Melding Software Processes into Software Object Databases

Ying Yang
Dept. of Computer Engineering and Science
Case Western Reserve University
Cleveland, OH 44106

Abstract
This paper describes a rule based approach to melding software processes into software object databases. The result is illustrated on some examples.

1 Introduction
Considerable attention has devoted to software process modeling during the past few years. A software process is a set of software engineering activities needed to transform a user's requirement into software. A software process may include requirement specification, design, implementation, verification, installation and operational support. It may also include repair or enhancement to meet continuing needs. Models for software processes are expected to provide a means of reasoning about the organizational processes to develop and maintain software. Since the user's needs and the operational environment may not be fully knowable in advance, the software processes need to be refined dynamically as more knowledge is gained during design and implementation.

We believe that software objects are inextricably bound to software development processes. That is, to properly define an object requires a careful understanding of how it will be developed and to define a development process requires a good understanding of the objects it will realize.

We have designed a model to describe various software objects such as management data, design documents, software modules. One of the novel features of our model is its ability to meld software processes into software object databases. In our model, software objects are divided into two primitive types: relationships and entities. A relationship class declaration consists of a class name associated with a list of attributes, some of which may be declared as flowing across the relationship. Real-world objects are modeled by entities. An entity class declaration consists a class name associated with attribute and port declarations. For example, we can represent software modules as entities. Each module defines a set of ports for its different interfaces. The module can get a list of import services that are required as data flowing out of these ports, and output a list of export services that are provided by it as data flowing into these ports. The interconnection of modules may be represented by relationships, each relationship specifies the type of services that are provided along it.

2 Rule Classes Process Specifications
In our model, software processes are specified as rule classes. Processes are executed through the firing of the rules attached to the objects of software object databases. The rules respond to dynamic changes in the environment and execute corresponding software processes. These dynamic changes may include users' operations on the software object databases, evaluation of type definitions, access controls and synchronization in the object management system, consistency checking, etc.

Now we demonstrate this approach by an example. Suppose that a software system consists of one main modules and several submodules, which in turn may have their own submodules. Simplistically, these modules are of type Module, and are connected by relationships of type Build. The declarations of these two classes are in Figure 1. Without losing generality,

```
define Build1
  ...
  status : (fail, success) to black
) inherits Relationship

define Module1
  ...
  sup: white Build1
  sub: black Build1
) inherits Entity
```

Figure 1: Declarations of Two Classes

we omit the possible attributes of entity type Module. The attribute flowing across the relationship Build1 is status from its white end to its black end. The entity class Module1 defines a port sup connectable to the white end, and a port sub connectable to the black end of a relationship of type Build1.
Now we try to model the following software process:

**Process 1:** The modules of a project stored in a software object database are connected by relationships of type Build as in Figure 2. When users give a command Rebuild on the main module, all the submodules should be recursively rebuilt, and the status of the rebuilding processes should be returned to the main module.

Process 1 can be described by the rule classes Make and Feedback in Figure 3.

```define Make1
  (on b1: Build1
    condition Rebuild([b1.black])
    action
      trigger(Rebuild, *[b1.white]);
    )
) inherits Rule
```

```define Feedback
  (on b1: Build1
    condition Finish([b1.white])
    action
      b1.status = [b1.white].status;
      trigger(Report_arrive, [b1.black]);
    )
) inherits Rule```

A similar notation may be used to manipulate multiple relationships connected to certain ports of an entity.

We now discuss the rule classes in Figure 3. The definition of a rule class consists of three parts:

- Each rule class has a domain, declared by the keyword on. The domain of a rule class specifies a set of entity or relationship classes over which the rule class acts. Rule classes are attached to entity and relationship classes declared on their domains. The rule classes **Make** and **Feedback** have an on domain **Build**, and they will be attached to every instance of **Build**.

In definitions of rule classes, we associate a role with each class in the on domain of the rule class. A role allows the instances of the associated class to participate in the rule class in this role. In the rule class **Make**, **b1** is the role for relationship class **Build**. The action part of **Make** is on individual mode so that at multiple connectable ends of relationships in rule classes. Let **R.e** be a multiple connectable end of a relationship **R** where **e** ∈ {black, white}. object **O** connected **R.e**. In order to process different entities connected **R.e**, we define two modes to refer these entities:

- **individual mode** **R.e**. **R.e** is a shorthand of any individual entity of all those connected at the port **R.e**. For example, in rule class **Make**, the statement `Rebuild([b1.black])` denotes the condition that an event **Rebuild** occurs at any entity connected to the black end of relationship **b1**.

- **broadcast mode** **R.e**. **R.e** denotes all entities connected at **R.e**. If we use **R.e** in place of **R.e** as a parameter of an operator **op**, we can apply **op** on all entities connected at **R.e**. For example, in rule class **Make**, statement `trigger(Rebuild, *[b1.white])` triggers an event **Rebuild** to every entities connected to the white end of relationship **b1**.

In more realistic situations, other rule classes would also be included to describe actions taken by the entity class **Module**. The resultant would be more complex without being more illustrative.
every entity \( m \) of type \( \text{Module} \), connected to the port \( \text{bl:.white} \) can be treated in the same way.

- The condition to fire a rule is a logic expression, called event expression, in which events may be used as logic operands. Events correspond roughly to software tool commands, such as create, delete, update, read, compile, link, test, deliver, etc. The occurrence of events may cause the evaluation of the event expressions of the rules depending on the events. This may further invoke the action parts of the rules. The effects may be propagated further to other rules. In above example, the condition to fire \( \text{Make} \) is an event expression \( \text{Rebuild}([\text{bl:.black}]) \), stating that if an event \( \text{Rebuild} \) occurs on an entity connected to \( [\text{bl:.black}] \), then the condition to fire the rule becomes true.

- The action part of a rule may be any object management actions, e.g., repairing actions to re-establish consistency with the given constraints, suspending a certain operation until the occurrence of some given events, refusal of an operation that results in the event becoming true, sending messages to the indicated destination objects and propagating a given event to indicated destination objects. The action part of rule class \( \text{Make} \) is trigger \( (\text{Rebuild}([\text{bl:.white}])) \), which triggers an event \( \text{Rebuild} \) on every entity connected to the white end of \( \text{bl} \).

When a user issues a command \( \text{Rebuild} \) on an instance \( m \) of type \( \text{Module} \), connected to the black end of \( \text{bl} \), the rule \( \text{Make} \) attached to \( \text{bl} \) is fired by the occurrence of event \( \text{Rebuild}([\text{bl:.black}]) \), and acts in two steps:

- After certain required actions (not presented here for simplicity), the \( \text{Make} \) propagates the rebuilding process along the relationship \( \text{bl} \) to the entities connected at the white end of \( \text{bl} \) by triggering events \( \text{Rebuild} \) on them. The process continues recursively on these entities along other relationships connected to them.

- After these entities finish the rebuilding processes, the occurrences of event \( \text{Finish} \) on them fire the rule \( \text{Feedback} \) attached to \( \text{bl} \). The status from these entities are passed back to the original entity along \( \text{bl} \). This may consequently cause other rules to be fired to carry out additional semantic actions (not specified here). This process may also be propagated recursively along the other relationships until the whole rebuilding process finishes.

3 Specialization of Processes

Software Engineering environments contain various kinds of entities and relationships, many of them contain common properties. For example, some entities are aggregations of other entities, e.g., stacks. Processes on these entities may differ only slightly. On the other hand, extending an environment often involves modifying some Processes so that they can be used with new entity types. The process modeling mechanism should make it easy to perform such modifications. In this section, we discuss one way to accomplish this: process specialization.

Let's look at an example first. Suppose we redefine Process 1 as follows:

**Process 2:** The modules of a project are stored in a software object database as in Process 1. But this time, when users give a command \( \text{Rebuild} \) on the the main module, the submodules should be recursively rebuilt only if the time stamp of the main module is out of date.

First, two inherited classes are introduced as in Figure 4. The relationship class \( \text{Build} \) is a subclass of \( \text{Build} \). In addition to the inherited attribute \( \text{status} \), \( \text{Build} \) defines a new attribute \( \text{last change} \). \( \text{last change} \) is defined to represent the last change times of submodules. It is defined as being transmitted from the \( \text{white} \) end to the \( \text{black} \) end of \( \text{Build} \). The entity class \( \text{Module} \) defines two new attributes: \( \text{time stamp} \) to record the last rebuilding time; and \( \text{lock} \) to prevent any possible accesses to the entity when it is rebuilding.

The definition of the class \( \text{Module} \) requires that the ports \( \text{sub} \) and \( \text{sup} \) form its parent class \( \text{Module} \), while we want these ports to be connectable to relationships of type \( \text{Build} \). The class \( \text{Module} \) inherits the ports \( \text{sub} \) and \( \text{sup} \) from its parent class \( \text{Module} \), while we want these ports to be connectable to relationships of type \( \text{Build} \). In order to enforce the restrictions on what an inherited class is allowed to do, we have the following compatibility rule for inheritance:

A subclass must have the same domain as the superclass, and for all attributes of the superclass, corresponding arguments of the subclass yield corresponding results.

By enforcing the compatibility rule, the attributes of a subclass can be freely manipulated as those of its superclasses with no fear of inadmissible behavior. This rule is similar to what called complete compatibility in [Wegner 88]. In our example, class \( \text{Build} \) is compatible to its superclass \( \text{Build} \), so it can be connected to the ports \( \text{sub} \) and \( \text{sup} \) of \( \text{Module} \). In fact, they can expect everything from \( \text{Build} \) as from \( \text{Build} \).

---

2 Here, we assume that there are other rules to update the time stamps when modules or their submodules are modified.
Consider the process of rebuilding a system, of which the modules are connected by the relationship \texttt{Make2} to represent the sub(sup)module relationship. The rule class \texttt{Make1} is already there, but is too un-restricted. In Process 2, an entity of type \texttt{ModuleC} should not be constantly rebuilt when an event \texttt{Rebuild} occurs on it. The time stamp of it should be tested to determine if it is out of date.

Extending the concept inheritance of classes from classical object oriented systems, our rule system allows the rule classes attached to one (relationship and entity) class \texttt{C} to inherit the rule classes attached to \texttt{C}'s superclass \texttt{S}, and the latter is called \texttt{virtual} rule class. The following modifications to the virtual rule class are allowed in our system:

- The condition to fire the rule can be strengthened, but not changed otherwise.
- New statements can be added to the action part, but these may not change the values of inherited attributes.

The process to define a rule class by modifications to its virtual rule classes is called \texttt{rule specialization}, and only the modifications are explicitly given and the rest is taken from the virtual rule classes.

\begin{verbatim}
define Make2
  (on b2: Build2
  condition
    out_of_date(b2.black).time_stamp)
  action
    [b.black].lock = lock
  inherits Make1

Figure 5: Rule for Process 2
\end{verbatim}

The restrictions on the rule specialization is motivated by the substitutability of compatible classes: an instance of the subclass \texttt{C} can always be used in any context in which an instance of its superclass \texttt{S} was expected, and the inherited rule classes for the subclass \texttt{C} are either the same rule classes as those attached to \texttt{S}, or the specializations of them. After the substitution of objects, the specialized rules may be executed as follows: testing the conditions of corresponding virtual rules from the superclasses, and enforcing additional restrictions of the specialized rules; then firing the virtual rules and executing the new statements of specialized rules, if both the conditions and restrictions satisfied.

For the specialization of rule class \texttt{Make1}, \texttt{Make2}, we strengthen the condition by "\texttt{out_of_date(b2.black).time_stamp}". The new condition tests whether the original condition "\texttt{Rebuild(b2.black)}"; the target entity must also satisfy the condition "\texttt{out_of_date(b2.black).time_stamp}" to be rebuilt. An additional statement is also specified to prevent other modules to access the entity when it is been rebuilding. The resultant rule class \texttt{Make2} are shown in Figure 5. Similarly, the rule class \texttt{Feedback} may also be specialized to update the time stamp of main modules and determine whether the main module should be rebuilt by testing the attribute \texttt{last_change} transmitted through the relationship \texttt{Make2}.

To formalize this process, we represent a rule class \texttt{R} as a 3-tuple \texttt{(D, C, A)}, where \texttt{D} is set of class names, \texttt{C} a conjunction of logic statements and \texttt{A} a sequence of semantic statements. They represent the \texttt{domain, condition} and \texttt{action} parts of \texttt{R}, respectively. For example, rule class \texttt{Make2} may be represented as

\begin{verbatim}
{b1 : Builds1,
  Rebuild([b1.black]),
  trigger(Rebuilt(*[b1.white])); ... }
\end{verbatim}

To interpret rule class \texttt{Make2}, it is necessary to access its virtual rule class \texttt{Make1}. This is achieved by supplying the child rule class as a parameter to an operator \texttt{V}, or the virtual rule class. For example,

\begin{verbatim}
V(Make2) =
( {b2 : Builds2},
  {Rebuild([b2.black])},
  {trigger(Rebuilt(*[b2.white])); ... })
\end{verbatim}

Tuple combination operation \texttt{\oplus} is a binary operator, which forms a new tuple \texttt{(D, C, A)} with the fields from its two arguments, where \texttt{D} is the union of the domains of both arguments, \texttt{C} is a conjunction of the condition parts of both arguments, and \texttt{A} is a concatenation of the action parts of both arguments. For example,

\begin{verbatim}
( {d1}, {c1}, {a1} ) \oplus ( {d2}, {c2}, {a2} ) =
( {d1, d2}, {c1 \land c2}, {a1; a2} )
\end{verbatim}

The rule class \texttt{Make2} specifies only the specialized part from its parent rule class \texttt{Make1}. The specialization rule may be indicated explicitly as \texttt{S}. In this case, the specialization made by \texttt{Make2} is

\begin{verbatim}
S =
( {b2 : Builds2},
  {out_of_date([b2.black].time_stamp}),
  {[b2.black].lock = lock}
)
\end{verbatim}

which states that an additional condition has to be satisfied, and a new semantic action should be executed if the rule is fired. After the combination with the original rule class, the specialization has the following form:

\begin{verbatim}
C = \forall(S) \oplus S
\end{verbatim}

where \texttt{V} refers to the inherited virtual rule class, and \texttt{S} refers to the specialized rule class. The two appearances of \texttt{S} are used in two contexts: for the interpretation of the virtual rule class \texttt{V} and to provide information about the specialization. Here, we use a
similar approach as that in [Bracha 90]. The example above has the following form:

\[
C = \{
\{b2 : Build2\},
\{\text{Rebuild([b2.black]\} out\_of\_date([b2.black].time\_stamp)})],
\{\text{Rebuild([b2.black])}; \ldots; [b2.black].lock = 1\operatorname{ock}\}
\]

4 Process Refinement

By process refinement, users can start from an initial solution satisfying some basic requirements, and impose further properties without violating those already been established. Process refinement is implemented by the rule refinement mechanism of our model. The relationship between a rule class \( R \) and its refinement \( r \) is as follows:

- \( r \) is eligible to be fired by the inference engine if and only if \( R \) is fired. In fact, the refinement(s) is taken into account in conjunction with \( R \).
- If a class \( C \) is in \( R \)'s on domain, then either \( C \) or a subclass of \( C \) is in \( R \)'s on domain.
- \( R \) may have several refinements, but they must be orthogonal. That is, if \( R \) is fired, then at most one of the refinements can be fired too.

Rule refinement is a natural construction for stepwise software refinement development. For example, we may further refine Process 2 as follows:

Process 3:
The modules of a project are stored in a software object database as in Process 2. But this time, when users give a command Rebuilding on the main module, all the submodules should not only be recursively rebuilding as in Process 2, but also be tested according to the attribute testing of the module. The test reports should also be returned.

Define Build3

\[
\begin{align*}
\text{define Build3} & \quad ( \\
& \quad \quad \text{test\_report} : \text{text to black} \\
& \quad \quad \text{inherits Build_2})
\end{align*}
\]

Define Module3

\[
\begin{align*}
\text{define Module3} & \quad ( \\
& \quad \quad \text{testing} : (\text{domain, functional}) \\
& \quad \quad \text{inherits Module_2})
\end{align*}
\]

Two more inherited classes are introduced as in Figure 6. The relationship class Build3 is a subclass of Build2. In addition to the inherited attributes status and last\_change, Build3 defines a new attribute test\_report representing the testing results of submodules. test\_report is defined to be transmitted from the white end to the black end of Build3. The entity class Module3 defines a new attribute testing to record the desired testing method.

We can model Process 3 by adding two refinements to rule class Make2 as in Figure 7. After a rule of type Make2 is fired, one of the two refinements will be fired to take additional test actions according to the value of attribute testing of the entity: functional, domain. To formalize the process of rule refinement, we introduce another operation, tuple union operation \( \cup \), which combines a set of tuples into a new tuple:

\[
\forall((\{d\}, \{c_1\}, \{c_1\}), \ldots, (\{d\}, \{c_n\}, \{a_n\})) = \\
(\{d\}, \{\text{TRUE}\}, \{c_1 \rightarrow a_1, \ldots, c_n \rightarrow a_n\})
\]

For example, the union of two classes Make2_1 and Make2_2 is:

\[
\forall(\text{Make2}_1, \text{Make2}_2) = \\
(\{b3 : Build3\}, \{\text{TRUE}\}, \\
\{b3\_black].testing == \text{domain} \rightarrow \\
\quad \text{trigger}([\text{domain}\_test(*[b3\_white]))))); \\
\{b3\_black].testing == \text{functional} \rightarrow \\
\quad \text{trigger}([\text{functional}\_test(*[b3\_white]))])
\]

The refinements \( R_1, \ldots, R_n \) of a rule class \( V \) specify a new rule class \( R \), in which actions specified \( R_1, \ldots, R_n \) are added to the action part of \( V \). The resulting new
rule class is following class combination:

\[ R = V(R_1) \oplus (w(R_1, ..., R_n)) \]

In the example above, two refinements, Make2-1 and Make2-2, of the rule class Make2 define a new rule class \( R \) as follows:

\[ R = V(\text{Make2-1}) \oplus (w(\text{Make2-1}, \text{Make2-2})) \]

\[ = \{ \{ \text{Build3}\} \\}

\[ \{ \text{Rebuild}([\text{b2.black}], \text{time}.\text{stamp}) \} \]

\[ \{ \text{Rebuild}([\text{b2.black}]) ; [\text{b2.black}] .\text{lock} = \text{lock} ; \]

\[ [\text{b3.black}] .\text{testing} = \text{domain} \rightarrow \]

\[ \text{trigger}(\text{domain}.\text{test}(*[\text{b3.white}])) ; \]

\[ [\text{b3.black}] .\text{testing} = \text{functional} \rightarrow \]

\[ \text{trigger}(\text{functional}.\text{test}(*[\text{b3.white}])) \}

5 Concluding Remarks

In the literature, several software process models have been proposed. For example, in the process programming approach [Osterw 87], a process is described as a program before the development is started. This program is then executed by the environment and the developer's role is to provide the information required during the process execution. Compared to the existing software process models, the main novel feature of our work is that our model is a uniform paradigm where the concepts of object management and software process modeling are integrated, and the dynamic behaviors of a software process and its impacts on the relevant objects can be described in a unified way.

Our basic object model is similar to the Cactis [Hudson 89] while our rule system can be seen as an extension of the Postgres [Stone 88] rule system. But ours has a different definition of events, domains and a different firing mechanism. The refinement and generalization of rule classes are also unique to our system. More importantly, our rule system plays a different role in object databases. The Postgres rule system is mainly used for supporting view systems of databases, while ours is designed to provide a unified way to dynamically model software processes. The dynamism of the process model is achieved by dynamic relationships between objects. This makes it possible to treat both passive and active objects equally. In our model all objects can be reviewed as actively seeking for partners to execute software processes by firing the rules attached to them. It is equally true to review all objects as passive resources manipulated by the software processes. All objects are maintained globally, and relationships between them can be defined and utilized. The other difference is that in our model one rule can act on arbitrary number of objects dynamically determined by the structure of object database, while the

rule class itself is expressed statically and is attached to individual (entity or relationship) object classes. In contrast, a rule in the Postgres rule system only acts on one class.

References


