Formalizing Rework in Software Processes

Aaron G. Cass, Stanley M. Sutton Jr., and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
+1 413 545 2013
{acass, ljo, suttonsm}@cs.umass.edu

Abstract. This paper indicates how effective software-process programming languages can lead to improved understandings of critical software processes, as well as improved process performance. In this paper we study the commonly mentioned, but seldom defined, rework process. We note that rework is generally understood to be a major software development activity, but note that it is poorly defined and understood. In this paper we use the vehicle of software-process programming to elucidate the nature of this type of process. In doing so we demonstrate that an effective language (i.e. one incorporating appropriate semantic features) can help explain the nature of rework, and also raise hopes that this type of process can be expedited through execution of the defined process. The paper demonstrates the key role played in effective rework definition by such semantic features as procedure invocation, scoping, exception management, and resource management, which are commonly found in programming languages. A more ambitious program of research into the most useful process-programming language semantic features is then suggested. The goal of this work is improved languages, for improved understandings of software processes, and improved support for software development.

1 Introduction

Rework is an ongoing problem in software development. Rework is generally considered to be potentially avoidable work that is triggered to correct problems or (sometimes) to tune an application [6, 10]. The cost of rework can approach or exceed 50% of total project cost [21, 10, 6]. Rework cost rises dramatically the longer the delay, relative to the life cycle, between the occurrence of a problem and its remediation. For example, for a problem that occurs in requirements analysis, the cost of repair may rise by one to two orders of magnitude depending on how late in the life cycle it is caught [6].

Research on rework has focused on minimizing the amount of rework that a project may incur [21, 10, 6, 29]. This is typically done through the introduction of earlier, more frequent, and more formal reviews, inspections, and tests; these aim to detect and enable the correction of problems as early in the life cycle as possible. A reduction in rework costs has also been shown for increases in the general level of process maturity [23, 14, 29]. Through such approaches the cost of rework has been reduced signifi cantly, in some cases to less than 10% of total project cost [21, 10].

Despite successes in reducing the amount of rework, it is generally accepted that rework cannot be eliminated entirely. Moreover, not all rework-inducing problems can be detected as soon as they occur; some problems will only be caught some distance downstream. Thus, some amount of rework, including some expensive rework, is inevitable.

In this paper we propose that the problem of rework can be attacked from another angle, namely, that of software-process programming. Process programming, which is the specification of software processes using formal, usually executable, languages, can contribute to the solution of the rework problem in two ways: It can help with the prevention of rework, and it can help in facilitating rework that cannot be prevented.

Towards the prevention of rework, process programming can help by enabling the precise definition of software processes, both before and after measures to mitigate rework are introduced. This has the benefits of showing precisely where in the development process measures to mitigate rework may be present or need to be introduced. In such cases, process programming can facilitate accurate tracking and evaluation of rework and rework-mitigation efforts, and it can enable precise cost accounting of these efforts (for example, [23, 10, 6]). With execution support, process fidelity can be increased, and monitoring and measurement efforts can be automatically supported, thus helping to ensure that planned efforts to reduce rework are effective. The development of precise and effective process definitions is also an important part of the effort to increase process maturity [33, 14].

Toward the facilitation of rework, process programming can contribute to the precise definition of rework processes, which in turn can contribute to the analysis and understanding of rework. Here too, execution support can make a contribution by enhancing the fidelity and effectiveness of rework processes, thereby improving their monitoring and reducing their costs. In combination, these measures further help to ensure the formal and actual correctness of rework processes and may indirectly lead to additional reductions in the amount of rework by reducing the need for rework.

The specification or programming of rework processes poses special challenges, however. These challenges arise, we believe, from what we see as an essential characteristic of rework, namely, that it entails repetition with variation. Typically in rework an activity must be repeated, a work product must be revised, or a goal must be reestablished. Thus, what was done or achieved before must be done or achieved again, but in a new context that reflects prior work, identified problems, dependent artifacts and activities, and other ongoing efforts.

In general, from initial development to rework (and from one rework context to another), some things are consistent and some things change. For instance, the same basic activity to define a requirement may be invoked during the initial specification of requirements as well as when a problem with the requirements is later discovered during the design phase. However, in the rework situation, developers with different qualifications may be required (say, expert versus intermediate), access to requirements documents may need to be coordinated with access by design engineers, dependent design elements may need to be identified, a specific sort of regression test (or review) of the revised requirements may need to be performed, and problems with the rework may require handling in ways specific to that context. Alternatively, perhaps the same

specific developers who created the initial specifications may be required for rework, but the process by which they will make the modifications may be different from the one they used to create the initial specifications. Or the same general review of the requirements will be performed after rework as after initial specification, but only after a suitable "batch" of requirements revisions has been accumulated.

In all of this, we are left with the impression that, while rework activities may vary from initial activities, the term *rework* suggests that these variations are just that, variations or modifications of the original. Thus, it is our surmise that it should be possible to specify these activities in such a way that the nature of the variations should be identifiable as, for example, alternation, parameterization, or elaboration. This suggests that rework activities are essentially reinstantiations, perhaps with context driven parameterized modifications. If this surmise is correct, it should lead to cleaner, clearer, more understandable development processes that, nevertheless, are complete and faithful to the many complexities of real development. This clarification should lead, moreover, to expedited automated support of the performance of real development. Thus, while a deeper understanding of the heretofore nebulously-defined notion of rework is a key goal of this work, so too is the provision of more effective automated support to real software development processes.

A key obstacle in achieving these goals is the ability to understand how to specify the contexts in which rework might occur in software development. This seems to us to entail the ability to capture precisely all of the relevant elements of that context, including both those that are retained from earlier development and those that are new. However, the ability to specify the relevant elements of rework processes, including their contexts and activities, in turn depends on having process languages with appropriate constructs and semantics.

This research makes two primary contributions toward these problems. First, we show that some kinds of rework can indeed be formalized by process programming, especially focusing on details of the rework context. Second, based on such experience, we identify features that process languages should incorporate in order to support real-world software processes including rework. Conversely, we recommend that those who are interested in specifying rework processes should look for process languages with these features. With such languages we should be able to give more complete and precise definitions of rework processes. In turn, with such definitions, we should be able to better understand the nature of rework and its relationship to other aspects of the software life cycle. Furthermore, by applying the definitions to the execution of processes, we hope to further reduce rework costs and to maximize rework efficiency in practice.

The remainder of this paper is organized as follows. Section 2 describes our approach to formalizing rework processes using process programming. It gives an example of a software process that includes rework; this process is specified using the process-programming language Little-JIL [43, 44] and presents an analysis of the language features used to capture various aspects of rework. Section 3 discusses related work on the definition of rework processes, and it surveys process languages for constructs that may be appropriate for rework process definitions.

Section 4 then presents our recommendations regarding language features for capturing rework processes and discusses applications of process programming to defining,

understanding, and executing rework. Finally, Section 5 presents our conclusions and indicates directions for future work.

2 Approach

Our approach to the formalization of rework is to represent rework using process programming languages in which the activities to be performed and especially the context of those activities can be precisely defined. This should allow us to be clear about how rework is to be performed, for example, when a flaw in the requirements specification is only discovered during design or coding and must be repaired at that later stage. It should also allow us to be clear about what happens when a design fails to pass review and must be revised still within the design phase but perhaps using a technique different from the one that gave rise to the problem.

2.1 Basis in Programming

We see significant analogies between common rework situations and typical programming language constructs. Rework activity definitions are analogous to procedures or methods. The context of a rework activity is analogous to the scope in which a procedure or method is invoked. The particular features or characteristics associated with these programming-language notions suggest benefits for the definition of rework processes.

For example, a procedure or method allows a flow of actions to be defined. In general this flow may be absolute or conditional, linear or nonlinear, deterministic or nondeterministic. Additionally, a procedure or method can be explicitly parameterized and may rely on additional information or services defined externally in the scope from which it is invoked. Mechanisms such as these enable the definition of activities that are adaptable to their contexts, in particular to factors that characterize and differentiate initial development from rework. Such factors may include the artifacts available, the state of these artifacts, the state of the process, the resources available, and so on.

Similarly, a scope in a programming language is associated to some defining construct, such as a procedure or object. Depending on the nature of that construct, a scope may contain data and control declarations, initializations and bindings, assertions, exception handlers, and so on. Additionally, the construct that defines the scope will typically invoke some activities as subroutines, controlling whether, when, and with what parameters a particular invocation occurs, and also determining how the results, whether normal or abnormal, are handled. These sorts of mechanisms enable the definition of contexts appropriate to both initial development and rework. For example, in a rework context compared to initial development, different data sources may be defined, perhaps representing problem reports or dependent artifacts; the rework may be accomplished by a rework-specific procedure or a general-purpose procedure with rework-specific parameters; the rework procedure may be invoked under particular scheduling constraints reflecting its relation to downstream development activities; and the human resources involved may be the original developers or a combination of original and downstream developers.

The analogies to general-purpose programming-languages concepts are thus suggestive of many benefits for the specification and control of rework processes. However, as with general-purpose programming languages, the particular design of a process-programming language affects the kinds of activities and contexts that can be readily described and automated. There have been many process-programming languages defined, based on many different computational and modeling paradigms.[1] These offer a variety of constructs and semantics that are suitable to varying degrees and in differing ways for specifying rework. To illustrate the formalization of rework through process programming, we will use a particular process-programming language, Little-JIL [43, 44], applied to an example of a phased software-development process that includes explicit rework.

The process we describe addresses requirements specification and design at a high level. This process description is intended as an example of how rework can be incorporated accurately into a software process. It is NOT intended to represent an ideal software process or even to imply that this is the only way that rework could be handled in this process. The purpose of the example is to be illustrative and suggestive. The approach we present can be applied to processes both more and less complex or restrictive than the example we give – different organizations will want differing processes, and an effective process language should enable a corresponding range of process characteristics to be addressed.

2.2 Example

For our example, consider the initial portion of a phased software development process. After requirements specification activities are completed, the developers proceed to design activities. During the requirements specification activity, as requirements elements are developed, they are reviewed, both independently and for inter-requirement consistency.

As design proceeds, design reviews are also conducted, including ones that check the conformance of the design with requirements. As a result of these reviews, it might be discovered that there are design elements based on design decisions that have no basis in the stated requirements. Additionally, it might be determined that the requirements specification needs some additional requirements to address the concerns reflected in these particular design decisions. At this point, the developers will engage in a rework activity. While still in the design phase, some requirements specification activities must be performed.

We believe that many aspects of this scenario will be easily described using a process-programming language that incorporates constructs (or variations thereof) from general-purpose programming languages, such as scoping, procedure invocation, exception handling, and so on. To investigate this hypothesis, we have used our process-programming language, Little-JIL [43, 44], to address the above scenario.

Figure 1 shows a Little-JIL program for a part of the phased software development process described above. A Little-JIL program is a hierarchy of steps, each of which has an interface and defines a scope in which a variety of additional elements are structured.

[1] Examples and citations will be given below.

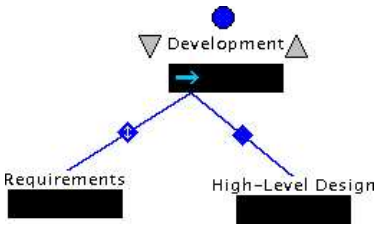

Fig. 1. A phased software development process

Little-JIL is a visual language in which every step is represented by a named black bar that is decorated by a variety of graphical badges. The step interface is represented by annotations on a filled circle above the name of the step. The interface represents the view of the step as seen from the caller of the step, including parameters, resources, messages, and exceptions that may cross the interface. Little-JIL passes parameters as value, result, or value-result. In Little-JIL a step is generally defined as either a root step or a substep in a process. However, a step defined in one part of a program can be referenced from other parts of the program. Such a reference represents a reinstantiation and invocation of the step in a new scope. Both the ability to construct process contexts and the ability to invoke steps in multiple contexts are important for describing rework, as shown in the example. Additional features of Little-JIL are introduced as the example is described.

The right-arrow in the root step in Figure 1 indicates that the substeps are to be executed in sequence from left-to-right, starting with Requirements and continuing with High-Level Design. Figures 2 and 3 show elaborations of the Requirements and High-Level Design steps, respectively.[2]

The triangle to the right of Declare and Define Rqmt in Figure 2 indicates a *post-requisite*, a step that is executed when Declare and Define Rqmt completes. In this case, the post-requisite is a Requirements Review. If the post-requisite completes without any errors, then Declare and Define Rqmt completes successfully. However, if errors are found in the Requirements Review, a rqmtReviewFailed exception will be thrown. In Little-JIL, exception handling is scoped by the step hierarchy. So, in this case, the rqmtReviewFailed exception will propagate to the Develop Rqmt Element step. The handler attached here (beneath the red "X" in the black bar) indicates that we should *restart* the Develop Rqmt Element step and recreate that requirement element.

Once requirements elements have been declared and defined, we proceed to High-Level Design. As can be seen in Figure 3, after all design elements have been declared (by Declare Design Elements), a design-rqmts conformance check post-requisite is executed. During this review, we could check that all design elements have associated requirements elements. If we discover that there are design elements that lack adequate support in requirements, we can throw a MissingRqmts exception. In this context, this

[2] In this and other Little-JIL figures, we show some information in comments (shaded boxes) because the editor we use for creating Little-JIL programs does not show all of the information pertinent to a step in one view. The use of comments here to reflect information that is more formally captured in other views does not reflect incompleteness in the overall process model.

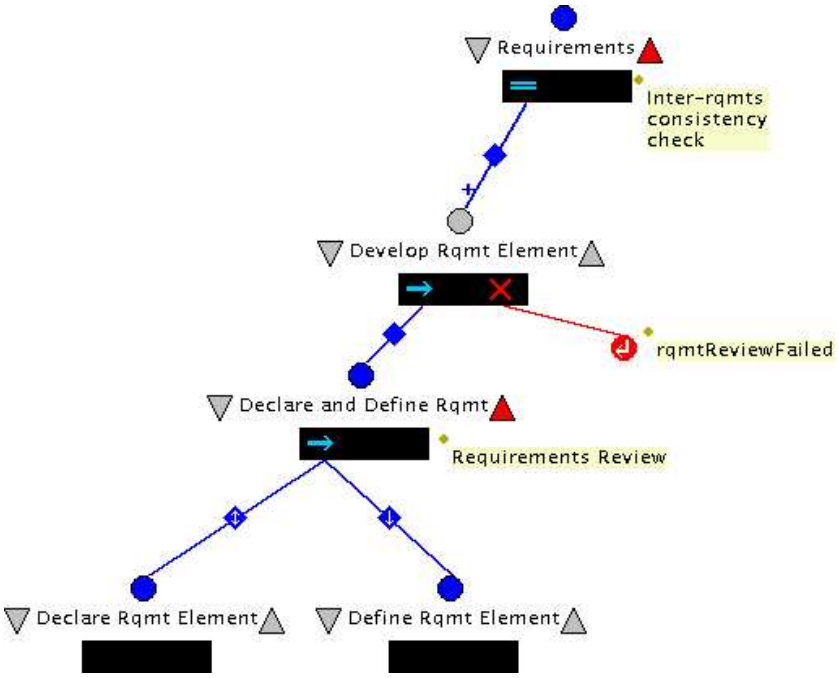

Fig. 2. Requirements activities

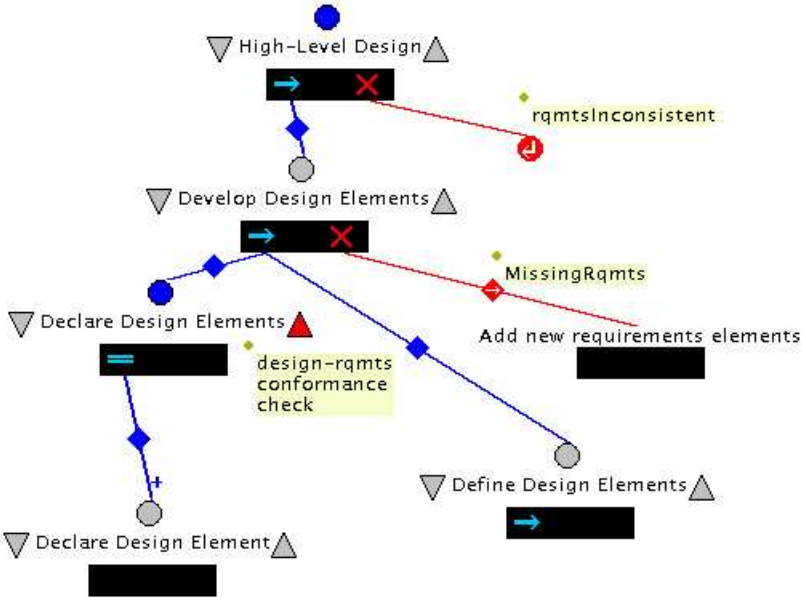

Fig. 3. High-Level Design activities

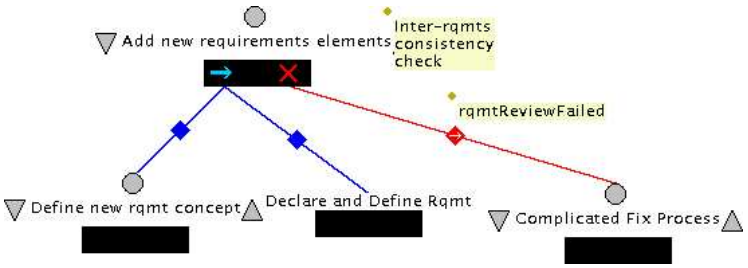

Fig. 4. Rework in the context of design

exception is handled by the exception handler called Add new requirements elements, which is elaborated in Figure 4. Add new requirements elements first defines a new requirements concept for use as input to the Declare and Define Rqmt step, which was defined earlier in the Requirements phase in Figure 2.

In addition to the features illustrated in the figures above, Little-JIL also allows the specification of resource requirements [35]. Managers of processes are very concerned in general with resource allocation and utilization, so resource specification has first-class status in Little-JIL. Resource-related cost issues are also of concern specifically in relation to rework, and the accurate definition of processes has been considered a prerequisite to first assessing the true costs of rework and then minimizing them [21].

Figure 5 shows a step Declare and Define Rqmt with a resource specification which indicates that the agent for the step must be a Rqmt Engr. In this example, we specify only the resource type Rqmt Engr. However, the specification can be any legal query that our externally defined resource manager can execute. For example, we could specify that Declare and Define Rqmt needs a Rqmt Engr with attributes indicating that he or she knows UML use cases. At run-time, the resource specification is interpreted by a management component of the Little-JIL runtime environment (Juliette) as a query for a resource from the resource model. The resource manager will choose a resource from those available in the resource model that satisfy the specification (or throw an exception if the request cannot be satisfied).

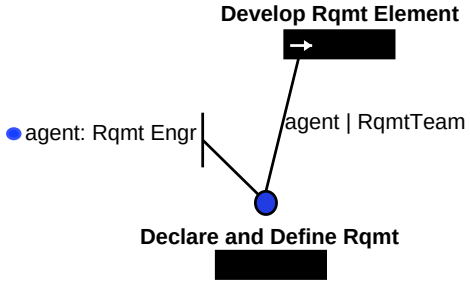

Fig. 5. Resource Specification

This example incorporates two occurrences of rework. The first is part of the requirements phase, when the failure of Declare and Define Rqmt triggers (via an exception) the reexecution of Develop Rqmt Element. In this case the rework is accomplished by repeating the same activity in the same context. Of course, the process will have progressed from the state in which the original work was done. The second occurrence of rework is when Declare and Define Rqmt is referenced from the design phase. In this case an activity that is first invoked in one context is later invoked in a different context, but for essentially the same purpose. That is, to complete a specific requirement that contributes toward the completion of the whole requirements specification.

2.3 Analysis

Based on analysis of the example-scenario program, what can we understand about rework in the process? What is distinctive about the rework context and what is retained from the context of the original work?

We can plainly see the contexts in which rework occurs, at specific points within the requirements phase (the original context) in one case and within the design phase (a different context) in the other case. The rework activity is explicitly represented by a step in the process. This step has a specific interface and (in this example) a particular postrequisite (entailing a requirements review when a new requirement is defined). The rework activity is the same one used in initial development.

During initial development the activity is invoked as part of normal control flow but both occurrences of rework are triggered by exceptions. Input parameters are passed from different sources in the different contexts. In the requirements phase the principal input is the result ultimately of requirements elicitation (not shown in the program), whereas in the design phase the input is the output of the step Define new rqmt concept.

In its invocations for both initial development and rework, the activity carries the same postrequisite, that is, it is subject to review of the requirements specification. This reflects the continuity of purpose of the activity. However, if the review is not passed, the exception that will be thrown will lead to very different exception-handling processes. In the requirements phase, the exception simply triggers a reexecution of the activity. In contrast, during the design phase, the exception may trigger a much more elaborate repair process. Details are not elaborated here but, for example, repair might involve changing the design elements, or even revisiting the requirements with the customer, activities that would not be necessary, or even sensible, during the requirements phase. The continuation after this handling is also different in this context: instead of repeating Declare and Define Rqmt, the process continues with the Add new requirements elements step (as indicated by the right arrow on the edge).

When we use step Declare and Define Rqmt in the design phase, compared to the invocation in the requirements phase, different resources can be acquired, for two reasons. First, the state of the resource model might be different – different resources might be available or new resources might have been added or removed, for example, due to staffing changes. Second, Little-JIL allows an invoking step to narrow the specification of the space of resources to be considered for an invoked step. The specification on the edge in Figure 5 indicates that the agent must come from the RqmtTeam, which is a resource collection passed from the parent step. In the reinvocation in the design phase,

however, we can constrain the agent specification with a different annotation on the edge (or, in fact, not constrain it at all to indicate that any Rqmt Engr will do).[3]

This example has shown that rework can be described with a variety of process language constructs and semantics, some derived from conventional programming languages, some derived from other sources. In this example we have used subroutine abstraction, invocation semantics, parameterization of data and resources, resource management, and exception handling. This has enabled us to write process programs for software development processes that seem to be substantially faithful to conventional notions of rework while adding significant precision to the description of rework processes.

Furthermore, because Little-JIL is executable, rework specified this way can actually be executed according to a rigorous semantics, thus providing effective, automated support. Little-JIL programs can be executed using Juliette [13], our interpretation environment for Little-JIL. Juliette faithfully executes the programs according to the Little-JIL semantics by assigning work to execution agents at appropriate times. The agents choose among those steps assigned to them which steps to execute, and when. In this way, we can allow the flexibility that is needed by opportunistic development, while still providing an overall framework for the process, including rework. Thus, by using rigorous semantics, both custom and borrowed from general programming languages, we can provide automated support for realistic software development processes.

3 Related Work

3.1 Rework Models

There is a common belief in the importance and inevitability of rework in real software development. But, in searching for a careful definition of this common activity, we were struck by its absence from the indices of many textbooks, for example, [20, 36, 39, 34, 38] and from descriptions of the Capability Maturity Model [33]. Thus, research into how to formalize rework processes can help to address an important gap.

Others have argued that software development does not proceed without rework, and have therefore argued the need for modeling of rework in general in software development processes [21, 10]. Rework is assumed in many of the popular software development lifecycles. These often describe the life cycle as a nominal flow with rework activities that cause cycles in the nominal flow. Other life cycle models are explicitly based on iteration, for example the Spiral Model [9] and the Unified Process [24]. However, in such life cycles the repeated iterations do not generally represent rework but new, incremental development. For example, an iteration of the prototyping phase in spiral development is not performed to repair earlier prototypes but to develop new

[3] Actually, different resources can be acquired for different invocations of the activity in the same context, as between the initial work and rework in the requirements phase. That is because the resource pool may change from one invocation to the next and the resource specification will be reevaluated upon each invocation. Of course, if the same resources should be used for each invocation in one (or more) contexts, this can be specified explicitly in the resource query.

prototypes to help assess new risks. In the Unified Process, the approach is not just iterative but explicitly "iterative and *incremental*" ([24], Chapter 5; emphasis added). However, even in incremental development, some rework is bound to occur, and how rework (as opposed to forward iteration) should be handled or represented is not well described. Additionally, life-cycle models are typically presented at a relatively abstract level that is independent of important, practical details of a rework context and activities that must be captured for definitions of rework processes to be useful.

Still others have argued that software development is inherently opportunistic. This is seen in design tools such as Argo [37] and its commercial counterpart Poseidon [5]. These provide automated support for design activities, and they even provide automated "critics" that can point out design problems that may entail rework. However, they do not support modeling of the design process in general or of design rework in particular. Specific kinds of rework have also been studied (for example, refactoring [19]). While this work is useful, automated support is generally lacking because triggers, and the resultant rework, are not formally defined and not integrated into the overall development process. The lack of automated support deprives practioners of those advantages cited in Section 1.

3.2 Modeling Languages

In contrast to the wealth of informal guidance in software development, workflow and process researchers have studied the software development process as a formal object. This research has produced many process languages. These languages have a variety of constructs, in various combinations, that make them more or less well suited for defining various sorts of rework processes.

A feature of Little-JIL that seems to offer much benefit is control abstraction and the ability to invoke a step from multiple contexts and with varying parameters. Many process languages and workflow languages offer some form of control abstraction with invocation semantics. Some of these are drawn directly from conventional programming languages, for example, HFSP [27], which is based on a functional computational model, and APPL/A [40], which incorporates and extends Ada control abstractions and invocation semantics.

Many other process languages offer comparable step or activity abstractions (such as Oikos [30], EPOS [17], ALF [11], and JIL [41]). Another common approach to process definition is based on Petri nets and comparable sorts of flow graphs, some of which also offer control abstraction (for example, SLANG [7], FUNSOFT Nets [18]). Process languages based on finite state models also sometimes offer control abstraction in the form of state or subchart abstraction (e.g., WIDE [12], state diagrams in UML [32], and current versions of StateMate [22]).

The main advantage of control abstraction and invocation semantics for definition of rework processes is in allowing an activity that is initially invoked at one point to be reinvoked at another if the work it achieved initially must be redone. Capturing rework in this natural way in languages that lack these semantics is problematic. For example, Kellner [28] used Statecharts [22] (a now relatively early version) to model processes to support timing simulations. The modeled processes included separate states for original

work and for modification of original work (i.e., for rework). Using Statecharts, transitions to these states are easy to define from multiple other states (representing multiple contexts). However, each state is only defined in one scope and the outgoing transitions from it are fixed (that is, "returns" from "calls" to a state do not in general go back to the "calling" state). Thus, rework has to be modeled with separate states for each context in which it may occur.

Another relatively common notion, comparable to that of step or activity, is the work context. A work context is a scope that groups tools, artifacts, tasks, and roles. Work contexts are often isolated scopes, sometimes with full transactional properties. Little-JIL lacks a notion of work context, but supports the specification of artifacts, resources, agents, and substeps from which a work context can be constructed by an external system. Some languages that support work contexts directly are Merlin [25], ALF [11], and Adele-2 [8]. Work contexts are often modeled (directly or indirectly) in terms of rules that may allow them to be invoked from (and return to) multiple contexts, thereby supporting rework by reinvocation. Additionally, the various transactional properties of work contexts help to shield rework efforts from conflicts with ongoing activities and also help to assure a measure of consistency in rework results.

Transaction management can often entail the automatic redoing of failed transactions. This can be interpreted as a kind of rework, even if the transaction is atomic and the results of the earlier execution are obviated. Apart from the languages that support work contexts, several others have transactional notions, including AP5 [16], Marvel [26], and APPL/A [40]. HFSP [42] supports rework directly with a redo clause which can be used to indicate reinstantiation of a step with different parameters. Compared to traditional transaction redo, this affords additional flexibility. This makes the HFSP redo especially appropriate for certain sorts of rework-in-context.

Scoping is particularly important for rework, for two main reasons. One is the introduction and binding of the various control and data elements that constitute a particular rework context.[4] With Little-JIL, for example, a step introduces pre- and postrequisites, substeps and their control flow, reactions and exception handlers, and resource and data parameters. A language like APPL/A [40], which is based closely on a general-purpose programming language, has the sorts of scoping constructs that one would expect in a general-purpose programming language, such as packages and subprograms.

The second reason that scoping is important for rework is as a means to control visibility. Control of visibility is especially important for rework because of the potential for interference between rework and ongoing activities and the need to manage impacts to shared and dependent artifacts. In Little-JIL, steps define a strictly local scope for data and a hierarchical scope for exception handlers and execution agents. Languages based directly on general-purpose programming languages or paradigms, such as APPL/A [40] and HFSP [27], can adopt scoping rules from those approaches. Many flow-graph and Petri-net based languages allow for hierarchical scoping of contexts by

[4] Strictly speaking it is not the scope that does this directly, but a construct, such as a step in Little-JIL, with which a scope is associated. Nevertheless, it is natural to think of such constructs as scopes, since we typically introduce them in a program for purposes of creating a scope containing certain elements.

nesting [7, 18, 31, 12]. AP5 [16], which is a combination of Lisp and relations, uses relations to build hierarchical scopes within an unscoped space.

A number of process and workflow languages include aspects related to data definition and flow. This is important for rework in representing the artifacts that are subject to rework, dependent artifacts, dependency relationships, and the flow of artifacts to (and from) rework activities. It also helps in coordinating rework and non-rework activities and in defining and evaluating conditions for the triggering of rework activities or acceptance of rework results. Little-JIL represents the flow of artifacts between steps, as do practically all software process and workflow languages. Little-JIL does not support data modeling but is intended to be used with external data definition and management services. Some process languages, such as Marvel [26] and Merlin [25], do not support full data definition (relying on external systems for that), but they do enable attributes to be associated to artifacts. These can be used to reflect the "state" of the artifact, including its state with respect to rework (for example, whether it is "complete", "outdated", "under revision", and so on). A few languages offer full data modeling capabilities (for example, APPL/A [40] and FUNSOFT Nets [18]), thus allowing the definition of rework processes based on details of the data affected.

Organization and resource modeling are somewhat analogous to product modeling and, like product modeling, provide the basis for an important form of parameterization of rework contexts. Many workflow languages especially support some form of organization, role, or resource modeling(for example, [1–4]). Little-JIL allows resources, including execution agents as a special case, to be specified for process steps, although resource modeling itself is intended to be supported by an external service. Some other process languages, such as StateMate [22] and FUNSOFT Nets [18], also incorporate organization modeling as a first-class element. Other software-process languages, for example, Merlin [25], EPOS [17], and ALF [11], allow the specification of particular kinds of "software-oriented" resources, such as user roles and tools, that may be involved in particular activities.

Finally, we view exception handling as an important part of rework processes. Rework is generally triggered by some form of exception, and rework processes can be viewed as a form of exception handling. Additionally, rework processes themselves may throw exceptions, for example, due to conflicts with other activities or the failure to produce satisfactory results, and so rework processes themselves may require exception handling. Exceptions and exception handling are common notions in programming languages (for example, C++, Java, and Ada, among others). APPL/A [40] adopted Ada-style exception handling, and its successors JIL [41] and Little-JIL continue to represent exception handlers as a first-class elements. With Little-JIL we have found them important for specifying rework processes (as discussed in Section 2). Surprisingly, few other software process languages include general exception handling. There are many languages, however, that provide support for the handling of exceptions that can be specified as consistency conditions (for example, AP5 [16], Marvel [26], Merlin [25], EPOS [17], ALF [11], and others). Exception handling in the programming-language style is uncommon in workflow languages, though it is not non-existent [2, 4, 1].

4 Recommendations and Applications

Rework processes exhibit a variety of characteristics, both within themselves and in relation to the overall software process. Many different kinds of language construct are thus potentially relevant to the formalization of rework processes. In formalizing rework processes in Little-JIL we have taken the approach of representing rework processes as steps that can be invoked from multiple contexts (parent steps). These contexts can be specialized in various ways to reflect the particular circumstances and requirements under which initial development and rework occur. These contexts may differ with respect to pre- and postrequisites, associated activities and reactions, and exception handlers. The contexts in general and the rework activities in particular can be parameterized with appropriate data, resources, and execution agents, according to their place and purpose in the overall process. The assignment of resources and agents can be made statically or dynamically, and control flow within the rework context and the rework process may be more or less strictly determined, as appropriate for the process. In particular, control flow may be ”hard coded”, left entirely open to the choice of the execution agent, or controlled or not to various intermediate degrees.

From a programming-language perspective, the sorts of constructs and semantics used in Little-JIL to support formalization of rework include control abstraction, interfaces, parameters, control and data flow, scoping, messages (events) and message handlers, and exceptions and exception handlers. We find that the context of rework, which abstractly is a rich notion, is best specified using a flexible combination of these constructs. We find that the rework activity, which can be highly parameterized, is well represented by a procedure-like construct, particularly as that can allow the rework activity to be invoked from and tailored to multiple points in a process at which rework may be found necessary.

The particular collection of mechanisms used in Little-JIL was chosen because the purpose of the language is to support specification of the coordination aspects of software development and similar processes. That is, Little-JIL is intended to support the specification of processes with particular emphasis on the orchestration of steps including especially the binding of agents, resources, and artifacts to those steps. Additionally, Little-JIL is intended to allow these aspects of processes to be specified and understood by people who might lack extensive training or experience in programming but who nevertheless require precise, rigorous process definitions. We believe that Little-JIL addresses these goals for rework processes.

As discussed in Section 3.2, there are a number of additional types of constructs that other sorts of process languages include that can also be usefully applied to the formalization of rework. These include (among others) transactional constructs, data modeling, and consistency specification and enforcement. Data modeling, for example, is appropriate for processes in which control depends closely on the details of data, whereas transactions can be important where data integrity and consistency need to be assured in the face of potentially conflicting activities. In general, constructs for the formalization of rework processes should be chosen according to the kind of information to be captured, the purposes for which it is to be captured, and the circumstances under which the process specification will be developed and used, including the availability

of supporting technologies and the qualifications of modelers, analysts, and others who work on or with the process specifications.

We believe that the formalization of rework processes using process-programming languages affords benefits in three main areas. The first is process understanding. The act of defining processes itself often brings a new understanding, and process definitions, once formulated, can be shared with process engineers, managers, planners, customers, and collaborators. Because the process definitions are formal, they have a precise and rigorous semantics that promotes clarity. This formality leads to the second area of benefit, which is analyzability. Process programs are amenable to automated analysis, which can be used to assess the correctness and other properties of rework processes. This can be useful in verifying and evaluating large process programs that may be beyond the scope of an individual to manually analyze in detail. Analysis of Little-JIL programs is described in [15]. Formality also enables benefits of the third kind, namely, those based on support for process execution. The executability of process programs can support process automation and guidance, thereby promoting process efficiency, control, and fidelity. Additionally, as a process executes, it can be automatically monitored, and information about the process can be gathered for purposes of measurement, analysis, evaluation, planning, and accounting.

Benefits in these areas are particularly applicable to rework. Formalization of a process in a process-programming language can clarify where various kinds of work and rework occur in a process and suggest opportunities to minimize and optimize rework efforts. Analysis can help to compare alternative process models and to verify properties relating to the sequencing, timing, impact, and other properties of rework tasks. Execution support can help to assure that rework activities are carried out as specified, supported appropriately, coordinated with other activities, and accomplished correctly and efficiently with minimal impact. Monitoring mechanisms can track initial development and rework costs, help to show the benefits of rework mitigation efforts, and facilitate planning and management for ongoing activities. For all of these reasons, we believe that using process-programming languages to formalize rework processes not only teaches us something about process languages but also has the potential for significant practical benefit in an area of software development that continues to be problematic and costly.

5 Conclusions and Future Work

Software-process programming seems to us to hold out the clear prospect of being effective in providing a number of benefits to software engineers. This paper provides tangible and specific evidence of some of these benefits. Chief among the benefits seems to us to be the prospect that, through the use of sufficiently powerful and precise languages, software-process programs can effectively elucidate the nature of key software processes. In this case, we have demonstrated the potential of an effective process programming language to elucidate the concept of rework. As noted, rework is a common software development concept, referred to often by practitioners, but seldom addressed, no less carefully defined, in software engineering textbooks. One explanation, to which we subscribe, is that the notion of rework requires specific semantic notions in order

to be adequately explained and understood. In this paper, we suggest that such notions include procedure invocation, scoping, and exception management, to name just a few. Thus, this paper has demonstrated that effective process languages can materially add to our understanding of commonly used software engineering terms and practices.

The paper has also demonstrated the potential for an executable software-process programming language to aid practice by being the basis for more powerful, precise, and effective guidance of practitioners. While the example given in this paper merely suggests at the potential complexity of rework processes, we believe that the example, nevertheless, does suggest that rework situations can become rather complex and intricate. In such situations, the availability of an executable process definition to help guide practitioners through the rework process, and then back to mainstream development would seem to have considerable value. In cases where there are multiple practitioners engaged in multiple rework instances driven by multiple contingencies, the value of a clear, disciplined process, supported by effective process interpretation would seem to offer the potential for important savings in effort, confusion, and errors.

While we are convinced of the potential for software-process programming to offer advantages of at least these two sorts (i.e. Clarification of the nature of key processes, and effective aid in performing them), we are aware of the fact that the work described here is hardly a definitive demonstration of these advantages. Thus, we propose to continue these investigations by using process-programming languages to program increasingly complex and realistic rework processes. In this work, we expect that our processes will entail more complexity, greater use of exception management, more involved examples of process variation, and more realistic specifications of resources. Having developed such processes, we expect to use our process interpretation facilities to support the execution of these processes in order to gauge the value of such process guidance to real practitioners engaged in real software development.

Through these continuing experiments we expect to gain greater understandings of the sorts of linguistic semantic features of most value in supporting the representation of rework processes, and indeed wider classes of realistic software processes. In addition, however, we expect that this research will lead to rework processes that are known to be of value and utility. As these processes will have been captured in a rigorous language, they would then be reproducible and usable by others, including real practitioners. In this way, we anticipate that we will be able to demonstrate how effective process languages can be useful vehicles in improving the state of software practice through improved understandings of the nature of software processes.

Acknowledgements

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, and by the National Science Foundation under Grant No. CCR-0204321. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

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 Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, The National Science Foundation, or the U.S. Government.

References

1. FileNet Business Process Manager; URL: http://www.filenet.com.
2. PAVONE Process Modeler; URL: http://www.pavone.com/web/uk/pages.nsf/goto/home.
3. Ultimus Process Designer; URL: http://www.ultimus1.com/.
4. MultiDESK Workflow; URL: http://www.multidesk.com/.
5. Poseidon for UML. http://www.gentleware.com.
6. *First CeBASE eWorkshop: Focusing on the cost and effort due to software defects*. NSF Center for Empirically Based Software Engineering, Mar. 2001. http://www.cebase.org/www/researchActivities/defectReduction/eworkshop1/.
7. S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second Int. Conf. on the Soft. Process*, pages 75–83. IEEE Computer Society Press, 1993.
8. N. Belkhatir, J. Estublier, and M. L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187 – 222. John Wiley & Sons Inc., 1994.
9. B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61 – 72, May 1988.
10. K. Butler and W. Lipke. Software process achievement at Tinker Air Force Base. Technical Report CMU/SEI-2000-TR-014, Carnegie-Mellon Software Engineering Institute, sep 2000.
11. G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A framework for building process-centred software engineering environments. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 153 – 185. John Wiley & Sons Inc., 1994.
12. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. on Database Systems*, 24(3):405–451, Sept. 1999.
13. A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. Technical Report 99-65, U. of Massachusetts, Dept. of Comp. Sci., Nov. 1999.
14. B. K. Clark. *The Effects of Software Process Maturity ON Software Development Effort*. PhD thesis, University of Southern California, Aug. 1997.
15. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. In *Proc. of the 2000 Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 96–101, Aug. 2000.
16. D. Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.
17. R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

18. W. Deiters and V. Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT/SIGPLAN Symp. on Practical Soft. Development Environments*, pages 193–205. ACM Press, 1990. Irvine, California.
19. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
20. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.
21. T. Haley, B. Ireland, E. Wojtaszek, D. Nash, and R. Dion. Raytheon Electronic Systems experience in software process improvement. Technical Report CMU/SEI-95-TR-017, Carnegie-Mellon Software Engineering Institute, nov 1995.
22. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Soft. Eng.*, 16(4):403 – 414, Apr. 1990.
23. J. Herbsleb, A. Carleton, J. Rozum, J. Siegel, and D. Zubrow. Benefits of cmm-based software process improvement: Initial results. Technical Report CMU/SEI-94-TR-013, Carnegie-Mellon Software Engineering Institute, aug 1994.
24. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.
25. G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 103 – 129. John Wiley & Sons Inc., 1994.
26. G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In B. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, Jan. 1990.
27. T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the Eleventh Int. Conf. on Soft. Eng.*, pages 343 – 353. IEEE Comp. Society Press, 1989.
28. M. I. Kellner. Software process modeling support for management planning and control. In *Proc. of the First Int. Conf. on the Soft. Process*, pages 8–28. IEEE-PRESS, Oct 1991. Redondo Beach, CA.
29. J. King and M. Diaz. How CMM impacts quality, productivity, rework, and the bottom line. *CrossTalk - The Journal of Defense Software Engineering*, pages 3–14, Mar. 2002.
30. C. Montangero and V. Ambriola. OIKOS: Constructing Process-Centered SDEs. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.
31. National Institute of Standards and Technology (NIST). *Integration Definition For Function Modeling (IDEF0)*, 1993. Federal Information Processing Standards (FIPS) 183.
32. Object Management Group. OMG Unified Modeling Language Specification. Technical Report formal/03-03-01, Carnegie-Mellon Software Engineering Institute, mar 2003. Version 1.5.
33. M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis. *Guidelines for Improving the Software Process*. Addison-Wesley, New York, 1995.
34. S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, second edition, 2001.
35. R. M. Podorozhny, B. S. Lerner, and L. J. Osterweil. Modeling resources for activity coordination and scheduling. In *Proceedings of Coordination 1999*, pages 307–322. Springer-Verlag, Apr 1999. Amsterdam, The Netherlands.
36. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, fourth edition, 1997.

37. J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Argo: A design environment for evolving software architectures. In *Proc. of the Nineteenth Int. Conf. on Soft. Eng.*, pages 600–601. Assoc. of Computing Machinery Press, May 1997.
38. S. R. Schach. *Classical and Object-Oriented Software Engineering with UML and Java*. WCB/McGraw-Hill, fourth edition, 1999.
39. I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.
40. S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Soft. Eng. and Methodology*, 4(3):221–286, July 1995.
41. S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. In *Proc. of the Sixth European Soft. Eng. Conf. held jointly with the Fifth ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 142–158. Springer-Verlag, 1997. Zurich, Switzerland.
42. M. Suzuki, A. Iwai, and T. Katayama. A formal model of re-execution in software process. In *Proc. of the Second Int. Conf. on the Soft. Process*, pages 84–99. IEEE-PRESS, Feb. 1993. Berlin, Germany.
43. A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci., Apr. 1998.
44. A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Software Engineering Conf.*, Sept. 2000. Grenoble, France.