Meta–Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes
Masato Suzuki, Takuya Katayama
Department of Computer Science, Tokyo Institute of Technology

ABSTRACT
Meta-operations are introduced into a software process model HFSP for modeling dynamic and flexible features of software processes. As process enaction is characterized as growing trees of activity decompositions in HFSP, these operations are most naturally considered as operations over these trees, which allow changing enaction status, creating new trees and communicating among them from inside and outside of the trees. A Formal description of these operations are given together with their application to the description of ISPW6 Example Process.

1 Introduction
Dynamics and flexibility are the fundamental characteristics of software processes, which make them different from usual pure computational processes and require special mechanisms for their enaction[1]. It refers, in general, to the fact that we need a dynamic and flexible control in their enaction. More specifically, it should allow that process scripts could be created, changed or refined incrementally during process enaction, processes could be dynamically instantiated from their scripts and the state of process enaction could be changed internally or externally for the process-process and process-human interaction.

These requirements come from several reasons. First, the behavior of software processes is dynamic in its nature. To modify a design, we will need the old design script to be changed and reinstantiated for the new design. To describe the situation that a manager monitors and controls a designer team, mechanisms for suspending, resuming, creating processes and changing process states, are required to control process behavior explicitly. Second, our knowledge of software processes is usually limited. We sometimes have to start with incomplete, partial and, even erroneous process scripts, and refine, augment and evolve them while they are enacted. Third, we, human, are involved in the processes as their essential components. Though we are creative, we are error prone and we often fail to perform activities correctly. We need a sophisticated redoing or backing-up mechanism which calls for a dynamic and flexible enaction of the software processes.

In this paper, we introduce meta-operations in the process model HFSP[2] as a mechanism for handling the dynamism and flexibility. HFSP is a software process model derived from the hierarchical and functional computation model HFP[3] and attribute grammars [4]. The primary concern of HFSP is the product-oriented aspect of the software processes which characterizes them in terms of their input/output.
relationships. These relationships are successively decomposed into simpler ones until they could be realized by invoking tools. Thus, HFSP describes the software processes in terms of tree structures (attributed with software products) representing the decomposition and their enaction is considered as growing these trees.

The meta-operations we introduce allow controlling the growth of the trees from the inside of the growing trees or from other trees. HFSP looks at a software process as a set of interacting trees. Meta-operations enable creating and changing the trees, suspending/resuming their growth and communication among them. HFSP with these meta-operations can be considered as a kind of reflective computational model in which programs can change their computational states and environments during their execution. The introduction of the meta-operations was found effective in describing higher level features of software processes.

The construction of this paper is as follows. Section 2 gives a brief description of the software process model HFSP. Meta-level features are introduced in Section 3. Their semantics are given in Section 4. Section 5 shows how they are applied to a process description using the ISPW6 Example Processes.

2 Software Process Model HFSP

2.1 Overview

HFSP (Hierarchical and Functional Software Process model) is a process model which describes software processes, in the first approximation, as a collection of activities which are characterized by their input and output relationship defined as mathematical functions. Though there might be many other aspects of the software process which have to be formalized, HFSP focuses on the following three aspects.

- Functional Aspect
  The functional aspect describes the input/output relationship realized by the software process. When the relationship is not simple enough, these activities are decomposed into subactivities together with the definitions of their input and output attributes.

- Behavioral Aspect
  The order of activity invocation is determined implicitly from the dependency among attributes of activities or explicitly specified by the notation which represents special ordering information such as sequencing, iteration and synchronization. In order to provide more flexibility and dynamism, meta-operations are introduced which enables dynamic control of process behavior.

- Enactional Aspect
  This aspect specifies a feature which are necessary to make written process scripts actually work in the real projects or organizations. Information about scheduling, resource allocation, collaboration, coordination and negotiation are described here.

2.2 Functional Aspect

2.2.1 Activity and Activity Decomposition

HFSP considers software processes as mathematical functions which map their input software products to their outputs, and define them through hierarchical functional definition. Activity $A$ with inputs $x_1, \ldots, x_n$
and outputs $y_1, \ldots, y_m$ is denoted by

$$A(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

$x_1, \ldots, x_n, y_1, \ldots, y_m$ are called attributes of $A$. When $A$ could be performed by software tools or human mental activity such as decision making, we call it a primitive activity. Otherwise, $A$ is decomposed into subactivities through an activity decomposition. Activity decomposition has to specify the way that the activity $A$ is decomposed into subactivities $A_1, \ldots, A_k$ and relationship among inputs and outputs of $A$ and $A_i$. So it takes the following form.

$$A(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

$$\Rightarrow A_1 \ldots A_k \text{ where } E$$

$E$ is a set of definitions of input attributes of $A_i$ and output attributes of $A$. This explicit definition of $E$ is sometimes omitted and use the following convention. When an output of $y$ of the activity $A_i$ is transferred as one of inputs $x$ of $A_j$, for example, we omit the definition $x = y$, and instead we put $y$ for $x$ in $A_j$. A pair of $A_1, \ldots, A_k$ and $E$ are called a decomposition and denoted by $D$ hereafter.

Usually, an activity decomposition is specified with a condition $C$ when this decomposition is applied, and the activity decomposition takes the following general form.

$$A(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

\begin{align*}
\text{when } & C_1 \Rightarrow D_1 \\
\vdots \\
\text{when } & C_n \Rightarrow D_n \\
\text{otherwise } & \Rightarrow D
\end{align*}

The condition $C_i$'s are tested sequentially and the decomposition $D_i$ is applied when $C_i$ holds. When none of $C_1, \ldots, C_n$ is satisfied, a default decomposition $D$ is selected.

### 2.2.2 Enaction Tree

The activity decomposition is applied until every activity is reduced to a primitive one. Thus, basically, enactment of HFSP could be looked as growing the tree of activity decomposition and we call this tree as an enaction tree. The following is an example HFSP script which describes a part of ISPW6 Example Problem, and Figure 1 illustrates the enaction tree corresponding to this script.

```plaintext
ModifyAndTest( changeReq | result.review, result.test )
=> ModifyAndReviewDesign( changeReq | design.out, result.review )
   ModifyTestPlans( changeReq | testplan.out )
   ModifyCodeAndTest( design.out, testplan.out | result.test )

ModifyAndReviewDesign( changeReq | design.out, result.review )
=> ModifyDesign( feedback, design.old | design.new )
   ReviewDesign( design.new | feedback, result )

ModifyCodeAndTest( design, testplan | result )
=> ModifyCode( design.in, code.old | code.new )
   ModifyUnitTestPackage( design.in, testpkg.old | testpkg.new )
   TestUnit( code.new, testpkg.new | result )
```

204
2.2.3 Persistent Object Access

Persistent object store is a part of the environment, inside which HFSP processes are embedded and work. Values of persistent objects are checked out and referred in an HFSP activity $A$ by simply writing them as inputs of $A$ with a special mark "*", and $A$’s output values could be checked into the objects by writing the object names as its outputs with the mark. Figure 2 illustrates the use of persistent objects.

```
ModifyCodeAndTest1( design | result )
  => ModifyCode( design.in, *code.old | *code.new )
  ModifyUnitTestPackage( design.in, *testpkg.old | *testpkg.new )
  TestUnit( code.new, testpkg.new | result )
```

Figure 2: Persistent Object Access

2.3 Behavioral Aspect

2.3.1 Ordering among Activities

In an activity decomposition, the order of subactivity invocation could be determined from attribute dependencies among subactivities. In the script shown at section 2.2.3, testpkg.new appears as an output of ModifyUnitTestPackage and an input of TestUnit. This dependency makes a partial
order between these two subactivities. Besides the implicit ordering derived from dependencies, HFSP has features explicitly ordering among activities such as strict sequencing. For example, we can enforce the enactment of $A_1$ before that of $A_2$ by adding the sequencing construct $\{\}$, where $\{A_1; A_2\}$.

2.3.2 Iteration

Iteration is a common and important operation in software process which is used for activities whose products are refined or produced incrementally. Though HFSP can express iteration by recursive activity decomposition, it has special forms of activity decomposition for directly expressing iterations. They are of the following form and decompose an activity into unspecified number of the same subactivities.

$$A \Rightarrow \text{repeat} \{A_1 \ldots A_k\} \text{ while } C \text{ where } E$$

$$A \Rightarrow \text{repeat} \{A_1 \ldots A_k\} \text{ until } C \text{ where } E$$

$A_1, \ldots, A_k$ are a set of activities iterated. $C$ represents the condition which terminates the iterations and $E$ is a collection of definitions for the initial value of input attributes in $A_1 \ldots A_k$ and the output attributes of original activity $A$. An example is shown in figure 3. In this example, the attribute design appears both the input and the output of the activity ModifyDesign. When the value of design.old is needed, it is automatically checked-out from objectbases and the modified version of the design will be checked-in as design.new. Through iterating this operation, the value of design might be refined and extracted as design.out at the end of iterations.

```
ModifyAndReviewDesign2( changeReq | design.out, result ) => repeat {
    ModifyDesign( feedback, *design.old | *design.new )
    ReviewDesign( design.new | feedback, result )
} until Approved( result )
where
    design.out = design.new
```

![Figure 3: Iterative Execution of Activities](image-url)
3 Meta-Operations

Though a straightforward aspect of process-centered approach can be described as an enaction tree in HFSP, more sophisticated devices are needed in describing real world processes which are flexible and dynamic. For example, when an execution of an activity is severely delayed from the plan, suspension of all currently enacted process may be required for re-scheduling, and creation and invocation of another process may be needed for compensating the delayed process.

To cope with this problem, meta-operations are introduced into HFSP, such as dynamically instantiating enaction trees from inside of an enaction tree, recording and manipulating the status of an enaction, suspending/resuming an enaction and sending/receiving information among enaction trees.

The unit of enaction, which is represented by an enaction tree, is called a process. In this view, the whole software process is represented by a set of processes which are dynamically created and interacting with each other.

A process communicates with outside of its enaction tree through meta-operations. Figure 4 illustrates what the meta-operations perform. When a meta-operation is detected during the enaction of a process, it is interpreted by an evaluator attached to the process. It causes such effects to the whole processes as creating a new process, controlling other process and sending/receiving messages.

1. Process Creation and Destruction
   A new process is created by @Create and restricted by @Destroy.
   @Create( A | p )
   Create a new process which has A as its initial activity and returns a process identifier p.
   @Destroy( p | )
   Kill the process p and remove it from the environment.

2. Process Enaction Control
   Process enacting is suspended by @Suspend and it is resumed by @Resume.
   @Suspend( p | t )
   Suspend the enaction of process p and returns the current p's enaction tree as t.
   @Resume( p, t | )
   Resume p's enaction from the state indicated by t.

3. Process Interactions
HFSP provides asynchronous communication by meta-operations \@Send and \@Receive.

\@Send( p, m | )
Send a message \(m\) to \(p\). Nothing returns.

\@Receive( | m )
Receive a message from the message queue of mine and stores it as \(m\).

4 Formal Description of Meta-Operations

In this section, we give a formal description of meta-operations.

4.1 Notations

(1) Classes and Instances

In the following definitions, every data object has a class to which it belongs. \(a : A\) denotes a data object \(a\) is a member of a class \(A\), and \(a\) is called an instance of \(A\).

(2) Powersets

A set of all subsets of a set \(A\) is called a powerset of \(A\) and denoted by \(\mathcal{P}(A)\). An element of \(\mathcal{P}(A)\) is denoted by \(\{a_1, \ldots, a_k\}\) where \(a_i \in A\).

(3) Sequences

A sequence whose elements are \(a_1, \ldots, a_n\) is denoted by \(<a_1, \ldots, a_n>\). The set of all sequences of elements from a set \(A\) is denoted by \(\text{seq}(A)\). Any sequence is convertible to a set. \(\text{ToSet}(s)\) which consists of all the different element of \(s\). \(\text{ToSet} : \text{seq}(A) \rightarrow \mathcal{P}(A)\).

(4) Tuples

A tuple whose components \(t_1, \ldots, t_n\) is denoted by \((t_1, \ldots, t_n)\). When \(t_i : T_i\), then \(t = (t_1, \ldots, t_n) : T_1 \times \ldots \times T_n\).

(5) Domain Restriction

When a function \(F : X \rightarrow Y\) is given, the function whose domain is restricted to \(R \subseteq X\) is denoted by \(R \circ F\). More precisely, \(R \circ F = \{x \mapsto y \mid x \in R, y = F(x)\}\).

(6) Function Overriding

For two function \(F_1, F_2\) whose domain type and range type are identical, the function in which the images by \(F_1\) of all elements which belong to the domain of \(F_2\) are overridden by the images by \(F_2\), is denoted by \(F_1 \oplus F_2\). More precisely, \(F_1 \oplus F_2 = (\text{dom} F_1 - \text{dom} F_2) \circ F_1) \cup F_2\).

4.2 Activity Definition

In HFSP, software processes are defined as a collection of activity definitions. Each of activity definition consists of a left hand side (lhs) and a right hand side (rhs) which represents a pattern of activity decomposition. Lhs consists of the name of activity itself and input/output attributes, and rhs consists of
a sequence represented by an activities and a collection of attribute equations. Each attribute equation represented by the name of attribute defined and an expression which denotes how the attribute is calculated.

**Definition 4.1** The class $\text{ACTDEF}$ of activity definitions is defined by

$$\text{ACTDEF} = \text{LHS} \times \text{COND} \times \text{RHS}$$

$$\text{LHS} = \text{ID} \times \text{seq} (\text{ATTR}) \times \text{seq} (\text{ATTR})$$

$$\text{RHS} = \text{seq} (\text{LHS}) \times \mathcal{P} (\text{ATTR} \times \text{Expr} (\text{ATTR}))$$

where $\text{COND}, \text{LHS}, \text{RHS}$ are the classes of activity decomposition condition, lhs of activity definition, rhs of activity definition respectively. $\text{ATTR}$ means a class of attribute symbols, and $\text{Expr} (\text{ATTR})$ denotes a class of expressions which is constructed from members of $\text{ATTR}$. $\text{COND}$ is a boolean expression of $\text{ATTR}$. Among them, $\text{ID}$ and $\text{ATTR}$ are primitives.

The following functions are defined for handling objects in these classes.

1. For each activity definition $\text{ade} = (l, c, r) : \text{ACTDEF}$,
   $$\text{LHSof}(\text{ade}) = l, \text{Cond}(\text{ade}) = c \text{ and } \text{RHSof}(\text{ade}) = r.$$  

2. For each decomposition $\text{r} = (d, e) : \text{RHS}$, $\text{Decomp}(\text{r}) = d$ and $\text{Equs}(\text{r}) = e$.

3. For each attribute definition $\text{eq} = (a, e) \in \text{Equs}(\text{r})$ where $a : \text{ATTR}$ is an attribute instance and $e$ is an expression, $\text{Args}(\text{eq})$ denotes a set of attribute instances appearing in $e$.

#### 4.3 Enaction Tree

An enaction tree represents the current status of an enaction of a single process. To formalize a tree structure, we use a tuple of the root node, a set of nodes, a mapping which represents children of each node, and a mapping from the node to their structure. Each node structure records its lhs, rhs and a mapping from attribute instances to their values. In order to denote nodes whose decomposition have not determined yet, a special rhs instance $\text{undef}$ is introduced.

**Definition 4.2** The class $\text{ET}$ of enaction trees is defined by

$$\text{ET} : N \times \mathcal{P} (N) \times (N \rightarrow \text{seq} (N)) \times (N \rightarrow \text{NODE})$$

$$\text{NODE} : \text{LHS} \times (\text{RHS} \cup \{\text{undef}\}) \times (\text{ATTR} \rightarrow \text{VAL})$$

where $N, \text{NODE}, \text{VAL}$ are the classes of node name, node structure and attribute values respectively.

The following functions are defined associated with the classes.

1. When $\text{et} = (r, \text{Nodes}, C, \text{NodeDefs}) : \text{ET}$ is given, following functions are used to get each elements of $\text{et}$:
   $$\text{Root}(\text{et}) = r, \text{Nodes}(\text{et}) = \text{Nodes}, \text{Child}(\text{et}) = C, \text{NodeDefs}(\text{et}) = \text{NodeDefs}.$$  

2. When $\text{node} = (\text{lhs}, \text{rhs}, \text{val}) : \text{NODE}$ is given and $(n \mapsto \text{node}) \in \text{NodeDefs}$, the following functions are used to get each elements:
   $$\text{LHSof}(n) = \text{lhs}, \text{RHSof}(n) = \text{rhs}, \text{Vals}(n) = \text{val}.$$  

3. A tree having only one node is often used. It is called a single node tree(SNT) and described by
   $$\text{SNT}(\text{node}) = (n_0, \{n_0\}, \phi, \{n_0 \mapsto \text{node}\})$$
   where $\text{node} : \text{NODE}$, $n_0$ is an arbitrarily chosen unique name which is attached to the node.
4.4 Operations on Enaction Trees

4.4.1 Attribute Calculation

Attribute values are calculated through the evaluation of attribute equations. When an attribute equation \((a, e)\) which is an element of \(\text{Equs}(r)\), is ready to evaluate, the attribute value \(v\) of \(a\) is computed according to \(e\) and a new correspondence \(a \mapsto v\) is added to the attribute value mapping of that node. This is characterized by a function \(\text{Assign}\).

**Definition 4.3** Let \(et = (r, Nodes, Child, NodeDefs)\), \(n \in Nodes\) and \(NodeDefs(n) = (lhs, rhs, vals)\). A function \(\text{Assign} : N \times ATTR \times VAL \times ET \rightarrow ET\) is defined by

\[
\text{Assign}(n, a, v, et) = SNT((lhs, rhs, vals \oplus \{a \mapsto v\}))
\]

4.4.2 Growth of Enaction Tree

An enaction tree grows whenever an activity decomposition is applied. Growth of the tree is achieved by an operation which substitute the node corresponding to the activity by a new enaction subtree. Substituting operation is performed by the following three steps: (1) pruning the node to be decomposed, (2) growing the node by applying the decomposition and making a subtree, and (3) grafting the subtree to the original tree.

**Definition 4.4** Let \(et = (r, Nodes, Child, NodeDefs)\) and \(NodeDefs(n) = (lhs, rhs, val)\). A function \(\text{prune} : N \times ET \rightarrow ET\) is defined by

\[
\text{prune}(n, et) = (r, Nodes, Child \oplus \{n \mapsto >\}, NodeDefs \oplus \{n \mapsto (lhs, undef, val)\})
\]

**Definition 4.5** Let \(et = (n, \{n\}, \phi, NodeDefs)\) and \(ets = < et_1, \ldots, et_k >\). A function \(\text{growtree} : ET \times \text{seq}(ET) \rightarrow ET\) is defined by

\[
\text{growtree}(et, ets) = (n, \{n\} \cup \text{ToSet}(Nodesof(ets)), \{n \mapsto Nodesof(ets)\}, NodeDefs \cup NodeDefsof(ets))
\]

where

\[
\begin{align*}
\text{Nodesof}(ets) &= < \text{Root}(et_1), \ldots, \text{Root}(et_k)> \\
\text{NodeDefsof}(ets) &= NodeDefs(et_1) \cup \ldots \cup NodeDefs(et_k)
\end{align*}
\]

**Definition 4.6** Let \(et_1 = (n_1, Nodes_1, Child_1, NodeDefs_1)\) and \(et_2 = (n_2, Nodes_2, Child_2, NodeDefs_2)\). A function \(\text{graft} : N \times ET \times ET \rightarrow ET\) representing a tree after grafting \(et_2\) below the node \(n\) of \(et_1\) is defined by

\[
\text{graft}(n_2, et_1, et_2) = (n_1, Nodes_1 \cup Nodes_2, Child_1 \oplus Child_2, NodeDefs_1 \oplus NodeDefs_2)
\]

**Definition 4.7** Activity decomposition is formalized as a function \(\text{Decompose} : N \times RHS \times ET \rightarrow ET\) defined by

\[
\text{Decompose}(n, rhs, et) = \text{growtree}(n, \text{instantiate}(\text{Decompof}(rhs))))
\]

where functions \(\text{instantiate}\) and \(\text{newtree}\) are given below:

For \(d = < a_1, \ldots, a_n >\),

\[
\begin{align*}
\text{instantiate}(d) &= < \text{newtree}(a_1), \ldots, \text{newtree}(a_n) > \\
\text{newtree}(a_i) &= SNT((a_i, undef, \phi))
\end{align*}
\]
4.5 Structure of Enaction Tree Evaluator

Enaction is considered as a process of transforming enaction trees by attribute calculation and decomposition, and could be formalized as a transform function $eval : N \times ET \rightarrow ET$. The whole evaluation of a single enaction unit is performed by repeated application of $eval$ to an initial SNT (Figure 5).

\[
\begin{align*}
\text{Decompose}(n_0) & \quad \text{(a)} \\
\text{Decompose}(n_1) & \quad \text{Assign}(n_2,y_2) \\
\text{(b)} & \quad \text{(c)} \quad \text{(d)} \\
\end{align*}
\]

Figure 5: Enaction Process

In this process, an evaluation of each node could be performed asynchronously and concurrently. Usually there might exist several nodes which can be decomposed or whose attributes can be calculated, concurrently, and the actual order of evaluation is nondeterministic. Consider an enaction tree (c). In this tree, decomposition of $n_1$ and calculation of the attribute $y_2$ can be done independently. The intermediate representation of enaction tree varies depending on which operation is performed first, though, the final enaction tree is identical.

For this reason, the whole operation $Eval$ of the evaluator is described as following procedure using parallel for which represents parallel nondeterministic operation for each elements of a list.

```plaintext
var
et, et' : ET, n : N
procedure Eval;
begin
repeat
parallel for $n \in \text{Nodes}(et)$ do
  eval($n, et, et'$)
end for;
until \{whole enaction finishes\}
end ;

procedure eval($n, et, \text{var } et'$);
begin
  if Decomposable($n$) then
    \{choose a rhs for decomposition\}
    $et' := \text{Decompose}(n, rhs, et)$
  end if;
parallel for $(a, e) \in \text{Equs}(RHSof(n))$ do
  if Evaluable($e$) then
    \{calculate v\}
    $et' := \text{Assign}(n, a, v, et)$
  end if
end for;
end if
end for
end;
```

Predicates $\text{Decomposable}(n)$ and $\text{Evaluable}(e)$ determine whether the node $n$ could be decomposed, and whether all attribute values appearing in $e$ are defined, respectively. The procedure $Eval$ terminates when no further decomposition and attribute calculation could be applied to any nodes.
4.6 Process Enaction Environment for Meta-Operations

Meta-operations are regarded as operations over multiple enaction trees. In order to formalize them, we have to define an enaction environment as a set of currently active processes. The meta-operations are formalized as functions on the environment. In this environment, a process consists of an enaction tree $et$ which represents the current enaction status, an evaluator which operates over the enaction tree $et$, a flag $f$ and a message queue $q$.

**Definition 4.8** Let $EV$ and $MES$ be classes of evaluators and messages. Then, a process class is defined by,

$$P = ET \times EV \times bool \times seq(MES)$$

and a environment class $W$ is defined by

$$W = P(P)$$

In the current version of HFSP, we have two evaluators, $Eval_{meta}$ and $Eval_{monitor}$. $Eval_{meta}$ is a standard one used in processes which are not monitored. $Eval_{monitor}$ is used for processes which are monitored by other process. $Eval_{monitor}$ is provided with a mechanism for notifying its operation to the monitoring process. Both evaluators are variants of $Eval$ defined in Section 4.4 and their details are given in the succeeding sections.

4.6.1 Process Creation and Destruction

A meta-operation $@create$ is defined by a function $Create$, given an enaction tree $ET$, an evaluator type $EV$ and an environment $W$, returns a process $p$ which has the given tree and evaluator, and a new environment containing $p$. $ev$ is a one of $EV = \{Eval_{meta}, Eval_{monitor}\}$. A function $Destroy$ simply erases process $p$ from the environment $W$.

**Definition 4.9**

$$Create : ET \times EV \times W \rightarrow W \times P$$

$Create(et, ev, w) = (w', p)$

where

$$p = (et, ev, true, <>), w' = w \cup \{p\}$$

$$Destroy : P \times W \rightarrow W$$

$Destroy(p, w) = w - \{p\}$

4.6.2 Process Enaction Control

In order to realize the suspension and resumption of enaction, the flag $f$ is used for prohibiting the change of enaction tree. Evaluator works that,

- if $f = true$, it operates normally.
- if $f = false$, it prohibits all operations which may cause a change of the enaction tree.

A function $Suspend$ is defined for changing a process $p$'s flag $f$ to $false$ for suspending the enaction of the process and returning the frozen enaction tree $et$. A function $Resume$ gets a tree $t$ and replace $p$'s enaction tree $et_{old}$ by $et$, and change $f$ to $true$. As the result, enaction is resumed from the state indicated by the tree.
Definition 4.10

\[ \text{Suspend: } P \times W \rightarrow W \times ET \]
\[ \text{Suspend}(p, w) = (w', 1) \]
where
\[ w' = w - \{p\} \cup \{p'\} \]
\[ p = (et, ev, true, q) \]
\[ p' = (et, ev, false, q) \]

\[ \text{Resume: } P \times ET \times W \rightarrow W \]
\[ \text{Resume}(p, et, w) = w' \]
where
\[ w' = w - \{p\} \cup \{p'\} \]
\[ p = (et_\text{old}, ev, false, q) \]
\[ p' = (et, ev, true, q) \]

4.7 Processes Interaction

The interaction between processes are performed through messages. Every process has a message queue in which messages are stored and process interact with each other through asynchronous communication realized by the message mechanism. A message is appended to receiver’s queue by \textit{Send}, and got from the top of the queue by \textit{Receive}.

Definition 4.11

\[ \text{Send: } P \times MES \times W \rightarrow W \]
\[ \text{Send}(p, m, w) = w' \]
where
\[ w' = w - \{p\} \cup \{p'\} \]
\[ p = (et, ev, f, <ms>) \]
\[ p' = (et, ev, f, <ms; m>) \]

\[ \text{Receive: } W \rightarrow W \times MES \]
\[ \text{Receive}(w) = (w', m) \]
where
\[ w' = w - \{p\} \cup \{p'\} \]
\[ p = (et, ev, f, <m; ms>) \]
\[ p' = (et, ev, f, <ms>) \]

\(<ms; m>\) denotes a queue such that \(m\) is appended after the tail of \(<ms>\)
\(<m; ms>\) denotes a queue such that \(m\) is appended before the head of \(<ms>\)

4.8 Evaluators with Meta-Operation Mechanism

To handle the meta-operations, we, first, have to extend the definition of the node class NODE to NODE\textsubscript{meta} and consider all enaction tree has nodes which belongs to this class.

\[ \text{NODE}_{\text{meta}} = (LHS \cup META) \times (RHS \cup \{\text{undef}\}) \times (\text{ATTR} \rightarrow \text{VAL}) \]

META is a special symbol class which denotes meta-operations.

\[ META = \{\text{@Create, @Destory, @Suspend, @Resume, @Send, @Receive}\} \]

Next, we change the evaluator \textit{Eval} into \textit{Eval}\textsubscript{meta} by replacing \textit{eval} procedure by \textit{eval}\textsubscript{meta}.

procedure \textit{eval}\textsubscript{meta}(n, et, et', var w);
begin
if LHSof(n) \in META then
\[ \text{case LHSof}(n) \text{of} \]
\[ @\text{Suspend} : \text{do.suspend(Processof}(n), et, w); \]
\[ \ldots \]
\[ \text{end case} \]
else
eval(n, et, et');
end if
end;
LHSof(Nodes(n)) ∈ META denotes the activity is meta-operation. A function Processof returns the target process id which recorded the meta-operation. A procedure do.suspension performs a suspension operation defined by function Suspend. The other meta-operations can be handled in the same way.

When flag = false, eval.meta prohibits all operations which cause change of the enaction tree by returning false whenever the predicate Decomposable(n) is called. Otherwise, Decomposable(n) returns same as eval shown in Section 4.5.

4.9 Monitoring

The evaluator changes enaction trees through enaction progresses. In order to monitor this, we have to expand the evaluator for notifying what operation is performed. The new evaluator Evalmonitor is obtained by replacing eval by evalmonitor which sends messages whenever the content of a node is changed by starting/finishing activity evaluation and attribute evaluation. Monitor is a constant and denotes the process identifier of the monitoring process.

procedure evalmonitor(n, et, w);
begin
  if domVals(n) = φ then
    do.send(monitor,"evaluation of n started", w)
  end if
  ...
  if Decomposable(n) then
    do.send(monitor,"n is decomposed", w);
    Decompose(n, rhs, et)
  end if
  parallel for r = (a, e) ∈ Equs(RHSof(n)) do
    if Evaluable(e) then
      Assign(n, a, v, et)
    end if
  end for;
  ...
  if domVals(n) ≥ Out(LHSof(n)) then
    do.send(monitor,"evaluation of n finished", w)
  end if
end;

A procedure do.send performs the operation defined by the function Send.

5 Examples of Meta-Operations in HFSP

Here, we show that meta-operations are useful for actual process description using ISPW6 Example Process[5].

5.1 ISPW6 Example Process

This example process contains a number of different types of process issues seen in real-world software processes. It consists of core part and several extensions. The core problem is scoped as a relatively confined portion of the software change process. It focuses on the designing, coding, unit testing, and
management of a rather localized change to a software system. In order to solve core problem, we have to describe several processes such as

1. Change a module according to a given requirement change.
2. Customize a generic change plan for the change.
3. Execute the customized plan.
4. Project manager monitors the process enactment.

(1) is a process performed by designers. As it does not involve any meta-operation and has no difficult problems concerning its behavior, we focus here only on (2),(3) and (4). The script for (1) is given in Section 2.2.

5.2 Project Manager’s Process

(2),(3) and (4) are performed by a project manager and expressed by the following script.

\[
\text{ProjectManager}( \text{changeReq} \mid \text{result} ) \\
\Rightarrow \text{ModifyProjectPlan}( \text{changeReq}, \text{plan.old} | \text{plan.new} ) \\
\text{AssignResources}( \text{plan.new}, \text{resources} | \text{plan.res} ) \\
@\text{Create}( \text{initialtree(plan.res)} | \text{process} ) \\
\text{MonitorProgress}( \text{plan.res}, \text{process} | \text{plan.out} )
\]

A plan and scheduling information of the change is represented by a data type \text{plan}. \text{ModifyProjectPlan} get a general project plan and adapt it to the requirement of change. \text{AssignResources} assigns resources such as designers or required time each activity. The designer’s process is created dynamically by the meta-operation \text{@Create} and it is referred by a process identifier \text{process}. The function \text{initialtree} returns the initial enaction tree whose root node is the initial activity of designer’s process embedded in \text{plan.res}.

![Figure 6: Creation of Designer’s Process](image)

5.3 Monitoring

The plan and the process identifier created by \text{AssignResources} are passed to the monitoring activity. Rescheduling or cancellation may happen during its enaction. If rescheduling happens, the modified project plan will be produced as an output \text{plan.out}.
The monitoring process MonitorProgress collects information, as message, about designer's process by the meta-operation @Receive.

\[
\text{MonitorProgress( plan.in, process | plan.out )} \\
= \rightarrow \text{repeat} \{ \\
\text{@Receive( | message.in )} \\
\text{monitor( message.in, *plan.in | message.out, *plan.out )} \\
\text{until bodyof( message ) == 'allFinished} \\
\}
\]

The message are received as message.in and passed to monitor which operates appropriate function according to the kind of the message. This operation sequence is performed repeatedly until a message 'allFinished, which means the completion of all activities, is received.

### 5.4 Rescheduling and Cancellation

Rescheduling and cancellation are handled by monitor.

\[
\text{monitor( process, message, plan.in | plan.out )} \\
\text{when kindof(message) == 'delayed} \\
\rightarrow \text{Analyze( message, plan.in | delay )} \\
\text{Control( process, delay | result )} \\
\text{where} \\
\text{plan.out = plan.in} \\
\text{otherwise} \\
\rightarrow \text{......} \\
\]

\[
\text{Control( process, delay | result )} \\
\text{case kind( delay ) == 'severe} \\
\rightarrow \text{@Suspend( process | tree )} \\
\text{DecisionByCCB( process, tree | command )} \\
\text{Recovery( process, command, tree | result )} \\
\text{otherwise} \\
\rightarrow \text{Reschedule( process | )} \\
\]

\[
\text{Recovery( process, command, tree | result )} \\
\text{case command == 'cancel} \\
\rightarrow \text{@Destroy( process | )} \\
\text{where} \\
\text{result = CanceledReport( process )} \\
\text{otherwise} \\
\rightarrow \text{@Resume( process, tree | )} \\
\text{where} \\
\text{result = ContinuedReport( process )} \\
\]

Assume that deviation from the schedule is notified to the monitor, which analyzes its effect. If it is too severe to recover, the enaction of designer's process is suspended by @Suspend operation.

The project manager requests CCB (Configuration Control Board) to decide whether the whole enaction is canceled or not. If CCB decides to cancel, project manager will abandon the designer's processes by @Destroy operation, otherwise, the enaction is resumed from the state indicated by tree.
6 Concluding Remarks

Meta-Operations required to handle the dynamics and flexibility of software processes have been introduced into the software process model HFSP. As the state of enactment in HFSP is represented by a tree of activity decomposition whose nodes are attributed by software artifacts and its progression is looked as growing the tree, we introduced a set of special operations which could create/destroy such trees, suspending/resuming their growth and communication among them. These operations are issued from inside the trees and the whole software process is considered as a set interacting trees.

We have applied our process model to describe the behavioral aspect of the ISPW6 example process and found that it enables clear and natural description of the process. Of course, we need much more experience to assess its effectiveness. Though we gave a naive view of how these processes are enacted, we have to study more about the structure of process environment and its mechanism. Meta-operations in distributed environment will be an interesting research topic.

References


