Flexible Process Enactment Using Low-Fidelity Models

John Noll
Computer Engineering Department
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053-0566
email: jnoll@cse.scu.edu

ABSTRACT
Attempts to extend process enactment to support dynamic, knowledge intensive activities have not been as successful as workflow for routine business processes. In part this is due to the dynamic nature of knowledge-intensive work: the tasks performed change continually in response to the knowledge developed by those tasks. Also, knowledge work involves significant informal communications, which are difficult to capture.

This paper proposes an approach to supporting knowledge-intensive processes that embraces these difficulties; rather than attempting to capture every nuance of individual activities, we seek to facilitate communication and coordination among knowledge workers to disseminate knowledge and process expertise throughout the organization.

KEY WORDS
Workflow Modeling, Cooperative Work Support

1 Introduction

The conventional workflow approach employs a server or execution engine that executes workflow specifications and stores documents produced by the workflows. Process participants (actors) interact with the engine through web browsers, environments, or task-specific tools, receiving guidance on what activities to perform, and how to perform them.

This approach has been successful for automating repetitive, routine processes, but attempts to extend it to support dynamic, knowledge intensive activities have not been as successful [1].

In part, this is due to the dynamic nature of such activities: actors in knowledge-intensive environments continually adapt their activities to reflect increasing understanding of the problem at hand, which results from performing the knowledge-intensive activities. Thus, the performance or enactment of knowledge-intensive work processes involves a cycle of planning, action, review, and refinement. This presents several problems for process management:

1. The activities performed in any cycle are difficult to describe in sufficient detail to be useful for conventional workflow enactment;

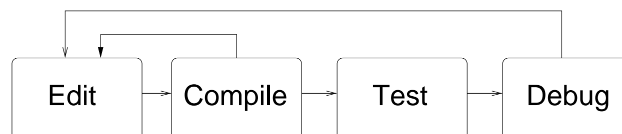

Figure 1. Edit-compile-debug process.

2. Experts perform these activities in a fluid, almost unconscious manner, rather than as discrete steps;

3. The cycle is repeated rapidly and continuously, so the set of activities evolves rapidly; therefore, any description of the process is immediately out of date.

Therefore, we propose an approach targeted to encouraging development of process expertise among knowledge workers. In this approach, actors are given high-level guidance about what activities to perform, and how to perform them, through the use of *low-fidelity* process models. These models specify a nominal order of tasks, but leave actors free to carry out their activities as their expertise and the situation dictates.

The approach comprises three key components:

1. Process specifications based on the notion of *low fidelity* process models;

2. A distributed process deployment and execution mechanism for enacting low fidelity process models;

3. A *Virtual Repository* of artifacts providing access to distributed collections, repositories, and databases of information objects related to the work to be performed;

The following sections discuss, in turn, the modeling approach based on low-fidelity models; coordination among concurrent processes; enactment based on low-fidelity models; related work; and, some conclusions.

2 Process Modeling

Our previous work with process modeling demonstrates the value of *low-fidelity* models for documenting and analyzing knowledge-intensive work [2, 3]. A low-fidelity

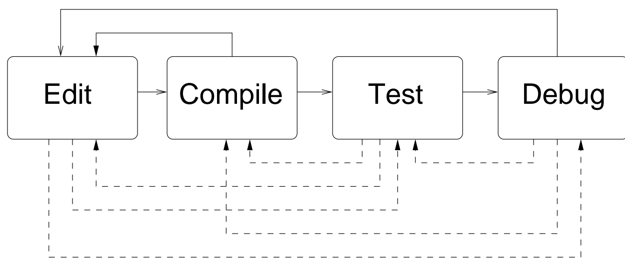

Figure 2. Edit-compile-debug process, augmented.

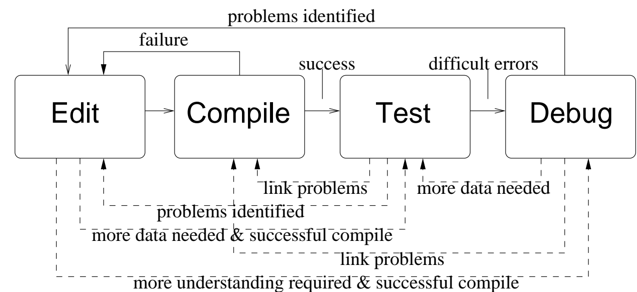

Figure 3. Edit-compile-debug process, augmented.

model does not seek to capture every detail and nuance of a knowledge-intensive process; rather, it documents the major activities of a process and the primary sequence in which they are performed.

An example of a model depicting software development is shown in Figure 1. This model shows the nominal sequence of activities involved in turning a design into working code: the programmer enters source code text using an editor, then compiles the code, iterating over these steps until the code compiles successfully. Then, he or she proceeds to test and debug, iterating over the whole sequence to fix the failures uncovered during testing.

This model captures both the important activities in code development, and the main sequence, and is thus useful for discussing the programming process. But it does not begin to capture all of the possible transitions between activities. Many experienced programmers switch frequently between debugging and editing, delaying the compilation step until several faults have been fixed. Occasionally, it is necessary to iterate over the compile-test-debug cycle; sometimes, a programmer will skip debugging and proceed directly back to editing. Figure 2 shows these additional transitions, represented by dashed edges.

While this depiction is more complete, in that it represents all of the plausible transitions between tasks, it is not entirely accurate. For example, although the graph shows a transition from "Edit" to "Debug", it is not possible to take this transition until the "Compile" step has been successfully completed at least once: "Debug" requires an executable program, which is the output of the "Compile" step. Figure 3 shows these conditions as labels on the edges.

However, it's not clear that the model depicted in Figure 3 is more useful than that in Figure 1; as a guidance tool, a novice programmer might find the numerous transitions confusing, while an expert would already know that these additional transitions are possible.

Our modeling approach is based on the notion of *low-fidelity* process models. A low-fidelity model seeks to capture the essence of a process, while abstracting away as many details as possible. The modeling language allows the modeler to capture both the nominal control flow (the solid edges in Figure 1), and the conditions that constrain transitions outside the nominal flow (the dashed lines in Figures 2 and 3).

Figure 4 shows a specification of the process of Figure 1, written in the PML process modeling language. The nominal control flow is represented explicitly by the *iteration* constructs and the ordering of *actions* in the specifications. The constraints on other transitions are expressed by the *provides* and *requires* statements. These are predicates that express the inputs and outputs of each step (*action*) in the process, and thus the pre- and post-conditions that exist at each step in the process. Note that this simple specification captures the constraint that testing and debugging cannot proceed until compilation is successful: the "Test" and "Debug" actions *require* a resource called "exec", which is produced (*provided*) by the "Compile" action. Thus, until this action succeeds, "Test" and "Debug" are not possible.

Modeling processes using low-fidelity models yields several benefits:

- Low-fidelity models are easy to specify, and can be generated rapidly.
- A low-fidelity model still captures the essential facets of a process, especially the resources consumed and artifacts produced by a given set of activities.
- Because they seek to represent only high-level detail, low-fidelity models are relatively stable; that is, they continue to be accurate descriptions of the high-level process, even as the details of process activities evolve in response to knowledge and experience gained with the problem.

2.1 Modeling Coordinated Activities

The development process of Figure 1 does not exist in isolation. The input to the process is a set of requirements, and the output is debugged source code. Other processes produce and consume these resources, as depicted in Figure 5. This figure shows two cooperating processes: the development process of Figure 1, and a parallel test development process that ultimately applies a test suite to the debugged code.

These processes cooperate to produce a tested product: both start with requirements to develop their respective

```
process edit-code {
    iteration {
        iteration {
            action Edit {
                requires { design }
                provides { code }
            }
            action Compile {
                requires { code }
                provides { exec }
            }
        }
        action Test {
            requires { exec }
            provides {
                code.status == "unit tested"
            }
        }
        action Debug {
            requires { exec }
        }
    }
}
```

Figure 4. Software Development Process Fragment.

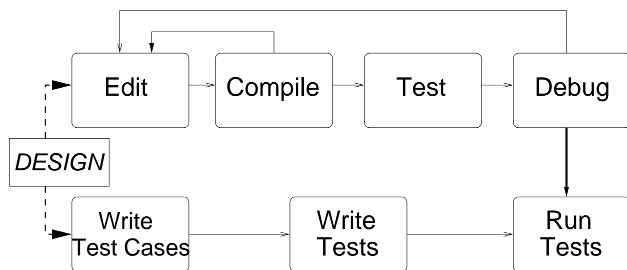

Figure 5. Coordinated processes.

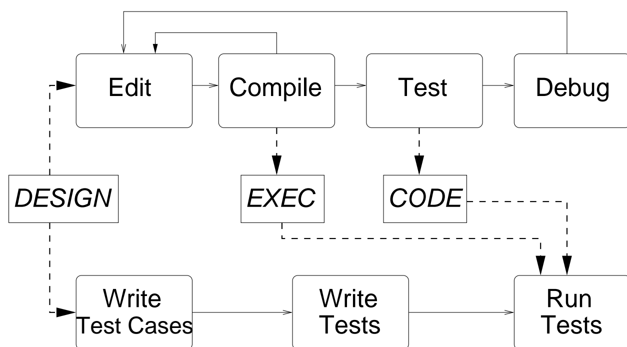

Figure 6. Resource flow between processes.

artifacts. In addition, the test process needs the output of the development process (the executable object) to run the tests. This dependency could be represented by an explicit link between the "Debug" and "Run Tests" actions (represented by the solid edge in Figure 5). But this approach has several difficulties.

First, it creates an explicit connection between the specific development process and the test process that does not always exist: developers could employ any of a number of different development processes (clean-room, test-first, even "chaotic" development) that could provide the object for testing.

Second, it requires both processes to be maintained as a single model, which is often not the case: different organizations are responsible for their respective processes which they develop and maintain independently.

Finally, it doesn't capture the true relationship between the processes: typically, testers require a compiled product to test, so that the actual relationship is between "Compile" and "Run Tests" as opposed to "Debug" and "Run Tests." But the compiled product must also have been debugged and tested. The essential relationship between the two processes is that the development process produces an executable product for the test process to test (Figure 6). It's not important to the test process how this product was developed, only that it exists in a state suitable for testing.

This relationship is easy to represent in PML, as shown in Figure 7. This specification shows that the beginning of the test process depends on the availability of the design document ("DESIGN"). More important, the "Run Tests" action cannot begin until the three conditions are met: the "Write Tests" action must complete, there must be an executable to run, and the code must be unit tested.

Note that because the processes are indirectly coupled, it is not necessary for all activity to be modeled, or enacted; enacted processes can be coordinated through a shared resource with ad-hoc work or activities in another organization. Thus, the "Run Tests" task can begin as soon as the "exec" and "code" resources are available; but these resources can be produced by any process, including a completely spontaneous ad-hoc process.

3 Flexible Enactment

Enactment is driven by events, which are classified as either *process* events or *resource* events. These are summarized in Table 1. A predicate in the process description specifies the state that the required resources must satisfy before the activity can proceed (the mechanism is described fully in [4]).

Process events signal action initiation and completion, and are generated directly by actors as a consequence of performing tasks. Resource events reflect changes in the environment, such as creation, deletion, and modification of resources, and time events such as deadlines or alarms.

The enactment mechanism enables flexible enactment through the way it handles process and resource events.

```
process test-code {
    action WriteTestPlan {
        requires { design }
        provides { test_plan }
    }
    action WriteTests {
        requires { test_plan }
        provides { test_suite }
    }
    action RunTests {
        requires { test_suite && exec
        && code.status == "unit tested" }
        provides { code.status == "tested" }
    }
}
```

Figure 7. Coordinated Test Process.

Process Events	
Create process	An actor requests instantiation of a new process instance.
Task Start	An actor has begun a task.
Task Suspend	The actor has suspended an active task.
Task Complete	The actor has completed a task.
Task Abort	The actor aborts a task that can't be completed.
Resource Events	
Object Creation	A new resource has been created (or detected).
Object Modification	An existing resource has been changed.
Object Deletion	An existing resource has been destroyed or removed.
Deadline	A time event (deadline, milestone, alarm) has passed.

Table 1. Process and Resource Events

Events affect process state in several ways:

1. Activation of the next task in the nominal control flow. The process description specifies the order in which tasks should be performed. A process event can trigger the transition from one task to the next; for example, the completion of the Edit task in Figure 1 causes transition to the Compile task.

2. Activation of other actions in the process. In Figure 4, completion of the "Compile" task *may* cause "Test" and "Debug" to become ready, if the "exec" object is successfully created.

3. Activation of actions in other processes. For example, when both the "Compile" and "Test" tasks of Figure 4 are successfully completed, resulting in products in the correct state, the "Run Tests" task in Figure 7 becomes ready.

The completion of an action can make a number of additional actions available for the actor to perform: a process event signals the completion of an action, which makes the next action in the nominal control flow available. In addition, the completion of an action may include the creation or modification of one or more resources as side-effects, generating resource events that make additional actions available. The enactment engine can then recommend that the next available action in the nominal control flow should be performed, and also show other actions available as the result of resource events.

As an example, suppose an actor has just completed the "Compile" action in Figure 6. This process event causes the "Test" action, which is the next action in the nominal control flow, to become available. The "Compile" action also creates an executable object ("EXEC" in Figure 6), a resource event that satisfies the "Debug" action's *requires* predicate (Figure 4). As a consequence, the "Debug" action also becomes available. Finally, the creation of the executable object satisfies part of the "Run Tests" action's *requires* predicate. This may cause "Run Tests" to become available as well. The result is several actions ready for the actor to perform, representing different process performance paths.

The enactment mechanism is depicted in Figure 8. The *User Interface* allows actors to generate process events, and invoke Tools to perform tasks. The *Process Engine* handles Process and Resource events, responding as described above. It includes a virtual machine to interpret PML process specifications to compute the state of actions in response to a process or resource event. The *Virtual Repository* is responsible for detecting resource events, and translating them into a representation-independent form suitable for interpretation by the *Process Engine*.

The Virtual Repository provides a logically centralized view of a set of distributed, heterogeneous information repositories. This enables the enactment mechanism to treat a variety of resources, such as email messages, Web

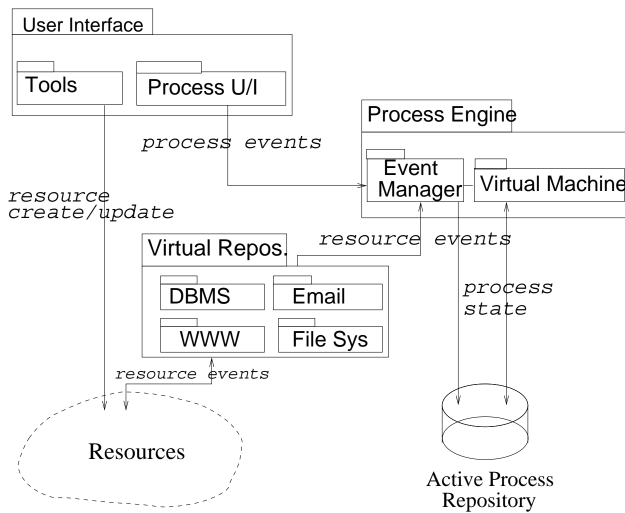

Figure 8. Process Enactment Architecture

pages, documents, etc. as information objects with a uniform format and access interface.

4 Related Work

There are three main approaches to enacting dynamic processes in a flexible manner.

The first treats deviations from the specified process as exceptions. This view supposes that there is a usual or "right" way to do things, from which deviation is occasionally required. For example, Milano augments enactment of Petri-net based models with the ability to jump forward or backward across several transitions in order to handle exceptions to the specified flow of control [5]. The philosophy of Milano is similar to the PML approach: use simple models to specify processes, and handle variation at runtime. However, Milano treats deviations as exceptions, rather than viewing them as alternate but normal variations as PML does.

The second approach models the process as a set of constraints; as long as the constraints are met, actors are free to perform tasks as they see fit, in the order that seems most appropriate. For example, Glance and colleagues propose a constraint specification language that can be used to specify the goals of a process, without specifying the order [6].

This approach provides a great degree of flexibility: as long as the goals are met, the actor is free to do any task in any order. However, the flexibility comes at the cost of guidance: it becomes difficult to advise the novice actor as to what step to perform at a given point in time. Dourish addresses this issue by adding constraints about the order in which tasks must be performed, if such an order is necessary [7].

Related to this approach are process-centered programming environments that employ rules to specify processes; examples include Merlin [8], which is based on Prolog, and Marvel [9].

The SPELL modeling language includes constraints on task pre- and post-conditions that are interpreted by the EPOS execution environment's planning mechanism to develop a sequence of activities to suit a specific situation [10]. While SPELL's constraints are similar to PML's *requires* and *provides* predicates, they are used to develop a specific sequence of tasks at runtime. In contrast, PML's predicates allow the actor to deviate from the nominal sequence when necessary.

The third approach attempts to model the process using specifications that inherently allow great flexibility in how tasks are performed.

For example, Bernstein proposes a hybrid model that combines constraints when processes are not well understood, with stronger control flow based specifications when the process matures [11]. This approach assumes that variation from the specification is a matter of process immaturity that will decrease over time as the process becomes better understood.

Jorgensen argues for enactment based on a dialog with the actor at runtime [12]. In this approach, processes are initially specified at a high level, and enactment takes place as a dialog between the actor and the enactment mechanism. Jorgensen also views variation as a side effect of process immaturity, so the goal of this dialog is gradual refinement of the process specification into a detailed model.

5 Conclusion

The approach described herein has several distinctive features.

First, low-fidelity process models specified using PML are both straightforward and enable flexible enactment.

Second, because the coupling between coordinating actors is indirect, through a shared resource, there is no requirement for trust (or even awareness) between coordinating peers. This enables extremely fluid, dynamic organizations in which participants can join at will without requiring administrative approval or action.

Finally, by relying resource events, process enactment can proceed in the background, out of the way of experienced users until they need explicit advice.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant No. IIS-0205679, through subcontract from the University of California, Irvine; and by an IBM Faculty Research Grant awarded by Santa Clara University. No endorsement is implied.

References

[1] Paul Dourish. Process descriptions as organizational accounting devices: The dual use of workflow technologies. In *Proceedings of the 2001 International ACM SIGROUP Conference on Supporting Group Work*, pages 52–60, Boulder, Colorado, USA, October 2001. ACM Press.

[2] Walt Scacchi and John Noll. Process-driven intranets: Life-cycle support for process engineering. *IEEE Internet Computing*, September/October 1997.

[3] John Noll and Walt Scacchi. Specifying process-oriented hypertext for organizational computing. *Journal of Networking and Computer Applications*, 24(1), January 2001.

[4] John Noll and Bryce Billinger. Modeling coordination as resource flow: An object-based approach. In *Proceedings of the 2002 IASTED Conference on Software Engineering Applications*, Cambridge, Massachusetts, USA, November 2002.

[5] Alessandra Agostini and Giorgio De Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3-4):335–363, August 2000.

[6] Natalie S. Glance, Daniele S. Pagani, and Remo Pareschi. Generalized process structure grammars (GPSG) for flexible representations of work. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, pages 180–189, Boston, Massachusetts, USA, 1996. ACM Press.

[7] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw. Freeflow: Mediating between representation and action in workflow systems. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, pages 190–198, Boston, Massachusetts, USA, 1996. ACM Press.

[8] Burkhard Peuschel and Wilhelm Schäfer. Concepts and implementation of a rule-based process engine. In *Proceedings: 14th International Conference on Software Engineering*, pages 262–279. IEEE Computer Society Press/ACM Press, 1992.

[9] Israel Z. Ben-Shaul, Gail E. Kaiser, and G. Heineman. An architecture for multi-user software development environments. *Computing Systems, the Journal of the USENIX Association*, 6(2):65–103, 1993.

[10] R. Conradi, M. L. Jaccheri, C. Mazzi, M. N. Nguyen, and A. Aarsten. Design, use and implementation of SPELL, a language for software process modelling and evolution. In J.-C. Derniame, editor, *Software Process Technology*, number 635 in Lecture Notes in Computer Science, pages 167–177. Springer-Verlag, 1991.

[11] Abraham Bernstein. How can cooperative work tools support dynamic group process? Bridging the specificity frontier. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, pages 279–288, Philadelphia, Pennsylvania, USA, 2000. ACM Press.

[12] Havard D. Jorgensen. Interaction as a framework for flexible workflow modelling. In *Proceedings of the 2001 International ACM SIGROUP Conference on Supporting Group Work*, pages 32–41, Boulder, Colorado, USA, October 2001. ACM Press.