# A Pattern for Modeling Rework in Software Development Processes

Aaron G. Cass[1], Leon J. Osterweil[2], and Alexander Wise[2]

[1] Department of Computer Science
Union College
Schenectady, NY 12308
cassa@union.edu
[2] Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{ljo, wise}@cs.umass.edu

**Abstract.** It is usual for work completed at one point in a software development process to be revisited, or reworked, at a later point. Such rework is informally understood, but if we hope to support reasoning about, and partial automation of, software development processes, rework be more formally understood. In our experience in designing formalized processes in software development and other domains, we have noticed a recurring process pattern that can be used to model rework quite successfully. This paper presents that pattern, which models rework as procedure invocation in a context that is carefully constructed and managed. We present some scenarios drawn from software engineering in which rework occurs. The paper presents rigorously defined models of these scenarios, and demonstrates the applicability of the pattern in constructing these models.

## 1 Introduction

Rework, the activity of reconsidering and modifying an earlier decision, is a feature common in software engineering and other creative processes. While reconsideration may be relatively straightforward, modifying earlier choices can be far more complicated, as it typically entails reconsidering and modifying other choices that had been made subsequent to the choice being modified. These subsequent modifications can then lead to still further reconsiderations and modifications, potentially creating a daunting collection of reconsiderations and modifications that can leave the participants confused and increasingly incapable of keeping the rework activity under control.

In software development, for example, it is not uncommon to find that a design under consideration is becoming unduly complicated because of requirements or architectural decisions that had been made previously. Often this causes the designer to revisit the requirements, for example, to see if they are needlessly restrictive, and to revisit either the requirements or architecture to see if they are, for example, inadventently inconsistent either internally or with each other. These revisitations often lead to changes, which in turn lead to others. Requirements changes then cause reconsideration of architecture decisions, which can cause the architecture to be inconsistent

with the changed requirements. Architecture changes can then require revisiting design and requirements decisions. The full effect of a single change is often referred to as the "ripple effect", and it can be highly disruptive to development processes that might otherwise seem to be orderly and systematic. Thus, the simplicity of a pure "waterfall" process is typically augmented by back-edges that denote iteration. But these iterations are typically driven by changes and cause other changes. Thus what may appear to be simple iterations are actually iterations needed to effect rework changes. Indeed it has been estimated that a very large percentage of the total effort in a complex development project is devoted to rework.

Because of the prevalence of rework, it seems important to better understand it. A clear understanding of the nature of rework could lead to stronger support for the activity, through carefully tailored tools and automated aids. Such an understanding might also help control the propagation of the ripple effect of modifications resulting from rework. It might also lead to better understanding of the relationship between rework and traditional development, perhaps helping to identify common subactivities and a smoother integration of these two dominant components of creative work.

In earlier work [1] we suggested that rework can be modeled as re-invocation of one or more development activities that had been carried out previously, but now must be carried out in a *context* different from the original execution, where a context is any aspect of the process step's dynamic, run-time environment (e.g. input parameters, resources, or personnel employed) that can cause the step to be carried out differently or produce a different result. In that approach, we used procedure invocation semantics to view rework as the invocation of a procedure in a context. So, instead of simply saying that rework is "going back" to a previously-executed development stage, we say that rework is a re-invocation of some activity, activities, or subphase(s) of that previously-executed stage. Invocation semantics make it clear where the rework activity gets its data from and where any output data is passed – down and up (respectively) the procedure invocation trace. This earlier work left largely unexplored, however, the specific details of how this rework context information was gathered, passed to, and used by the rework activity itself. In this paper we present examples of rework, define a rework process pattern based on re-invocation in a managed context, and use this pattern to specify rework in some software development activities.

## 2 Motivating Examples of Rework

As suggested above, rework happens when it becomes clear that a previous decision has become problematic, often because it has come into conflict with subsequent decisions. In such cases, rework is undertaken to resolve the conflict. In this section we present several concrete examples of rework that we have encountered, and which seem amenable to solution using the pattern presented later in this paper. Though we have found rework in dispute resolution [2], scientific data processing [3, 4], health care delivery [5], and other domains, the examples here are drawn from software engineering.

Although there is a lack of agreement about the exact way to develop software, there is little disagreement that a finished product consists of a set of different types of artifacts, usually including a specification of requirements, a design meeting those

requirements, executable code, and evidence that the code satisfies the requirements. Ultimately, these artifacts must be acceptably consistent, both internally and with each other. For example, the design should be shown to specify an approach that enables the requirements to be met, and the executable code should be shown to be a correct implementation of the design.

While all of these dimensions of consistency should be achieved by the end of a development project, there is no expectation that all of these consistencies will be achieved easily or straightforwardly. For example, developers expect that initial design decisions will be inconsistent with requirements, initial code might not correctly implement designs, and that executing code may make clear the unreasonableness of requirements specifications. Indeed, the process of developing a finished product inherently entails the more or less continuous reevaluation and reconsideration of all prior decisions.

For example, a requirements specification may be reconsidered because it is apparently inconsistent with other requirements, because subsequent design efforts are complicated by the requirement, or because code seems unable to satisfy the requirement. Identifying the inconsistency is what we will call a **triggering event**. In such cases, the nature of the conflict is a key part of the **context** under which developers re-evaluate and possibly re-produce the problematic requirement. We view this re-evaluation and re-production as re-instantiation of the activity that produced the faulty requirement in a new context, with different input parameters. Re-instantiation in the new context leads to what we will call **re-invocation** of the decision process, resulting in rebinding of the results of the execution. The re-invocation may or may not result in a modification of the requirement. In either case, execution resumes where it left off, at the site of the triggering event. This may entail re-evaluating the triggering condition, and that might trigger further rework. In software development, a modification of an artifact can wind up triggering a long sequence of rework activities, which can entail multiple reconsiderations of a decision about a single artifact. The fact that there are additions and changes to the context for subsequent reconsiderations of the same artifact improves the chance that previous experiences will inhibit making the same decision multiple times, reducing the chances of unproductive loops in the process.

## 3 A Pattern of Managed Rework

The previous examples suggest a pattern of rework, which we will now define using the vocabulary of Gamma et al. [6]. Fig. 1 shows the structure of the pattern using a UML activity diagram [7].

### 3.1 Applicability

- Use the pattern if internal consistency of work products must be maintained and one wants to handle the inconsistencies that arise before continuing.
- Use the pattern if rework will trigger a long, complex sequence of consequences, whose management will be facilitated by an at least semi-automated process.

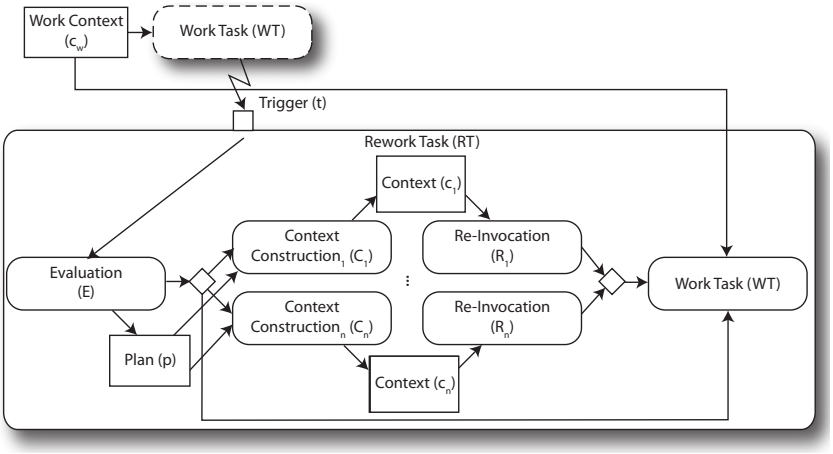


**Fig. 1.** Structure of the pattern.

### 3.2 Participants

**Work task** $(WT)$ The process activity in progress when the need for rework is noticed.

**Trigger** $(t = \{e, i\})$ The trigger is a message that is sent in response to the identification of an inconsistency that seems to require rework. The trigger should identify the entity to be reworked $e$, and the inconsistency detected $i$.

**Rework task** $(RT = \{E, \{(C_1, R_1), \ldots (C_n, R_n)\}\})$ The process model must specify how to respond to each trigger by specifying the following parts:

- **Evaluation** $(E(t) \rightarrow p)$ An activity to be carried out to evaluate the trigger $t$ in order to create a plan $p$ that specifies what, if any, response is required as a response to the trigger. Process models can omit an explicit evaluation activity if there is only a single possible plan that could result from a trigger, in which case $p = t$.
- **Context Construction** $(C_j(p) \rightarrow c_j)$ An activity to create an appropriate calling context $c_j$ for the re-invocation.
- **Re-invocation** $(R_j(c_j))$ The re-invocation of a previously executed activity.

### 3.3 Collaborations

- A process model using to the pattern must define a triggering mechanism. Typically, the trigger is the result of a checking activity and the trigger is represented as an exception object – e.g., a design review might check the internal consistency of a design and throw an exception if the review fails.
- When trigger $t$ is fired, work task $WT$ is suspended and the rework task $RT$ begins.
- Rework begins with the evaluation of trigger $t$ by $E$ to determine whether re-invocation of a previously-executed activity is needed. Based on the trigger, Evaluation will create plan $p$, which may consist of a choice among several different

re-invocations to address the trigger. Evaluation may involve human effort, or the activity may be (at least partially) automated. If re-invocation is needed, an appropriate context $c_j$ for re-invocation $R_j$ is constructed by context construction $C_j$. Re-invocation $R_j$ can then be undertaken, using parameters provided by context $c_j$. Once re-invocation is complete, rework task $RT$ is complete, and the process proceeds where it left off by re-invoking work task $WT$ in its original context $c_w$ (i.e. with its original parameters).

### 3.4 Consequences

- A precise model describes both the context in which rework occurs and how to proceed after the rework has been completed.
- Processes using rework implemented with this pattern may be executable. Because rework is explicit in the process model, such executable processes could be used to monitor how well an activity is progressing.
- Activities that may be carried out in rework contexts must be designed to be carried out in all possible re-invocation contexts. For example, a requirements specification activity within a development process must be designed to allow for modification of the requirements during phases other than the initial requirement specification.

### 3.5 Related Patterns

- When the immediate reworking of an entity is not desired, the exception handling pattern Deferred Compensation may be used [8]. This pattern breaks rework tasks into two disjoint activities – one contains the evaluation activity, while the other, the deferred activity, contains context construction and re-invocation activities.
- Object Derivation [9] offers an approach in which requests for inconsistent objects serves as a trigger for *backward chaining*.
- Observer [6] may be used as a vehicle for creating triggers when reworking an item can result in the need for *forward chaining*.
- Task Deferral [10] is another mechanism for triggering rework. In this pattern, the availability of data is itself a trigger for the forward chain.

## 4 Managing the Context

It is the context in which the re-invocation occurs that differentiates rework from simple procedure invocation. The context defines the entities to be modified, the information available to support this modification, and the constraints placed on the re-invoked activity. Therefore, the re-invocation context must define the binding of objects to both the in- and out-parameters of the re-invocation, ensure that any appropriate constraints are enforced during the re-invocation, and specify the response to any events that arise while the re-invocation is in progress. Formally, we define the context as:

- a clear designation of the entity that is to be the subject of the rework activity.
- a set of bindings between the formal parameters to $R_j$ and the actual arguments in the plan $p : \{(f_1, a_1), \ldots (f_n, a_n)\}$.

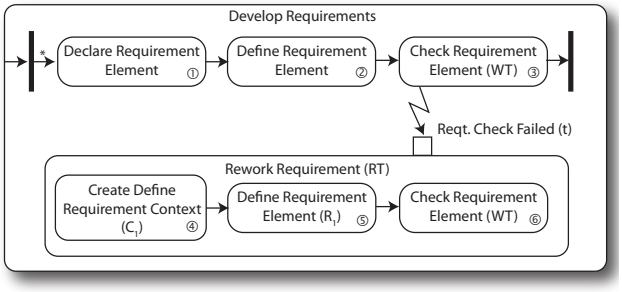


**Fig. 2.** Control flow in the requirements development process.

- a set of constraints $B$ that are in force during the re-invocation.
- a set of handlers $H$ for any signals $s$ that may occur during the re-invocation.

The set of bindings in the context specifies the information available, and the destination of entities that are created during the re-invocation. When creating this binding, the difference between pass-by-reference and pass-by-value is particularly important. Since rework often involves the modification of existing entities, is is usual to pass these entities by reference. We note, however, that the derivation history [9, 3, 4] of these entities is often particularly important in the creation of a rework plan as that history is often necessary in order to guide the re-invocation away from repetition of choices that have previously been shown to have led to later problems.

```
try
    element ← Declare Requirement Element
        (informal requirements)                    ①
    Define Requirement Element (element)           ②
    Check Requirement Element (work context)       ③
catch failure : Reqt. Check Failed
    context ← Create Define Requirement
        Context (failure)                          ④
    Define Requirement Element (context)           ⑤
    Check Requirement Element (work context)       ⑥
end try
```

**Fig. 3.** Pseudo-code for the requirements development process. Step numbers correspond to those in Fig. 2.

In our experience, exceptions are often used as triggers of rework, and as seen in the examples provided in Section 5, changes made in later phases of a larger activity often create inconsistencies or problems that are detected as the violations of constraints that then generate exceptions that in turn initiate rework sequences that "ripple" through the larger activity. The pattern we define here allows the set of handlers $H$ to control this rippling by treating re-invocations as "work tasks" from which triggers may be emitted, each of which associates a set of parameter bindings, specifications of which entities are to be reworked, and other components of the context for the re-invocation.

## 5 Examples Using the Pattern

We begin with rework in requirements specification, shown in Figures 2 and 3. This activity consists of the parallel creation of a set of Requirement Elements, each of which consists of a Requirement Specification Declaration (created by Declare Requirement

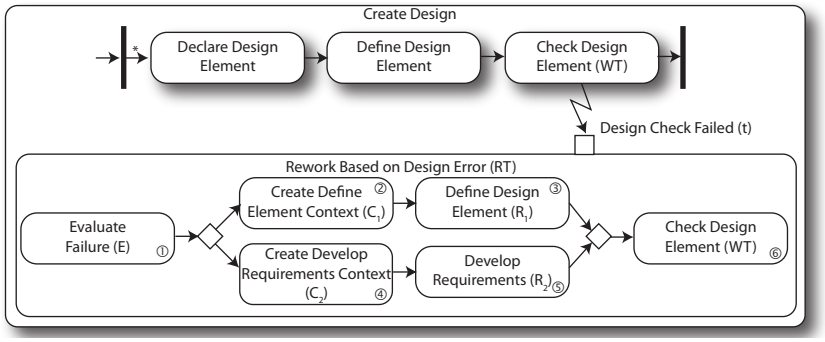


**Fig. 4.** Control flow in the design process.

Element) and a Requirement Specification Definition (created by Define Requirement Element). Each Requirement Element is reviewed in Check Requirement Element, and reworked if the review indicates any deficiencies.

The failure of Check Requirement Element serves as the trigger. As the Requirement Element must always be reworked if its review fails, this use of the rework pattern contains no evaluation activity – responding to the trigger immediately results in the creation of an appropriate context. In this example, we assume that the failure indicates the need to rework a specific Requirement Specification Definition whose identity is passed as part of the trigger. The rework then begins with creation of the appropriate context by Create Define Requirement Context, then proceeds with the re-invocation of Define Requirements Element, and then resumption of the execution of Check Requirement Element in the context from which the initiating trigger occurred.

```
try
    element ← Declare Design Element
        (requirement)
    Define Design Element (element)
    Check Design Element (work context)
catch failure : Design Check Failed
    plan ← EvaluateFailure(failure)                    ①
    if plan is rework the design then
        context1 ← Create Define Element
            Context(plan)                              ②
        Define Design Element (context1)               ③
    else rework the requirements
        context2 ← Create Develop Requirements
            Context(plan)                              ④
        Develop Requirements(context2)                 ⑤
    end if
    Check Design Element(work context)                 ⑥
end try
```

**Fig. 5.** Pseudo-code for the design process. Step numbers correspond to those in Fig. 4.

In reinvoking Define Requirements Element as the response to the detection of a difficulty with the result of a prior invocation, this process definition demonstrates our view of how rework can be defined accurately. In order to explain this approach to defining rework adequately it is necessary to elaborate upon the way in which this process definition manages its artifacts and their flow. Develop Requirements takes as input an informal set of requirements and produces a set of Requirement Elements as output. When, as described above, Check Requirement Element fails it fires a trigger $t$ this results

in an instance of the rework pattern. As there is only a single response to the failure, the evaluation activity $E(t)$ has been left out, and no plan is created, and the re-invocation context is created using information from Reqt. Check Failed directly. While the context should include a range of information to support the modification of the element such as the reason the check failed, of particular interest to us is the requirement element to be defined. In the original work task Check Requirement Element $(WT)$, the element is defined by the normal flow of Develop Requirements but here is defined to be the element $e$ that is part of the trigger Reqt. Check Failed. Because the context binds the formal parameters in an activity to the actuals in the calling context, the changes to the element are reflected in the set of requirement elements.

As previously noted, Develop Requirements takes an informal requirements description and produces requirement elements by, in parallel, defining individual elements by creating a requirement element for part of the informal requirements ①, defining the requirement ②, and finally reviewing the defined requirement ③.

In the event that the requirement is inadequate, Check Requirement Element signals this by throwing *failure*, an instance of the trigger Reqt. Check Failed that includes the failed element. The trigger is the input to Create Define Requirement Context which creates a context including the element to be passed ④ to the re-instantiation of Define Requirement Element ⑤ and finally, control is returned to Check Requirement Element ⑥.

We now use the pattern to specify a more complicated form of rework. Figures 4 and 5 present a design activity, the overall structure of which is similar to that of the Develop Requirements activity presented above. Specifically, after the design elements are declared and defined, there is a review activity, Check Design Element that may trigger rework. In Create Design however this review activity incorporates a determination of whether or not the design element should be revised (by the re-invocation of Define Design Element) or if the requirements must be reworked by Develop Requirements to accommodate discoveries about Requirement Elements that have been made during design. This is an example of the not-uncommon situation where detecting the existence of a problem is only the first step in a process that leads to a re-invocation.

Before re-invocation can take place, participants must determine an appropriate solution to the problem. The pattern represents this as evaluation activity Evaluate Failure ① that takes the failure and produces a *plan* to assess the problem $(E(t) \rightarrow p)$. In this example, in response to the Design Check Failed trigger, Evaluate Failure creates one of two possible plans: one that requires reworking the design element in much the same way as the requirement element was reworked above ② ③, and a second in which new requirement elements are created in response to new informal requirements created as part of the plan by re-invocation of Develop Requirements ⑤ in a context created by Create Develop Requirements Context ④ that includes the new informal requirements, and then finally returning to the work context Check Design Element ⑥.

## 6 Related Work

Others have studied process patterns, notably Russell et al., who have specifically studied workflow structures that can be used for exception handling [11]. They derive several patterns that can be used to handle exceptions caused by a single work item. While

our pattern is rightly seen as one way of handling a kind of exception – namely those exceptions that cause a previous decision to come under suspision – our work differs from that of Russell et al. in that we observe a pattern in existing process models instead of deriving possible patterns from low-level considerations.

The pattern presented in this paper helps to describe rework in formal process models. As such, it aims to help solve a long-standing problem in process improvement. It is generally accepted that rework is a feature (or a bug) in real-world software development, and that modelling processes therefore requires modeling rework carefully [12, 13]. And yet, most life-cycle models do not formally model rework and rework is not formally treated in popular software engineering texts (for example, [14–16]). Many life-cycle models (e.g. the Spiral Model [17]) assume that steps are repeated many times with different contexts, but do not formally model how context is managed.

In order to implement the pattern described here in a process or workflow model, the modelling language must support a trigger mechanism and a mechanism for managing the parameter binding needed for a re-invocation context. Some modelling languages support this approach more directly than others. For example HFSP's [18] `redo` clause allows reinstantiation of a step with different parameters. With other languages, especially those with semantics similar to general-purpose programming languages, this pattern can be implemented using exception handling and scopes.

### 6.1 Implementing Triggers as Exceptions

Because triggers are seen as devices for initiating activities that are considered to be outside the normal flow of control, it seems natural to implement a trigger with an exception handling mechanism. Several languages provide such a mechanism, borrowing from general purpose programming-language semantics [19–21]. Other languages allow the specification of consistency conditions that produce exceptions when violated (for example, AP5 [22], Marvel [23], Merlin [24], EPOS [25], and ALF [26]).

Wang and Kumar [10] propose a different approach to exception handling that could also be used to support rework. Their approach assumes a data-flow based workflow system, in which control flow is inferred from data dependencies between activities – if an activity $B$ must occur after activity $A$, even with no data flow between $A$ and $B$, a *soft* data dependency is added. Then, if $A$ should fail, the soft dependency can be relaxed and activity $A$ can be *deferred* to such a time as it can be safely executed (reworked). However, their approach does not seem to allow specifying a change in the invoking context of $A$ – the source of $A$'s data is fixed. Therefore, it seems that some kinds of rework could be modelled in this approach but it is not flexible in managing the context for rework.

### 6.2 Implementing Context with Scope

An invocation context must bind parameters and exceptions and provide an environment in which the re-invocation is carried out. It seems natural to use a scope to define such an environment. Any inputs an activity needs or outputs it produces can be found by searching within a scope. Therefore, by providing a different scope through re-invocation, we provide a different environment in which the activity can be performed.

Little-JIL [19] supports hierarchical scoping for parameter binding and exception handling. Also, several languages based on flow graphs and Petri-nets allow nesting of activities, where each nesting provides a scope [27–29].

## 7 Conclusions and Future Work

The examples provided in this paper fit nicely into the pattern that we have presented. Moreover the examples seem to us to provide elegant representations of the actual nature of rework. The pattern makes it clear that rework does indeed entail repeating activities and steps that had been executed previously, but it also shows that the re-execution is not exactly a repetition, but is a revisitation of previous work now with new knowledge, as contained in context information such as calling arguments. The pattern makes clear how the new knowledge is created and brought to bear.

The pattern makes it clear that rework does not entail "returning to a previous phase". In an important sense it has always been obvious that reworking a requirement because of a problem found in design did not cause a "return to the requirements phase", but rather a pause in the activities involved in design while activities involved in requirements were revisited. This intuition now seems to be very well represented in the pattern of "re-execution in a managed context", expressed precisely and elegantly through the semantics of procedure invocation. In short, the rework examples we have shown are some form of carefully managed, potentially recursive, procedure invocation.

The pattern presented here suggests opportunities to improve development environments. In recognizing that rework often entails creating new contexts for previously executed process steps, this work seems to highlight the importance of maintaining the information basis for constructing such contexts. This information basis may consist of specific instances of types of software development artifacts such as design components and design decisions, or of large and elaborate structures of such instances that have arisen during extensive development and rework activities. This suggests to us that future work aimed at creating powerful development support systems might do well to focus on how to maintain precise and articulate information about these artifact instances, and the histories of their development. As most of our previous work has focused on defining process steps, the work indicated here suggests a complementary focus on the artifact instances that they require and generate. Such complementary work might then focus on how to store, structure, and present artifact structures in ways that enable tools to better support development, which inevitably includes rework. Such tools would not simply present a developer with the need to revisit a previously executed step, but would supplement that with an articulate description of the circumstances under which the step had previously been carried out. This would enable the developer to make better informed decisions about how to address the needed rework.

The modest number of examples provided in this paper are only a representative sample of a larger number of examples that seem to fit into the pattern that has been presented here. These examples all seem to be cleanly and clearly represented as instances of the pattern that we have presented. It is our conjecture that this pattern will suffice to describe many other instances of rework, found both in software engineering and in other disciplines. We remain interested in examining other instances in order to

explore our hypothesis that this pattern might well serve as a definition of the term "rework". Should our conjecture prove to be correct, then we expect that this work could lead to more effective support for rework as is needed in many of the varied domains in which it is a central feature of how work is carried out.

## Acknowledgements

The authors wish to express gratitude to Stanley M. Sutton, Jr, Reda Bendraou, Barbara Staudt Lerner, Stefan Christov, Lori A. Clarke, and members of the Laboratory for Advanced Software Engineering Research at the University of Massachusetts Amherst who have participated in this research, and clarified the points made in this paper.

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 author 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. Cass, A.G., Sutton, Jr., S.M., Osterweil, L.J.: Formalizing rework in software processes. In: Proc. of the 9th European Workshop on Soft. Proc. Technology. (September 1–2, 2003) Helsinki, Finland.
2. Clarke, L.A., Gaitenby, A., Gyllstom, D., Katsh, E., Marzilli, M., Osterweil, L.J., Sondeimer, N.K., Wing, L., Wise, A., Rainey, D.: A process-driven tool to support online dispute resolution. In: Intl. Conf. on Digital Government Research, ACM Press (2006) San Diego, CA.
3. Osterweil, L.J., Wise, A., Clarke, L.A., Ellison, A.M., Hadley, J.L., Boose, E., Foster, D.R.: Process technology to facilitate the conduct of science. In: Soft. Process Workshop (SPW2005), Spring-Verlag (2005) 403–415 Beijing, China.
4. Osterweil, L.J., Clarke, L.A., Podorozhny, R., Wise, A., Boose, E., Ellison, A.M., Hadley, J.: Experience in using a process language to define scientific workflow and generate dataset provenance. In: Proc. of the 16th ACM SIGSOFT Intl. Symp. on Foundations of Soft. Engineering (FSE16), ACM Press (2008) Atlanta, GA.
5. Christov, S., Chen, B., Avrunin, G.S., Clarke, L.A., Osterweil, L.J., Brown, D., Cassells, L., Metens, W.: Rigorously defining and analyzing medical processes: An experience report. In: 1st Intl. Workshop on Model-Based Trustworthy Health Information Systems (MOTHIS), Springer-Verlag (2007) Nashville, TN.
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Object Management Group: OMG Unified Modeling Language (OMG UML) Superstructure. Technical Report formal/2007-11-02, Object Management Group (November 2007) Version 2.1.2.
8. Lerner, B.S., Christov, S., Wise, A., Osterweil, L.J.: Exception handling patterns for processes. Technical Report 08-06, UMass Dept. of Comp. Sci. (March 2008)
9. Clemm, G., Osterweil, L.: A mechanism for environment integration. ACM Trans. on Prog. Lang. and Systems (TOPLAS) **12**(1) (1990)

10. Wang, J., Kumar, A.: Exception handling using task deferral in document-driven workflow systems. In: Proc. of the Annual Workshop on Information Technology and Systems (WITS). (2005)
11. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Exception handling patterns in process-aware information systems. Technical report, BPM Center (2006)
12. Haley, T., Ireland, B., Wojtaszek, E., Nash, D., Dion, R.: Raytheon Electronic Systems experience in software process improvement. Technical Report CMU/SEI-95-TR-017, Carnegie-Mellon Software Engineering Institute (November 1995)
13. Butler, K., Lipke, W.: Software process achievement at Tinker Air Force Base. Technical Report CMU/SEI-2000-TR-014, Carnegie-Mellon Software Engineering Institute (September 2000)
14. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, Englewood Cliffs, NJ (1991)
15. Pressman, R.S.: Software Engineering: A Practitioner's Approach. Fourth edn. McGraw-Hill, New York (1997)
16. Sommerville, I.: Software Engineering. 5th edn. Addison-Wesley (1996)
17. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Computer **21**(5) (1988) 61–72
18. Suzuki, M., Iwai, A., Katayama, T.: A formal model of re-execution in software process. In: Proc. of the 2nd Intl. Conf. on the Soft. Process, IEEE-PRESS (February 1993) 84–99 Berlin, Germany.
19. Wise, A.: Little-JIL 1.0 Language Report. Technical Report 98-24, UMass Dept. of Comp. Sci. (April 1998)
20. Sutton, Jr., S.M., Osterweil, L.J.: The design of a next-generation process language. In: Proc. of the 6th European Conf. on Soft. Engineering, Springer-Verlag (1997) 142–158 Zurich, Switzerland.
21. Sutton, Jr., S.M., Heimbigner, D., Osterweil, L.J.: APPL/A: A language for software-process programming. ACM Trans. on Soft. Engineering and Methodology (TOSEM) **4**(3) (July 1995) 221–286
22. Cohen, D.: AP5 Manual. USC, Info. Sci. Institute. (March 1988)
23. Kaiser, G.E., Barghouti, N.S., Sokolsky, M.H.: Experience with process modeling in the MARVEL software development environment kernel. In Shriver, B., ed.: 23rd Annual Hawaii Intl. Conf. on System Sci. Volume II., Kona HI (January 1990) 131–140
24. Junkermann, G., Peuschel, B., Schäfer, W., Wolf, S.: MERLIN: Supporting cooperation in software development through a knowledge-based environment. In Finkelstein, A., Kramer, J., Nuseibeh, B., eds.: Soft. Process Modelling and Technology. Wiley (1994) 103 – 129
25. Conradi, R., Hagaseth, M., Larsen, J.O., Nguyên, M.N., Munch, B.P., Westby, P.H., Zhu, W., Jaccheri, M.L., Liu, C.: EPOS: Object-oriented cooperative process modelling. In Finkelstein, A., Kramer, J., Nuseibeh, B., eds.: Soft. Process Modelling and Technology. Wiley (1994) 33 – 70
26. Canals, G., Boudjlida, N., Derniame, J.C., Godart, C., Lonchamp, J.: ALF: A framework for building process-centred software engineering environments. In Finkelstein, A., Kramer, J., Nuseibeh, B., eds.: Soft. Process Modelling and Technology. Wiley (1994) 153 – 185
27. Bandinelli, S., Fuggetta, A., Grigolli, S.: Process modeling in-the-large with SLANG. In: Proc. of the 2nd Intl. Conf. on the Soft. Process, IEEE Computer Society Press (1993) 75–83
28. Deiters, W., Gruhn, V.: Managing software processes in the environment melmac. In: Proc. of the 4th ACM SIGSSOFT/SIGPLAN Symp. on Practical Soft. Dev. Environments, ACM Press (1990) 193–205 Irvine, CA.
29. Casati, F., Ceri, S., Paraboschi, S., Pozzi, G.: Specification and implementation of exceptions in workflow management systems. ACM Trans. on Database Systems (TADS) **24**(3) (September 1999) 405–451