Enactment Control in Interact/Intermediate

Dewayne E. Perry

AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 USA

Abstract. Interact/Intermediate supports goal-directed process modeling in such a way as to maximize the concurrency of activities and to minimize the direct control of humans in the process. In this context, there are three different and interacting loci of control which we illustrate and discuss with an example: an implicit, internal locus of control; an external, arbitrary locus of control; and an explicit, internal locus of control.

1 Introduction

The philosophy of Interact/Intermediate [1-3] is to support goal-directed process modeling in such a way as to maximize the concurrency of activities and to minimize the direct control of the human element in the process. Interact is the process description language by which the necessary artifacts, project structures, organizational structures, and process activities are defined; Intermediate is the support environment which, via the process models, provides the context for the process being enacted by both humans and tools.

Interact's emphasis is on specifying the assumptions and goals of the various process activities while, in general, leaving the details of the activities implementation to the enactor. Providing guidance in implementing the structure of an activity is one of the primary purposes of Intermediate.

However, there are times when the modeler will want to define in some detail the enactment structure of an activity. For example, the detailed description may be the result of a desire for a particular approach to avoid some particular problem, a desire for some standard steps in certain activities across a particular project, or a desire to automate a particular routine activity.

Two further considerations are important in the design of the enactment control mechanisms of Interact: dynamism and reflectivity. The process by which we build and evolve large software systems is of necessity an extremely dynamic one. The activities required at any particular point in time are dependent on the state of the artifacts being produced, the state of the project and organization, and the state of the process itself. Only by being able to reify both the state of a process and the process itself can the process be sufficiently adaptive and dynamic. Not only will the human enactor dynamically create processes out of the activity fragments, but the process itself will dynamically create the necessary enactment control structures for the particular state of the product, project and process.

Given these various requirements on the design of the description language and the support environment, it is worth noting that there are three different loci of control that will interact with each other in controlling the enactment of any given process: an implicit, internal locus of control, an arbitrary, external locus of control, and an explicit, internal locus of control.

In the subsequent sections, we discuss these three loci and indicate how they interact with each other.

2 Implicit, Internal Locus of Control

Activity descriptions are the basic process fragments from which an enacted process is constructed. An activity description consists of an activity name, a set of typed parameters, a set of policies (policies are first order predicates over finite sets) that represent the assumptions that must be satisfied before enactment of the activity can begin (that is, preconditions), the internal structure of the activity (which may be **primitive** — meaning that the implementation is left to the enactor — or some explicit control structure), and a sequence of results (one or more of which may be designated by the enactor as the results of the enacted activity) each of which consists of a set of policies that represent assertions about the resulting state of the activity (that is, postconditions), and a set of of policies that must be eventually satisfied by enactors of the process (that is, obligations). Consider the example activity description.

Because of the preconditions, postconditions and obligations, there are dependencies among the defined activities in a process model that yield a partial-ordering of those activities. Preconditions and obligations must be satisfied by some set of state and postconditions provide that state. Thus there is an implicit ordering, an implicit control, that is exerted on the activities that are enacted within a particular model. An activity instantiation (such as Determine-Dependencies) can only be enacted when its assumptions are satisfied. Until that happens, the activity cannot proceed no matter what explicit control structure it may be embedded in.

This partial order is the fundamental global form of enactment control.

3 Arbitrary, External Locus of Control

There are three different kinds of user-directed (and apparently, arbitrary) enactment control: activity elaboration, state restoration and enactment scheduling. The first is considered to be normal enactment control; the second is considered to be abnormal enactment control (that is, it is the handling of exceptional circumstances by moving the process state to some — possibly previous — consistent state) by the human enactor; the third is an external constraint on either the beginning or the completion of an enactment.

In the example, we have a detailed activity structure specified for the integration activity. Given that our approach is to underspecify rather than overspecify activity structures, activity structures will range from the primitive structure (which must be elaborated at enactment time) through a variety of incompletely

```
activity Integrate ( )
  preconditions { Release-Approved(Tool-Release-Board) }
  {
    for each tool t in {tool t | submitted(t) }
    until Current-Time == Deadline:
      <
        Determine-Dependencies(t, dependencies) ,
        let testset' = testset + t ,
        Build(testset', result) ,
        ( result == false, tool-rejected(t) ) ,
        ( result == true,
          <
            < for each person P
                in {person p | owner[t1] == p & t1 in dependencies }:
                  bind Evaluate(t, t1) to P
            > ,
            Await-Acceptance/Rejection(t)
          >
        )
      >
  }
  results
    <
      ( postconditions {
            approvedset = { tool t | tool-approved(t) },
            exportset = exportset + approvedset,
            tools-released(exportset) } ,
        obligations { }
      ) ,
      ( postconditions { rejectset = { tool t | tool-rejected(t) } } ,
        obligations { for each tool t in reject-set: modify-tool(t) }
      )
    >
```

specified structures to completely specified structures. Since the general intent is for an incomplete structure rather than a complete one, the human enactor will interact heavily with the support environment to elaborate the incomplete activity structures by means of the various Intermediate environment commands and the various language control statements.

The interaction between the enactor and the internal and implicit partial-ordering of activities is a direct one. The primary purpose of the enactor is to achieve one or more results within a given activity. One way to achieve those results is to backchain through the activity dependency graph to find activities that will produce the desired results. This backchaining results in a non-classical transformation of an incomplete activity structure into a complete one.

For those activities that are independent and also for those activities that have been enabled, the enactor is free first to choose which activities to enact and in which order. Their freedom is constrained only by the implicit priority defined by the specific schedule applied to those activities. For those activities that are in the process of enactment, the enactor is free to multiplex between them arbitrarily by means of activity suspension and resumption. It should be noted, however, that mere caprice is not likely to result in an effective or efficient process.

It may well be the case that because of the incompatible dependencies or conflicting enactments, a particular activity may get permanently stalled (or, if you will, starved or deadlocked). Often the way to recover from this situation is to change the current state of the process by enacting various environmental commands or recovery activities.

Again, the interaction between this form of control and the implicit partial-order control is a direct one. The recoverer must understand those dependencies and, via recovery activities, adjust the process state to some state in which progress can be made.

4 Explicit, Internal Locus of Control

Explicit enactment control is provided in Interact via basic enactment commands, enactment control, and control generation. Basic enactment commands are the units of work within an activity and are performed either by the human, a tool or the Intermediate support environment. Enactment control indicates how these individual enactment commands will be executed in relation to each other. The control generation mechanisms enable one to generate families of structures with a basic structure that is individuated either statically with distinct arguments or dynamically with different values. We see illustrations of each of these in the example.

4.1 Enactment Commands

There are three kinds of enactment commands: local equations, activity instantiations, and Intermediate commands. Local equations enable the modeler to bind values to names. On the one hand, this enables one to construct or deconstruct values for local names used only within the context of the local equation — that is, import values. On the other hand, this enables one to construct or deconstruct values for global names — that is export values. The constructive aspects of local equations enable one to construct larger structures out of component parts; the deconstructive aspects enable one to extract desired pieces from complex structures.

The local variable, testset', is defined and and assigned its value of the current test set with the addition of the particular tool t.

Activity instantiations are analogous to function or procedure calls: the activity is named and given a set of arguments. However, as we mentioned in the section on implicit control, the activity cannot be enacted until the assumed policies have been satisfied. Once the assumptions have been satisfied, the activity can be enacted. If the activity is bound to a tool, then that tool will be executed by the support environment; if the activity is bound to a human, then it is the responsibility of the that human to enact that activity.

Determine-Dependencies and Build are two examples where the instantiated activities are bound to tools. Evaluate, however, is bound to the appropriate owners of tools that are dependent on the tool to be evaluated.

An Intermediate command is one of the basic underlying process enactment and inspection primitives (such as binding commands, event commands, eval, etc.). These commands are the primitives for process definition, administration and enactment. We leave the discussion of them to a later paper.

4.2 Enactment Control

There are three enactment control structures: guarded enactment, sequential enactment and arbitrary enactment. Guarded enactment is represented by a tuple in which the first element in the tuple is the guard and the second is the enactment statement (which is either an enactment command, enactment control or control generation statement).

We test the results of the the build activity with two guarded enactment statements: the first specifies what to do if the build fails, the second specifies what to to if the build succeeds.

Sequential enactment is denoted by the structure $< \ldots >$ — literally, this is a sequence of enactment statements which are to be executed in sequence. Note, however, that the semantics of the individual enactment statements must be observed. In particular, if the assumptions of an activity have not been satisfied, the execution sequence will not proceed further until that does occur. Thus some other activity or set of activities will eventually have to be enacted to satisfy the assumptions of the stalled activity.

We have three sequential enactments in the example. The first is the sequence of steps that applies to each submitted tool. The second is the sequence of steps that applies when the build succeeds. The third, we defer to the discussion of control generation.

Arbitrary enactment is denoted by the structure $\{ \ldots \}$. Where in sets of data, we have no particular ordering of that data, so in sets of enactment statements, we have no particular ordering of those statements. The enactor (whether it is a human, a tool, or the support environment in the case of local equations and primitive commands) is free to select the order arbitrarily doing as much concurrently as is desired. Remember, however, that activities can only proceed when their assumptions have been satisfied. The defined activities' partial ordering takes precedence over the arbitrary ordering.

We have a single arbitrary enactment statement in the example. It is coupled with a set generation statement which we will discuss in the next section. In this case, the enactments generated are to be done in arbitrary order. It is likely that each of the enactments generated is independent of the others and can be done in truly arbitrary order — concurrently, randomly, etc.

Note that we have combined sequential and arbitrary enactments: we have an arbitrary enactment of sequences as the basic structure of the activity. We could just as easily have had a sequence of arbitrary enactments. Note that a arbitrary enactment of arbitrary enactments just reduces to a single arbitrary enactment. That is,

{ {A, B} {C, D} } is effectively { A, B, C, D }.

4.3 Control Generation

Given the set orientation of Interact, a useful thing to be able to do is to apply some enactment to each member of the set. We do that with the set generation command. This command generates an enactment for each member of the specified set. The order in which the enactments are done depends on the encompassing enactment control.

We have two such instances in the example. The first applies the sequence enactment to each submitted tool. Note that this set generation command is contained within the arbitrary enactment command: the evaluation sequences for each tool may be done arbitrarily (though, of course, the sequences themselves must be done in sequence). Note also that we have a termination expression that allows the set generation to continue until the deadline has been reached. As additional tools are submitted (that is, as the set of tools satisfying the policy tool-submitted gets new members), the evaluation sequence is applied to those newly submitted tools.

In the second instance, we have an illustration of the set generation command within a sequence enactment. This produces an enactment sequence that is analogous to iteration (though strictly speaking, it is iteration unrolled). An instantiation of the evaluation activity is bound to each person who owns a tool dependent on the particular submitted tool. Note that it is the binding of the activity not the enactment of the activity itself that is done in sequence. [1] The actual enactment of each of those instantiated activities takes place outside the scope of this activity and is subject to the standard partial ordering rules.

5 Summary

We have illustrated three loci of enactment control: internal and implicit, external and arbitrary, and internal and explicit. The partial order inherent in the precondition and obligation dependencies is paramount. It is fundamental in the

[1] Note that we could just have easily done the binding in arbitrary order — that is, enclosed this part of the activity implementation in { } instead of < >. We enclosed it in a sequence here for pedagogical reasons.

sense of global control of activity enactment. Within this global control, the human has a great degree of latitude to choose which of the prescribed activities to enact as long as it is consistent with the modeled enactment ordering.

References

1. Perry, Dewayne E.: Policy and Product-Directed Process Instantiation. Proceedings of the 6th International Software Process Workshop, 28-31 October 1990, Hakodate, Japan.
2. Perry, Dewayne E.: Policy-Directed Coordination and Cooperation, Proceedings of the 7th International Software Process Workshop, October 1991, Yountville CA.
3. Perry, Dewayne E.: Humans in the Process: Architectural Implications, Proceedings of the 8th International Software Process Workshop, March 1993, Schloss Dagstuhl, Germany.

This article was processed using the LaTeX macro package with LLNCS style