Using Process Modeling for Process Understanding

Dewayne E. Perry

Systems and Software Research Laboratory, Bell Laboratories, Murray Hill NJ 07901
dep@research.bell-labs.com

Abstract

An important step to improving processes is first understanding them. I report here a case study in process understanding using the process modeling language Interact. I illustrate both the language and the process iteratively, somewhat in the way it might actually be done, and then provide the completed model. I close with lessons to be drawn from this study about understanding and improving processes, the flaws in the process that were exposed as a result of the modeling process and the utility of the modeling formalism used in exposing those flaws.

1. Introduction

An important step to improving processes is to first understand what the current processes are [5] and then use that understanding as the basis for measured process improvement. The advantage of using a process formalism as the means of modeling the current processes is that the formalism encourages you to think about certain aspects of the process (and if it is a good formalism, the most important aspects).

In this paper, I report on an experiment using the process modeling language Interact [3, 4] as the means of understanding the current tool release process for a research group residing in a development organization. The tools are built on top of the UNIX operating system and interact with each other in various ways --- that is, they are either mutually supporting tools that depend on each other for their full functionality or they are tools that can be composed together via scripts to form more sophisticated tools. In either case, it is of paramount importance that the release process guarantee that the tool set plays together properly.

2. Modeling the Process

The tool release process is an informal process used by a set of researchers who build useful software development tools. The primary means of process understanding of this informal process is the process flow diagram showing the major steps using ‘libx’ as an example.

I will describe tool release process using Interact as the modeling language. The various aspects of Interact will be introduced as we explore various parts of the modeled process. In a sense this section will be a dual tutorial on both the modeled process and the modeling language.

2.1 Overview of Interact/Intermediate

The general goals in Interact and Intermediate are to support goal-directed process modeling in such a way to maximize concurrency of activities and to minimize control of the human element in the process. To do this, I have separated the model specification from the enaction — that is, I have separated the modeling from the support and in doing so have separated the (mostly) static aspects of process modeling from the (mostly) dynamic aspects of model enactment.

Interact provides facilities for defining objects, policies, and activities: object definitions are used to model both the product, the project, and the organization; policy definitions are used to model various facts and relationships about both the state of the product, the project and the organization as well as to define synchronization, interaction and cooperation abstractions; activity definitions are used to model the process activities that transform the product and project from one state to another. Activities are defined in terms of the activating policies, the defined goals and resulting obligations. Where desired, the process designer may bind the user to a particular implementation of the activity by supplying some structure to what is normally considered a primitive entity.

Object declarations include type definitions, type instances, and object definitions. Types and type instances enable the process designer to define the appropriate abstractions that are necessary for the model and to define the values for those abstractions. Objects have types and may assume the values

defined for those types. For example, the model of a software artifact serves as the coordinating object for the various activities that transform the product from one state to another; the model of the project defines the objects by which non-product related communications take place.

Policies define various facts and relationships among objects in several different ways. First, policies may be primitive. These serve as base abstractions which are asserted as results of activities. Second, policies may encapsulate logical expressions that relate (in various ways) base abstractions, facts about the state of the product or facts about the state of the project. Finally, there may be multiple policy definitions, any number of which may be active at any time. These various definitions serve as one of the primary means of customization and evolution of the model for particular instantiations.

Activities then define the basic work that needs to be done in terms of what must be true before an activity can be executed (that is, the assumed policies -- the preconditions -- that must be satisfied), of what is guaranteed to be true as a result of executing the process, together with associated obligations placed on the executor as a result of executing that activity (the postconditions and obligations — again specified in terms of policies), and of what the actual work is that must be done if the implementation is not defined to be primitive (that is, unspecified and left to the executor).

2.2 The Project and Product Model

A good place to start in doing any modeling activity is that of modeling the project and the product [1] to a level that you think is initially necessary. You may then refine that model is you elaborate the activities and policies.

```
MODEL Release ()
data {
-- Model of the Project --

  type    role        primitive,
  type    group       {person},
  type    roleset     {(person p, role r)},

  value   role        initiator,
  value   role        integrator,
  value   role        submitter,
  value   role        tooluser,
  value   role        builder,
  value   role        shipper,

  value   person      AlexanderWolf,
  value   person      DavidRosenblum,
  value   person      Bala...,
  ...

  object  dept        group,
  object  roles       roleset,
```

Note that we do not define a role except by enumerating it by means of value definitions. A group is modeled as a set of person (person is a primitive concept in Interact). A set of roles is defined as a set of tuples which consist of a person and a role. Thus a person may have several roles and a role may be played by one or more persons. We have also indicated how you would enumerate the values for the primitive type person by listing its values.

We then have to define data objects that we use in the model to contain the actual values of the department (modeled as a group) and the roles persons play in that department. The values of these objects may be specified either at model specification time (exactly as we have illustrated the values for several of the types) or at model instantiation time when the model administrator customizes the model for execution.

```
-- Model of the Tools  --

  type    source      primitive,
  type    toolset     {tool},

  value   tool        libx,
  value   tool        app,
  value   tool        persi,
  value   tool        yeast,
  ...

  object  owner
          {(person p, tool t)} [t],
  object  depends
          {(tool t, toolset dtools)} [t],
  object  testset     toolset,
  object  exportset   toolset
}
```

We model the tools (another basic concept in Interact) in terms of their source code, who owns them (that is, who is responsible for them) and which tools depend on them. Note that in the object definitions for owner and depends there is the notation ‘[t]’ — that means that the set is indexed by by the tool t and hence only own person may be the owner of a tool (since there can only be one element of the set for that tool) and that each tool has only one set of dependent tools. The objects testset and exportset are used for testing the buildability of a tool and the accumulated acceptable tools for export, respectively.

2.3 Activities and Policies in the Model

The easiest place to start the modeling activity once you have a project and product models in place is the main piece of work that the process is supposed to do — in this case, the activity Integrate distributes the submitted tools for approval and collects those that have been approved for release. This will enable us to determine what is needed as support for this activity and what policies need to be defined for it to work properly. We can then incrementally expand the process model from there.

```
Integrate ()
preconditions { cycle-initiated () }
{
  foreach tool t in { tool t | submitted ( p, t ) }
  <
    Determine-Dependencies ( t ),
    let testset' = testset + t,
    Build ( testset', result ),
    ( result == false, reject-tool ( SELF, t ),
    ( result == true,
    <
      < foreach person p
        in { person p | owner[t1] = p and t1 in depends[t] }
          instantiate Evaluate ( t ),
      >,
      Await-Acceptance/Rejection ( t )
    >)
  >
}
results<(postconditions {
            exportset = { tool t | tool-approved ( t ) },
            tools-released ( exportset ) },
          obligations { }
        )>
```

Integrate (see [4] for a more complete discussion of the details of this activity and related matters) requires only that the release cycle have been initiated to begin. Then for each tool submitted, a parallel thread (indicated by the delimiters { } around the foreach statement) of a sequence (indicated by the delimiters < > as the body of the foreach statement) is generated that determines the dependencies on that tool, builds it (and rejects it if it does not build properly), then instantiates an Evaluate activity for each owner of the dependent tools to approve or reject the submitted and built tool, and finally awaits its acceptance or rejection.

We now have a list of things that need to be interpreted and modeled: the policies cycle-initiated and tools-released, and the activities Determine-Dependencies, Build, Evaluate, and Await-Acceptance/Rejection. Following our initial approach of primary work first, let us look at the policy 'tools-released' and what it implies.

```
tools-released (toolset ts) :: primitive
```

For purposes of this model we only wanted to look at what sorts of interactions the various tool builders have to do to get their tools released, not what the mechanisms are that followed that creation of the set of tools for release (that is, the actual mechanics of distributing the tool set — that is a different process). Thus, we define the ultimate policy as a primitive one, one that primarily asserts the fact that the tool set exportset has been accumulated and released for distribution.

The policy 'cycle-initiated', on the other hand implies quite a bit more in terms of both policies and implied actions.

```
cycle-initiated ( ) ::
  < notified ( dept, "Request submission of tools" ) >
```

Initiating the release cycle means that everyone in the department has been notified that they are to submit new versions of their tools. The question then is how does this come about and what does the policy 'notified' mean. First there is the implication that the release cycle has somehow been triggered and that there is some means of notifying everyone in the department. Thus, we need to define these two activities. We look at Initiate first.

```
Initiate ()
preconditions { allow-initiation () }
primitive
results<(postconditions { cycle-initiated () },
          obligations { tools-released ( exportset ) }
        )>
```

To begin this process we must determine what it means to allow initiation of the entire release cycle. There are two means of initiation: either on the established 6 month cycle beginning in either May or November, or whenever it is arbitrarily decided to do a release. We define these two definitions for 'allow-initiation' and either one or both can be in vogue for that process model instantiation (or it may even change over time which one is in vogue).

```
arbitrary-initiation ( ) :: primitive,
allow-initiation ( ) ::
  < arbitrary-initiation (),
    CurrentDate="1 May" or "1 November"
  >
```

Initiate does not specify how to proceed, but entails two things: a postcondition and an obligation. The obligation is that we now need to satisfy the policy 'tools-released (exportset)' and the only way that we can do that (because of the way we define our model) is to execute the activity Integrate.

The primary question is how to satisfy the postcondition of broadcasting the request for submission message? For that we need an activity Notify which does precisely that. The most likely implementation of this activity is an automated agent — for example, email. The process administrator would use Intermediate to bind this activity to a broadcast mailer.

```
Notify ( group g, message m )
preconditions { }
primitive
results<(postconditions { notified ( g, m ) },
          obligations { }
        )>
```

We now must determine what the policy 'notified' means and we define that in terms of a more primitive notification policy, one for individual persons, which in turn is considered to be a basic fact, not an interpreted one.

```
notified (group g, message m) ::
   < foreach person p in g: notified(p, m) >,
notified (person p, message m) :: primitive
```

We still have on our stack of work the basic activities that we need to implement Integrate. Let us sketch them out first and then worry about the associated policies.

```
Determine-dependencies ( tool t )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions { depends[t] = { tool t1 | includes(t1, t) },
          obligations { }
        )>,

Build ( toolset ts, boolean result )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions {
             result == true
             tool-built ( t, ts ),
             tool-dependencies-notified ( t ) },
           obligations { }
        ),
        (  postconditions {
             result == false
             reject-tool ( SELF, t ) },
           obligations { }
        ),
        (  postconditions {
             SELF in { person p | (p, Integrator) in roles },
             notified ( owner[t], "Tool %t did not build" ) },
           obligations { }
        )>,

Evaluate ( tool t )
preconditions { tool-dependencies-notified ( t ) }
primitive
results<(postconditions { approve-tool ( SELF, t ) },
           obligations { }
        ),
        (  postconditions { reject-tool ( SELF, t ) },
           obligations { }
        )>,

Await-Acceptance/Rejection ( tool t )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions { tool-rejected ( t ) },
           obligations { }
        ),
        (  postconditions { tool-approved ( t ) },
           obligations { }
        )>,
```

Build requires that the tool first have been submitted and results in a set of dependencies that have been set in the data object depends for that tool. It is an activity that at first sight looks like it ought to be bound to the appropriate automated agent (ie, tool) for its implementation, and its results bound to one of the three possible results of the activity. However, if we look at the first result, we note that one of the postconditions is that the dependent tool owners be notified that the tool has properly built. This is a case where the results imply what the implementation must be: a use of an actual build tool to see if it indeed builds and then the use of Notify to broadcast the appropriate tool owners.

The remaining two results are those which report the fact that the tool did not build: one for the normal build used in private integration and the other for the case where it is the Integrator who is doing the build. In the former case, it results in rejecting the tool by that dependent tool owner; in the latter case, it results in the Integrator informing the submitter that the tool did not build. Note that in the latter case, the Integrator would produce two results instead of just the one: the second to reject to the tool formally, the third to inform the submitter that it failed to build.

Evaluate requires only that all the tool dependencies have been notified and then results in either approval or rejection. Nothing else is implied about the implementation of the evaluation activity. Note that the precondition is satisfied if the too builds properly (the first result of Build).

Await-Acceptance/Rejection of a tool has two results: either someone has rejected it who depends on it, or every dependent tool owner approves its release. This latter we encapsulate in the policies 'tool-rejected' and 'tool-approved'.

```
tool-approved (tool t) ::
   < foreach person p
```

```
    in { p | owner[t1].p = p and t1 in depends[t].dtools }:
      approve-tool(p, t)
  >,
tool-rejected (tool t) ::
  < forsome person p
    in { p | owner[t1].p = p and t1 in depends[t].dtools }:
      reject-tool(p, t)
  >,
approve-tool (person p, tool t) ::
  < notified({q | (q,Integrator) in Roles}, "%p approves tool %t") >,
reject-tool (person p, tool t) ::
  < notified({q | (q,Integrator) in Roles}, "%p rejects tool %t") >,
tool-dependencies-notified (tool t) ::
  < do = { p | owner[t1].p = p and t1 in depends[t].dtools } and
    notified (do, "Tool %t successfully submitted - Evaluate %t")
  >
```

Note that tool approval and rejection imply that the Notify activity has been executed to inform the Integrator that the tool is either approved or rejected. Similarly, the same is implied to achieve the policy 'tool-dependencies-notified'.

This leaves us with one policy that is the assumption of three of the above activities: 'submitted (p, t)'. We find that we need an activity Submit to supply this policy.

```
Submit ( tool t )
preconditions { submitting ( t ) }
primitive
results<(postconditions { submitted ( SELF, t ) },
          obligations { }
        )>
```

The assumption is that the tool owner is considering submitting the tool for which this activity is the official act. These two policies then imply other activities that must take place. In the case of 'submitting' the activities are either Consider-Submission or Consider-Resubmission (of a tool that had previously been rejected).

```
submitting (tool t) ::
  < notified( {p | (p,Integrator) in Roles}, "Submitting %t") >,

Consider-Submission ( tool t )
preconditions { cycle-initiated () }
primitive
results<(postconditions { submitting ( t ) },
          obligations { submitted ( SELF, t ) }
           ),
        (  postconditions { not submitting ( t ) },
          obligations { }
        )>,

Consider-Resubmission ( tool t )
preconditions { cycle-initiated () }
primitive
results<(postconditions { submitting ( t ) },
          obligations { submitted ( SELF, t ) }
        ),
        (  postconditions { not submitting ( t ) },
          obligations { }
        )>
```

Note that their is an obligation for choosing the first result of each of these activities: that of satisfying the policy 'submitted'. An obligation moves the process along; it obliges us to do another activity to satisfy that obligation. It is like a liveness condition, it guarantees that eventually the executor must do something.

```
shipped-to-test ( person p, tool t ) ::
  < notified({q | (q,Integrator) in roles}, "%p intending to ship %t")
    and testset = testset + t
    and notified({p | (p,Integrator) in roles}, "%t shipped by %p")
  >,
submitted ( person p, tool t) :: < shipped-to-test( p, t) > ,

Ship-to-Test ( tool t )
preconditions { submitting ( SELF, t ) }
<
  Notify({person p | (p,integrator) in roles},
         "%SELF intending to ship %t")
  Copy ( t, testset )
  Notify({person p | (p,integrator) in roles},
         "%t shipped by %SELF")
>
results<(postconditions { shipped-to-test ( SELF, t ) },
          obligations { }
        )>,

Copy ( tool t, toolset ts )
preconditions { }
primitive
results<(postconditions { ts = ts + t },
          obligations { }
        )>
```

Note that we have supplied the implementation of Ship-to-Test. We did this primarily to show that sometimes you will want to constrain the execution of the activities to a particular order: first notify the Integrator of intention, doing it, and then notifying that it was done: < > indicates that the activities are to be done in sequence. The set of activities is implied by the postcondition but not the order in which they are done.

2.4 The Complete Model

```
MODEL Release ()
data {
-- Model of the Project --

  typeroleprimitive,
  typegroup{person},
  typeroleset{(person p, role r)},

  value  roleinitiator,
  value  roleintegrator,
  value  rolesubmitter,
```

```
   value  roletooluser,
   value  rolebuilder,
   value  roleshipper,

   value  personAlexanderWolf,
   value  personDavidRosenblum,
   value  personBala...,
   ...

   object deptgroup,
   object rolesroleset,

-- Model of the Tools  --

   typesourceprimitive,
   typetoolset{tool},

   value  toollibx,
   value  toolapp,
   value  toolpersi,
   value  toolyeast,
   ...

   object owner
      {(person p, tool t)} [t],
   object depends
      {(tool t, toolset dtools)} [t],
   object testsettoolset,
   object exportsettoolset
}

policies {
notified (person p, message m) :: primitive,
notified (group g, message m) ::
   < foreach person p in g: notified(p, m) >,
notified (role r, roleset rs, message m) ::
   < foreach person p in { p | (p, r) in rs }:
      notified ( p , m )
   >,
arbitrary-initiation ( ) :: primitive,
allow-initiation ( ) ::
   < arbitrary-initiation (),
      CurrentDate="1 May" or "1 November"
   >,
cycle-initiated ( ) ::
   < notified ( dept, "Request submission of tools" ) >,
submitting (tool t) ::
   < notified( {p | (p,Integrator) in Roles}, "Submitting %t") >,
shipped-to-test (tool t) ::
   < notified({p | (p,Integrator) in Roles}, "Intending to ship %t")
      and testset = testset + t
      and notified({p | (p,Integrator) in Roles}, "%t shipped")
   >,
submitted (tool t) :: < shipped-to-test(t) >,
includes (tool user, tool used) :: primitive,
tool-built (tool t, toolset ts) :: primitive,
tested (tool t, toolset ts) :: primitive,
tool-dependencies-notified (tool t) ::
   < do = { p | owner[t1].p = p and t1 in depends[t].dtools } and
      notified (do, "Tool %t successfully submitted - Evaluate %t")
   >,
approve-tool (person p, tool t) ::
   < notified({q | (q,Integrator) in Roles}, "%p approves tool %t") >,
reject-tool (person p, tool t) ::
   < notified({q | (q,IntegratoR) in Roles}, "%p rejects tool %t") >,
tool-approved (tool t) ::
   < foreach person p
      in { p | owner[t1].p = p and t1 in depends[t].dtools }:
         approve-tool(p, t)
   >,
tool-rejected (tool t) ::
   < forsome person p
      in { p | owner[t1].p = p and t1 in depends[t].dtools }:
         reject-tool(p, t)
   >,
tools-released (toolset ts) :: primitive
}

activities {

Initiate ()
preconditions { allow-initiation () }
primitive
results<(postconditions { cycle-initiated () },
            obligations { tools-released ( exportset ) }
         )>,

Notify ( group g, message m )
preconditions { }
primitive
results<(postconditions { notified ( g, m ) },
            obligations { }
         )>,

Copy ( tool t, toolset ts )
preconditions { }
primitive
results<(postconditions { ts = ts + t },
            obligations { }
         )>,

Ship-to-Test ( tool t )
preconditions { submitting ( SELF, t ) }
<
   Notify({person p | (p,integrator) in roles},
         "%SELF intending to ship %t")
   Copy ( t, testset )
   Notify({person p | (p,integrator) in roles},
         "%t shipped by %SELF")
>
results<(postconditions { shipped-to-test ( SELF, t ) },
            obligations { }
         )>,

Consider-Submission ( tool t )
preconditions { cycle-initiated () }
primitive
results<(postconditions { submitting ( t ) },
            obligations { submitted ( SELF, t ) }
             ),
         (  postconditions { not submitting ( t ) },
            obligations { }
         )>,

Consider-Resubmission ( tool t )
preconditions { cycle-initiated () }
primitive
results<(postconditions { submitting ( t ) },
            obligations { submitted ( SELF, t ) }
         ),
```

```
        (  postconditions { not submitting ( t ) },
           obligations { }
        )>,

Submit ( tool t )
preconditions { submitting ( t ) }
primitive
results<(postconditions { submitted ( SELF, t ) },
           obligations { }
        )>,

Determine-dependencies ( tool t )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions { depends[t] = { tool t1 | includes(t1, t) },
           obligations { }
        )>,

Build ( toolset ts, boolean result )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions {
              result == true
              tool-built ( t, ts ),
              tool-dependencies-notified ( t ) },
           obligations { }
        ),
        (  postconditions {
              result == false
              reject-tool ( SELF, t ) },
           obligations { }
        ),
        (  postconditions {
              SELF in { person p | (p, Integrator) in roles },
              notified ( owner[t], "Tool %t did not build" ) },
           obligations { }
        )>,

Evaluate ( tool t )
preconditions { tool-dependencies-notified ( t ) }
primitive
results<(postconditions { approve-tool ( SELF, t ) },
           obligations { }
        ),
        (  postconditions { reject-tool ( SELF, t ) },
           obligations { }
        )>,

Test ( tool t, toolset ts )
preconditions { tool-built ( t, ts ) }
primitive
results<(postconditions { tested ( t, ts ) },
           obligations { }
        )>,

Await-Acceptance/Rejection ( tool t )
preconditions { submitted ( p, t ) }
primitive
results<(postconditions { tool-rejected ( t ) },
           obligations { }
        ),
        (  postconditions {
              tool-built ( t, testset ),
              tool-approved ( t ) },
           obligations { }
        )>,

Integrate ()
preconditions { cycle-initiated () }
{
  foreach tool t in { tool t | submitted ( p, t ) }
  <
    Determine-Dependencies ( t ),
    let testset' = testset + t,
    Build ( testset', result ),
    ( result == false, reject-tool ( SELF, t ),
    ( result == true,
    <
      < foreach person p
        in { person p | owner[t1] = p and t1 in depends[t] }
          instantiate Evaluate ( t ),
      >,
      Await-Acceptance/Rejection ( t )
    >)
  >
}
results<(postconditions {
              exportset = { tool t | tool-approved ( t ) },
              tools-released ( exportset ) },
           obligations { }
        )>
}
EndModel Release
```

3. Lessons Learned

There were a number of interesting lessons learned about the process we modeled, about the process of using a process formalism to understand processes, and about the utility of the formalism itself.

3.1 Lesson 1: Understanding and Modeling are Iterative, Interacting and Concurrent Processes

There were two strands running in parallel in this case study: the strand of trying to understand and formalize the tool release process and the strand of understanding the process formalism and how it could be used to model the process. This is often the case during the initial phases of process improvement in trying to define the process as it is and using a new technique for defining the processes and reasoning about the processes.

The process of modeling itself is an iterative one, especially in this case where there are three logical parts to the model: the submodels of both the product and the organization, the policies that are used within the processes, and the activities that carry the bulk of the work. This is compounded by the iterative nature of deepening the understanding the existing processes that is going on concurrently and interacting with the process of modeling the release process.

I have tried in the presentation to illustrate some the kinds of iterations that might take place showing how the definition of an activity causes an iteration over required activities and policies. For this example, the project and product model remained static, but very likely you will have to iterate over them as well.

3.2 Lesson 2: Limiting the Process to Understanding takes Discipline

One of the hardest things we found in this study was to keep our attention focused on trying to define the process as it existed. We constantly found ourselves trying to define a model of what it ought to be rather than what it was. I think the main reason for this is a natural one: as one discovers the flaws in the process there is a tendency to want to correct them, especially if the solutions are seen as obvious ones.

Thus, we found it very hard to separate the process of improvement from the process of understanding the process as it is. We had to keep pulling ourselves back to the narrow focus of the original intent. Obviously, we kept track of what we considered to be problems with the current processes and what we thought of as solutions to these problems. We strongly encourage you to do the same. But, we also feel strongly about the need to document the existing process so that there is a firm basis for defining and measuring the various changes to the processes that will arise from this exercise. Baseline it first [5], then improve it by substantiated and measured changes [2].

Projecting into the future of our enterprise, I would also extend this lesson: limiting improvements to substantiated and measured ones takes discipline as well. It is all too easy to rely on your intuition (even as accurate as it may be) and anecdotal evidence (even as good as it seems to be) as the basis for process improvement. As a cautionary word, remember that organizational and technological factors are also critical in process improvement [6].

3.3 Lesson 3: Modeling Enabled Us to Find Serious Flaws in the Process

The flaws discovered in the process of understanding came in several flavors from problems about how to organize the process, through somewhat obvious omissions in critical policies, to very subtle problems about tool interdependencies.

One of the problems about process organization was that the definition of roles and their use in the process, and the people that filled those roles. The problem was that they did not have a clear view of how a role should function. Indeed this is a serious problem since there are at least a dozen ways in which one might define the function of roles in processes. For example, a role might mean any one of the following things:

- the embodiment of a process — this is the definition of a role as we find it in a craft: an Xr is one who has been apprenticed in the arts of X and has demonstrated that he or she has mastered the materials and processes of doing X;
- a job description — an X is one who does ... and as such aligns with job categories and hiring practices;
- a set of activities — for example, a designer is one who does design and in this view the definition tends to be finer grained than the preceding;
- a particular responsibility — finer grained yet where we have document reviewers, moderators, etc;
- an execution list — this separates the role from the process and defines the role in terms of the activities that must be executed;
- a permission list — slightly different from the one above in that the role is not required to execute the activities, but allowed to if desired; and
- an abstraction for people — a minimal definition to hide the fact that people get reassigned or that more than one person may actually do it and as such functions like a variable in a program that can take various values at various times due to a binding mechanism.

A case of a rather obvious omission was the fact that testing was not a necessary part of the acceptance evaluation. One could accept or reject a tool for release without actually testing it. This was a case of too much reliance on the ‘‘reasonable person’’ principle.

A number of serious issues arose in handling the subtle interactions between the tools. This is primarily because there was not a good model of interdependencies among the tools and the evolution of those interdependencies over time. Thus there were serious problems in that resulted from this deficient dependency model:

- tools could be released that were dependent on tools not shipped;
- there was no way to indicate shared dependencies; and
- many dependencies were implicit, not explicit, and as such not caught by the dependency model.

One further important omission was that concerning versions and configurations. There was only one version: the current one. That alone causes serious problems distinguishing between the currently exported version and the new one submitted for approval. There is nothing in the model that distinguishes between these two. The other major problem resulting from this is that the release process focuses on the release of individual tools, not on the release of consistent configurations. Moreover, the process of approval is time dependent since a later submission might break earlier approved tools and hence be rejected without any requirement that the earlier approved tools be fixed in order for it to be accepted.

3.4 Lesson 4: An Emphasis on Policy and Product Directed Modeling was Extremely Useful

While part of any understanding comes as a result of merely thinking about the process no matter how, uncovering a number of problems can be attributed explicitly to the formalism of Interact.

Interact requires three primary sections to be part of the model: a data model section (used to model the artifacts, the project and/or organization, etc as well as process state), a policy section in which to define the important policies that define various assumptions and goals and govern the execution of activities, and the activities themselves (where the emphasis is on assumptions and goals rather than just on implementations).

The focus on data modeling of the artifacts uncovered a number of the problems relative to the subtle interactions of interdependencies. More accurately, the lack of a sufficient model was the cause of these problems. The artifact model was too simple for the reality of the various kinds of interdependencies.

Another deficiency in the project model is the lack of any notion of deadline. Thus in the activity Integrate, there is no notion of how long the loop should go on waiting for tools to be submitted and evaluated.

The focus on policies was instrumental in uncovering several serious shortcomings in the release criteria. One such policy is related to the model deficiency mentioned in the preceding paragraph: there is a clear policy for when the release cycle can begin, but there is no policy at all for what defines the end of the cycle (a deadline, all the tools having been approved, etc).

Another serious policy deficiency is that there is no policy about the problems that cause a tool to be rejected and what should be done about them: whether they should be fixed, how they should be fixed, etc.

The focus on activity descriptions was extremely useful in clarifying activity interdependencies without overly constraining the process. Where there are several activities providing the same goals, the people have a choice of which to do. Otherwise, the activities are partially ordered according to the dependencies defined by the activity assumptions and results (goals).

We saw some example were the implementations of the activities were given. In one case it was because of the sequential ordering that was essential but which would not have been guaranteed by the resulting policy. In the other case, that of Integrate, it was to provide the basic structure and to control the mixture of parallelism and sequentialization in the activity — to make it easier for the executer to understand.

4. Conclusions

This study was an exercise in using a process modeling formalism as the basis for understanding a process as a prerequisite to improving it. Interact was extremely useful both in documenting the process as it exists and in uncovering serious problems with the process that need to be fixed. Thus we consider the exercise to have been successful both for gaining an understanding of the process as it exists (and providing ways of improving it) and for exercising Interact as a means of process modeling.

Acknowledgements This work would not have been possible without the help of the other members of the discussion group at the time of this experiment in process modeling and understanding: Alexander Wolf (CU Boulder), David Rosenblum (UC Irvine) and Balachander Krishnamurthy (AT&T Research Labs).

References

[1] Ashok Dandekar and Dewayne E. Perry, ‘‘Barriers to Effective Process Architecture’’, Software Process: Practice and Improvement, 2.1 (January 1996).

[2] Ashok Dandekar, Dewayne E. Perry and Lawrence G. Votta, ‘‘A Study in Process Simplification’’, Software Process: Improvement & Practice, 3:2 (June 1997).

[3] Dewayne E. Perry. ‘‘Policy-Directed Coordination and Cooperation’’, Proceedings of the 7th International Software Process Workshop, October 1991, Yountville CA.

[4] Dewayne E. Perry, ‘‘Enactment Control in Interact/Intermediate’’, in Software Process Technology, Third European Workshop, EWSPT’94, Brian C. Warboys, ed., Springer Verlag, February 1994 .

[5] Dewayne E. Perry, Nancy Staudenmayer and Lawrence G. Votta, ‘‘People, Organizations, and Process Improvement’’, IEEE Software, July 1994.

[6] Dewayne E. Perry and Lawrence G. Votta, ‘‘The Tale of Two Projects -- Abstract’’, European Software Engineering Conference/Foundations of Software Engineering Conference 1997, Zurich CH, September 1997.