

# Process Inheritance and Instance Modification

Guangxin Yang

Bell-Labs Research, Lucent Technologies  
600 Mountain Ave., Murray Hill, NJ 07974 USA  
gxyang@acm.org

## ABSTRACT

Process technologies play an increasingly important role as the world is being digitalized in nearly every corner. The major obstacles to their massive deployment include reusability and adaptivity. This paper addresses the two crucial problems with one single solution: process inheritance. We discuss what process inheritance is, what mechanisms are needed to support it, and how it can be used to handle exceptions effectively. The ideas and mechanisms are implemented in the runtime system of a process language named P.

## Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Construct and Features – *inheritance*

**General Terms:** Languages

**Keywords:** Process Language, Inheritance, Dynamic Modification, P

## 1. INTRODUCTION

Process management investigates how to model <sup>[9,33]</sup>, enact <sup>[1,3]</sup>, analyze <sup>[12, 32]</sup>, and discover <sup>[34]</sup> complex cooperative tasks that usually consist of smaller steps, involve groups of geographically distributed people, and last as long as days, months, or even years. Examples include *online order processing, medical care, insurance claims, mortgage applications, publication reviews, software developments, etc.* The list goes on as the Internet spreads to nearly every corner on this planet. Process technologies aim at better control over and higher efficiency in carrying out these tasks by automating some, often mechanically repeated, steps, breaking down and managing their interdependencies, e.g. *sequencing, splitting, and joining steps*, and reengineering their overall structures. We shall call a task of this kind a (*cooperation*) *process* and a smaller step that contributes towards the accomplishment of the task an *activity*.

Despite the huge research and development efforts, process management is facing two crucial problems to gain its popularity. One of them is *reusability*, i.e. building new process definitions based on existing ones; the other is *flexibility*, i.e. handling flexibly various exceptions occurred during instance enactments. Numerous solutions have been proposed. This paper tackles the two challenges with one stone: process inheritance. Inheritance is a commonplace in today's object-oriented programming (OOP) <sup>[28]</sup>. Though several authors have advocated the idea of process inheritance <sup>[31, 22]</sup>, it is not available in any of today's process languages. We present how the

mechanisms supporting inheritance are implemented in a process language, *P*, which allows a process, or a *subprocess*, to share and reuse the definitions, e.g. *task decomposition, control and data flow specification*, of other processes, or *superprocesses*. A subprocess is a specialization of its superprocesses and can customize them by *adding new activities, or changing the control and data flows*. When an exception occurs during the enactment of a *P* process instance, we specialize this instance by creating a new process definition, which inherits its current one, and migrating the instance to the new definition. This way exception handling in *P* becomes a matter of specialization and can take the full advantage of inheritance.

## 1.1 Process Management

Process management related methodologies and systems have been investigated and developed under different banners, including *workflow management, business process reengineering, software process engineering*, etc. However, these artificially separated areas actually try to achieve similar goals and share certain basic approaches. We can classify existing work according to the following three related dimensions:

- *Process meta-models* define the basic facilities including *artifacts, agents, roles, transition rules, forms, etc.* and their interrelationships for modeling processes.
- *Process languages* provide language constructs for programming/defining processes based on certain meta-models.
- *Process runtime systems* implement the computing services for enacting instances of processes, usually by interpreting process programs written in process languages.

Numerous alternatives have been explored. At the *meta-model* dimension, there have been various paradigms, including *Petri-net* <sup>[33]</sup>, the *directed graph* <sup>[7, 31]</sup>, the *grammar* <sup>[16]</sup>, the *mathematical function* <sup>[20]</sup>, the *production rule* <sup>[25]</sup>, etc., whose constructs are usually given special semantics meanings. For example, in a Petri-net based meta-model, a Petri-net represents a process, a transition an activity, a place a pre/post-condition and/or a resource needed for an activity, and an arc a logical relationship and/or a flow of work. In a directed graph-based meta-model, a process takes the form of a directed graph, a node usually denotes an activity, while a directed edge denotes the control/data flow between two activities. Note that some early systems don't define a meta-model. Instead, process logic is hard coded in program logic. Because of the equivalence of their computing power, these process meta-models may have equivalent expressiveness.

At the *language* dimension, different constructs and formalisms have been developed for describing the details of meta-model entities and how they relate to each other in forming process programs. Some work bases its constructs on classical programming languages. For example, APPL/A <sup>[29]</sup> extends Ada with *relations, triggers, predicates, and composite statements for persistency, reaction, constraint, and integrity and consistency* respectively. Other examples in this direction include Ada95-based CSPL <sup>[6]</sup>, Prolog-based Merlin <sup>[25]</sup>, XML-based BPEL4WS <sup>[7]</sup> and WfMC PDL

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GROUP '03, November 9–12, 2003, Sanibel Island, Florida, USA.  
Copyright 2003 ACM 1-58113-693-5/03/0011...\$5.00.

<sup>[31]</sup>. Some work defines entirely new constructs. For example, Vortex <sup>[17]</sup> offers *modules*, each of which represents an activity and produces output values for one or more attributes and may have side effects. In this sense, it is very similar to the *activity construct* of HFSP <sup>[20]</sup>. Other non-text based examples include directed graph based VPL <sup>[27]</sup>, ADEPT <sup>[26]</sup>, Petri-net based SLANG <sup>[2]</sup>, tree-based Little-JIL <sup>[5]</sup>, which offer visual components as the basic constructs for defining process programs.

At the *runtime* dimension, process instance enactment services, e.g. *instance initialization and state transition, interaction with agents and external tools, evaluation of activity de/activation conditions*, have been implemented with many different approaches. In early systems where process logic is hard coded, these services may be implemented as integrated parts and usually not all of them are available. Obviously if process logic changes, the whole system needs to be recompiled. Process systems developed with APPL/A has a similar problem because APPL/A process programs are compiled into executables. Today's dominant approach is to abstract these services from application systems and implement them as stand alone process management systems (PMS), e.g. *general-purpose workflow management systems, software process environments*, which can manage a large number of instances, of either the same or different processes. Because process enactment services are usually built atop other basic services, e.g. *data management, object system*, another direction in building PMSes is to introduce into the underlying systems certain kinds of process supports. For example, [10] shows how triggers and transactions in an active database system can be used to support processes consisted of multiple activities. [23] presents that how two of Smalltalk's advanced futures, namely *continuation and future objects*, can be used to build workflow applications.

## 1.2 Adaptivity

Despite the huge amount of work on process technologies and many reportedly interesting stories on applying them, their massive deployment still seems far away. The primary reason roots in process technologies themselves, which get a bad fame in trying everything to reinforce strictly human activities with formal meta-model based rules. It is nice to base a technology on a solid mathematical foundation. It is also nice to help humans to finish their work accurately. However, humans are not machines. Machines can perform a task, e.g. *a computation based on certain algorithms*, repeatedly and strictly. Humans cannot. They tend to, or will be asked or forced to, alter their ways to do things. If a PMS cannot adapt to these changes, or exceptions, sooner or later, it will be bypassed and abandoned. This is exactly the root of the problems of today's process technologies, as have been observed by numerous authors <sup>[14, 21, 32]</sup>. The process community offers *adaptive process (workflow)* as a solution, with an aim to remove the rigidity of PMSes so that they can handle more flexibly various changes and exceptions during instance enactment.

Solutions for adaptivity have also been developed at each of the three aforementioned dimensions. At the *meta-model* dimension, for example, [32] argues that "*workflows (processes) should not be driven by pre-specified control flows but by the products they generate*" and develops a new process meta-model, *PDCH*, which gives a central role to data objects generated during instance enactments. With *GPSG* <sup>[16]</sup>, processes are defined as grammar rules, from which process instances can be derived as *sentences*. This allows a theoretically infinite number of execution alternatives. More recently, [19] proposes to use interaction as a framework for process modeling to achieve better flexibility. The trend seems to be that the strict dependency among activities should be loosened or removed.

At the *language* dimension, new constructs and facilities are introduced to enable more dynamic and flexible process definitions. For example, HFSP defines a special data type, *status*, to represent the status of an instance and two operations, *snap* and *resume*, to record and change the status <sup>[20]</sup>. This allows some kind of control over process instance from within a process definition. [21] develops a knowledge base of exception descriptions that can be incorporated into process definitions as advanced plans for handling exceptions. [11] defines for specifying process change and change policy a language, *ML-DEWS*. Introducing runtime information into process modeling seems to be another important requirement.

At the *runtime* dimension, existing work has investigated issues related to migrating a bunch of running instances to a new process definition <sup>[12, 31]</sup> and dynamic modifications to individual running instances <sup>[3, 26]</sup>. Runtime support is extremely important because there are always some exceptions that cannot be planned in advance, no matter how flexible process meta-models are and/or how careful process designers/ programmers are. A Gartner report dated Jan. 2003 <sup>[13]</sup> predicts that "business and workflow logic will increasingly be represented as explicit and more easily modified knowledge." To accomplish this, we need more advanced runtime support, which generally requires that the runtime control structures for process instances be more easily reconfigured.

## 2. AN OVERVIEW OF P

P is designed for developing cooperative applications <sup>[40]</sup>. The strength lies in its distinctions at the three dimensions. Its meta-model relaxes activity dependency by making a process a *set* of activities that communicate with each other by message exchanges, which are triggered by various events occurred during instance enactment. Its language offers constructs that enable accessing from within process definitions the runtime information about process and activity instances and the runtime services for instance enactment. Its runtime system represents process instances in a format that enables process-level inheritance and easy reconfiguration. This section describes these distinctions briefly.

### 2.1 The P Process Meta-Model

Figure 1 shows the relationship among the meta-model's elements, which include *process, activity, class, member, and message exchange* definitions, as well as *process and activity instances, activity objects, and events*. A process definition contains multiple activity definitions, each of which has a class designation that defines the semantics of the activity object, which records the product of the activity; a set of member definitions that defines the participants of the activity; and a set of message exchange definitions that defines how this activity communicates with other activities for control and data flows.

Parallel with *process, activity, and class, ProcessInstance, ActivityInstance, and ActivityObject*, record dynamically generated information, e.g. *when an instance was created, what is its current state, who actually participated an activity instance, what has been produced so far*, and provide the interfaces for accessing this information and manipulating instances, e.g. *retrieving or changing the state of an instance, modifying process or activity definitions*. They are created the first time they receive a message exchange.

Events occur at specific points during instance enactment and can trigger message exchanges. Currently we identify four types of events, *process or activity instance state change related*, which occur whenever the state of an process or activity instance changes; *operation related*, which occurs whenever a specific operation is performed on an activity object; and *time-related*, which occur once at specific time points or periodically at specific time intervals.

Message exchanges establish among multiple activities different interconnections, via which activities are glued together to form a process. Responding to a specific event, a message exchange originates from within an activity and carries certain information, which is usually generated based on the information known to the originating activity, to a destination activity, which processes what it receives, in a way it prefers, to change its state. Because the meta-model includes runtime control structures, message exchange in P can achieve both control and data flows. At the same time, because message exchanges to different destination activities can be generated simultaneously when an event occurs, different routings among activities can be achieved flexibly.

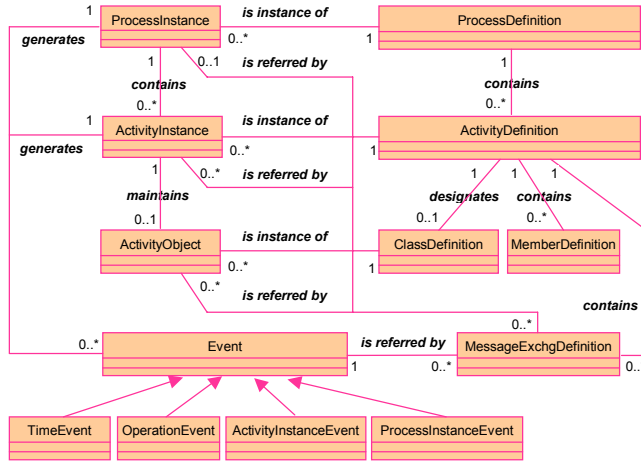


Figure 1. Relationship among elements of P's process meta-model.

We define some notations to facilitate the discussions that follow:

- $P$  a process definition
- $\tilde{P}$  a process instance
- $e$  an event
- $c$  a class definition
- $t$  a message exchange specification
- $A$  an activity definition
- $\tilde{A}$  an activity instance
- $o$  an activity object
- $u$  a member name

Based on these notations, we can refine  $P$  as  $\{A \mid A = \langle c, U, T \rangle$ , where  $U = \{u_1, \dots, u_m\}$ ,  $T = \{t_1, \dots, t_n\}$  and  $\tilde{P}$  as  $\{\tilde{A} \mid \tilde{A} = \langle \tilde{A}, o \rangle\}$ . Each process instance is assigned an integer as its unique identifier. For an instance of process  $P$  with an identifier  $id$ , we usually denote it as  $\tilde{P}^{id}_P$ . An activity instance in  $\tilde{P}^{id}_P$  is more specifically denoted as  $\tilde{A}^{id}_{P::A}$ . Its activity object is usually denoted as  $o^{id}_{P::A}$ , which is allocated and maintained in a P interpreter's object heap and assigned a unique identifier within range 0 to  $2^{31}-1$ .

## 2.2 The P Process Language

P defines the language constructs for creating process definitions based on the above meta-model. More specifically, it has:

- A **class** construct for class definitions. The *class* construct is a full-featured object-oriented language (OOL) with support to collection, such as **set**, **bag**, **array**, **list**, and **dictionary**.
- An **activity** construct for P activity definitions. Each activity can produce an object of a certain class.
- A **member** construct for defining expected activity participants as expressions evaluated to string values.
- A **process** construct for process definitions, each of which can have an arbitrary number of activities that can establish interconnections with each other in a certain way.

- A **trigger** construct for message exchange definitions. A *trigger* is part of an activity and takes the form of an *event-condition-action* rule developed in the Active Database (ADB) community [24]. However, in P triggers, all the four types of events are related to instance enactment. Conditions and actions are expressions based on invocations to methods of activity objects, process and activity instances, and configurable global objects.

The P language specification is provided in [37]. Figure 2 shows a simple P process definition. For simplicity, we omit such details as the definitions of classes CDocument and CReview. The process, PApplication, has two activities, AProposal and AReview, which produce a CDocument and a CReview object, and starts at AProposal. The trigger TProposalSubmit in AProposal states that after this activity is submitted, the generated document will be visible to participants of activity AReview, which will then be activated so that a review can be generated. The trigger TReviewSubmit in AReview states that after a review is generated, the summary is fed back to the participants of the proposal activity via email and if the result is negative, activity AProposal is restarted so that the object can be modified and resubmitted. Neither trigger has a condition.

```

1  class CDocument { ... } //details omitted
2  class CReview { ... } //details omitted
3  process PApplication start AProposal
4  {
5    //activity to produce a CDocument object
6    public activity AProposal produce CDocument
7    { //define a trigger named TProposalSubmit
8      trigger TProposalSubmit as //name
9        AReview.activity.Resume() //action
10     after submit activity; //event
11   }
12   //activity to produce an CReview object
13   protected activity AReview produce CReview
14   { //define a trigger named TReviewSubmit
15     trigger TReviewSubmit as
16       email.SendMessage( //action# 1
17         AProposal.activity.GetParticipants(),
18         "Review Result for" +
19         AProposal.GetName(), //subject
20         AReview.GetReviewSummary() ),
21       AReview.Approved() ? //action# 2
22       AProposal.SetApproval():
23       AProposal.activity.Resume()
24     after submit activity; //event
25   }
26 }

```

Figure 2. A simple review process in P.

The example also shows how to access activity objects and activity instances. In the two trigger definitions, AProposal and AReview denote the activity objects, which are instances of CDocument and CReview. AReview.GetReviewSummary() is a method call to the activity object AReview. The keyword **activity**, when used after the name of an activity, represents the corresponding activity instance. AReview.activity.Resume() represents an action to activate the activity instance AReview. The keyword **process**, not shown here, represents the corresponding process instance and can be used in any expression of an activity definition. This way runtime information and services are accessible from within process programs. email is a system-wide global object that can be used in any P program. At this point, we can ignore the access modifiers **public** and **protected**.

## 2.3 The P Runtime System

The runtime system's major work is to create and maintain the control structures for process instances; to monitor various events occurred within the context of an instance; and to execute triggers to push instance state changes. In our current implementation, a centralized process server, a P server, is responsible for all these tasks. It keeps the compiled P process programs and the persistent states of instance control structures. Currently, the server is implemented with C++ and runs on Win32 platforms.

### 2.3.1 Supporting Classes for Processes

At the center of our solution is the *supporting class* for a process, which is an ordinary P class and describes the internal control structure for a process instance. It defines member variables representing a process instance and its activity instances and activity objects. For example, Figure 3 shows the supporting class for process PApplication, which has five member variables. The 2<sup>nd</sup> and the 4<sup>th</sup> ones represent the activity objects produced and maintained by activities AProposal and AReview. The 1<sup>st</sup> member, **process**, represents the process instance and has type CProcessInstance, which is a P class that defines the interface for retrieving information about and manipulating a process instance. For example, we can get the time the instance was created or ended, its current state, the name of the process definition, etc.; we can also change its state forcefully by calling one of the four methods, i.e. Abort, Terminate, Resume, and Suspend; we can modify the process definition by calling ModifyDefinition. Similarly, the 3<sup>rd</sup> and 5<sup>th</sup> variables represent the activity instance AProposal and AReview respectively. In trigger definitions, references like AProposal.**activity** or AReview.**activity** is changed by a P compiler to AProposal.**activity** and AReview.**activity** respectively. Detailed definitions of CProcessInstance and CActivityInstance can be found in [37].

```

1 //supporting class for process PApplication
2 class PApplication_class4process
3 { //the process instance
4     public CProcessInstance process;
5     //the activity object of AProposal
6     public CDocument AProposal;
7     //the activity instance of AProposal
8     public CActivityInstance AProposal_activity;
9     //the activity object of AReview
10    protected CReview AReview;
11    //the activity instance of AReview
12    protected CActivityInstance AReview_activity;
13 }

```

Figure 3. The supporting class for process PApplication.

### 2.3.2 Process Objects

According to the supporting class definition, when initializing a process instance, a P server allocates two objects for it: one is a C++ object, which resides in the server's memory space and records the instance's runtime information; the other one is a P object, which is an instance of the supporting class and resides in the P interpreter's object heap. We call the P object a *process object* and denoted it as  $O_{PApplication}^d$ . Fields of a process object is filled with appropriate values as the instance is enacted. Figure 4 shows one state of  $O_{PApplication}^0$ , for process instance  $\tilde{P}_{PApplication}^0$ . Each cell represents a 4-byte value and corresponds to a field of the supporting class. Initially, only the first cell, representing the field **process**, is assigned a value converted from the pointer to the C++ object representing  $\tilde{P}_{PApplication}^0$ . (We signify the converted value with a black background.) Similarly when an activity instance is initialized, a P object, i.e. its activity object, and a C++ object, i.e. the activity instance, will be allocated. Its two

corresponding fields in the process object are assigned the identifier of the activity object and the converted C++ object pointer.

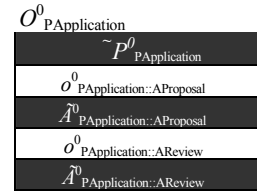


Figure 4. The memory structure and state of  $O_{PApplication}^0$  after both activity instances are created.

The conversion is done by changing the highest bit of the C++ object pointer from 0 to 1, which, on Win32 platforms, makes the converted value fall within  $2^{31}$  to  $2^{32}-1$ . The purpose is to allow the P interpreter to differentiate to which type of object a method call is made when an expression is evaluated. If the object is a P object, the method call is executed by interpreting compiled P code. Otherwise, it is redirected to the corresponding C++ object. More details about the redirection are found in [37].

### 2.3.3 Executing Triggers

The P Server monitors various events that occur during the enactment of a process instance. Each event has a distribution range. Events like *suspending/terminating/resuming/aborting* a process instance will be distributed to all activity instances of that process instance; *time-related* or *activity instance* related events are restricted within a specific activity instance. Upon the occurrence of an event in an activity instance, the trigger list of its activity definition is searched. If there is a match, the trigger condition, if specified, will be evaluated first. If the result is **true**, the trigger action is executed. We are especially interested in how trigger conditions are evaluated and trigger actions are executed.

The solution is a *temporary method* added to the supporting class. The P compiler ensures that the condition expression evaluates to a **boolean** value. To evaluate it, a temporary method, which has a **boolean** return value and takes a form as the following P source code:

```

public boolean TempMethod4TriggerCondition()
{
    return TriggerConditionExpression;
}

```

is added to the supporting class and is executed. If the return value is **true** (for triggers that don't have a condition expression, the return value is always **true**), another temporary method is added to the supporting class and executed. For example, after the activity AProposal is submitted, the supporting class will be added a temporary method shown in Figure 5:

```

1 class PApplication_class4process
2 {
3     ... //member variables same as these in Figure 3
4     public void TempMethod4TriggerAction()
5     { //trigger action expression
6         AReview.activity.Resume();
7     }
8 }

```

Figure 5. A temporary method is added to the supporting class for executing trigger actions.

A temporary method is removed from the supporting class definition once its execution is finished. The same mechanism is used for evaluating other expressions such as member definitions.

### 3. PROCESS INHERITANCE

The features and mechanisms outlined above have made P capable of modeling a wide range of cooperative applications [36]. So why inheritance? The answer lies in the similarities and distinctions among processes. For example, the review process defined in Figure 2 actually abstracts certain common characteristics of many processes, including *paper reviews*, *mortgage applications*, *online orderings*, etc, in each of which a document is produced first and then reviewed, after which a result is generated. However, they do have some significant differences. In a paper review process, the document is a paper, perhaps with some standard format; the result typically includes an acceptance or rejection decision and some review comments. In a mortgage application process, the document is an application form; the result may include whether the application is approved or rejected and some detailed information regarding the approval or rejection. In an online ordering process, the document is an order form, which may include a number of fields; the result may be a message saying that the good has been shipped. The reviews, of course, are done in very different ways. These similarities and differences suggest that inheritance is potentially a very good feature for process programming.

#### 3.1 Process Inheritance

We define *process inheritance* as a mechanism that allows a process A to inherit, or reuse without redefinition, the properties, e.g. activity definitions, of another process B. Seemingly, this definition looks like a duplication of what inheritance means in OOP, where it is usually defined as a mechanism that allows a class A to inherit the properties, e.g. attributes and methods, of another class B [28]. Indeed, process inheritance and class inheritance are parallel concepts. They have quite similar formalism and purpose. However, they differ entirely in their contents, i.e. *what is inherited* and *where they are used*, and implementation mechanisms, as we will discuss later. We call process B the *superprocess* of process A, the *subprocess*. Currently, we support only single inheritance, where a subprocess can have only one direct superprocess. We sometimes call an activity in a subprocess a *subprocess activity*, and an activity in a superprocess a *superprocess activity*.

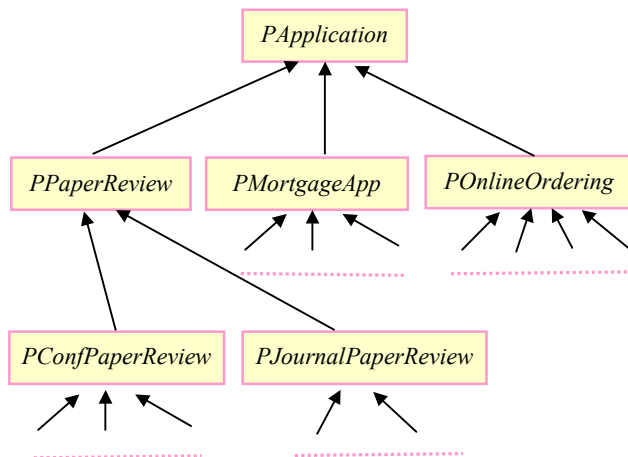


Figure 6. Processes organized in an inheritance tree.

Process inheritance introduces a new dimension to process modeling, in addition to what process meta-models provide. New processes can be defined based on existing process definitions as well as the facilities of process meta-models. In this way, process definitions can be organized in inheritance trees, where a parent node represents a more general process and a child node represent a more specific one.

For example, according to earlier discussion, we can define the paper review, the mortgage application, and the online ordering processes as subprocesses of PApplication. Based on them, more specialized processes may be defined. For example, we can define a conference paper review process and a journal paper review process based on the paper review process. The inheritance tree is shown in Figure 6. Note that this is just an example. Certainly there are other ways to construct inheritance trees and this is basically a choice of programmers.

In P, the inheritance relationship is expressed by the keyword **extend**. For example, we may define PPaperReview as follows:

```

process PPaperReview extend PApplication
{
    ... //subprocess codes go here
}
  
```

#### 3.2 Process Specialization

Process inheritance offers a great way to reuse existing process definitions. However, its power will be seriously compromised if a subprocess cannot specialize its superprocesses flexibly to address the differences among processes. With OOP, we can specialize a class by defining new member variables and methods, or redefining its member variables or methods in a subclass. Similarly, with P, we can specialize a P process definition in the following ways at the language level:

- *Adding new activity definitions* that describe how tasks specific to a more specialized process can be finished. For example, in a mortgage application process, we may need to add a new activity to do a credit check for the applicant. In an online ordering process, an activity that bills the customer may be added. This may be the most common specialization we need to do in subprocesses.
- *Extending a superprocess activity by defining new triggers or members*. This is useful because on one hand, a superprocess activity may not define the desired member or the trigger to respond to an event that is of interest to a subprocess; on the other hand, a superprocess activity may need to communicate with an activity that is newly introduced to a subprocess. For example, if we add an credit check activity to a mortgage application process, we need to extend the proposal activity by adding a new trigger so that after the proposal (application) is finished, the credit check activity is activated and passed with necessary parameters such as the applicant's social security number.
- *Enabling/disabling a trigger*. Triggers service as the communication channels among activities, which may need to be shutdown in some cases and brought up in others. The action of a disabled trigger will not be executed, even if its condition expression can evaluate to *true*. For example, if we want to rule out an activity, we can disable all triggers in other activities that refer that activity so that it will never get any input thus will never be activated. Enabling a trigger will pull it back to be effective.
- *Changing the class of a superprocess activity object* so that the semantics of the object can be more specific. For example, in a mortgage application process, we can change the class of the proposal activity from a general class CDocument to a more specific one, say CMortgageApplicationInfo, that describes the contents of the application. Note that we require the new class to be a subclass of the original one (a P compiler will check this). The purpose is to maintain the correctness of the triggers in superprocess activities that refer the activity object. If arbitrary change is allowed, the new activity object may not comply with the interfaces of the original one, which will disable the triggers that refer them. Obviously this is not desirable.

### 3.2.1 An Example

A formal specification of the semantics of these specializations is beyond the scope of this paper. As an example, we give in Figure 7 the definition of process `PPaperReview`, which inherits process `PApplication` and specializes both of its two activities. Firstly, we define a subclass, `CPaper` (line 1), of class `CDocument`, and use it as the class of activity `AProposal` (line 5). Therefore, for an instance of `PPaperReview`, instead of producing a `CDocument` object, the `AProposal` activity produces a `CPaper` object, which may have additional methods (such as `GetKeywords()` used in line 12) that can be used in trigger or participant definitions. Secondly, activity `AReview` is extended by adding a member definition, which specifies a set of expected participants according to the keywords of the `CPaper` object, and a trigger `TNotifyReviewers`, which automatically sends to the expected reviewers an email notification that a paper is ready for their review once the `AReview` activity is activated. The activity modifier **extending** (line 5 and 9) indicates that definitions in the subprocess activity will be merged with these in the corresponding superprocess activity. If we don't want the merging, **overriding** can be used instead, in which case all definitions in the overridden subprocess activity will not be used.

```

1  class CPaper extend CDocument { ... }
2  process PPaperReview extend PApplication
3  {
4      //changing the class of the activity object
5      extending activity AProposal produce CPaper
6      {
7      }
8      //adding new trigger and participant defs
9      extending activity AReview produce CReview
10     {
11         member reviewerdb.GetReviewers(
12             AProposal.GetKeywords());
13         trigger TNotifyReviewers as //header
14             email.SendMessage( //action
15                 reviewerdb.GetReviewers(
16                     AProposal.GetKeywords()),
17                 "Paper for your review",
18                 AProposal.GetPaperURL())
19         after resume activity; //event
20     }
21 }

```

Figure 7. With inheritance, process `PPaperReview` can base on another one, i.e. `PApplication`, and specialize it as needed.

## 3.3 Supporting Inheritance

Two problems need to be solved to support inheritance in the runtime system. The first one is how to represent an instance of a subprocess. The second one is how expressions, such as trigger conditions and actions, member expressions, can be correctly executed, regardless of whether they are defined in the subprocess or superprocess. The solutions are tightly related.

### 3.3.1 Supporting Classes for Subprocesses

The solution is again based on the supporting class for a subprocess, which will be a subclass of the supporting class for its direct superprocess. For example, the supporting class for `PPaperReview` is shown in Figure 8. Here we may note two major changes. The first one is that there is no such a member variable named **process**. The reason is that this variable represents the most specific process instance. Since it is already defined in the supporting class for

`PApplication`, we don't need to define a duplicated one in the subclass to represent the same thing. The second change is the type of the member variable `AProposal` (line 5), which is now `CPaper`, rather than `CDocument`. This reflects the change to activity `AProposal`'s definition at line 5 in Figure 7. Each member variable pair in the subclass corresponds to an activity in the subprocess itself. For example, if activity `AReview` is not customized in `PPaperReview`, there will be no such variables as `AReview` and `AReview_activity`.

```

1  //supporting class for process PPaperReview
2  class PPaperReview_class4process extend
3      PApplication_class4process
4  { //the activity object for AProposal
5      public CPaper AProposal;
6      //the activity instance for AProposal
7      public CActivityInstance AProposal_activity;
8      //the activity object for AReview
9      protected CReview AReview;
10     //the activity instance for AReview
11     protected CActivityInstance AReview_activity;
12 }

```

Figure 8. The supporting class for `PPaperReview` is a subclass of that for its superprocess `PApplication`.

### 3.3.2 Process Objects for Subprocess Instances

According to the inheritance hierarchy of the supporting classes, the logical structure of the process object for an instance of a subprocess now changes from a single memory block, as the one shown in Figure 4, to a chain of memory blocks, each of which represents a node in the inheritance hierarchy. For example, after both activities are activated, the process object for an instance  $\tilde{P}^1_{PPaperReview}$  is shown in Figure 9. Here we have two nodes in the chain, which is formed by recording in the first four bytes of the process object for a subprocess (in this case `PPaperReview`) the identifier of the process object for its direct superprocess (in this case `PApplication`). Note that each block contains only variables defined in the corresponding supporting class itself, though in our example, both classes happen to have variables for the two activities.

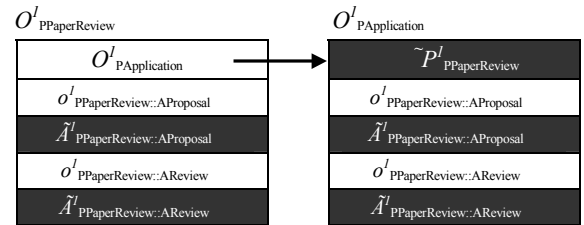


Figure 9. The memory structure of the process object for a `PPaperReview` instance.

We may notice that each pair of corresponding variables in  $O^I_{PPaperReview}$  and  $O^I_{PApplication}$  has the same value. This is a deliberate setup, which is based on our design strategy that specializing an superprocess activity should not introduce any new activity to the process. Therefore, all activities with the same name at different levels of a process inheritance hierarchy represent the same activity. Therefore, an instance of `PPaperReview` can have at most two activities, not four. Consequently the variables of the supporting classes representing the activity instance or activity object should have a same value. This setup is done by a simple deep-first-search along the chain (a single branch tree in the case of single inheritance) after an activity instance is activated, at which time the instance object and activity object will be allocated.

Note that in classical OOP, member variables with a same name in a superclass and a subclass usually have different values.

### 3.3.3 Executing Triggers and Other Expressions

Similar to no inheritance cases, trigger conditions and actions and other expressions are evaluated by adding temporary methods to the supporting class of subprocesses. The way process objects are setup is the key for evaluating these expressions correctly. This is because the setup ensures that a variable referred in an expression, whether it is defined in a superprocess or subprocess activity, can be resolved to the right object. Take PApplication in Figure 2 as an example. Consider the variable AProposal used in the second action of trigger TRReviewSubmit. After compilation, the binary code records this is a reference to the 2<sup>nd</sup> variable (indexed as 2) of the supporting class. When this trigger is executed in the context of a PPaperReview instance, the reference will be resolved to the second variable in  $O^j_{PApplication}$  (the first variable of  $O^j_{PPaperReview}$  is indexed as 5 by the P compiler), which is  $o^j_{PPaperReview::AProposal}$ , a CPaper object and exactly the right object on which the trigger action should be executed. Since CPaper is a subclass of CDocument (checked by a P compiler), any invocation on AProposal in the superprocess can be corrected executed.

### 3.4 Activity Accessibility

We now turn to the activity modifiers such as **public** and **protected** we have ignored in Section 2.2. Process inheritance raises the issue of activity accessibility, i.e. whether a subprocess can communicate with the activities in its superprocesses. For example, a process programmer may want an activity to be used for a completely internal purpose and doesn't want any activity in a subprocess to communicate with it. P uses a mechanism that is very similar to how some OOLs like C++ and Java restrict the visibility of class variables and methods. It defines three accessibility modifiers, i.e. **private**, **protected**, and **public**. A **private** activity is only accessible by the triggers, member expressions, etc. of the activities in the process itself. A **protected** activity can be accessed by activities in the same process and its subprocesses. Although not discussed in any detail here, P supports defining an activity to be a nested process, just as most OOL allows a member variable to have a class type. Only **public** activities of a nested process can be accessed by other activities of the nesting process.

Activity accessibility in P is a parallel concept of member/field accessibility in classical OOLs. As we have already seen, access modifiers on the member variables of supporting classes are the same as those on activity definitions. A P compiler ensures the accessibility rules when P programs are compiled.

## 4. INSTANCE MODIFICATION

We now take a look at the dynamic modification challenge we discussed at the beginning. As we have mentioned, a subprocess customized its superprocesses to describe a more specialized situation. An exception occurring in a process instance means the instance needs to be specialized to cope with a more special situation. The similarity implies that process inheritance can also be effective for handling exceptions. However, there are some significant differences on how this mechanism is used. Creating a process with inheritance is still an in-advance planning: We know the special case and how to model it. However, using inheritance to handle exceptions is an ad-hoc processing: We don't know which exceptions and when they will happen. As a result, enacting an instance of a subprocess is quite different from modifying an instance: In the former case, control structures like process objects can be allocated in advance when instances are initialized; in the latter case, everything needs to be built up little by little.

This section discusses how we dynamically modify an instance running on a P server. As we will see, the major work is to reconfigure instance control structures based on the mechanisms that implement process inheritance. To illustrate our ideas, we begin with instance  $\tilde{P}^0_{PApplication}$ , with an initial process object shown in Figure 4. We do two representative dynamic modifications to make it an equivalent to an instance of process PPaperReview. We may simply denote the instance as  $\tilde{P}^0$ .

### 4.1 Changing the Class of Activity AProposal

Changing the class of an activity addresses the need to specialize the semantics of the result, i.e. the activity object, produced by an activity. This may happen if during the enactment of an instance, the participants of an activity find that they need to add more information to the activity result and/or introduce new ways to operate the result. In these cases, they may create a subclass derived from the current class of the activity object and use a function in P Client API to make the modification. The required parameters include the *instance identifier*, the *name of the activity whose class is to be changed*, and the *name of the new class*. We assume that the compiled form of the new class has been saved on the P Server where  $\tilde{P}^0$  is running and that this is the first modification to it.

The P Server needs several steps to finish this task. The first step is to construct based on the supplied parameters a new process definition, which is a subprocess of PApplication and whose source code is shown below:

```

1  process PApplication_0 extend PApplication
2  { //changing the class of the activity object
3      extending activity AProposal handle CPaper
4      {
5      }
6  }
```

The source is compiled and if there is no error (the only possible error here is that CPaper is not a subclass of CDocument), a new process named PApplication\_0 and its supporting class PApplication\_0\_class4process will be generated. The modified instance will be an instance of this new process.

The major work of the second step is to migrate the control structures for  $\tilde{P}^0$  to a new one that complies with the new process definition. One option to do this is to allocate a new set of control structures, initialize them with the values in the existing ones, which are then released. Obviously this is costly. We choose another option that reuses existing structures. We create a new C++ process instance object, denoted as  $\tilde{P}^0_{PApplication_0}$  and set its superprocess instance pointer to  $\tilde{P}^0_{PApplication}$ . This way the two instance objects are chained as a list. We also add to  $\tilde{P}^0_{PApplication_0}$ 's activity instance set the pointer to activity instance  $\tilde{A}^0_{AProposal}$ . Accordingly, an object of class PApplication\_0\_class4process,  $O^0_{PApplication_0}$ , will be created. However, we don't allocate a new P object for its superclass. Instead, the ID of  $O^0_{PApplication}$  is recorded in the first four bytes of  $O^0_{PApplication_0}$ . This way it becomes a chained list with two nodes, reflecting the fact that process PApplication\_0 inherits PApplication.

However, there are some changes to  $O^0_{PApplication}$ . The first one is that its first four bytes, which originally record the converted pointer to  $\tilde{P}^0_{PApplication}$ , will be changed to the converted pointer to  $\tilde{P}^0_{PApplication_0}$ . Therefore, all references to **process** will be resolved to the newly created process instance. The second change is made to the second four bytes, which represents PApplication\_\_class4process::AProposal and originally records the ID of  $o^0_{PApplication::AProposal}$ , a CDocument object. Since the class of activity AProposal is changed to CPaper, we need to

construct a CPaper object and replace the CDocument object with it. The construction is similar to that of  $O^0_{PApplication\_0}$ . We only allocate the memory blocks for classes along the inheritance hierarchy from CPaper up to the direct subclass of CDocument, the first four bytes of whose memory block will record the ID of  $o^0_{PApplication::AProposal}$ . This way the existing activity object of AProposal is preserved<sup>1</sup>. The ID of the new activity object, denoted as  $o^0_{PApplication\_0::AProposal}$ , is written in both  $O^0_{PApplication\_0}$  and  $O^0_{PApplication}$ . Accordingly, the activity object ID recorded in  $\tilde{A}^0_{PApplication::AProposal}$  will be updated to that of  $o^0_{PApplication\_0::AProposal}$ , to represent the new activity instance.

After these reconfigurations, the process object for the modified instance will be the one shown in Figure 10. Note that the identifier of the instance remains unchanged, however, its process definition has been changed from PApplication to PApplication\_0.

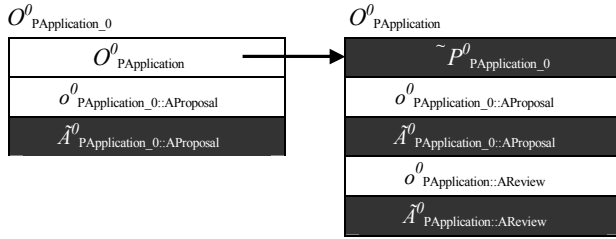


Figure 10.  $O^0$  after the class of activity *AProposal* is changed.

## 4.2 Adding a New Trigger to *AReview*

We further discuss how to add a trigger to activity *AReview* of  $\tilde{P}^0_{PApplication\_0}$  so that when this activity is activated, review notices are sent to selected reviewers via email. The required parameters are the *instance identifier*, the *activity name where the new trigger is added*, the *new trigger name*, and the *trigger code*. The procedure is again a two-step task: First an intermediate process is compiled and then  $\tilde{P}^0_{PApplication\_0}$ 's control structures are reconfigured.

The intermediate process definition is a subprocess of PApplication\_0, whose source code is listed below:

```

1  process PApplication_0_modified extend
2      PApplication_0
3  {
4      extending activity AReview handle CReview
5      {
6          trigger TNotifyReviewers as //header
7              email.SendMessage( //action
8                  reviewerdb.GetReviewers(
9                      AProposal.GetKeywords()),
10                 "Paper for your review",
11                 AProposal.GetPaperURL())
12          after resume activity; //event
13      }
14 }

```

After a successful compilation of the code above, we may follow a similar procedure outlined in the previous subsection to reconfigure  $\tilde{P}^0_{PApplication\_0}$  and  $O^0_{PApplication\_0}$  to two three-node lists. To keep the lengths of these lists as short as possible, we can merge the definitions of PApplication\_0\_modified into PApplication\_0 by ensuring that the new PApplication\_0 is equivalent to PApplication\_0\_modified. The merging is done by: ❶ moving the activity definition of *AReview* from

<sup>1</sup> A potential problem here is that the new activity object constructed this way may have some semantic inconsistency. We leave this problem to client applications.

PApplication\_0\_modified to PApplication\_0; ❷ moving the variable definitions of *AReview* and *AReview\_activity* from PApplication\_0\_modified\_class4process to PApplication\_0\_class4process; ❸ releasing the intermediate process definition and its supporting class definition.

After the merging, we reconfigure the process object by: ❶ expanding  $O^0_{PApplication\_0}$  by eight bytes; and ❷ storing the ID of  $o^0_{PApplication::AReview}$  and the converted pointers to  $\tilde{A}^0_{PApplication::AReview}$  to the expanded area. After these steps,  $O^0_{PApplication\_0}$  will be something like the one shown in Figure 11. Note we change the subscript of  $o^0_{PApplication::AReview}$  and  $\tilde{A}^0_{PApplication::AReview}$  to reflect the fact that they now represent a more specific activity in PApplication\_0.

It is interesting to compare Figure 11 with Figure 9. We may notice that except the process name, the logical structures of the two process objects are equivalent. A less obvious fact is that their process definitions are equivalent as well, except the member definition in activity *AReview* of PPaperReview. Other modifications such as *adding new activities or members, enabling or disabling triggers* can be done in similar ways.

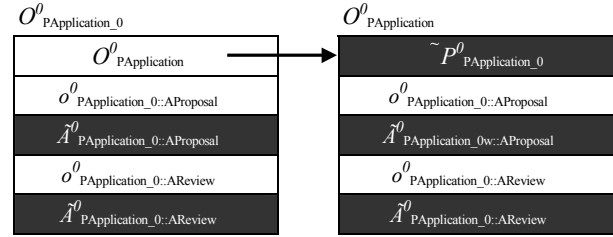


Figure 11.  $O^0$  after a new trigger is added to activity *AReview*.

## 5. COMPARING WITH RELATED WORK

We have introduced P, defined informally what process inheritance is in P, discussed how P implements inheritance, and developed an inheritance-based approach for modifying dynamically running instances. We are now ready to compare P with other interesting work on related topics, including *process inheritance*, *dynamic instance modification*, and *active database systems*.

### 5.1 Related Work on Process Inheritance

Process inheritance has been explicitly addressed in several process management related work. *wOrlds* [3] uses networked *obligations*, i.e. *requests* from one agent to other agents, as its process meta-model. An obligation is actually a description of a certain task and can be divided into a network of sub-obligations connected by links via sub-obligations' input/output ports. The execution of an obligation is driven by tokens similar to those in Petri-nets. Inheritance is supported by generating a composite obligation from a stack of obligation networks, with the most general one at the bottom and the most specific one (local modifications) at the top. For the composition purpose, at each obligation layer, network entities are marked as a *Deleter*, a *Creator*, a *Deleter/Creator*, or a *Modifier*, which makes it possible for an upper obligation to customize certain entities in lower obligations. This is very similar to what process inheritance can do in P. However, due to the rigidity of the process meta-model (basically it is Petri-net based), inheritance in *wOrlds* has many restrictions that can be easily violated and are the source of many errors, including *invalid entity reference (dangling entity)*, *links to incompatible ports*, *missed token*, etc. A deeper reason for these errors is the composition way in which *wOrlds* implements inheritance: obligations are created independently at each layer and then combined. P's pure language based approach ensures at

compilation time that each subprocess is correctly based on its superprocesses and rules out any possibilities for these errors. A related problem is that although *wOrlds* claims that it supports dynamic modifications to running instances by allowing updates to its local modification obligation layer, a re-composition procedure is required to generate a new composite obligation for the instance. This re-composition is not as smooth as P's reconfigurations to instances' control structures, which can be done internally by the system.

*Process Handbook* [22] aims at building a depository of business process descriptions, which are in natural English, and uses inheritance as a second dimension, in addition to *task decomposition*, to organize these descriptions into a hierarchy. It uses inheritance as a way of *classification*, rather than *programming*. As such, they did not address the mechanisms that actually implement process inheritance, nor to mention mechanisms for dynamic modifications.

*WF-net* [31], a variant of Petri-net, is the process meta-model based on which four types of inheritance, namely *protocol*, *projection*, *protocol/project*, and *life-cycle inheritance*, are defined to analyze the relationship of two process definitions. These inheritance relationships are further used to define a set of transformation rules, each of which has complex preconditions, for migrating a running instance from one process definition to another or doing ad-hoc change to running instances. The focus of this work is the theoretical analysis of the correctness of these rules. We instead develop the mechanisms to support process inheritance and our experiences shows that dynamic modifications can be done in a far less restricted manner.

Several earlier process language and models, such as *EPOS* [8], *E<sup>3</sup>* [18], and *CSPL* [6], claims to be object-oriented. However, their capability in object-orientation is limited to describing in an object-oriented fashion various entities, e.g. *artifacts*, *roles*, *tasks*, involved in cooperative processes. None of them has ever tried to develop mechanisms for supporting process-level inheritance.

## 5.2 Related Work on Dynamic Modification

Dynamic modification is a less studied area where few systematic techniques have been developed. The seminal work [12] discusses how to migrate running instances modeled with Petri-net from one process definition to another. Its major contribution is a theoretical analysis on the correctness of replacing an old subnet in a process definition with a new one. It does not provide any detail on how to do the migration, which is addressed partly in our inheritance-based approach. [31], though based on inheritance, is also a theoretical analysis.

ML-DEWS [11] develops a language dedicated for describing dynamic change and change policy. It does not explicitly support inheritance but offers a capability similar to that of inheritance in the sense that it can augment a process definition step by step. This gives the approach of ML-DEWS a flavor of *mixin inheritance* [4], with which a special *mixin class* is created to model the difference between a superclass and a subclass. The strong point is that a *mixin class* can be repeatedly used for similar modifications. Obviously ML-DEWS programs can achieve the same effect if they are designed carefully. P's inheritance is based on specialization, where modifications are embedded directly in subprocess definitions.

[26] develops a set of modification primitives, e.g. *adding/deleting tasks*, *changing task sequences*, which can be applied to individual instances of processes modeled with a directed graph based meta-model. These primitives directly change the structure of the graphical representation of an instance and are not inheritance based. Since control and data flows are explicitly expressed in process

definitions, to maintain their correctness, each primitive may have some global effects to the process structure due to graph substitutions and reductions. In our solution, the effects of each change are strictly restricted within the affected activity.

## 5.3 Related Work on Active Database

Triggers are a concept originated in the ADB community, where they are often a part of data models and are used to perform some computation in response to an operation on a data object. For example, in O++, trigger definitions are part of a class definition [15]. In relational databases, triggers are usually attached to a relation definition. P uses triggers in a totally different way: They are part of an activity definition and serves as the communication channels among activities. For example, we can use triggers to control the state of another activity, or to pass/get some information to/from another activity. Not defining triggers in classes gives P programmers the flexibility to reuse classes in different activities.

Triggers have also been shown capable of supporting cooperation processes in ADBs [10], where nested transactions are organized as a transaction tree and each transaction represents an activity. A tree, of course, can be seen as a special directed graph. Therefore, the underlying process meta-model is highly structured and difficult to be modified. What makes the situation even worse for modifying instance is that a transaction is not an entity that can be explicitly manipulated.

## 6. CONCLUSIONS AND FUTURE WORK

P contributes to process management by being the first process language that supports process inheritance, which is conceived as a new dimension to process modeling, programming, and reusing. We also show that mechanisms supporting process inheritance are also very effective for modifying dynamically individual running instances. In another work [39], we reported how inheritance can be used as a key mechanism for building a library for process programming. A binary release of P runtime system, related documents, and some sample process programs are available at <http://blrc.edu.cn/research/p/>.

We skipped many details regarding *how activity objects are presented and accessed*, *how triggers are searched along the inheritance hierarchy*, *how concurrency is handled in trigger execution and dynamic modification*, *how global objects are un/installed and developed*, *how an activity can be defined as an process so that processes can be nested*, *the programming interface for modifying instances*, etc. These aspects are crucial to the correct functioning of the system. However, they are relatively easy to understand and implement. Some of these topics are described in more detail in [37, 38].

We believe that inheritance is a very important factor to boost the popularity and flexibility of process management technologies. Though the idea of process inheritance is brought directly from classical OOP, we do need novel mechanisms to support process-level inheritance. We provide a first step and hope the ideas and mechanisms presented here would be helpful for extending other process languages to support process inheritance. We also hope that this work would trigger further research on different patterns of inheritance, such as these surveyed in [30], for programming processes and their effects on reusability and flexibility.

## ACKNOWLEDGEMENTS

This research is based on the author's PhD work, which was supervised by Prof. Shi, Meilin and funded by *China NSF* and *Hi-tech R&D Plan* under several grants. The author appreciates the

anonymous reviewers' insightful and critical comments. The author thanks the students at Tsinghua University who have used the system for their course work for helpful feedback, Alfred V. Aho for valuable discussions about the system, and the management team for its continuous support to this research.

## REFERENCES

1. Ambriola, V., Conradi, R., and Fuggetta, A. Assessing process-centered software engineering environments. *ACM TOSEM*, 1997, 6(3) : 283-328
2. Bandinelli, S., Nitto, E. D., and Fuggetta, A. Supporting cooperation in the SPADE-1 environment. *IEEE TOSE*, 1996, 22(12) : 841-856
3. Bogia, D. P. and Kaplan, S. M. Flexibility and control for dynamic workflows in the wORlds environment. In: *Proc of ACM COOCS*, Milpitas, 1995, 148-159
4. Bracha, G. and Cook, W. Mixin-based inheritance. In: *Proc of ACM OOPSLA/ECOOP*, Ottawa, 1990, 303-311
5. Cass, A. G., Lerner, B. S., McCall, E. K., et al. Little-JIL/Juliette: a process definition language. In: *Proc of ACM/IEEE ICSE*, Limerick, 2002, 754-757
6. Chen, J. CSPL: an Ada95-like, Unix-based process environment. *IEEE TOSE*, 1997, 23(3) : 171-184
7. Curbera, F., Golland, Y., Klein, J., et al. Business process execution language for web services, Version 1.0. 2002 ([http://www.ibm.com/developerworks/library/ws-bpel/.](http://www.ibm.com/developerworks/library/ws-bpel/))
8. Conradi, R., Hsgaseth, M., Larsen, J.-O., et al. EPOS: Object-oriented cooperative process modeling. In: *Software Process Modeling and Technology*, Finkelstein, A., Kramer, J. and Nuseibeh, B. Eds. Research Studies Press, London, U.K., 1994
9. Curtis, B., Kellner, M. I., and Over, J. Process modeling. *CACM*, 1992, 35(9) : 75-90
10. Dayal, U., Hsu, M., and Ladin, R. Organizing long-running activities with triggers and transactions. In: *Proc of ACM MOD*, Atlantic City, 1990, 204-214
11. Ellis, C., and Keddara, K. ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. *CSCW: the Journal of Collaborative Computing*, 2000, 9(3-4) : 293-333
12. Ellis, C., Keddara, K., and Rozenberg, G. Dynamic change within workflow systems. In: *Proc of ACM COOCS*, Milpitas, 1995, 10-21
13. Fenn, J. and Linden A. CIO update: key technology predictions, 2003 to 2012. *Gartner Report*, Jan. 2003 (<http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2910700-1,00.html>.)
14. Fuggetta, A. Software process: a roadmap. In: *Proc of ACM/IEEE ICSE*, Limerick, 2000, 27-34
15. Gehani, N. H., Jagadish, H.V., and Shmueli, O. Event specification in an active object-oriented database. In: *Proc of ACM MOD*, San Diego, 1992, 81-90
16. Gance, N. S., Pagani, D. S., and Pareschi, R. Generalized Process Structure Grammars (GPSG) for Flexible Representations of Work. In: *Proc of ACM CSCW*, Cambridge, 1996, 180-189
17. Hull, R., Lirbat, F., Simon, E., et al. Declarative workflows that support easy modification and dynamic browsing. In: *Proc of ACM WACC*, San Francisco, 1999, 69-78
18. Jaccheri, M. L., Picco, G. P., and Lago, P. Eliciting software process models with the E<sup>3</sup> language. *ACM TOSEM*, 1998, 7(4) : 368-410
19. Jørgensen, H. D. Interaction as a framework for flexible workflow modeling. In: *Proc of ACM GROUP*, Boulder, 2001, 32-41
20. Katayama, T. A hierarchical and functional software process description and its enactment. In: *Proc of ACM/IEEE ICSE*, Pittsburgh, 1989, 243-252
21. Klein, M. and Dellarocas, C. A knowledge-based approach to handling exceptions in workflow systems. *CSCW*, 2000, 9(3-4) : 399-412
22. Malon, T. W., Crowston, K., Lee J., et al. Tools for inventing organizations: toward a handbook of organizational processes. *Management Science*, 1999, 45(3) : 425-443
23. Manolescu, D. A. Workflow enactment with continuation and future objects. In: *Proc of ACM OOPSLA*, Seattle, 2002, 40-51
24. Paton, N. W. and Diaz, O. Active database systems. *ACM Computing Surveys*, 1999, 31(1) : 63-103
25. Peuschel, B. and Schafer, W. Concepts and implementation of a rule-based process engine. In: *Proc of ACM/IEEE ICSE*, Melbourne, 1992, 262-279
26. Reichert, M. and Dadam, P. ADEPT-supporting dynamic changes of workflow without losing control. *Journal of Intelligent Information Systems*, 1998, 10(2) : 93-130
27. Shepard, T., Wortley, C., and Sibbald, S. A visual software process language. *CACM*, 1992, 35(4) : 37-44
28. Stroustrup, B. What is object-oriented programming? *IEEE Software*, 1988, 5(3) : 10-20
29. Sutton, S. M. Jr, Heimbigner, D., and Osterweil, L. J. APPL/A: a language for software process programming. *ACM TOSEM*, 1995, 4(3) : 221-286
30. Taivalsaari, A. On the notion of inheritance. *ACM Computing Surveys*, 1996, 28(3) : 438-479
31. van der Aalst, W. M. P. and Basten, T. Inheritance of workflows: an approach to tackling problems related to change. *TCS*, 2001, 270(12) : 125-203
32. van der Aalst, W. M. P. and Berens, P. J. S. Beyond workflow management: product driven case handling. In: *Proc of ACM GROUP*, Boulder, 2001, 42-51
33. van der Aalst, W. M. P. and van Hee, K. M. Workflow management: models, methods, and systems. MIT press, Cambridge, MA, 2002.
34. van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., et al. Workflow mining : a survey of issues and approaches. *DKE*, 2003 (to appear)
35. WfMC. Workflow process definition interface—XML process definition language. WfMC-TC-1025, 2002.
36. Yang, G. A uniform meta-model for modeling integrated cooperation. In: *Proc of ACM SAC*, Madrid, 2002, 322-328
37. Yang, G. P language specification and library reference. 2003 (<http://blrc.edu.cn/research/p/documents/pspec.pdf>.)
38. Yang, G. P API reference. 2003. (<http://blrc.edu.cn/research/p/documents/papi.pdf>.)
39. Yang, G. Towards a library for process programming. In: *Proc of BPM*, Eindhoven, 2003, 120-135 (LNCS 2678)
40. Yang, G. and Shi, M. Cova: a programming language for cooperative applications. *Science in China, Series F*, 2001, 44(1) : 73-80