Programming Rework in Software Processes

Aaron G. Cass
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
acass@cs.umass.edu

Leon J. Osterweil
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
ljo@cs.umass.edu

ABSTRACT

Our long-term research in process programming is based on the hypothesis that software processes can and should be captured accurately and formally, using executable formalisms to support execution, analysis, and understanding. Many process languages have been developed over the years for modeling processes formally. In this paper, we argue that for automated support, we need not a process *modeling* language, but a process *programming* language – a language with semantics sufficient to support the *execution* of process definitions. Specifically, we argue that invocation semantics are required for accurately describing real-world development processes.

We use the common phenomenon of rework as an example of a process feature which requires invocation semantics for adequate specification. In this paper, we argue that rework can only be accurately described using invocation semantics borrowed from general purpose programming languages. We argue this in the general case and demonstrate a specific case via an example using Little-JIL, our hierarchical process programming language.

1. INTRODUCTION

Our long-term research in process programming [10] is based on the hypothesis that software processes can and should be captured accurately and formally, using executable formalisms to support execution, analysis, and understanding. In this paper, we demonstrate that these formalizations of processes benefit from semantic features from general purpose programming languages. In particular, we demonstrate the use of invocation semantics for clearly describing software processes with rework.

Software development is the controlled construction of many intricately interconnected artifacts (e.g. requirements, designs, test cases, and code) that together describe a problem to be solved and its software solution. At any particular point during development, a developer will be working on a particular artifact, and might discover a need to revisit previously developed artifacts, and thus re-perform some activities associated with those artifacts. For example, at design time, we might discover missing or ill-defined requirements and need therefore to "go back" to revisit and revise requirements artifacts. This activity is commonly referred to as "rework", a term that is used widely and loosely, but seems to us to be generally misunderstood. In this paper we demonstrate how rigorous process formalisms can be used to provide deeper understanding of this, and similar, software engineering terms. These deeper understandings can also lead to more effective support for software development.

Because of the key role that rework plays in real-world software development processes, any software process model should accurately describe rework. However, rework seems to be a challenge to model in available work-flow and process formalisms. Such shortcomings are unfortunate, not just because they fail to elucidate, but also because they fail to provide the basis for effective automated support.

It is our contention that, during rework, developers do not simply "go back" to earlier development phases. For example, in a design activity, if a requirements artifact must be revised, the engineers do not abandon the design phase and go back to plunge into the midst of the requirements phase in order to re-do requirements. Rather, rework is *triggered* by some event(s) during design with the result that some activities that had initially been carried out during the requirements phase, are now reexecuted, but now in the different *context* of the design phase. In this new context, the artifacts on which the activity is performed may be different, may come from a different activity, and may be output to a different activity. The activity may also consume different resources and errors may be handled differently in this new context. The essence of the concept of rework is the provision of a new context within which previous activities are re-performed. This understanding leads us to conclude that, to provide effective automated support for software development, context must be allowed to vary for different executions of software development activities.

Our hypothesis is that by integrating specification of triggers and contexts, we can accurately describe rework, thereby

enabling the accurate definition of real-world software processes. This will thus allow execution, analysis, and understanding of those processes as they are actually carried out. We argue that in order to specify process activities that behave differently in different contexts, the equivalent of procedure invocation borrowed from general purpose programming languages is required. Therefore, invocation semantics are required to define and provide automated support for real software development processes.

In this paper, we describe our approach to define rework accurately and completely, using process definition languages incorporating invocation semantics and adequate concepts of scoping. To make these points clear, we give an example which employs the reinstantiation and scoping mechanisms of Little-JIL [16], our hierarchical process programming language.

2. RELATED WORK

Others have argued the need for rework in general in software development processes. Rework is assumed in many of the popular software development lifecycles. Some of these have described the life-cycle as a nominal flow with rework activities that cause cycles in the nominal flow, while others show development as a continuous cycle of development and rework [1, 8]. In much of this work, however, the processes are not modeled precisely enough to capture the commonly understood concept of rework. As a consequence, attempts to provide automated support for software development based upon these definitions fail to support rework, as current practitioners know it.

Still others have argued that software development is inherently opportunistic [12] and therefore provide automated support for development without fully modeling the development process in which this support is provided. Specific kinds of rework have also been studied (for example, refactoring [6]). 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.

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 special-purpose process languages.

For example, there are a large number of workflow modeling languages based mostly on data flow diagrams (DFDs) (WIDE [3] is an example from this class of system). A key deficiency of such languages is their lack of scoping semantics. While DFD languages often support decomposition, invocation in differing scopes is not supported. Thus, an activity in a DFD may be represented by a box, which can then be elaborated into an entire sub-DFD that is used to provide elaborative detail. Certainly the elaborative diagram is defined to be within the scope of the box that is its parent. But most DFD languages require that such elaborations be defined only once, and always as the elaboration of the single parent. Incorporation of the elaboration as a sub-activity of another parent is illegal within the simple semantics of such languages. But it is precisely this incorporation of a single structure of sub-activities within various contexts that is the essence of real rework.

Kellner [9] uses Statecharts [7] to model processes to support timing simulations. The modeled processes include rework by separate states at which modification occurs. Using Statecharts, transitions to these states are easy to define. However, the states are only defined in one scope and the outgoing transitions are fixed. Therefore, the models cannot show how developers can change to the rework context, and then resume the context from which they came upon completion of rework.

We note that the semantics being described here are inherent in the classical notion of procedure invocation, but are absent from most modeling formalisms (e.g. data flow diagrams). In this paper, we argue that invocation semantics are required for accurately describing real-world development processes which, by nature, must include rework. Some process languages have been developed that have invocation semantics, as they are based on general purpose programming languages[13, 14]. However, these often do not have the right abstractions to support effective process description.

HFSP (Hierarchical and Functional Software Process) [15], a functional process programming language, supports rework directly with a redo clause which can be used in an HFSP program to indicate reinstantiation of a step with different parameters. However, instead of reinstantiating the rework activity in the new context, thus preserving the history of the previous executions of the activity, the redo is a rewriting of the enaction tree as if the original activity was executed originally with the new input.

3. OUR APPROACH

Our approach is to represent rework as a reinstantiation (or reinvocation) of a previously instantiated step in a (potentially different) context. A rework context is much like a scope in a general purpose programming language – the behavior of activities can be modified by changing the scope from which they obtain certain information. In a procedure invocation, the statements in the procedure might exhibit different behaviors based on the parameters passed to the procedure. This allows the reuse of the procedure to obtain different behavior. Additionally, the computation is decoupled from where the results are needed – the procedure does the computation, but multiple callers can make use of it.

As with general purpose programming languages, the design of a process programming language affects the kinds of activities that can readily be described and automated. We propose that a process programming language with invocation semantics that allow the parameterization of a process step in different contexts can be used to accurately describe rework and therefore provide the basis for automated support.

Consider a portion of a phased software development pro-

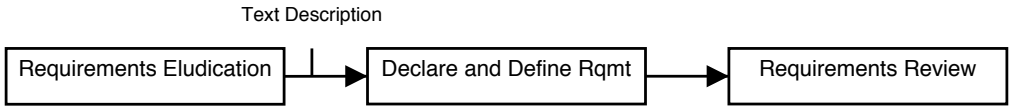

Figure 1: Example Requirement Phase DFD

cess. 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-requirements 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 with no corresponding requirements elements, indicating that there are missing requirements. At this point, the developers should engage in a rework activity. Without leaving the design phase, some requirements specification activities must be performed.

Figure 1 shows a data flow diagram (DFD) for an iteration of requirements activities. The DFD shows a Requirements Elucidation step followed by a Declare and Define Rqmt step, which takes a text description of what requirement to develop. When the requirement is done, a review is conducted. All of this is within the scope of the requirements phase. However, if we later discover during the design phase that the requirements must be re-visited, some of the activities will have to be reexecuted in new scopes. For example, we might find in a design review that a new requirement is needed. This need can be addressed by reexecuting Declare and Define Rqmt, giving it a text description from a design step (instead of the requirements elucidation step).

This rework is difficult, if at all possible, to describe accurately using traditional DFD semantics. If a step in viewed as a procedure, in a DFD the procedure definition is not separable from the invocation – the step is only performed in a single context. It might seem that adding a data-flow edge from the design step to Declare and Define Rqmt with new parameters would be adequate. However, in addition to having the inputs come from the design step, we need the outputs to go to the design step. In other words, we need to *resume* where we left off after reworking the requirements. The traditional DFD semantics do not support this resumption without copying the rework step statically into the new context. Copying is particularly troublesome and error-prone if the rework step has sub-steps.

Even though DFD semantics don't support rework very well, rework is very easy to describe using invocation semantics from general purpose programming languages. Procedures can be defined that can later be invoked from multiple different scopes. The procedures can therefore be parameterized differently for different invocations. A process programming language with invocation semantics similar to those in general purpose programming languages can thus support the following kinds of desired parameterizations:

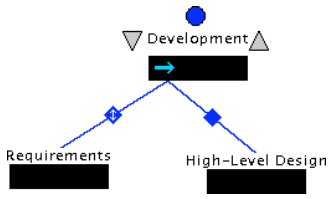

Figure 2: A phased software development process

- **Parameter flow.** Parameter flow consists not just of the values of the parameters, but also the sources and destinations of those parameters. In a procedure invocation, the caller passes parameters and gets the results.
- **Exception handling.** A process step, much like a procedure, might throw an exception if some portion of the activities cannot be performed correctly. In a programming language like Ada or Java, the exception handling is scoped by placing the procedure invocation within a block with an attached exception handler. Therefore, to change the way that exceptions from a procedure are handled, we can simply scope the different invocations with different containing blocks. This is useful for processes as well. For example, in the requirements phase we might wish to respond to a failed requirements review by redeveloping the offending requirement and then continuing with the rest of requirements. If we are re-reviewing the requirements as part of reworking requirements in the context of design, we might instead consider a more complicated process involving changes in design as well as requirements.

Therefore, it seems that a general purpose programming language could support rework with invocation semantics. Process languages based on standard programming languages have been proposed [13, 14] that could therefore support rework. We argue, however, that these languages have tended not to have the right abstractions to support effective process definition. Therefore, we propose that a special-purpose process language borrow invocation semantics from general purpose programming languages in order to support rework.

There are, of course, issues to consider when integrating invocation semantics. For example, what parameter passing modes are useful for process programs? Are there restrictions on parameter types (for example, can process steps be passed to other process steps)?

4. EXPERIENCE

To investigate the approach, we have applied it, using our hierarchically scoped process programming language, Little-JIL [16, 17], to develop software process programs including

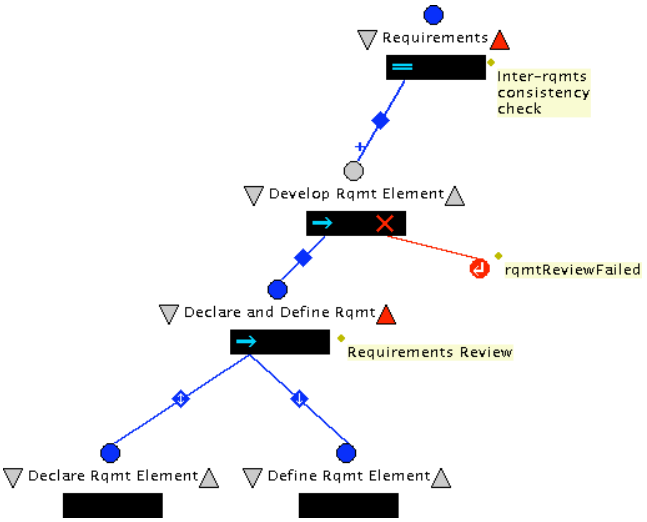

Figure 3: Requirements activities

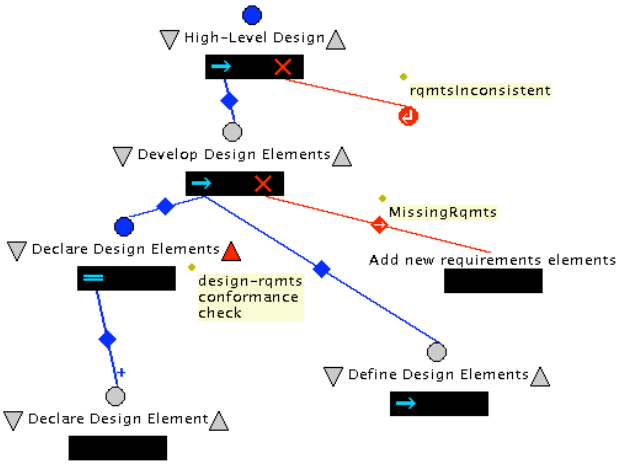

Figure 4: High-Level Design activities

rework. Figure 2 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. Little-JIL is a visual language in which every step is represented by a named black bar. The step interface is represented by annotations on the filled circle above the name of the step. The interface represents the view of the step as seen from the parent step. Little-JIL also allows steps to be referenced from other parts of the program. A reference represents a reinstantiation and invocation in the new scope, and is key to describing rework. Little-JIL passes parameters as value, result, or value-result.

In Little-JIL, we have tried to give first-class status to those abstractions that help describe real-world processes. Because of this, the aspects of rework contexts described above are easily captured in a Little-JIL program. In this section, we describe how scopes define parameter flow and exception handling for a reinstantiated step, using a Little-JIL elaboration of the process described above.

In Figure 2, the right-arrow in the root step indicates that the substeps are to be executed left-to-right, starting with Requirements and continuing next with High-Level Design. Figures 3[1] and 4 show elaborations of the Requirements and High-Level Design steps, respectively.

The triangle to the right of Declare and Define Rqmt in Figure 3 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 can be thrown.

[1]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 information in one view. The comments are not meant to indicate that the program is incomplete or imprecise.

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 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 4, 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 without associated requirements, we can throw a MissingRqmts exception. In this context, this is handled by the exception handler called Add new requirements elements, which is elaborated in Figure 5.

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 3. This reference to Declare and Define Rqmt is an example of rework.

4.1 Varying Parameter Flow

As shown in Figure 5, the Declare and Define Rqmt step is used within the scope of Add new requirements elements. This means that the design concept input to Declare and Define Rqmt is passed from Add new requirements elements. This allows varying of parameter values and additionally allows the source and destination of to vary in much the same way as a procedure call in a general purpose programming language.

4.2 Varying Exception Handling

Little-JIL's exception handling mechanism is slightly more complex than parameter flow, but especially important in supporting real-world processes. There are four aspects of exception handling that must be specified in a Little-JIL process program:

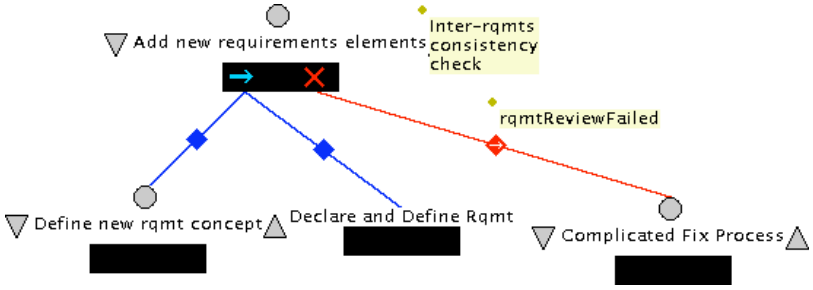

Figure 5: Rework in the context of design

1. Which exceptions a step can throw
2. At which scope they will be handled
3. What steps will execute to handle them
4. How will the process continue after exception handling

Recall that Declare and Define Rqmt has a Requirements Review post-requisite that can throw a rqmtReviewFailed exception. In the Requirements phase, the exception is handled at the scope defined by Develop Rqmt Element, no step is executed to handle the exception, and the process continues by redefining the requirements element (indicated by the angled arrow). However, in the design phase rework context, we can define a different exception handling rule. As shown in Figure 5, we have defined a Complicated Fix Process handler for the rqmtReviewFailed exception. Instead of simply re-doing the requirements element, Complicated Fix Process, which could be further elaborated, might involve changing the design elements related to this requirement element. The continuation after this handling can also be different in this new context – in this case, we continue the Add new requirements elements step (as indicated by the right arrow on the edge). Thus, we can change the scope at which the exception is handled, what steps handle the exception, and how the process continues.

This example has shown that rework can be described with invocation semantics that allow parameterization of data flow and exception handling. This allows the development to proceed as a phased development process, getting the benefits of those phases, while still allowing activities to be reexecuted as needed. Furthermore, because Little-JIL is executable, rework specified this way can actually be executed according to a rigorous semantics, thus providing automated support.

4.3 Additional semantics

In addition to parameter flow and exception handling borrowed from general purpose programming languages, Little-JIL allows the specification of resource management [11] – managers of processes are very concerned about resource allocation and utilization, so resource management has first-class status is Little-JIL. Figure 6 shows a step Declare and Define Rqmt with a resource specification which indicates that the agent for the step must be a Rqmt Engr[2]. At run-time, the

[2]In this example, we give only the 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.

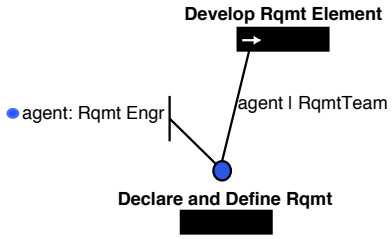

Figure 6: Resource Specification

specification is interpreted by a resource management component 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.

When we use step Declare and Define Rqmt in the design phase as a rework activity, 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 due to staffing changes. Second, Little-JIL allows specification of a narrowing of the space of resources to be considered for a particular acquisition. The specification on the edge in Figure 6 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).

We should note that, even though resource allocation is not borrowed directly from general purpose programming languages, it is analogous to shared-memory approaches like tuple-spaces in Linda [2]. A Linda function can use the shared tuple space by retrieving a tuple from the tuple space by specifying a template. The current state of the shared memory will determine what tuple is retrieved, just as the state of the resource model can affect the acquisition of resources in Little-JIL. Also, the template could be derived from input parameters to the function, so the function's use of the shared space can be modified by changing input parameters, much like a resource acquisition is affected by the

narrowing specification on the incoming edge to a Little-JIL step. One major difference, however, between a Little-JIL step with a resource allocation specification and a Linda function that uses a shared tuple space is that the use of the resource manager is shown directly in the interface to the Little-JIL step – resource management is a first-class part of the interface.

It is our experience that affording resource management first-class status in this way allows us to more accurately define rework by allowing an important parameterization of behavior in the rework context. This, combined with the parameter flow and exception handling parameterizations that we accomplish by step invocation, has enabled us to effectively write process programs for software development processes that seem to us to be significantly more faithful to conventional notions of rework.

These process programs can then be executed using Juliette [4], 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.

5. FUTURE WORK

Formalization of rework as reinstantiation in context has not only allowed a description of more realistic processes, but has also helped to focus our language design effort on giving first-class status to those abstractions that support even more realistic process specifications, thus allowing them to act as parameters in a reinstantiation of a step. We are currently investigating ways to add a data sharing mechanism for steps and it will be represented in the interface of a step and its behavior will be subject in part to the context in which the step is used.

This paper has focused on the invocation of rework activities based on design and requirements review triggers. In order to provide even more accurate representation of software development decision making, we are working of formalizing the triggers as constraints on and between artifacts [5].

Other future work will be focused on more reactive process descriptions. In this paper, we have shown rework triggered by post-requisites to steps. This proactive control is useful for describing rework caused by planned reviews, but it does not support more opportunistic triggers – for example, a developer discovers alone while thinking about a project that some artifact previously developed must be reworked. We would argue that these opportunistic reworkings should be supported in process programs and still require contexts that can be parameterized. However, a reactive mechanism for the trigger is needed.

Additionally, because invocation semantics have proved so useful for process descriptions, we continue to explore the need for other semantics that are common to programming languages (e.g. various forms of iteration), but generally absent from modeling languages like data flow diagrams. Our work is driven by the hypothesis that processes can be programmed, not just modeled, and that automated support requires executable semantics. Based on the success of this work, we expect to find other semantics from programming languages useful in describing a full range of software development processes.

6. CONCLUSIONS

Our experience writing several process programs with this approach has shown that important parameterizations of context must be described in order to accurately describe real-world software processes. In order to allow these parameterizations to affect the behavior of activities in different contexts, we need an equivalent to procedure invocation. Thus, our work has shown that the concepts of scope, abstraction, and invocation which have long been staples of general purpose programming languages are also useful in process programming languages for capturing the semantics of real-world processes.

By integrating specification of rework triggers and parameterization of contexts into executable process programs, we have provided a framework to support more realistic rework in processes. Therefore, using semantics borrowed from general purpose programming languages along with some that are unique to process programming, we can support the execution, analysis, and understanding of real-world software development processes.

Acknowledgements

We would like to thank Alexander Wise for many fruitful conversations and for his efforts both supporting Little-JIL and writing process programs that use the idioms developed in this work. We also thank Heather M. Conboy for her suggestions for improving this paper.

This research was supported in part by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032 and by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231. 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 U.S. Dept. of Defense, the Air Force Research Laboratory/IFTD, the Defense Advanced Research Projects Agency, the U. S. Army, or the U.S. Government.

7. REFERENCES

[1] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[2] N. Carriero and D. Gelernter. *How to Write Parallel Programs A First Course*. MIT Press, 1990.

[3] 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.

[4] 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.

[5] A. G. Cass and L. J. Osterweil. Design guidance through the controlled application of constraints. In *Proc. of the Tenth Int. Workshop on Soft. Specification and Design*, Nov. 5–7, 2000. San Diego, CA.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] 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.

[8] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[9] 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.

[10] L. J. Osterweil. Software processes are software, too. In *Proc. of the Ninth Int. Conf. on Soft. Eng.*, Mar. 1987. Monterey, CA.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci., Apr. 1998.

[17] 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.