The Object Role Software Process Model

Noureddine Belkhatir and Walcélio L. Melo
Laboratoire de Genie Informatique
BP 53X
38041 Grenoble Cedex France
{belkatir,wmelo}@imag.imag.fr

Published in J.-C. Derniame (Ed.)
Software Process Technology.
Lecture Notes in Compute Sciences. Volume 635. Spring-Verlag.

Introduction

In our previous papers[2, 1, 4, 3], we report our use of event-condition-action formalism for modeling software process, and the Adele trigger mechanism to control this execution. Although, this approach produced very good results, it has various drawbacks, including: (1) the formalism is low level; (2) trigger execution may be difficult to control; and (3) the software process description is distributed over different object and relation types. For human understanding, it would be preferable to have all the information relevant to a given process together in single unit. We are currently trying to solve the problems we found during our experiments using a high level process formalism. In the remainder of this position paper, we summarize the main design issues of this software process formalism.

The Adele's Process model

Each identifiable software process step in a software development process is represented in Adele's modeling language as a complex object. A process step is recorded in the Adele-DB as an instance of a standard object type. Thus, a process can be instantiated, characterized by attributes, versioned (storing the history of the different states of the software processes), removed and connected with other processes or software entities. Through the multiple inheritance mechanisms, a process type can be refined and specialized. New attributes, roles, methods and rules can be defined, modified and overloaded. Therefore, process customization will achieved by process type specialization.

The role concept

A role makes it possible to adapt the vision of an object in an process step, i.e. which specific types of the product model are able to be adapted to process execution context using the *roles*. Thus, a process becomes a list of roles where each role is customized in order to satisfy the process requirements. That is, the properties and behavior of an object are characterized in the process execution context. The data driven process model specifies how an object, in a specific step of the life cycle is manipulated. The access and modification of roles are mapped to the corresponding objects in the product base.

Each role has methods which are used to adjust the behavior of the original object to the execution context. That is, a role can redefine the original methods or define new ones in order to customize the object behavior for the context where the object is used. For example, the `module` type has methods associated with it, that are independent of the context where a module is used. However, when a module is manipulated by a process step, other methods may be needed, e.g., the method `compile` associated with the module type may be different from the one used by the debug activity (compilation flags, etc). On this way, methods can be overloaded, i.e. a method defined in the module type can be overloaded by the method defined in the role types.

In order to control method execution, event-condition-action (ECA) rules are used to envelope methods. For each method, a set of rules (private and react rules) can be specified for automatically taking actions before and/or after method execution as well as in the case of error situations. The purpose of specifying rules as a method envelope is to provide better and more flexible control during process execution. Thus for each process step, the operations that can be mechanized, the situation to which such operations can be applied, the chaining of operations execution, and what to do when such operations do not run correctly are described using both methods and rules.

Although private and react rules are defined in the same way, their semantics are slightly different. Private rules are activated only when the current process is the originator the method execution, i.e. when a method is called within of the process execution context, the private rules of the process instance are hired. React rules are activated only when another process instance is the method call source. This different semantic has been imposed in order to improve method execution control and provide an implicit mechanism for process synchronization. We are currently extending our ECA formalism to better control synchronization and coordination of operations in long transaction. This extension is based on a subset of first-order temporal logic (use of some modal operators). We have felt the need to express constraints, such as:

1: AF (checkout (M) $\rightarrow$ SF(checkin(M)) 2: AF (checkin (M) $\rightarrow$ SP(checkout(M)

Line 1 stipulates that every time object M is checked-out (AF), there will be a reference point in the future (SF) where M is checked-in. Line 2 stipulates that every time checkin is executed on M (AF), there were checkout M in a

reference point in the past (SP).

AF	always in the future	AP	always in the past
SF	sometime in the future	SP	sometime in the Past

An Example

Two software engineers (**A**, **B**) work in parallel on the same module copy in their respective work spaces (WSs), A is responsible for module interface modification, while B is responsible for module body modification. when **A** checks the module part in the DB, an unitary test triggering automatically to verify if the modification is correct. In the positive case, the module part is recorded in the DB, creating a new revision; **B** must be alerted about the modification; and the new module part revision (either interface or body) is copied automatically to **B**'s WS. When both the module interface and body are considered OK, then the module is validated and the process finishes.

```
TYPEPROCESS WS-change IS PROCESS;
 ROLE view = module;
         LOCALATTRIBUTE status = tested, not_tested := not_tested;
         METHOD         (defmethod check-in ... )
         PRIVATE-RULE
(1)        check-in PRE:
(if (not (apply-test-in self_role))
      abort)
(2)        check-in POST:
(mda  self_role (('status 'tested)))
(3)      REACT-RULE
    check-in:
(progn (send-mail ...)
       (check-out self_role))
END WS-change;
TYPEPROCESS manager IS PROCESS;
  ROLE domain = module;
  ROLE implement = WS-change;
    REACT-RULE
      check-in:
(4)   (if (== (lsa (lsrole (lsrole self 'implement) 'view) 'status) 'tested)
    (mda self_role (('status 'tested))))
END manager;
```

In our example, each SE finishes its activity considering that the module is either `tested` or `not_tested`, however only the manager is able to release the module status in the public database. From a technical point of view, we can see each process step like a long transaction which is carried out in a private database. Thus, depending on the policy described, the changes made inside a process step may not interfere with other activities carried out in parallel during the software process.

Whenever the module is checked-in by a WS-change, using the private rule, a foreign tool (`apply-test-in`), is applied to the module in order to test the modification . If the tests are not validated, the check-in operation is aborted (1). Otherwise, after the check-in operation the local `status` attribute is up-to-dated to `tested` (2). Using the `react rule` after the check-in operation, all other instances of `WS-change` are alerted about the module modification, and the module part (interface or body) is automatically replaced by the new tested revision (3). When these operations are finished, the manager is also alerted by the react rule about module modification. It verifies whether all WS-change process steps have tested the module; in the positive case it releases the module in the public database to `tested` (4). Note that the WS-change is considered like a role by the manager process.

Status

This formalism is currently under work. On parallel, we are extending our event-driven process virtual machine and object model to support this formalism mainly temporal constraints to control long transactions. It will be possible, because: (1) We keep track the evolution of the objects (state and content) manipulated during the software processes; (2) Every significant event produced during the software processes is also recorded in the database. Through these two kind of object historic we will be able to interpret the temporal logic specification in order to assure the operations synchronization.

References

[1] N. Belkhatir, J. Estublier, and W. L. Melo. Activity coordination in Adele: a software production. In I. Thomas, editor, *Proc. of the 7th Int'l Software Process Workshop*, San Francisco, CA, October 16–18 1991. IEEE CS Press.

[2] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In M. Dowson, editor, *Proc. of the First Int'l Conf. on the Software Process*, pages 159–170, Redondo Beach, CA, October 21–22 1991. IEEE CS Press.

[3] N. Belkhatir, J. Estublier, W. L. Melo, and M. A. Nacer. Process-centered SEE and Adele. In N. H. Madhavji G. Forte and H. A. Muller, editors, *Proc of the 5th Int'l Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 156–165, Montréal, Québec, Canada, July 6–10 1992. IEEE CS Press.

[4] N. Belkhatir, W. L. Melo, J. Estublier, and A.-M. Nacer. Supporting software maintenance evolution processes in the Adele system. In C. M Pancake and D. S. Reeves, editors, *Proc. of the 30th Annual ACM Southeast Conf.*, pages 165–172, Raleigh, NC, April 8-10 1992.