Policy-Directed Coordination and Cooperation

Dewayne E. Perry
Software and Systems Research Laboratory
AT&T Bell Laboratories,
Murray Hill NJ 07901

Of paramount importance in modeling an evolutionary software process for large systems are the following observations:

- the software artifact is dynamic and changing;
- the artifact is (or rather, various parts of the artifact are) in multiple and different states simultaneously; and
- the model of the product state induces a partial order on the possible process activities.

Modeling such a process requires design decisions about the granularity of the process activities, about the degree of prescription defined within those activities, and about the policies that govern the triggering and termination of the activities and that define the nature of interaction and communication among the programmers evolving the system. Whatever choices are made for these issues, the resulting process is of necessity one with concurrent, independent, and asynchronous activities — that is, there will be multiple and different activities acting on the multiple and different states of the product. The critical issue is that of coordinating and synchronizing these independent activities.

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 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 and the project; policies definitions are used to model various facts and relationships about both the product and the project as well as to define synchronization (or interaction) 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 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.

Of paramount importance, however, are the policy definitions. They define the relationships among objects in several 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 and 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.

```
-- Product and Project Model

        type    tool                    primitive;
        tool    inscape;
        tool    infuse;
        . . .

        type    role                    primitive;
        role    integrator;
        . . .

        type    roles                   set of (person, role)

        obj     dept-roles              roles
        obj     owner[tool t]           person;
        obj     dependencies[tool t]    set of tool;
        obj     exportset               set of tool;
        . . .

-- Policies

        notify(person p, string s) ::
                1.  primitive.
        approve-modification(person p, tool t) ::
                1.  for person i | (i, integrator) in dept-roles,
                        notify(i, "%p approves %t").
        reject-modification(person p, tool t) ::
                1.  for person i | (i, integrator) in dept-roles,
                        notify(i, "%p rejects %t").
        modifications-approved(tool t) ::
                1.  for each person p in { p | owner[t1] = p and t1 in dependencies[t]},
                        approve-modification(p,t).
                2.  for each person p in { p | owner[t1] = p and t1 in dependencies[t]},
                        approve-modification(p,t),
                        tested( {t, t1}, errors ),
                        errors = {}.
        modifications-rejected(tool t) ::  . . .
        . . .
```

The recommended modeling method is one that in general keeps that activity descriptions fairly straightforward and encodes the coordination abstractions in the policies.

This example does not require that the participants in the process synchronize with each other in approving a particular modification, but merely that they follow certain rules in approving that modification. When and exactly how the activities are performed are up to the participants, who are constrained only by the allowed results of the instantiated activities. Should there be a number of different ways to satisfy a particular result, the participant is free to chose among the possibilities. In the case above, it is up to the process administrator to constrain the process to either require testing as a part of the approval process or not. The result of the integration activity requires that only approved modifications are allowed to be exported.

It should be noted that what the process model description provides are the pieces with which an actual process is instantiated dynamically. The description defines the modeled objects and the activities that may be applied to them. How the model is enacted is due in part to the process administrator, the state of the product, project and process, and the actions of the individual participants. The model constrains the

```
Evaluate-Modification(person p, tool t)
        pre:  modification-submitted(p, t)
        { primitive }
        1.   post:     not tool-built(t, testset),
                       reject-modification(p, t).
             obl: <none>
        2.   post:     tool-built(t, testset),
                       reject-modification(p, t).
             obl: <none>
        3.   post:     tool-built(t, testset),
                       approve-modification(p, t).
             obl: <none>

Integrate-Modifications()
        pre:  modification-cycle-initiated()
        { parallel:
             for each tool t in { t | modification-submitted(p, t) },
                  for each person p in { p | owner[t1] = p and t1 in dependencies[t]},
                       instantiate Evaluate-Modifications(p, t)
        }
        1.    post:    for each tool t in { t | modification-approved(t)},
                           t in exportset,
                  not some tool t in { t | modification-rejected(t)},
                           t in exportset.
```

process to some desired degree, but within those constraints, the order of enactment, the activities instantiated, the number of activities instantiated is dependent on the dynamics of the product, the process and the participants.

Intermediate, the process support mechanism, consists of three components: model analysis, model administration, and model enactment. The analysis component takes an Interact model description, analyses it for consistency, satisfiability, reachability, etc, and produces the model representation needed for administration and enactment. For example, the analysis component checks to ensure that it must be possible to satisfy the preconditions of an activity by some combination of results from other activities.

The administration component enables the process administrator to initialize, customize and evolve the model for a particular use, instantiating particular parts of the model and dynamically adjusting the model as it is enacted. Among the basic administrative operations on a process model are populating the various objects in the model with values (for example, assigning roles to the people on a particular project), supplying arguments for the parameters of the instantiated models and activities, determining which policy definitions are active at a particular time, and binding certain activities to tools to be activated automatically.

The enactment component supports the actually processing of the model. As such, the focus is on the interactive support of the human element in the process. This component manages and monitors activity instantiations and the current model state, determines when activities are startable or completable, provides facilities for user customization and instantiation, manages the automated activities, and maintains process history. In the example above, the activity Evaluate-Modification is instantiated automatically (as a result of the Integrate-Modifications activity) for each person who has a tool that is dependent on a submitted tool. Each these instantiated activities must be completed. The submission of modified tools, however, is the result of voluntary activities instantiated individually by those who have new versions to release. In both cases, the enactment component provides interactive guidance, monitors progress, and enforces the policies for activation, progress and completion.