An Approach to Modeling and Supporting the Rework Process in Refactoring

Xiang Zhao
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
xiang@cs.umass.edu

Leon J. Osterweil
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
ljo@cs.umass.edu

***Abstract*—This paper presents the definition of a process for performing rework, and a tool that executes the process in order to support humans seeking help in being sure that they are carrying out rework completely and correctly. The process definition treats rework as the reinstantiation of previously-performed activities in new contexts, which requires the careful specification and management of the values of the artifacts that comprise key process execution history and contextual information. The rework tool exploits access to this information to provide human reworkers with guidance about the rework tasks to be done and with context and history information expected to be useful in guiding superior rework decisions. The paper presents a detailed example of the use of the process and tool in supporting a particular kind of rework, namely the refactoring of the design of an Object-Oriented program.**

***Keywords*-rework; software process; refactoring**

I. Introduction

The reconsideration and revision of decisions and activities that had taken place previously is a central activity in creative processes such as the development of software. We suggest that this notion of reconsideration and revision be taken as an intuitive description of the activity that is commonly referred to as rework. Rework seems inevitable in virtually any activity that entails exercising judgment, being creative, or speculating about imponderable future events. Thus, for example, civil engineers might need to rework plans for digging a building foundation when subsurface features (e.g., large rocks) are discovered. Doctors might need to rework treatment plans when patients develop unexpected reactions and symptoms, and educators might need to rework course plans when classes turn out to have unexpectedly large numbers of underprepared students. In virtually all of these situations, it should be expected that rework will needed, probably on an ongoing basis. Rework should be expected because understandings of problems and proposed solutions grow over time, invariably creating new perspectives on prior approaches and decisions. These future understandings are impossible to predict. Rework should be expected to be ongoing at least partly because these solution activities are aimed at solving problems arising in the real world, and the real world changes constantly. Ongoing changes in the world create changes in context that affect the suitability of choices made about how to solve problems.

Of greatest interest to us is the need for rework that arises in software development projects. In software development initial decisions (e.g. design choices or assumptions about user communities) often turn out to be incorrect or unworkable, typically necessitating rework. Rework becomes necessary when design considerations indicate that initial requirements may have been incomplete or inconsistent. Conversely requirements changes often necessitate rework of previous architecture and design decisions. Code rework is necessitated when design specifications change. And system testing outcomes often necessitate design, coding, and requirements rework. Because software systems address real world problems and operate under real-world constraints, the changes that take place continually in the real world typically necessitate continual software rework. Indeed, the need for rework is so prevalent, that it has been suggested that software developers spend most of their time doing what we characterize here as ”rework”.

In earlier work we have suggested that the most commonly enunciated characterizations of rework are insufficient and can lead to inadequate attempts to support rework [1] [2]. Thus, typically [3] it is suggested that rework consists of ”going back to a previous phase” of software development to redo decisions made or work carried out in that previous phase. But it is rarely the case that a software project really ”goes back” to a previous phase of development. If requirements specifications are found to be incomplete, inadequate, or incorrect during design, it is rare for the design phase to be aborted in favor of reopening the requirements phase. Instead activities more typically associated with the requirements phase are repeated, but now in the context of the design phase. Thus, for example, a requirement specification task will be instantiated to create a new requirement specification element whose need had not been previously appreciated, but whose need is now understood in the context of subsequent design activities. The context of the further design is essential to the more effective specification of the missing requirement. Such examples have led us [4] to propose that rework is more accurately characterized as the reinstantiation of tasks pre-

vious carried out in earlier development phases in the richer context that is provided by the activities and artifacts that had been performed and created during subsequent phases. A troublesome consequence of the need to make such changes is that these changes often create inconsistencies with other decisions, creating the need to revisit those other decisions. We characterize this need to address inconsistencies created in this way as *consequential rework*, which leads to what is often characterized as "ripple effect", with rework leading to consequential rework that can lead to further consequential rework, potentially ad infinitum.

In this paper we explore and evaluate our previously proposed characterization of rework by exploring in more detail a kind of rework prevalent in modern software development, namely refactoring the class structure of an Object-Oriented (OO) program. This is a good example of rework because the class structure of an OO program comprises a key element of the program's design, and yet refactoring typically takes place either during the coding phase, or even after deployment, when experience indicates that initial design decisions need reconsideration, revision, and improvement. In this case, refactoring does not involve leaving the coding or deployment phase, but rather entails instantiating design tasks in the context of those later phases. In this case, the context of subsequent experience (coding difficulties, performance issues, etc.) creates the richer context needed to inform the rework that must be done. Note also that the need to modify a single class definition often creates consequential rework and ripple effect.

This paper provides a more complete and specific discussion of our proposal about the nature of rework, through presentation and evaluation of a system that adopts our notion of rework as the basis for its architecture. Section II presents our notion of rework more carefully. Section III then describes the system itself, and its conceptual basis drawing upon our ideas about rework. Section IV presents an example of how our system has been used to support the refactoring of an OO class structure. Section V presents the capabilities of some other refactoring tools, comparing them to our own tool. The paper provides suggestions for future work in section VI and concludes in section VII.

II. An approach to representing rework and providing automated support

A. A Rigorous View of Rework

In earlier work [5] we have suggested that rework can be modeled as the handling of an exception that is thrown when an inconsistency has been identified. Inconsistencies of many kinds can arise at many places in software development, and accordingly various consistency checks are typically incorporated throughout development processes (often these checks are made during periodic reviews). The failure of a consistency check (e.g., for the adherence of behavior to requirements) then becomes an event that triggers the instantiation of a process aimed at remedying the inconsistency. That process is typically a software development task that had been performed previously. But the triggered new performance of the task is expected to have a better outcome because the new performance will take place in a richer context that includes the outcome of the failed consistency check, new software artifacts, and consistency checks that have occurred subsequent to the previous instantiation of the task being undertaken.

This view of rework suggests that the context provided by such things as newly created artifacts, recently performed consistency checks, and the outcomes of previous decisions are central to an appreciation of the nature of rework. Simply repeating a task that had been carried out previously could possibly deliver the same outcome, unless that task has the benefit of more knowledge, provided by subsequently generated artifacts and understandings. Thus, it is the provision to revisited tasks of contextual information carried in the form of artifacts and outcomes that seems to us to characterize rework. Being precise and complete in describing this context becomes increasingly important (and difficult) in dealing with iterated rework (the need to rework decisions and activities that had already been reworked previously), consequential rework, and ripple effects. In these cases entire histories of decisions, activities, and resulting artifacts are increasingly needed in order to be sure that complete and correct contextual information is made available to increasingly complex reconsiderations.

In other prior work [6] [7], we have also suggested that abstractions and capabilities that have been previously developed for programming languages can be quite useful in supporting the precise specification of complex processes such as software development. Rework seems to provide a good example of this, in that the intuitive notions just presented seem likely to be described far more precisely and usefully using programming language constructs. Thus, for example, as suggested in [5], consistency checks can be modeled as postconditions or **if** statements, and the triggering of rework can be modeled as the throwing of an exception. But the above discussion of the essence of rework as being the provision of context suggests that, in addition, mechanisms for using scoping concepts to manage the access to software artifacts could be of central importance as well. Appropriate abstraction mechanisms, using scoping information to control which artifacts are to be made available to which instantiations of rework activities, seem particularly useful in supporting rework. The support of iterated rework and ripple effect would seem to be facilitated, in particular, by visibility rules that are central to supporting nested scoping. Moreover, as shall be seen, the importance of being able to benefit from inspecting the outcomes of previous activities and decisions, also suggests the value of incorporating historical retrospection into capabilities for supporting rework.

In view of the previous discussion, we suggest that processes and tools for supporting rework are more likely to be more effective if they support the semantic features just described. In particular we hypothesize that the performance of rework processes can be materially facilitated by executing process definitions that incorporate appropriately strong and precise specifications of context by means of process language features such as scoping, abstraction, parameter passing, recursion, and historical retrospection. A key goal of this paper is to explore that hypothesis.

Exploration of this hypothesis thus requires the use of a process definition language that is executable and that supports the semantic features just enumerated. The Little-JIL process definition language offers these features and thus has been taken as a key vehicle for the evaluation of our hypothesis. A full description of Little-JIL can be found in [8]. Here we summarize very briefly some of the key features of the language that are most relevant to exploring our hypothesis. Additional language features will be presented in the context of the rework process descriptions to be found in the next section of this paper.

B. Using Little-JIL to Define Rework

Little-JIL is a rigorously defined language intended to be used to support the definition of complex processes. The language incorporates such semantic features as abstraction, concurrency, hierarchical decomposition, exception management, and human-user-driven choice in order to facilitate the clear and detailed specification of complex processes down to low levels of detail. Little-JIL process definitions are comprised of three main components, specifying an **activity structure**, an **artifact space**, and a **repository of resources**, some of which can serve as agents for executing activities. The Little-JIL activity structure is the central feature of a definition, and has a visual representation intended to make process definitions more accessible to domain experts.

An activity specification is a hierarchical decomposition of steps. The visual representation of a step is shown in Figure 1. A Little-JIL step is best thought of as a procedural abstraction defined by a hierarchical decomposition of substeps, with parent and child steps communicating with each other through artifacts that are passed back and forth

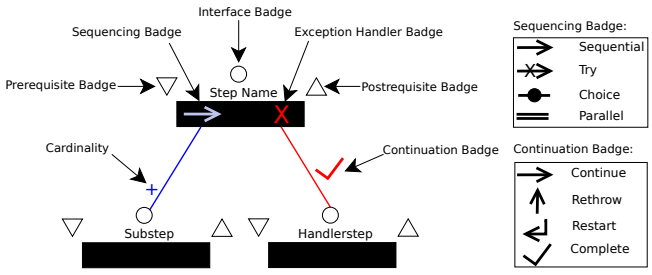

Figure 1. Little-JIL Step

as arguments, bound to formal parameters that are part of each step's definition. In Figure 1, a step is represented by a black rectangular step bar, with the step's name shown above the bar. Parameters are incorporated into the step's external interface, represented iconically by a small circle above the step bar. Substeps are represented as steps shown below the parent, but connected to the parent by edges. The edges support specification of argument artifacts passed and bindings of arguments to formal parameters. Steps have two types of substeps, ordinary substeps, attached to the parent by edges emanating from the left of the step bar, and exception handlers, attached to the parent by edges emanating from the right of the step bar. The left side of a non-leaf step bar contains an iconic representation of the order in which substeps are to be executed. There are four order specifications: sequential (represented by a right arrow), parallel (represented by an equal sign), and two specifications that allow human process participants to make choices, namely the choice step and try step kinds (iconically represented respectively by a slashed circle and an arrow with an X on its tail).

Every Little-JIL step is to be executed by an agent, whose specification is incorporated as part of the step's external interface. A step's agent can be either a human or an automated device (hardware or software). Agents can throw exceptions, which are typed objects. Exceptions can also be thrown during the execution of step prerequisites or postrequisites, step structures optionally incorporated into the execution of a step by being executed either before (prerequisite) or after (postrequisite) the step executes. Exceptions are handled by exception handlers, substeps emanating from the right of ancestor step bars. Each exception handler handles exceptions of a specific type specified as part of the handler's definition. Because exception handlers are Little-JIL steps they have external interface specifications that define how they handle artifacts passed as arguments. Steps are true procedural abstractions, and thus can be instantiated multiple times, with each instantiation being differentiated (e.g., by parameter bindings and exception handler access) according to the context in which the instantiation is done. Recursion is defined straightforwardly by instantiation of a step as its own descendant.

These language features seem particularly effective in supporting the needs of rework definition. The hierarchical structure of Little-JIL supports nested scoping. Rigorous parameter passing semantics (Little-JIL uses copy-and-restore) support rigorous control of the visibility of artifacts by step instances, particularly important to the creation of contexts for activities. Recursion seems important for support of nested rework. Treating exceptions as typed objects provides a vehicle for separating the handling of different instances of consequential rework from each other.

The next section presents an example that makes clear the relevance of these kinds of language features to supporting the specification and support of rework.

III. Refactoring as an Example of Rework

In this section we present a Little-JIL definition of a process to support one form of rework, namely refactoring the class structure of an Object-Oriented (OO) program, and a tool to support execution of the process.

A. Refactoring as an Example of Rework

Decisions about the concepts to be captured as classes and objects are among the most important decisions made in creating an OO program. A good choice of classes makes an OO program more readable, more evolvable, and more straightforward to program. Bad choices lead to programs that are problematic. It is neither unusual nor surprising to find that initial decisions have led to later problems. The accumulation of knowledge, insights, and subsequent decisions can suggest how to reduce these problems by rebundling methods and capabilities into classes. This form of rework is commonly called refactoring. Initial refactoring changes may be incorrect, causing consequential rework. Moreover, these changes in class definitions may themselves spawn the need for more subsequent changes (the previously-described ripple effect).

We now present an example of a specific form of refactoring called *separating query from modifier* [9]. This kind of refactoring deals with a method that is used both to query, and also to change the state of an object. Combining these two capabilities into a single method may have seemed like a good idea initially, but subsequently it may turn out that the method is used mostly as a query, where side effects are awkward. An obvious response is to refactor the method into two methods, a query and a modifier. We now present a carefully and precisely defined Little-JIL process definition that can be useful in helping assure that all necessary changes are made correctly.

B. A Little-JIL Refactoring Process Definition

Figure 2 shows the top hierarchical level of a Little-JIL definition of a process for guiding the refactoring just described. The process is represented by the step **Separate Query from Modifier**, which is defined to be the sequential execution (note the right-facing arrow in the left of the step bar) of five child substeps, **Specify QueryModifier Method**, **Create Query**, **Modify Original Method**, **Update References**, and **Deal with Original Method**. Each of these substeps must be further decomposed in order to support the kind of detailed refactoring support that we seek. Because of space limitations we focus only on the third and fourth substeps. Figure 3 shows the third substep, **Modify Original Method**, decomposed into three ordinary substeps, and two exception handler substeps. The three ordinary substeps describe the sequence of activities taken under nominal circumstances, namely performing sequentially namely making the change in the method (**Make original method return a call to the new query**), then compiling the newly modified code (**Compile**), and then running a suite of test cases to assure that the change has been made correctly (**Run unit tests**). Note that some of these steps are performed by a human agent (the refactorer), and some are performed by automated agents (e.g. a compiler, and an automated test aid). Essential to effective support of this step, however, is providing support in the not-unlikely case that the change has not been made correctly. Thus, the **Modify Original Method** step incorporates as substeps **Handle Compilation Error** to deal with exceptions raised when the changed code fails to compile, and **Handle Unit Tests Failure** to deal with exceptions raised when execution of one or more unit test cases does not deliver correct results.

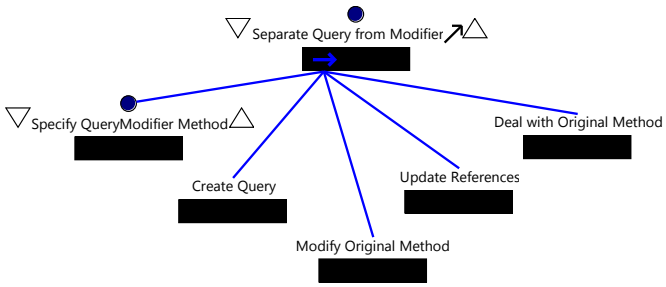

Figure 2. Top-level Process Definition

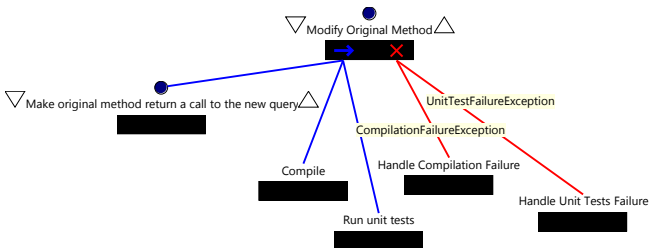

Figure 3. Modify Original Method Step Definition

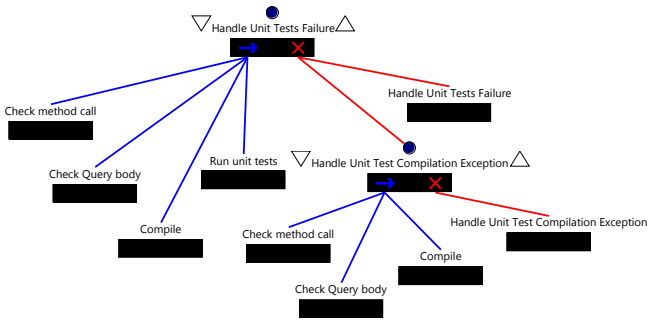

Figure 4. Handle Unit Test Failure Step Definition

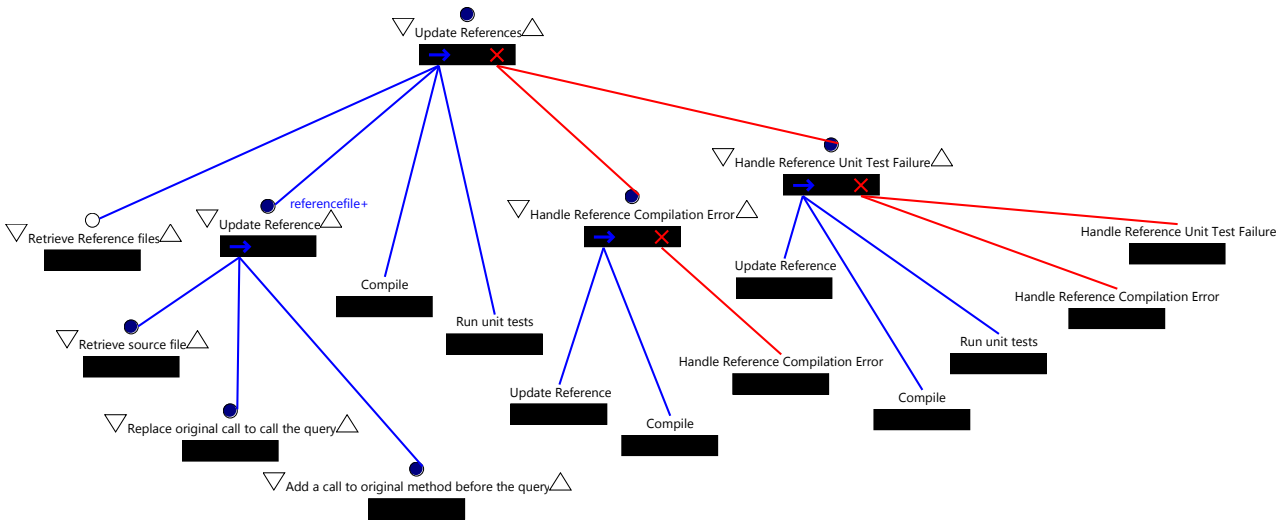

Figure 5. Update References Step Definition

One way this process supports rework is shown by how the process handles these exceptions. Figure 4 provides further details about this by showing how unit test failures are handled. Note that more than one unit test case may fail, and that each failed test case must be examined individually. Little-JIL supports specifying this by treating an exception as an object whose type is used to select the appropriate handler and whose argument artifacts are used by the exception handler to support appropriate responses. In Figure 4 the **Run unit tests** step throws one **UnitTestFailureException** for each unit test case that fails, causing each unit test case failure to instantiate a separate **UnitTestFailureException** handler.

The handling of a single unit test case failure is defined by the elaboration of the **Handle Unit Tests Failure** step shown in Figure 4. This consists of performing in sequence the steps **Check method call**, **Check Query body**, **Compile**, and **Run unit tests**. Examination of this step shows both the complications that can arise during refactoring, and the process language features that can be effective in addressing this complexity. Note that the first two substeps support examination of the specification and the body of the modified method in the hope that a defect will be found and fixed. But then recompilation of the method and rerunning of the unit tests follow to confirm that defects have been found and fixed. In the not-implausible case that one or more defects remain, compilation error exceptions and/or unit test failure exceptions will be thrown. In case a compilation error is thrown, it is handled by the **Handle Unit Test Compilation Exception** exception handler, whose substep decomposition is almost identical to the **Handle Compilation Error** substep decomposition that caused the initial consideration of the compilation error. In case a unit test error is thrown, it is handled by a recursive call to the **Handle Unit Test Failure** exception handler. In both of these situations previously executed debugging steps are now being executed again, but now in the context of additional information (e.g. new defects observed, or indications of why previous attempts to fix previously-identified defects have not succeeded). These new context can be seen clearly from examining the Little-JIL diagrams. The values that comprise these contexts are constructed from the values of the artifacts that are visible from these step instantiation sites.

A similar situation can be seen in the elaboration in Figure 5 of **Update References**, the fourth substep of **Separate Query from Modifier**. This substep is the sequential execution of four substeps, with the additional specification of two exception handlers. The first substep, **Retrieve Reference files**, identifies all classes that refer to the changed class, the second substep defines an iteration through each such class, in which the **Update Reference** subsubstep of **Update References** is responsible for making the actual changes to each class, sequentially one at a time, the third substep **Compile** checks to be sure that all changes actually compile correctly, and the fourth substep **Run unit tests** reruns all unit tests. The exception handlers **Handle Reference Compilation Error** and **Handle Reference Unit Test Failure** include recursive invocations of **Update Reference** and **Compile**, which may cause further compilation errors thereby causing the recursive invocation of **Handle Reference Compilation Error** and **Handle Reference Unit Test Failure**. There is clearly a strong analogy between the internal structures of the third and fourth substeps of **Separate Query from Modifier**. Clearly the recursive invocations of steps in both cases can continue without limit. Clearly such recursive invocation sequences can go on in parallel for each of the exceptions that has been thrown. And, in addition, there is the clear possibility that steps taken to address one defect

may interfere with addressing other defects. Thus, these are two very representative examples of how rework, and potentially quite complex rework, arises quite naturally in the course of performing refactoring.

Clear understanding of what is needed to support human refactorers who undertake complex rework requires more than examination of the steps required to support rework. As the current example shows, the sequence of steps performed to do rework is essentially an iteration. But the essence of the rework is not the iteration, but rather the sequence of contexts within which each iteration is carried out. Thus a major challenge in supporting rework effectively is the ability to specify, create, and control the contexts within which process steps execute. A key component of this context is the set of values of the variables and artifacts to which the step and its performer have access. A Little-JIL process seems capable of managing these scopes and artifacts, and thus seems promising as the basis for a tool that can support human refactorers by providing clear access to them. We describe such a tool in the next section.

C. A Tool to Support Execution of the Process

Careful execution of the process just described (e.g. by Juliette [10], the Little-JIL interpreter) is necessary to support human refactorers, but far from sufficient. It is certainly necessary for Juliette to advise humans when each step is to be performed, and which artifacts (e.g. bodies of code) to perform the steps on. But the essence of the support we propose to provide is to be sure that humans have sufficient context and history to support a deep understanding of the changes needed. Accordingly, it seems necessary to also provide humans with clear visibility of that contextual and historical information as well.

Figure 6 is a high-level diagram of a prototype tool that provides such support. The righthand side of the diagram shows the Little-JIL process and an interpreter that performs needed delivery of the appropriate steps to the appropriate agents (in this case only one human agent is depicted) by placing the steps, and their associated artifacts, on the agents' agendas. But the diagram also shows that the artifacts included in the steps carry pointers that enable the human to access considerable historical and contextual information. That access is through a dynamic execution trace recording that we refer to as the *Data Derivation Graph (DDG)*.

The DDG is a vehicle for keeping track of how the values of process artifacts change as a process executes. Because the Little-JIL activity diagram is a static structure of types (e.g., types of steps that are to be instantiated, and types of arguments that are also to be instantiated, at runtime) it is unsuitable for representing artifact value evolution during process execution. The DDG has been developed to represent artifact value evolution as a structure that records how each instance of each artifact in a Little-

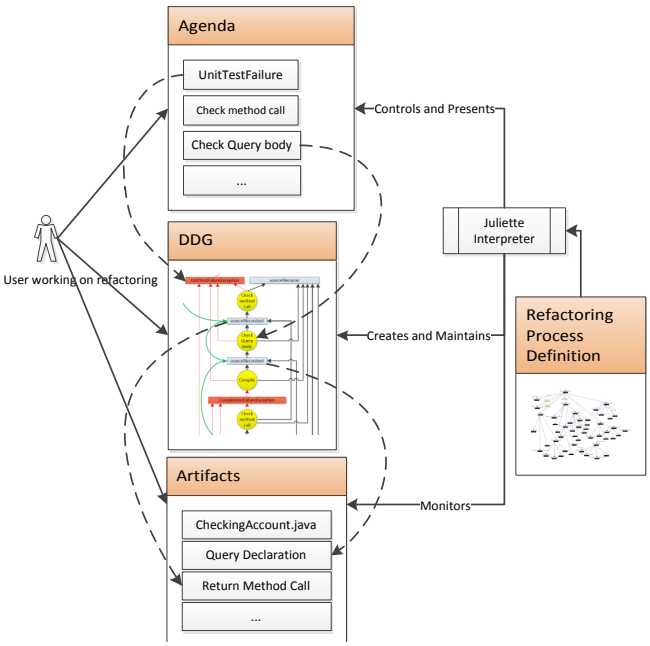

Figure 6. System Architecture

JIL process has been derived and is then used in subsequent derivations of new values. Further details about DDGs can be found in [11] [12]. A basic understanding of the nature of a DDG can be gained with the help of an example such as is shown in Figure 7. Figure 7 shows a DDG that it is a small portion of an idealization of the DDG that is built during execution of the part of the refactoring process shown in Figure 4. Rectangle blue and red nodes represent artifact values and are at the tails of edges whose heads represent the process step(s) that created them (represented by yellow round boxes). Thus, for example, the bottom node in Figure 7, **Unit Test Success**, represents the creation of a token indicating that all unit tests have succeeded, and this token was created by executing the **Run Unit Tests** step, which in turn required access to the **Unit Test Suite** artifact. The figure shows that this step also required a **Compilation Success** token, generated by the prior **Compile** step.

The **Compile** step itself required access to the name of the source file and the content of the file (**sourcefilename** and **sourcefilecontent** respectively). Note that the step **Check Method Call** required access to similar artifacts, but also required that a **CompilationFailureException** exception token had been generated.

Note that the top half of Figure 7 looks quite similar to the bottom half of the figure, and that is because each half represents a different iteration under the **Handle Unit Tests Failure** step shown in Figure 4. What is depicted is the response to a failure of the **Run Unit Tests** step, where the first response was a change that caused a compilation failure. On the next iteration, the unit compiled and execution of

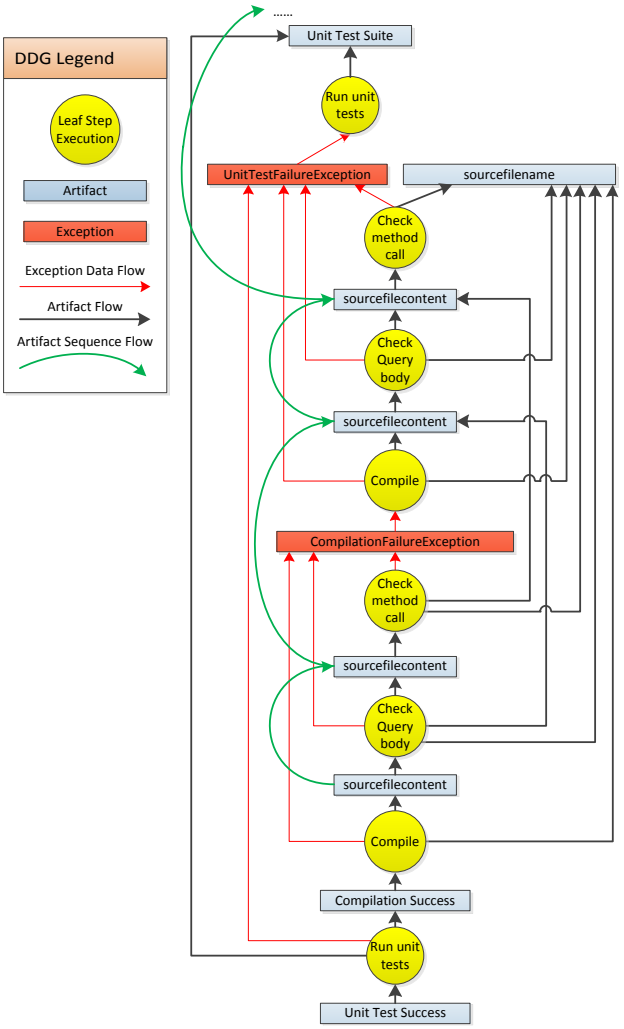

Figure 7. A DDG Example

Run Unit Tests also succeeded. We suggest that access to previous values of the compilation unit, represented by the **sourcefilecontent** artifact, was expected to be helpful, and so the figure depicts these values as being linked to each other (by curved green edges) for easy perusal by the user. In addition, the tool currently incorporates Meld [13], an open source visual diff and merge tool to help the user visualize artifact changes more easily. This expedited access to past values is an example of the kind of historical information that seems helpful in supporting rework. The artifacts managed by ancestor steps provide further context, and are also accessible through navigation up the DDG to the ancestor steps. Figure 7 shows that the DDG is managed as a structure of tokens, with actual artifact values managed separately (accessed through DDG links) because of the expectation that they may be large and require storage optimization.

IV. Evaluation of the Process and the Tool

In this section we present a specific example of our experience in the execution of the refactoring process, and our experience with the support tool just described, in the context of our efforts to refactor a simple class called **CheckingAccount**, which holds the balance and provides methods for withdrawing and depositing from a bank checking account, as shown in Listing 1.

Listing 1. CheckingAccount.java

```
1  public class CheckingAccount{
2      private float balance;
3      public CheckingAccount(float balance) {
4          this.balance = balance;
5      }
6      public float checkForBalanceAndWithdraw(
           float amount) {
7          if (balance >= amount) {
8              balance -= amount;
9              System.out.println("Widthdraw done.
                   Current balance: " + balance);
10             return balance;
11         } else {
12             System.out.println("There is not
                   enough balance!");
13             return balance;
14         }
15     }
16     public void deposit(float amount) {
17         balance += amount;
18     }
19 }
```

This class has combined the balance checking and withdrawal features in one method, which is to be refactored into two methods, a query of the current balance, and a withdrawal of money from the account.

Process execution begins with some steps that are performed by automated agents. The first step assigned to the human refactoring agent is **Specify Query Modifier Method**. A Linux terminal interface feature of our tool provides the user with the argument to this step, which is the full source code of the class and its method list. The terminal also shows a terse instruction on what to do in this step:

```
Please enter a query modifier method name:
```

The user is then directed to specify the full method signature of the method needed for refactoring as the output of this step, **querymodifiername**. The process then transmits this parameter to the parent step **Separate Query from Modifier**, which distributes it to its other substeps as input parameters. Next the process execution places on the user's agenda a step requiring the user to declare a query method in the class source file that returns the desired value, which in this case is the balance of the account. As this step is specified to require a code editor as a resource, the process execution system acquires a code editor, opens it with the source file being edited, and makes this available to the user through the user's agenda. In a similar way, the execution

system will in sequence ask the user to declare the query method, and modify the query method to return the same value as the original method. After the user has performed these steps, the Little-JIL interpreter then automatically calls the Java compiler to do the first check on the new artifacts. If there is a compilation failure, one or more **CompilationFailureException** instances will be thrown by the **Compile** step. A new step instance **Handle Compilation Error** will be initiated to respond to the exception, causing a form of rework that requires examining the previous decisions and making new decisions about modifying the artifact.

During rework, the user is guided by the tool to perform checks on the previous steps' artifacts from different perspectives. To illustrate this, suppose the user had created the query definition shown in Listing 2:

Listing 2. Sample Query Declaration

```
1 public float checkForBalance(){
2     return balance++
3 }
```

In this case the user has left a semicolon out of the declaration and wrote the wrong return statement, resulting in a compilation exception. The tool will then execute the appropriate exception handler by placing the appropriate step on the agenda of the human refactorer. Executing the first step will show the detailed compilation error message as output from the Java compiler, and indicate that **Check Query declaration** should be performed first, followed by the **Check Query body** step. At the same time, the user can check the currently generated DDG to review all past decisions and artifact changes. In this example, the DDG will show the change to the artifact is the newly added query declaration, and the change was done by having performed **Declare Query Method** and **Modify Query to return same value as original one** steps sequentially. This rework instance is very easy to deal with, and the user can simply append a semicolon at line 2. Note after the rework, the tool will initiate another compilation step to check for errors. If the user's fix still does not compile, another instance of **CompilationFailureException** will be thrown and handled by the nested **Handle Compilation Error** step. Our experience suggests that things become more difficult when a user thinks a problem was fixed but another exception is raised because the fix was defective. In this case, our tool's provision of contextual information becomes much more valuable in supporting the making of correct decisions, as will be described later in this paper.

Suppose exception handlers have now guided the correction of all compilation errors. In this case the complete history of all the changes the user made will have been recorded in the DDG for possible future reference. The process continues to guide the user to refactor this method by placing the next step **Make original method return a call to the new query** on the user's agenda. After the user makes the requested change, the tool will invoke the compiler to check for errors and jUnit to perform unit tests. Any number of instances of two different kinds of exceptions, either **CompilationFailureException** or **UnitTestFailureException** can be thrown here. Each exception instance will result in one or more rework instances. The **Handle Compilation Error** step follows the same pattern as before. We will assume that the user did some rework there and fixed the compilation issue. As is shown in Listing 2, the balance is not returned as it should be, so the **Run unit tests** step detects the error and a more complicated step **Handle Unit Tests Failure** is initiated. Figure 4 shows the fixing process involves revisiting two previous decisions that may be related to the unit test failure, **Check method call** and **Check Query body**. After the user checks the corresponding aspect in the code and fixes it, the source files will be compiled and the unit tests will be run again. In this context, assume that in the process of fixing the unit test failure, the user has accidentally caused another compilation exception. In that case a **CompilationFailureException** instance will be created in the course of this rework process. As is shown in Figure 4, another rework process is then invoked within the current one, to revisit **Check method call** and **Check Query body** and fix the local exception. The fixing step will be iterated if more exceptions are thrown and more rework instances are created. Once the tool has ensured that the code compiles, it returns to its previous high-level task of running the unit tests to see if the error has been fixed. If not, the tool executes a new **Handle Unit Tests Failure** step instance to guide further changes until the source files pass the unit tests. The tool will then proceed to direct that the user update references to the refactored method. The execution of this step is similar to what has been described previously. Note in Figure 5 shows *referencefile+* notation in the edge from **Update References** to **Update Reference**. This is an artifact binding *referencefile-referencefilelist[]* that specifies iteration over an artifact collection and causes the tool to fetch a list of source files that reference the method, iterate over the file collection, and instantiate one **Update Reference** step for each file. The potential rework in this step creates a more complicated situation where method references are updated in parallel, but changes to one artifact in the context of one reference may necessitate changes to other artifacts connected with other references. Although Little-JIL is able to define this complex rework, our DDG builder is currently not able to provide complete support.

To finalize the method refactoring process, the original method is modified to assign a void return type, also removing its return expressions so that it will no longer be used as a query method. The tool will notify the user of the completion of the process, and show the user the final DDG depicting the whole process execution.

Certainly the example just given is relatively straightforward, but the assistance provided by the DDG and Little-JIL

interpreter, were positively helpful. The tool assured that the user was always able to find out what the next step should be and enabled the user to examine previous decisions and current context. This was helpful even in this small example, and is expected to be even more helpful in larger examples that are to be carried out in future work.

Although this example never required recursive invocations of depth greater than five layers, the DDG generated consists of 60 data nodes and 147 procedure nodes. Clearly it will be important to devote serious attention to the efficient management of the DDGs generated by these sorts of rework processes.

V. Related Work

A number of tools are available to provide automated support for refactoring. While the degree of automation and the coverage of refactoring process vary, we note that a DESMET [14] evaluation of refactoring tool support [15] showed that none of the tools fully supported refactoring. Common software IDEs and plugins like Eclipse Java development tools (JDT) [16], JBuilder [17] and IntelliJ IDEA [18] offer refactoring as one of their key capabilities, but these tools focus on a selective set of refactoring tasks and selected parts of the refactoring process. For instance, Eclipse JDT supports some refactoring tasks like renaming fields and extracting methods, allowing users to specify intended changes with the tool automating the propagation of the changes. RefactorIT [19] supports more automation like detecting code smells. However, the refactoring support provided by these tools focuses only on automating some of the common operations while not providing support for the entire refactoring process. Without a rigorously defined process, these tools are not able to offer a complete solution to specific refactoring problems, and without the proper collection of history information, they are not able to provide context information to help users understand how their current decisions might be guided by their previous decisions.

Some other tools provide better support for the refactoring process. Guru [20] is derived from research on automatic refactoring of OO programs by restructuring inheritance hierarchies. Its Java equivalent Condenser [21] finds and removes duplicate code. These tools perform an entire refactoring process without human interaction. But because the process is fully automated, users cannot review refactoring progress during rework, and no user customization is possible. This can cause users to lose control over their source code.

Technologies for capturing data provenance during process execution have been studied by others. VisTrails [22] implements a provenance mechanism that captures changes to the data and displays them in a history tree. Callahan et al. [23] proposed a framework adapting this approach in process driven settings by creating a uniform environment. Kepler [24] provides a mechanism for integrating a broad range of supporting tools for specification, execution, and visualization of scientific data processes, and provenance data is built incrementally much as is done by our own DDG. Some of the other approaches to provenance are summarized in [25]. But these other process provenance systems are based upon process languages with less comprehensive semantics, which cause the provenance records that they generate to have fewer details, for example about the contexts in which data management events occur. We argue that our more comprehensive process definition language leads to a more complete and more useful data provenance model.

The Little-JIL language and DDG provenance technology address important shortcomings of these other approaches. Little-JIL processes such as the one described here, provide support for entire refactoring processes, while keeping users engaged and in control of the refactoring process. In addition our technologies afford users access to extensive historical and contextual information that make it easier to learn from past decisions and outcomes.

VI. Future Work

This work presented here is still preliminary, but it already suggests many future research directions. Most immediately, there should be much more experimentation with our approach and our tool. We will undertake larger and more complex refactoring examples, contriving more intricate rework scenarios. For example, in our current process multiple exceptions thrown by multiple compilation or testing errors are handled sequentially. But the exceptions should be handled in parallel, perhaps by multiple humans. This will require enhancement of our current tool support.

We plan to enhance our tool by automating more operations and invoking other useful tools for each step in our refactoring processes. For instance, in **Modify Original Method**, we could automate the modification of the return statement and make it return a call to the query method we declared. We will also add more of the refactoring processes described in [9] to our refactoring process library and specifically study their use of rework.

We will also address DDG scalability problems. Because DDGs preserve all artifact values, step execution sequences, and exception information, DDGs quickly become very large, causing querying to take longer. We will explore providing optimized DDG query support from within process definitions to address this problem.

VII. Conclusion

We have shown that an articulate process language can be used to create clear definitions of processes that seem able to support humans in doing complex rework tasks. An example of the rework entailed in OO code refactoring was presented. Initial experience suggests that this approach and tool support can provide valuable assistance to rework in

refactoring, but considerable additional research and evaluative experience are indicated.

ACKNOWLEDGMENT

The authors thank Sandy Wise for his strong support of this project in providing advice, expertise, and support for understanding Little-JIL and Juliette. We would also thank Barb Lerner for her advice, expertise, and insights into the design and implementation of the DDG.

The authors also thank the National Science Foundation for its support of this research through grant #CCF-0905530. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] A. G. Cass, L. J. Osterweil, and A. Wise, "A pattern for modeling rework in software development processes," in *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes*, ser. ICSP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 305–316.

[2] A. G. Cass, S. M. S. Jr., and L. J. Osterweil, "Formalizing rework in software processes," in *EWSPT*, ser. Lecture Notes in Computer Science, F. Oquendo, Ed., vol. 2786. Springer, 2003, pp. 16–31.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering*. Prentice Hall, 2003.

[4] A. Wise, A. Cass, B. Lerner, E. McCall, L. Osterweil, and J. Sutton, S.M., "Using Little-JIL to coordinate agents in software engineering," in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, 2000, pp. 155 –163.

[5] B. S. Lerner, S. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise, "Exception handling patterns for process modeling," *IEEE Transactions on Software Engineering*, vol. 99, no. RapidPosts, pp. 162–183, 2010.

[6] L. Osterweil, "Software processes are software too," in *Proceedings of the 9th international conference on Software Engineering*, ser. ICSE '87. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 2–13.

[7] L. J. Osterweil, "Software processes are software too, revisited: an invited talk on the most influential paper of icse 9," in *Proceedings of the 19th international conference on Software engineering*, ser. ICSE '97. New York, NY, USA: ACM, 1997, pp. 540–548.

[8] A. Wise, "Little-JIL 1.5 language report," Department of Computer Science, U. of Massachusetts, Amherst, Tech. Rep. (UM-CS-2006-51), 2006.

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[10] A. Cass, A. Lerner, E. McCall, L. Osterweil, J. Sutton, S.M., and A. Wise, "Little-JIL/Juliette: a process definition language and interpreter," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 754 –757.

[11] L. Osterweil, L. Clarke, A. Ellison, E. Boose, R. Podorozhny, and A. Wise, "Clear and precise specification of ecological data management processes and dataset provenance," *Automation Science and Engineering, IEEE Transactions on*, vol. 7, no. 1, pp. 189 –195, jan. 2010.

[12] B. Lerner, E. Boose, L. J. Osterweil, A. Ellison, and L. Clarke, "Provenance and quality control in sensor networks," in *Proceedings of the Environmental Information Management Conference*, ser. EIM 2011, Santa Barbara, CA, USA, 2011.

[13] K. Willadsen. Meld. [Online]. Available: http://meldmerge.org/

[14] B. A. Kitchenham, "Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 11–14, January 1996.

[15] E. Mealy and P. Strooper, "Evaluating software refactoring tool support," in *Software Engineering Conference, 2006. Australian*, april 2006, p. 10 pp.

[16] D. Megert *et al.* Eclipse Java Development Tools. [Online]. Available: http://www.eclipse.org/jdt/overview.php

[17] Embarcadero Technologies. Jbuilder. [Online]. Available: http://www.embarcadero.com/products/jbuilder

[18] JetBrains. IntelliJ IDEA. [Online]. Available: http://www.jetbrains.com/idea/

[19] RefactorIT. [Online]. Available: http://sourceforge.net/projects/refactorit/

[20] I. Moore, "Automatic inheritance hierarchy restructuring and method refactoring," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '96. New York, NY, USA: ACM, 1996, pp. 235–250.

[21] I. Moore. Condenser. [Online]. Available: http://condenser.sourceforge.net/

[22] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. T. Vo, "Managing the evolution of dataflows with vistrails," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, 2006, p. 71.

[23] S. P. Callahan, J. Freire, C. E. Scheidegger, C. T. Silva, H. T. Vo, and V. Inc, "Towards process provenance for existing applications," in *Proceedings of IPAW*, 2008, pp. 120–127.

[24] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the kepler scientific workflow system," in *Proceedings of IPAW*, 2006, pp. 118–132.

[25] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, pp. 31–36, September 2005.