verlag, 16-31.

[6] Christov, S.C., B. Avrunin, G.S., Chen, B., Clarke, L. A., Osterweil, L.J., Brown, D., Cassells, L., Mertens, W. 2007. Rigorously Defining and Analyzing Medical Processes: An Experience Report. LNCS Volume on Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers (to appear).

[7] Coplien, J.O., 1994. A Development Process Generative Pattern Language. in Pattern Languages of Programs, (Monticello, Il., 1994).

[8] Curbera, F., Khalaf, R., Leymann, F. and Weerawarana, S., 2003. Exception Handling in the BPEL4WS Language. in Conference on Business Process Management, (2003).

[9] Fong, P. and Brent, J., 2007. Exception Handling in WebSphere Process Server and WebSphere Enterprise Service Bus.

http://www.ibm.com/developerworks/websphere/library/techarticles/0705_fong/0705_fong.html

[10] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.M. Design Pattenrs: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[11] Golani, M. and Gal, A., 2005. Flexible Business Process Management Using Forward Stepping and Alternative Paths. in Business Process Modeling, (2005), Lecture Notes in Computer Science, 48-63.

[12] Goodenough, J.B. 1975. Exception Handling: Issues and a Proposed Notation. Communications of the ACM, 18 (12). 683–696.

[13] Hagen, C. and Alonso, G. 2000. Exception Handling in Workflow Management Systems. IEEE Transaction on Software Engineering, 26 (10). 943-958.

[14] Hagen, C. and Alonso, G. October 2000. Exception Handling in Workflow Management Systems. IEEE Transaction on Software Engineering, 26 (10). 943-958.

[15] Henneman, E.H., Cobleigh, R.L., Frederick, K., Katz-Bassett, E., Avrunin, G.A., Clarke, L.A., Osterweil, L.J., Andrzejewski, C., Merrigan, K. and Henneman, P.L. 2007. Increasing Patient Safety and Efficiency in Transfusion Therapy using Formal Process Definitions. Transfusion Medicine Reviews, 21 (1). 49-57.

[16] Lerner, B.S. Verifying Process Models Built Using Parameterized State Machines. Rothermel, G. ed. ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, 2004, 274-284.

[17] Liskov, B.H. and Snyder, A. 1979. Exception Handling in CLU. IEEE Transactions on Software Engineering, SE-5 (6). 546–558.

[18] Osterweil, L.J., 1987. Software Processes are Software, Too. in Ninth International Conference on Software Engineering, (Monterey, CA, 1987), IEEE Computer Society Press, 2-13.

[19] Russell, N., van der Aalst, W.M.P. and ter Hofstede, A.H.M. 2006. Exception Handling Patterns in Process-Aware Information Systems.

[20] Wise, A. 2006. Little-JIL 1.5 Language Report. Department of Computer Science, University of Massachusetts, Amherst, MA 01003

[21] Wise, A., Aaron, C.G., Lerner, B.S., McCall, E., J., O. and Sutton, S.M., 2000. Using Little-JIL to Coordinate Agents in Software Engineering. in 15th International Conference on Automated Software Engineering, (Grenoble, France, 2000), IEEE Computer Society 155-163.

[22] Yemini, S. and Berry, D.M. 1985. A modular verifiable exception handling mechanism. ACM Trans. Program. Lang. Syst., 7 (2). 214-243.