

Process Library

Guangxin Yang

Bell-Labs Research, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974 USA

Abstract

Process programming is a critical approach in many process management related areas including workflow management, software engineering, etc. Numerous process models, languages, and systems have been developed. Comprehensive libraries for process programming are essential for the acceptance, popularity, and success of this new programming paradigm. This paper identifies four important mechanisms, i.e. inheritance, nesting, integration, and reflection, for building process libraries and presents how these mechanisms are implemented in the context of P, a process language and system for developing cooperative applications.

© 2004 Elsevier Science B.V. All rights reserved.

Keywords: Process Library, Process Reuse, Process Management, P

1 Introduction

Process programming is a vital approach in a number of process management related areas, such as workflow management (WfM)[34], process-centered software engineering environment (PSEE) [1][7][8], business process reengineering (BPR)[9]. It concerns primarily with describing formally

-
- Corresponding author. Phone: 1-908-582-7830. Fax: 1-908-582-3104
Email address: gxyang@acm.org

various aspects, e.g. activities, artifacts, roles, tools, and the interrelationship among them, of a complex and long duration procedure within which a group of people cooperate with each other to finish a certain task. The resulting description often takes the form of process programs, which are enacted by a process management system (PMS) to coordinate the activities of various process entities. We call a formal description of a process a process definition. We call a software system that enacts instances of cooperation processes according to their definitions a process management system, or PMS for short.

There have been numerous process-modeling techniques, among which language-based ones are dominant. For example, in the WfM area, workflow processes have been described with various Petri-net, directed graph, formal grammar, etc. based formalisms, among which some representative ones include wOrlds[3], IBM WebSphere MQ Workflow [12], Staffware[26], WF-net[31], FLOWer[32], WASA₂[35]; in the PSEE area, more than a dozen process languages[1], including SPADE-1[2], Little-JIL[4], CSPL[5], EPOS[6], E³[13], HFSP[14], OIKOS[22], APPL/A[27], have been developed; in the BPR area, representative work includes Process Handbook[21], and Process Access Library [10] for systematic business process modeling and analyzing. These efforts have deepened substantially the understandings of processes and produced many interesting process management mechanisms and systems.

Process programming is perhaps based on the understanding that "business processes are software too", a generalization of the well-received proposition that "software processes are software too" in the software process community[23]. Though the power of software reuse is self-evident [19] and many have recognized that it is critical to share and reuse process programs among different systems to promote efficiency and interoperability[16][25], few systematic efforts have been put into developing advanced mechanisms that enable such sharing and reuse, perhaps partly due to the wide variance in the formalisms, capabilities, and semantics of process languages. We believe that serious research on the mechanisms for process reuse will not only be a key enabler to the popularity of process technologies, but also will lead to better understanding of and more flexible support for processes.

Library is the dominant reuse vehicle in today's software practice. A software library is a collection of code assembled to perform a set of related computing tasks. Virtually every programming language in use today offers sophisticated ways and tools for creating libraries. Well-established and widely adopted mechanisms and alternatives for library creation and maintenance include components, classes, subroutines, static and dynamic linkage, various calling conventions, etc. Carefully designed, a software library brings many advantages such as sharing code easily, increasing modularity, and sharing code in binary form among different languages. The importance of these advantages to the success of software research and development can hardly be overestimated.

Process libraries can be developed to gain similar benefits in the process management area. We define a *process library* as *a collection of code assembled to perform a set of related coordinating and computing tasks*. The code in a process library will be used as the building blocks for process management applications. Since the application domains can obviously be very different, it is likely that the contents in process libraries may vary widely. Therefore, we will not discuss what should be included in process libraries. Instead, we are concerned primarily with the mechanisms on how to glue these building blocks together so that developing process management applications can be easier and more flexible.

The purpose of this paper is to investigate systematically these mechanisms in general and to present how they are implemented in a process language and system, namely P, in particular. Conceptually, process libraries are a superset of traditional software libraries. Therefore, mechanisms

for building process libraries are a superset of the mechanisms for software libraries. We group these mechanisms into the following four categories:

- *Inheritance* to explore the similarities among processes and these among various process entities
- *Nesting* to incorporate smaller processes into a large one to decrease the complexity
- *Integration* to include various computing resources and facilities into PMSes in which process programs are enacted
- *Reflection* to make process management services accessible from within process programs

A preliminary version of this paper is published as [39]. The rest of this paper is organized as follows. The coming section discusses some important requirements for each of the four mechanisms and then proposes a framework for process libraries. The third section details how the architecture is implemented in the context of P. Comparisons with related work are presented in the forth section. Concluding remarks and some considerations for future work are given in the last section.

2 Process Library: A Framework

To develop effective mechanisms for process libraries, we need to identify the types of various library elements and analyze their relationship. A process programmer sees two things: a set of process programs and a process management system. From an intra-process point of view, a process may consist of some smaller ones, each of which may also consist of even smaller ones. This is known as *nesting*, which many process languages support. Decomposition relates processes in a tree structure, which we call an *n-tree*. The formation of an n-tree is completely determined by application semantics. From an inter-process point of view, a process may resemble another one in certain aspects, e.g. task decomposition, coordination patterns. This makes it possible to organize similar processes in a hierarchy of abstraction, according to which a more specific process may inherit certain properties of a more general one. We call this hierarchy an *i-tree*, whose structure is primarily determined by the similarities among processes. Obviously n-trees and i-trees are independent, but they may interleave.

A process management system creates and controls the enactments of instances of various processes. It plays a coordinator's role and therefore must rely on the things it coordinates to get its job done. The things may include various, either standardized or proprietary, software systems, such as operating systems (OS), database management systems (DBMS), object systems, email systems, web systems, and hardware systems controllable via certain programming interfaces. They reside externally to a PMS. It is important for a PMS to provide mechanisms to integrate these things so that they can be accessed or controlled from within process programs. On the other hand, internal functionalities for instance enactment may allow process programmers to gain more flexible control over instances of the processes they write, right from within process programs. For this purpose, a reflection mechanism is highly desirable.

We will elaborate on these topics and then present our process library framework.

2.1 Inheritance

Generalization-and-specialization is a natural and powerful approach. Its application in computer science resulted many methods and techniques prefixed by object-oriented (OO). OO design, OO analysis, OO language (OOL), OO programming (OOP) are just a few of them. Inheritance plays a

characterizing role in the OO world [28]. Many have believed it will bring many benefits to process programming and have introduced it into several process languages. For example, CSPL, which is based on object-oriented Ada95, claims to support inheritance based on Ada95's OO feature [5]. EPOS defines an inheritance hierarchy among various entities, including task entities such as project, develop, design, and compile and data entities such as design documents, executable, source code and object code [6]. E³ follows a similar approach but classifies these entities in a more fine-granular manner [13]. These types of inheritance support have demonstrated their strength in flexible process modeling and reusing. We call this type of inheritance *process entity level inheritance*, or *entity level inheritance* for short.

Support for entity level inheritance is important and necessary, but not sufficient. If we look at different processes carefully, we may also find the similarities among their process behaviors. For example, a paper review, online ordering, and mortgage application process resemble each other in that in each process, a document will be produced, reviewed, and then a result is generated. We should try to explore these similarities and invest more on supporting inheritance of properties and behaviors at process level. These properties include task decomposition and descriptions. The behaviors include the control and data flows among different tasks. We call this kind of inheritance *process level inheritance*, which has completely difference semantics, as compared with class inheritance in today's OOLs. Class inheritance focuses on specializing object states and ways for changing and retrieving these states, which are described with member variables and methods. Consequently, process level inheritance requires novel mechanisms for enacting process instances.

To inherit is to receive, and to change. If a process definition inherits another one, we call the former a *subprocess*¹ and the latter its *superprocess*. A subprocess will have all the property and behavior definitions of its superprocesses along its inheritance hierarchy. At the same time, process programmers should be able to and usually they will customize these definitions in a subprocess to describe a more specialized situation. For example, task decomposition can be specialized by adding new task definitions to subprocesses; task descriptions can be specialized by adding more details and/or overriding it with a new task definition; control and data flows can be customized by changing the contents exchanged and conditions under which the flows occur. Besides the necessary language constructs that are used to describe the inheritance at the process level, new mechanisms are required to implement the new inheritance semantics. Of course, the specifics depend heavily on the underlying process meta-models and languages.

2.2 Nesting

Divide-and-conquer is another natural and powerful technique and has many applications in computer science. One representative application is subroutine, which is supported by nearly all of today's programming languages. Its application in process programming would mean dividing a complex cooperation process into smaller ones, which are programmed separately and then glued together. We call this *nesting*, which is supported by almost all process languages and has different forms. For example, in Petri-net or directed graph based languages, a small process is usually represented as a sub-net or a sub-graph; in tree-based languages like Little-JIL[4], a small process is a node with descendants; in traditional language based process languages like Ada-based CSPL[5] and

¹ Some workflow literature such as [11] uses the term subprocess to denote a process encapsulated in a larger one. In this paper, we call the encapsulated process a nested process. See Section 2.2 for more details.

APPL/A[27], a small process is mapped to their base languages subroutine constructs, e.g. procedure and task. We shall call the complex process a nesting process and its constituent smaller ones nested processes.

Process nesting is an important mechanism for process libraries because it would allow incorporating easily processes in a library accomplishing certain tasks into a new process to be programmed. Despite the widely varied forms, we think three issues are critical to a successful nesting mechanism. The first one is parameter passing. Similar to a subroutine, a nested process usually needs context information, i.e. inputs from its nesting process. When specifying a process as its nested process, a nesting process must have a way to designate these input parameters, which can be constant values or dynamically generated information specific to instances of the nesting process. The second issue is how a nesting process can retrieve certain information, such as instance states and application data, from a nested process. This information is similar to the output/return values of a subroutine to its callers. If it is not stored in a centralized database which both nesting and nested processes have access, the language designer must devise specialized mechanisms and language constructs for this purpose.

The third one is the visibility of the internal structure and state information of a nested process to its nesting processes. Existing process languages seem to have ignored this aspect completely by allowing a nesting process to gain full control over a nested process. This may do no harm if all these processes are developed by the same group and/or in the same process language. However, if we put a process program in a library for reuse purpose, things need to be changed. For example, we may want to hide the internal structure of a reusable process definition for certain organizational or other non-technical considerations, especially in inter-organization process management. This way not all state information of the instance of a nested process is revealed to its nesting process. For this purpose, language constructs should be provided so that process programmers can specify the visibility of the elements of a process that can possibly nested.

2.3 *Integration*

Collaborative processes are a combination of coordination, i.e. sequencing and information passing among tasks, and computation, i.e. using software tools and systems to finish individual tasks. Although there are strong trends and evidences that in the design of some cooperation languages people tend to make a separation between these two parts, from the point of view of constructing process libraries, we think both parts are of equal importance. This is because coordination, in some sense, relies on computation. For example, end users often use some software tools, e.g. a word processor, a compiler, and access some resources, e.g. a database, a file, to get their work done. Therefore, elements for accessing these services should be an integral part of process libraries. Consequently, appropriate mechanisms must be developed so that process programs can access various external resources, e.g. operating systems, file systems, databases, directories, Internet services, or even other process systems. Components for accessing these resources in process libraries are virtually unlimited due to the wide application areas of process technologies.

In designing these mechanisms, we need to take into account a number of factors to address the diversities of external services. We discuss two of them here. The first one is the representation, i.e. the way in which external services are represented in a process system. For example, HFSP has a tool interface construct; CSPL has a very similar tool unit construct. However, from a more general point of view, we believe that the mechanism should be extended beyond handling only standalone programs to dealing with external computing services in a truly programmable manner by utilizing the application

programming interfaces exported by these services, if any. The second factor is the protocol via which a process system talks to the outside world. In whatever the case, it is the responsibility of the process runtime system to interpret a process program and to communicate with external computing services when needed, during which a specific protocol may be followed. Since the protocols used by external services may vary greatly from low level protocols like TCP/IP to high level standard ones such as RPC, DCOM, CORBA, and XML-based Web Services, or proprietary ones such as Sun Tooltalk and DEC FUSE, a process management system should be able to accommodate to this diversity.

2.4 Reflection

A process program differs from an ordinary software program in that the former is not directly scheduled by an operating system, but by a PMS, which provides services for managing process instances, e.g. creating, destroying, or changing the state of instances, and holds dynamically generated information about them, e.g. when an instance is created, who created it, which state it is currently in, who are participating in an activity. These services and information can help to bridge the somewhat artificial gap that usually exists between the runtime and build-time functions of PMSes. They should be made available to process programs so that process modeling could be more flexible. We believe that by introducing these services and information into process programming, the rigidity of process programs can be significantly decreased and they can be more flexible and adaptive to the ever-changing work conditions.

The mechanism is reflection [20], which has demonstrated its power and benefits in the areas of operating systems and traditional programming languages and systems. For example, process/thread management services of operating systems, such as creating or terminating processes/threads, are available to any program via certain system calls. Smalltalk and Java offer specialized interface so that programmers can modify the class definition of an object or retrieve its meta-information at runtime. We believe reflection can also benefit process programming. For example, by using the runtime information of activity instances, we can restrict, at build time, the prospective participants of an activity to be the actual participants of another activity; we can force a program-defined action after a certain time interval by using the creating time of an activity or process instance. The runtime information obviously brings more flexibility to process modeling.

Gaining control over process or activity instances from within process programs via reflection can make running instances more adaptive to dynamic situations. Lacking this ability does no harm for toy applications. However, complex real world processes do need this kind of control to manage the creation and enactment of process instances, as observed in [15]. If a PMS allows dynamic modifications to the definitions of instances, accessing to this functionality from within process programs would make these programs more adaptive to various exceptions and increase their flexibility. Adaptivity and flexibility have been a hot topic in workflow research [18] and recognized as future direction for software process research [8]. Existing research has not addressed the possibility and advantage of modifying process programs from within themselves. Our experience has shown that this capability is very useful [40].

2.5 The Framework

Fig. 1 shows the building blocks of libraries for programming cooperation processes and the ways in which these blocks relate to each other. The blocks under the caption *Reflection* make it possible for process programmers to access runtime information about and gain control over instances from within

their programs. The blocks under *Integration* incorporate into a PMS various external computing resources that are relevant to process enactment. Within the top block are the various process definitions represented as ovals and interconnected through *i-trees* and *n-trees*, which are differentiated with solid upwards and dotted downwards arrow lines. We omit in this block various entities involved in processes and their interrelationship, as the techniques to handle them are relatively mature and well understood.

This framework focuses on four mechanisms, i.e. *inheritance*, *nesting*, *reflection*, and *integration*, as the major vehicles for constructing process libraries. We believe it provides a relatively comprehensive view of what should be provided in process libraries. However, because of various implementation details, a PMS may choose not to and sometimes cannot implement all of them. And, as we have mentioned earlier, the technical details on how these mechanisms are implemented depend heavily on the PMS's process meta-model and language. The names of these mechanisms are by no means novel, however, a careful implementation of them in the context of process management does require significant innovations. In the next section, we present how P implements these mechanisms. In Section 4, we will discuss what the novelty of these implementations is when compared with other related work.

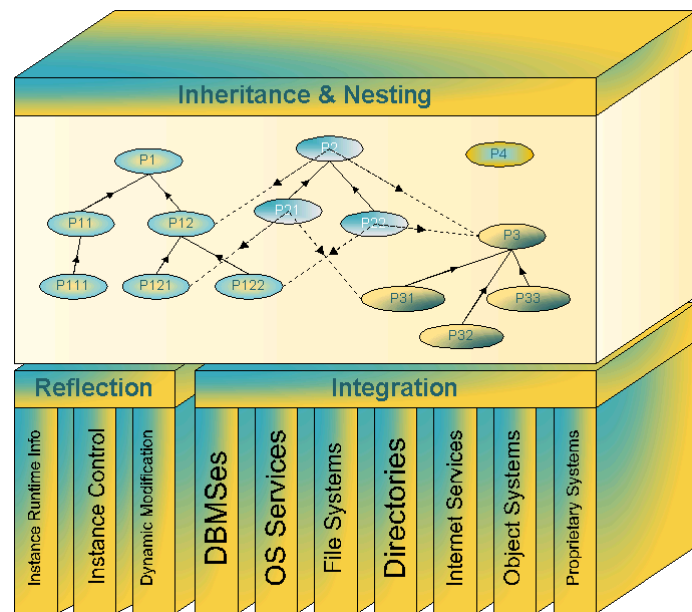


Fig. 1. The building blocks of process libraries

3 Process Libraries in P

We have identified our key mechanisms for process libraries and discussed some general requirements about their implementations. To further our understanding of these mechanisms, this section concretizes the ideas by presenting how these mechanisms are implemented in P, a programming language and system for developing cooperative applications [37]. P's pure language-based approach makes it possible to implement all these mechanisms. And it addresses these requirements in a unique, and in some sense, a uniform way. For example, all reflection and integration blocks will be encapsulated as respective P classes, P's base part for describing the computing behavior of various process entities. Before we present the details, it is necessary to give a brief overview of P. Readers familiar with any OOL like C++ or Java will find it is easy to understand.

3.1 A Overview of P

A process meta-model defines the basic elements for process modeling. Fig. 2 shows the relationship among the elements of P's process meta-model, 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 initiated 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.

The P language provides the constructs for modeling various aspects of a process, e.g. its process structure, the rules regarding message exchanges among activities, and the structure and operations of the results produced by its activities. More specifically, P has the following five key constructs:

- A *class* construct for class definitions. The *class* construct is a full-featured object-oriented language (OOL) with support to collection, such as *array* and *dictionary*. It can be used to program the structures and operations of various entities involved in processes.

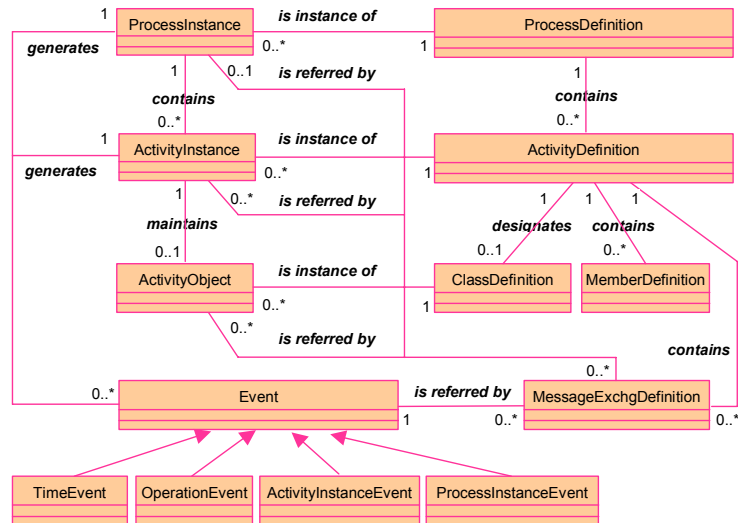


Fig. 2. P's process meta-model.

- An *activity* construct for P activity definitions. Each activity can produce an object of a certain class. We call the object an *activity object* and denote it as o_A , where A is the name of the activity. An activity can also be a process.
- A *member* construct for defining anticipated 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 among activities. A *trigger* is part of an activity and takes the form of an *event-condition-action* rule developed in the Active Database (ADB) community (refer to [24] for a wonderful survey). 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.

A full specification of the P language is provided in [38].

3.1.1 An Example Process Definition in P

The code in Fig. 3 describes an application process, in which a document is first proposed then reviewed. For simplicity, we omit the details 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 the 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  { //activity to produce a CDocument object
5    public activity AProposal produce CDocument
6      { //define a trigger named TProposalSubmit
7        trigger TProposalSubmit as AReview.activity.Activate()
8          after submit activity;
9      }
10 //activity to produce a CReview object
11 protected activity AReview produce CReview
12 { //define a trigger named TReviewSubmit
13   trigger TReviewSubmit as
14     email.SendMessage(AProposal.activity.GetParticipants(),
15       "Result for"+AProposal.GetName(),AReview.GetReviewSummary()),
16     (!AReview.Confirmed())?AProposal.activity.Activate():true
17   after submit activity; //event
18 }
19 }
```

Fig. 3. The definition of process *PApplication*.

3.1.2 Supporting Classes

An instance is enacted by a P server, which knows the instance as a combination of related runtime information. We need a way to organize this information for both instance recording and enacting. P's solution is to define a *supporting class* for each process, which is a standard P class and describes the internal control structure for instances of that process. It defines member variables representing a process instance and its activity instances and activity objects. For example, the supporting class for PApplication is shown in Fig. 4. It has five member variables. The second and the fourth ones denote the activity objects produced and maintained by activities *AProposal* and *AReview*. The first 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. Similarly, the third and fifth variables represent the activity instance *AProposal* and *AReview* respectively. During compilation, the P compiler changes references like *AProposal.activity* and *AReview.activity* to *AProposal_activity* and *AReview_activity*. Variables such as *process* and **activity* are the main vehicle for reflection, which will be elaborated together with *CProcessInstance* and *CActivityInstance* later in Section 3.5.

```

1  class PApplication_class4process
2  {
3      public CProcessInstance    process;    //the process instance
4      public CDocument          AProposal;  //the activity object of AProposal
5      public CActivityInstance  AProposal_activity; //the activity instance
6      protected CReview        AReview;    //the activity object of AReview
7      protected CActivityInstance AReview_activity; //the activity instance
8  }
```

Fig. 4. The definition of the supporting class for process *PApplication*.

3.1.3 Process Objects

We denote an instance of a process *P* as \tilde{P}^{ID_P} , or \tilde{P}^{ID} for simplicity, where *ID*, an integer, is the instance's system-wide unique identifier. 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 object heap of the P interpreter associated with the server. We call the P object a *process object* and denote it as O^{id_p} , or O^{ID} for short. Fields of a process object is filled with appropriate values while the instance is being enacted. Fig. 5 shows one state of O^0 , the process object for instance 0 of PApplication. 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}^0 . The converted value is signified with a black background. Similarly when an activity instance is initialized, a P object, i.e. its activity object, denoted as $o^{\text{ID}_P::A}$, and a C++ object, i.e. the activity instance, denoted as $\tilde{A}^{\text{ID}_P::A}$, are allocated. Its two corresponding fields in the process object are assigned the identifier of the activity object and the converted C++ object pointer.

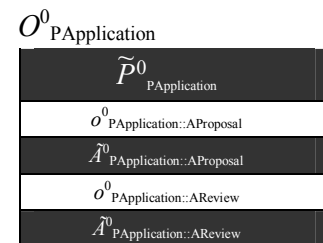


Fig. 5. The process object for \tilde{P}^0 .

3.1.4 Executing Triggers

A 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:

```

1   public boolean TempMethod4TriggerCondition()
2   {
3       return TriggerConditionExpression;
4   }

```

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 as shown below:

```

1   public void TempMethod4TriggerAction()
2   { //trigger action expression
3       AReview.activity.Activate();
4   }

```

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.

3.2 Inheritance

In P, process inheritance is a mechanism that allows all definitions, i.e. *activities* and *local classes* and *processes*, of one process, the *superprocess*, to be part of those of another, the subprocess. In the subprocess, these definitions can be used in a way as if they were defined in the subprocess itself. Meanwhile, the subprocess can define new definitions and/or to customize some definitions in its superprocess/es to fulfill its own specific needs. In this way, a process definition can be reused without losing the capability to be customized. Currently, we support *single inheritance*, which means that a process definition can have only one direct superprocess. With OOP, programmers 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 process definition in the following ways:

- 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 customization programmers 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

We will not give a formal specification on how to do the customizations. Fig. 6 gives an example that shows how to do some of them. We first define a new class, *COrder*, which inherits *CDocument* and models the orders. We omit the details for simplicity. The new process, *POrdering*, customizes its superprocess, *PApplication*, by first changing the class of the *AProposal*'s activity object to *COrder*; and second by adding a new trigger to activity *AReview*. After activity *AReview* is activated, the new trigger checks the availability of the ordered good and set the confirmation flag of *o_{AProposal}* and submit it to indicate the end of processing. The keyword *extending* specifies that definitions in the subprocess will be combined with those in the superprocess. The way the combined process works is after an order is submitted, the review activity is activated, the availability is checked (done by *TCheckInventory*), and an order status is sent to the customer (by *TReviewSubmit* in *PApplication*).

```

1  class COrder extend CDocument { ... } //details omitted
2  process POrdering extend PApplication
3  { //make the activity object more specific
4    extending activity AProposal produce COrder
5    {
6    }
7    //add participant specification and a new trigger
8    extending activity AReview produce CReview
9    {
10     trigger TCheckInventory as      //trigger header
11       AReview.SetConfirmation(
12         dbInventory.CheckAvailability(AProposal.GetInventoryNo()),
13         AReview.activity.Submit()
14       after activate activity;      //event
15   }
16 }
```

Fig. 6. The definition of process $P_{Ordering}$, which inherits process $P_{Application}$.

Accordingly, the supporting class for a subprocess will inherit that of its direct superprocess and it will have two member variables for each activity defined in the subprocess. The difference is that it will not have a member named *process* because the super supporting class already has it and we don't need a duplicated one to denote the same thing. At the same time, it is our design choice that overriding or extending an activity does not introduce a new activity (task) to the process. This means that for that activity, only one activity object and only one activity instance object will be created. For example, $P_{Ordering}$ will have only two activities, $A_{Proposal}$, which produces a C_{Order} object and A_{Review} , which produces a C_{Review} object. The P runtime system properly configures the supporting structures for a process instance to reflect this fact. For example, Fig. 7 shows the state for $O^1_{P_{Ordering}}$, after both of the two activities are activated. Note that $O_{P_{Ordering}}$ has four variables for the two extending activities. However, the value of each variable of $O_{P_{Application}}$ will be set to that of the corresponding variable of $O_{P_{Ordering}}$. This way the corresponding variables represent the same entity (an activity object or an activity instance object).

Overriding and extending differ in how they affect a P server in searching the triggers in activities with the same name in the process inheritance hierarchy. Starting from the leaf node, the searching ends at the first overriding activity. To ensure that trigger actions and conditions referring the activity can be correctly executed or evaluated, we require that the class type of the overriding or extending activity is the same as or a subclass of that of the activity being overridden or extended. When evaluating a trigger condition or executing a trigger actions, a P server adds the temporary methods to the supporting class of the subprocess. The configuration of the process object ensures that each reference resolves to the right value.

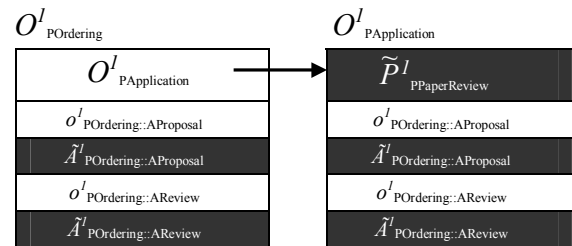


Fig. 7. The process object of $\tilde{P}^1_{P_{Ordering}}$.

3.2.2 Activity Accessibility

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 same process. A *protected* activity can be accessed by activities in the same process and its subprocesses. Only *public* activities of a nested process (see 3.3) 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.

3.3 Nesting

We have mentioned that an activity can also be a process. This opens a way for programmers to logically group a set of related activities into a reusable and, to some extent, autonomous entity, which can be embedded in larger process to achieve certain collaboration functions. For example, we can nest *POrdering* into a larger process, *POrderProcessing*, as defined in Fig. 8. The nested process will coordinate the order submission and availability checking. Beyond that, two extra activities, *ABilling* and *AShipping*, are defined for billing and shipping. Both activities will start when the ordering process is completed and the availability is confirmed. This condition is stated with the *Boolean* expressions at lines 7-8 and 13-14. These expressions are based on the state of the nested *POrdering* instance and a flag of the activity object of *POrdering::AProposal*. Both new activities can have their own trigger and member definitions. For simplicity, they are omitted here.

```

1  class CBill      { ... } //details omitted
2  class CShipping { ... } //details omitted
3  process POrderProcessing
4  {
5      public activity AOrdering as POrdering;
6      public activity ABilling produce CBill
7          start ( AOrdering.process.GetState() == COMPLETED &&
8                  AOrdering.AProposal.Confirmed() )
9      {
10         ... //activity definition goes here
11     }
12     protected activity AShipping produce CShipping
13         start ( AOrdering.process.GetState() == COMPLETED &&
14                 AOrdering.AProposal.Confirmed() )
15     {
16         ... //activity definition goes here
17     }
18 }

```

Fig. 8. The definition of process *POrderProcessing*.

3.3.1 Supporting Nesting

An immediate question is how nested processes are supported in the runtime system. Again we need turn to the supporting class for the nesting process. For each nested process, the supporting class of the nesting process will have a member, which has the nested process's supporting class as its type. For example, the variable for *POrderProcessing::AOrdering* is defined at line 4 in Fig. 9. Note that definitions of the activities in *POrderProcessing* can only refer to the public activities in *POrdering*.

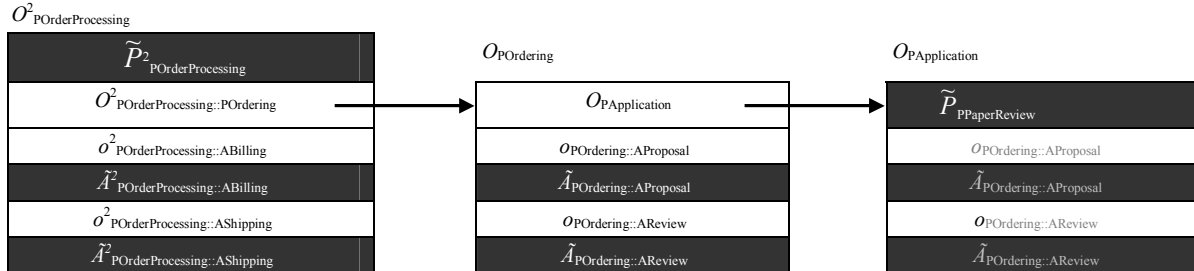
```

1  class POrderProcessing_class4process
2  {
3      public CProcessInstance      process; //the process instance
4      public POrdering_class4process AOrdering; //for the nested process
5      public ABilling              ABilling; //the activity object of ABilling
6      public CActivityInstance     ABilling_activity; //the activity instance
7      protected CShipping          AShipping; //the activity object of AShipping
8      protected CActivityInstance  AShipping_activity; //the activity instance
9  }

```

Fig. 9. The definition of the supporting class for process *POrderProcessing*.

According to the supporting class definition, the logical structure of the process object of an instance of *POrderProcessing* is shown in Fig. 10.

Fig. 10. The process object of a *POrderProcessing* instance \tilde{P}^2 .

A few words about passing parameters to nested processes as well as atomic activities. A class can have a constructor that takes necessary parameters. There is an optional part in the activity header that designates parameters that will be passed to the constructor of an activity object to set its initial state. For example, if we define a constructor for *CBill* that takes the credit card information of the customer as a *CString* type parameter, the header of activity *ABilling* can be defined as:

```
public activity ABilling produce CBill (AOrdering.AProposal.GetCreditCardInfo ())
```

A P server will evaluate the parameter and pass it to the constructor when initializing the activity object. For a nested process, parameters can be specified in a similar way and will be passed to the constructor of the nested process's start activity. A P compiler will check whether the activity class has a constructor that takes the specified parameters.

In a subprocess, when a nested process activity is redefined, the new nested process has to be a subprocess of the old one. The purpose of this restriction is to ensure that the nested process activity maintains a consistent "interface", through which other activities of the superprocesses where the nested process activity resides may still communicate correctly with the nested process. Again this restriction is checked and enforced by a P compiler.

3.4 Integration

As we have discussed, integration mechanisms in a PMS must deal with the diversity of external resources. What matters is not the quantity of integrated resources, but the extensibility of integration. P offers three ways.

3.4.1 COM/DCOM Support

Microsoft COM/DCOM is a de facto standard in the component software industry. The original intention was for the smooth interoperation of different object systems. Currently, P's COM/DCOM support allows a P programmer to describe with a standard P class the interested behaviors of a COM/DCOM component that supports the `IDispatch` interface and to use it in a P program as an ordinary P object. This way nearly any COM/DCOM component that is useful for the enactment of a process instance can be put under the control of the P runtime system. For this purpose, P has a predefined class named `uuid`, each instance of which represents a universal unique identifier that

COM/DCOM uses to uniquely identify a component. The wrapping P class for a component should have a constructor that takes a parameter of class `uuid` and defines the methods that will be used.

Fig. 11 shows the P wrapper class for an Acrobat PDF ActiveX control, which actually has more than two dozens of methods. The wrapper defines only two methods, one for loading a PDF file and the other one for printing it. Of course, new methods can be conveniently added.

```

1  class CPDF
2  {
3      public          CPDF(uuid clsid);
4      public  boolean LoadFile(CString csFileName);
5      public  void   printAll();
6  }
```

Fig. 11. The P wrapper class for an Acrobat PDF control.

To use the control, we simply declare a variable, say `pdf4Print`, of class `CPDF` and initialize it as:

```
pdf4Print = new CPDF(new uuid(0xCA8A9783, 0x280D, 0x11CF, 0xA24D444553540000));
```

where the four hex integers denote the class ID of the ActiveX control. After that `pdf4Print` is used in the same way as other ordinary P variables. A P interpreter takes care of all the details, such as COM/DCOM un/initialization and value conversion. Currently parameters and return values passed between a P interpreter and a COM/DCOM component can only be values of certain primitive types, e.g. *boolean*, *char*, *short*, *int*, and the class *CString*. This is partly due to the limitations of the *IDispatch* interface.

3.4.2 Native Methods/Objects

P supports a mechanism that allows the body of a method of a P class to be a function residing in a Win32 dynamical link library (DLL). We call it a *native method*, which can be called in the same way as an ordinary P class method. When a native method is called, a P interpreter manages preparing the parameters, turning control to the actual implementation, and converting the return value and output parameters into the interpreter's stack. Native functions can be written in any language, provided the calling convention requirement is met.

As a more advanced form of the native method mechanism, *native object* allows to introduce a C++ object running in the memory space of the P runtime system into the object space of the P interpreter as a P object. We call such a C++ object a *native object*. Similar, a P interpreter manages all the details to direct a call to a method of a native object to the corresponding method of the C++ object. For this purpose, we need to write a P class definition that services as an interface description of the C++ object. *CProcessInstance*, *CActivityInstance*, *CInstanceEnactor* in the next subsection are good examples.

The native method/object mechanisms offer a fully extensible and programmable way for accessing any external services from within P process programs. They are consistent with the object-oriented pattern for programming P classes. For a certain resources, all we need to do is to define as a P class an interface for it and implement the native functions or objects. Based on this mechanism, we have successfully developed some P classes, e.g. *CTime*, *CMath*, *CIO*, *CFileSystem*, *CEMailAgent*, and *CDatabase*, and the related native codes for time conversion, mathematical calculation, input/output, accessing files, sending emails, and accessing a relational database, with several hours work.

3.4.3 Global Objects

In some applications, some resources may be shared by many process instances. In these cases, we can install dynamically in a P runtime system objects representing these resources as *global objects*, which are accessible from within any process instance enacted in that runtime system. The P runtime system has the programming interface for dynamically install and uninstall global objects. A global object can be either a native object, or a normal P object whose definitions are given in the P language.

The processes *PApplication* and *POrdering* contain examples on how to use global objects. There we use two global objects, *email*, which is an instance of *CEMailAgent* for sending Internet email messages, and *dbInventory*, which is an instance of *CDatabase*. These objects should be properly installed when instances of *PApplication* and *POrdering* are enacted.

3.5 Reflection

The P runtime system allocates in its memory space a C++ object that records the runtime information of a process instance. It also allocates for each activity instance a C++ object representing the activity instance. For reflection purpose, it installs these C++ objects as native objects in the P interpreter. In a process program, these objects can be accessed via two predefined identifiers, i.e. *process*, *activity*, in the same way as ordinary P objects are accessed.

3.5.1 *process*

process can be used in a P process definition and represents an instance of the process. It is a native object, which is mapped to a C++ object maintained for the instance in the P runtime system. *process* is an instance of class *CProcessInstance*, whose methods can be divided into three categories:

- **Retrieving Information** for retrieving certain information, e.g. process name, display name, state code, start time, and end time, of the process instance which *process* represents.
- **Manipulating Instances** for changing forcefully, i.e. suspending, resuming, terminate, and aborting, the state of the process instance.
- **Modifying Definition** for modifying dynamically the process definition of this instance so that the instance can be adaptive to changing working setting. Details on this topic are found in [40].

The definition of *CProcessInstance* is given in Fig. 12.

```

1  class CProcessInstance
2  {
3      //the name of the process definition
4      public CString  GetProcessName();
5      //return the user friendly name
6      public CString  GetInstanceName();
7      //return the state of this process instance
8      public tiny     GetState();
9      //get the start time of the process instance
10     public int      GetStartTime();
11     //get the end time of the process instance
12     public int      GetEndTime();
13     //abort this process instance
14     public boolean  Abort();

```

```

15     //terminate this process instance
16     public boolean Terminate();
17     //activate this process instance
18     public boolean Activate() ;
19     //suspend this process instance
20     public boolean Suspend();
21     //modify the definiton of this process
22     public boolean ModifyDefinition(
23         tiny tAction, CString csObjectName,
24         CString csSubObjectName, CString csCode);
25     }

```

Fig. 12. The definition of class *CProcessInstance*.

3.5.2 activity

Similar to *process*, *activity* can be used in an activity definition and represents the current instance of that activity. It is also a native object, which is mapped to the corresponding C++ object created for that activity instance in the P runtime system. The P interpreter records the value converted from the pointer to the C++ object in the process object for that process instance. Operations available on *activity* are described as class *CActivityInstance*, which has methods for retrieving certain information about an activity instance and manipulating it. Both *PApplication* and *PPaperReview* have examples on how *activity* can be used. The P compiler will convert a notation like *AReview.activity* into *AReview_activity*, which is a member in the supporting class for the process. Fig. 13 gives the definition of class *CActivityInstance*.

```

1     class CActivityInstance
2     {
3         //get the name of this activity instance
4         public CString GetActivityName();
5         //get current state of this activity instance
6         public tiny GetState();
7         //get the start time of the activity. Return 0 if not started
8         public int GetCreateTime() ;
9         //get end time of the activity. Return 0 if not finished
10        public int GetEndTime();
11        //get users current taking part in this activity
12        public CString GetParticipants() ;
13        //submit this activity to indicate end of processing
14        public boolean Submit();
15        //abort this activity
16        public boolean Abort();
17        //terminate this activity instance
18        public boolean Terminate();
19        //suspend this activity instance
20        public boolean Suspend() ;
21        //activate this activity instance
22        public boolean Activate();
23    }

```

Fig. 13. The definition of class *CActivityInstance*.

3.5.3 *enactor*

enactor is a pre-defined global object that encapsulates some of the P runtime system's most important instance enactment services and makes them available to P process programs. These encapsulated services include:

- Creating a new instance for creating dynamically a new process instance to coordinate the accomplishment of a new task. The newly created instance will be a top-level instance and enacted independently.
- Removing an existing top-level instance for destroying the instance's persistent representation.
- Modifying the definition of a top-level instance for coping with dynamic changes. The same function is also available via *CProcessInstance::ModifyDefinition(...)*.
- Changing forcefully the state of a process instance into a specific state.
- Retrieving references to process or activity instances for other functionalities available via a *process* or an *activity* object.

The interface of *enactor* is described with class *CInstanceEnactor*, whose definition is given in Fig. 14.

```

1  class CInstanceEnactor
2  {
3      //create a new standalone process instance and returns its instance ID
4      public int      CreateInstance(CString csProcessName,
5                                  CString csDisplayName, CString csArguments=null);
6      //delete an existing process instance from the system
7      public boolean  RemoveInstance(CString csInstanceUniqueName);
8      //modify the process definition of the specified process instance
9      public boolean  ModifyDefinition(int nInstanceID, tiny tAction,
10                                 CString strActivityName, CString strSubObjectName, CString strCode);
11     //abort the specified process instance
12     public boolean  AbortInstance(CString csProcInstName);
13     //terminate the specified process instance
14     public boolean  TerminateInstance(CString csProcInstName);
15     //Activate the specified process instance
16     public boolean  ActivateInstance(CString csProcInstName);
17     //suspend the specified process instance
18     public boolean  SuspendInstance(CString csProcInstName);
19     //return the specified process instance object
20     public CProcessInstance  GetProcessInstance(int nInstanceID);
21     public CProcessInstance  GetProcessInstance(CString csProcInstName);
22     //return the specified activity instance object
23     public CActivityInstance  GetActivityInstance(CString csActInstName);
24 }

```

Fig. 14. The definition of class *CInstanceEnactor*.

3.6 *P Library Tools*

P provides the necessary tools for creating and utilizing P libraries. The P compiler supports an *import* directive, which can introduce the class and process definitions in the designated P source file

into the current one. We are currently working on a tool that can archive a bunch of compiled definitions into a library file and necessary compilation support so that the definitions can be managed more neatly. We are also considering a tool that generates automatically the P wrapper class for a COM/DCOM component from its type library. More sophisticated tools with graphical user interface can be built to allow visual manipulations on library elements.

4 Comparisons with related work

We have discussed *inheritance*, *nesting*, *integration*, and *reflection* as the key mechanisms for constructing libraries for process programming and presented how they are implemented in P, a language and system for developing cooperative applications. Though the four identified mechanisms are by no means new and have been exploited to some extent historically, a comparison with related work would reveal that our solutions are in some sense novel. Considering the fact that the number of related systems in BPR, PSSE, and WfM is huge, it is impossible for us to exhaust all these systems here. Instead, we choose some representative systems in the three areas and compare P with them in five important perspectives. Of course there are many other perspectives such as modeling capability in terms of patterns [33], but we ignore them here. Table 1 summarizes the comparison. The next three subsections explain it in some details.

4.1 Related work in BPR

In BPR, the Process Handbook aims at a repository of business process descriptions to help people redesign existing and invent new organizational processes and to share ideas of organizational practices. It uses inheritance as a second dimension, in addition to task decomposition, to explore the similarities among different processes. In an similar effort, the Process Asset Library [10], the goal is to organize all assets, e.g. generic process architectures, cycle models, process elements (or subprocesses or steps), of business processes into one easily available online database. The Process Handbook project has collected a large number of business process descriptions (see the website at <http://ccs.mit.edu/ph/>). Some companies are marketing web-based software for building and maintaining process asset libraries (see <http://www.missionsystems.lockheedmartin.com/pal/> for an example). Their primary goal is to develop rich on-line repositories of business knowledge, which is expressed in natural languages. Therefore, techniques for process enactment and support for integration and reflection are out of their scopes.

4.2 Related work in PSSE

P is more tightly related to process-centered software engineering environments because they share a similar language-based approach. For inheritance, six out of the eight systems we list here support it. However, most of this support is limited to describing and using documents, artifacts, or process entities in an object-oriented fashion. Obviously P can deal with this kind of inheritance with its class construct. Inheritance at the process level, i.e. inheriting process properties such as task decomposition, control and data flow among activities is basically not addressed in these systems. The relationship between process level inheritance and E³'s associate level inheritance is somewhat subtle. Association in E³ is a mechanism used to define the relationship among different entities, which are described with classes. Association inheritance means that a class inherits all association definitions from its

superclasses. For example, if *Task11* inherits *Task1* and there exists a binary association *preorder(Task1, Task2)*, then there exists implicitly another association *preorder(Task11, Task2)*. However, this kind of semantics seems to be problematic to process behavior: *Task11* may have other properties of *Task1*, however, it is not necessarily that in all cases, *Task11*'s termination must precede the activation of *Task2*. Similar problems exist for other associations. Association level inheritance may be good for process analysis, but seems inappropriate for process enactment. EPOS describes everything, including tasks, with classes. A task class may have a *decomposition* attribute for defining nested tasks, a *formals* attribute for specifying the task's in/output parameters, and *pre/post static/dynamic* attributes for various conditions. These attributes can be inherited. In this sense, EPOS supports inheritance of certain process behavior. However, for a task class describing a complex process, if its *decomposition* attribute is redefined, the task decomposition of the super task class is actually lost. Therefore, decomposition as one attribute is too coarse. P allows customization to individual activities. Obviously this gives programmers more flexibility.

Nesting is supported by nearly all systems, though in different forms, which are basically equivalent in expressiveness. This evidences that it is indeed an important mechanism. Integration has also been a major concern. For those based on classical languages, such as Ada-based APPL/A, Prolog-based Merlin, it is possible to use the libraries of the base language directly to access any external resources. Therefore they don't need any special construction. For newly defined languages such as SPADE-1, CSPL, and HFSP, they need to introduce new constructs for specifying external tools. For example, CSPL has a tool construct and HFSP programs can define a tool section to describe external, mostly standalone applications, used in processes; OIKOS defines a proprietary protocol to talk to external applications; SPADE-1 and EPOS uses more powerful tool integration facilities, such as Sun Tooltalk, Microsoft OLE2, DEC FUSE, and HP BMS. None of them has ever tried to use standard platforms such as COM/DCOM or CORBA. This may be partly because at the time these systems were developed, these standard platforms have not been ready yet.

Reflection in certain limited forms is found in some process languages. For example, HFSP has a special data type *status* and two special operations *snap* and *resume* for recording and changing the enactment status of a process; EPOS'es task classes have instance level attributes and procedures which can record and change task state. P's has similar facilities with *process* and *activity* constructs. Both EPOS and SPADE-1 allows changes to process definitions in the name of process evolution. EPOS achieves this with its meta-class approach, which allows a task class to be modified with special procedures. The idea is borrowed directly from Smalltalk. SPADE-1 allows process evolution by treating activity definitions as tokens, which can be modified by special external tools through its black transitions. However, from existing literature, it seems that these modifications cannot be done from within process programs. P's *process* and *activity* constructs and the global object *enactor* offer a more comprehensive reflection mechanism to use from within process programs all the enactment services provided by the P runtime system. Modification is done to *running instances*, not *process definitions*, and based on process inheritance.

4.3 Related work in WfM

Most systems in this area focus more on representing processes with certain graphical notations, which are usually based on directed graph or Petri-net. The advantage is visual representation makes process definitions easier to build and understand. The disadvantage is that the less language-based approach weakens programmability. The direct result is that the WfM systems seem to have weaker

reflection capability. According to the documents available to us, only IBM WebSphere MQ Workflow provides elementary reflection support by means of certain predefined data variables, e.g. `_PROCESS`, `_ACTIVITY`, denoting the names of the current process or activity instance, which can be used to retrieve or specify attributes of process or activity instances. However, changing instances' structure, or creating new instances from within process definitions is not supported, though in some systems, some of these functions are available via their client application programming interfaces. The weakness is offset by WfM systems' strong integration capabilities, which are usually based on the latest interoperability standards such as CORBA, COM/DCOM, XML and support important information infrastructure like various DBMSes. This is desirable because WfM systems need to deal with a wider range of applications, not just often standalone tools for developing software in PSSE.

Inheritance seems to be another weakness, even at the process entity level. Systems implemented atop of a distributed object system, like the CORBA-based WASA₂, can at best support entity level inheritance. Process behavior level inheritance is generally not supported. The exceptions are WF-net and wOrlds. *WF-net* is a Petri-net based theoretical model for process modeling. Four types of inheritance, namely *protocol*, *projection*, *protocol/projection*, and *life-cycle*, are defined based on the test of *branching bisimilarity*, an equivalence relationship about the observable behaviors of labeled Petri-nets, between two processes by blocking or abstracting tasks in one of them. Though there are transformation rules that define the conditions under which a process inherits another one in terms of the four inheritance semantics, it is not clear how these rules can be used to build new process definitions by customizing existing ones. The model seems more appropriate for reasoning about the similarities and differences of the process behaviors of two processes. P provides the language constructs for building new process definitions and actually implements these constructs. However, similar to any traditional OOL, there is a possibility that programmers may "abuse" inheritance to build a subprocess that may behave very differently from its superprocesses. This is actually a violation to the generalization-and-specialization approach. Unfortunately, from the language and system perspectives, there is little we can do to prevent this sort of abusing.

wOrlds 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*, *Creator*, *Deleter/Creator*, or *Modifier*. This makes it possible for an upper obligation to customize certain entities in lower obligations, a feature 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 is done internally by the system [40].

Table 1. A Comparison of the library capabilities of different process systems.

Perspective System	Base Language		Inheritance	Nesting	Integration	Reflection
<i>P</i>	ODMG model OOL	object based	Process level Object level	Subprocess as an activity	COM/DCOM Native methods Native objects	<i>process activity enactor</i>
for business process reengineering						
<i>Process Handbook</i> [21]	Natural English		Process level	<i>Processes</i> broken into <i>parts</i>	No support	No support
<i>PAL</i> [10]	Natural Language		No support	No support	No support	No support
for software engineering						
<i>APPL/A</i> [27]	Procedural Ada		No support	Process steps as Ada subunits	Ada packages	Stored process state data
<i>CSPL</i> [5]	OO Ada95		Object level	Process steps as Ada tasks	Ada packages <i>tool</i> for external apps.	No support
<i>E³</i> [13]	Graphical with OO capability		Entity level Association level	<i>subtask(...,...)</i>	No support	No support
<i>EPOS</i> [6]	Prolog-based SPELL		Entity level Task level	<i>decomposition repertoire(...)</i>	HP BMS	Meta-class
<i>HFSP</i> [14]	Functional		No support	Activity decomp. as hierarchy func. definition	A <i>tool</i> section for external apps	<i>status snap(...)</i> <i>resume(...)</i>
<i>Little-JIL</i> [4]	Graphical		Sub-typing of certain process entities like exceptions and messages	Sub-steps as child nodes	No support	No support
<i>OIKOS</i> [22]	Limbo: tuple space, angle, coordinator		Document level	Composed angles	A proprietary comm. protocol	Process def. as rule repositories
<i>SPADE-I</i> [2]	O ₂ C + Petri-net based SLANG		Artifact level	Activity as nested Petri-net	Black-box trans. DEC FUSE, OLE2	Activity def. and state as tokens
for workflow management						
<i>FLOWer</i> [32]	Directed Based	Graph	No support	Graph node as a sub-plan or a sub-process	COM/DCOM DLL, CLI Standalone apps	No support
<i>WebSphere MQ Workflow</i> [12]	Directed Based	Graph	No support	Block as a set of activities	IBM MQ Series XML	Predefined data members like <i>_RC</i> , <i>_PROCESS</i> , <i>_ACTIVITY</i> , etc.
<i>Staffware</i> [26]	Directed based	Graph-based	No support	Graph node as a nested graph	COM+/CORBA RDBMSes	No support
<i>WASA₂</i> [35]	Directed Based	Graph	Object level	Graph node as a nested graph	CORBA Business Objects	No support
<i>WF-net</i> [31]	Petri-net based		Process level	A WF-net connected to a larger WF-net via its in/output	No support	No support

Perspective System	Base Language	Inheritance	Nesting	Integration	Reflection
			places		
wOrlds[3]	Petri-net based	Obligation level	An obligation as a network of sub-obligations	CORBA in HP DST	No support ²

5 Concluding Remarks and Future Work

Process programming as a new programming paradigm is believed to be critical in many tightly related areas, e.g. software engineering, workflow management, and business management. We believe that a library with reusable elements for programming cooperative processes flexibly is essential for the acceptance, popularity, and success of this new paradigm. For this purpose, we identify four important mechanisms, i.e. *inheritance*, *nesting*, *integration*, and *reflection*, discuss the general requirements for each of them, and concrete the ideas by showing how they are implemented in P, a programming language and system designed for developing cooperative applications. Comparisons with related work in various process management related areas show that P is the first system that offers a comprehensive solution for building and using process libraries. A public release of the P runtime system that contains all the work discussed here is available at <http://blrc.edu.cn/research/p/>. The system has been successfully used as the programming environment for a graduate level course on CSCW in Tsinghua University and as the software platform for several projects in e-commerce and workflow management.

We are trying to apply the system into more application areas to further test the ideas. We believe there is still a long way to go. Making our library much richer, e.g. by adding more reusable process definitions into our library, by developing special-purposed libraries for different application areas, is in our near-term plan. A long-term work we are currently considering is whether it is necessary to define a standard binary format of process programs, and if the answer is yes, what format should we define. We believe that a standard format will promote significantly process sharing and reusing. This attempt conforms to NIST's goal on Process Specification Language [<http://ats.nist.gov/psl/>] and WfMC's effort on developing a common process interchange standard based on XML [36]. However, our effort differs from these in that it tries to achieve sharing from a much lower level.

The work itself is very challenging due to many, both technical and non-technical, factors. Technically, process programs are executed by process systems, which may adopts different process meta-models. It may be difficult to map the concepts of one meta-model to those of another. However, if this can be done, all process systems will have a common ground to play on. As a result, libraries for process programming may then be developed with and shared among different process languages. Whether this challenge is achievable or not seems dependent on whether we can proof, theoretically, that seemingly very different process meta-models are actually equivalent in their expressiveness.

Acknowledgements

² wOrlds has another part, Introspect, for defining collaborative work within the framework. For reflection, Introspect has a meta-level architecture similar to that of Smalltalk[30]. However, it is not clear, at least from existing literature, how Introspect is related to wOrlds' process management functionalities.

This research is based on the author's PhD work, which was supervised by Prof. Shi, Meilin at Tsinghua University and funded by China NSF and Hi-tech R&D Plan under several grants. The author appreciates the anonymous reviewers' insightful and critical comments on both this paper and its preliminary version in BPM 2003. 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):841-865
- [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] Cass, A. G., Lerner, B. S., McCall, E. K., et al. Little-JIL/Juliette: a process definition language and interpreter. In: *Proc of ACM/IEEE ICSE*, Limerick, 2000, 754-757
- [5] Chen, J. Y. CSPL:an Ada95-like, unix-based process environment. *IEEE TOSE*, 1997, 23(3):171-184
- [6] 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
- [7] Curtis, B., Kellner, M. I., and Over, J. Process modeling. *CACM*, 1992, 35(9):75-90
- [8] Fuggetta, A. Software process:a roadmap. In: *Proc of ACM/IEEE ICSE*, Limerick, 2000, 27-34
- [9] Hammer, M. Reengineering work: don't automate, obliterate. *Harvard Business Review*, 1990, 46(4):101-112
- [10] Hart, H., Doland, J., Drake, D., et al. STARS process concepts summary. In: *Proc of ACM TRI-Ada*, Orlando, 1992, 570-594
- [11] Hollingsworth, D. The workflow reference model. *WfMC-TC00-1003*, 1995
- [12] IBM WebSphere MQ Workflow. <http://www.software.ibm.com/ts/mqseries/workflow>, 2003
- [13] Jaccheri, M. L., Picco, G. P., and Lago, P. Eliciting software process models with the E³ language. *ACM TOSEM*, 1998, 7(4):368-410
- [14] Katayama, T. A hierarchical and functional software process description and its enactment. In: *Proc of ACM/IEEE ICSE*, Pittsburgh, 1989, 243-252
- [15] Katayama, T. and Motizuki, S. What has been learned from applying a formal process model to a real process. In: *Proc of IEEE ISPW*, Washington D.C., 1991, 79-81
- [16] Kellner, M. I. Connecting reusable software process elements and components. In: *Proc of IEEE ISPW*, Dijon, 1996, 8-11
- [17] Kellner, M. I., Feiler, P. H., Finkelstein, A., et al. ISPW-6 software process example. In: *Proc of IEEE ISPW*, Redondo Beach, 1991, 176-186
- [18] Klein, M., Dellarocas, C., and Bernstein, A. eds. Special issue on adaptive workflow systems, *CSCW*, 2000, 9(3-4):265-455.
- [19] Krueger, C. W. Software reuse. *ACM Computing Surveys*, 1992, 24(2):131-183
- [20] Maes, P. Concepts and experiments in computational reflection. In: *Proc of ACM OOPSLA*, Orlando, 1987, 147-155
- [21] 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

- [22] Montangero, C. and Ambriola, V. OIKOS: Constructing process-centered SDEs. In: *Software Process Modeling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Research Studies Press, London, U.K., 1994
- [23] Osterweil, L. Software processes are software too. In: *Proc of ACM/IEEE ICSE*, Monterey, 1987, 2-13
- [24] Paton, N. W. and Diaz, O. Active database systems. *ACM Computing Surveys*, 1999, 31(1):63-103
- [25] Perry, D. E. Practical issues in process reuse. In: *Proc of IEEE ISPW*, Dijon, 1996, 12-14
- [26] Staffware. Staffware 2000/GWD User Manual. *Staffware Plc*, UK, 2000
- [27] Sutton, S. M., Jr., Heimbigner, D., and Osterweil, L. J. APPL/A: a language for software process programming. *ACM TOSEM*, 1996, 4(3):221-286
- [28] Taivalsaari, A. On the notion of inheritance. *ACM Computing Surveys*, 1996, 28(3):438-479
- [29] Tibco. TIB/InConcert process designer user's guide. *Tibco Software Inc*, US, 2000
- [30] Tolone, W. J., Kaplan, S. M., Fitzpatrick, G. Specifying dynamic support for collaborative work within WORLDS. In: *Proc of ACM COOCS*, Milpitas, 1995, 55-65
- [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., ter Hofstede, A.H.M., Kiepuszewski, B., et al.. Workflow Patterns. *Distributed and Parallel Databases*, 2003, 14(3):5-51
- [34] van der Aalst, W. M. P. and van Hee, K. M. Workflow management: models, methods, and systems. MIT press, Cambridge, MA, 2002
- [35] Vossen, G. and Weske, M. The WASA₂ object-oriented workflow management system. In: *Proc of ACM MOD*, Philadelphia, 1999, 587-589
- [36] WfMC. Workflow process definition interface – XML process definition language. *WFMC-TC-1025*, 2002
- [37] Yang, G. and Shi, M. *Cova*: a programming language for cooperative applications. *Science in China, Series F*, 2001, 44(1):73-80
- [38] Yang, G. P language specification and library reference. 2003 (<http://blrc.edu.cn/research/p/documents/pspec.pdf>.)
- [39] Yang, G. Towards a library for process programming. In: *Proc of BPM*, Eindhoven, 2003, 120-135
- [40] Yang, G. Process inheritance and instance modification. In: *Proc of ACM GROUP*, Sanibel Island, 2003, 229-238