Producing and Managing Software Objects in the Process Programming Environment OPM [1)]

Yasuhiro Sugiyama

Department of Computer Science
Nihon University
Koriyama, Fukushima 963, Japan
e-mail: sugiyama@ce.nihon-u.ac.jp

Abstract

This paper reports my experience on applying the process programming environment OPM to the production and management processes of software objects. There are tools and environments that support software production and management processes. Process programs may interact with these tools and/or environments to produce and manipulate software objects. However, it is not always the case that the software processes can get the most desirable support from the tools and environments, because the support is hard-coded in the tools and/or environments. OPM includes the ability to model software objects as well as the production and management processes of these software objects in its process programming language *Galois*. Process programmers can develop process programs that (i) define their own software objects, and (ii) produce and manage the software objects in their own ways. As a result, OPM's support for software objects is highly configurable to meet the diversified requirements of various software processes.

1. Introduction

Process programming [10] is an attempt to describe software development activities by a script written in a programming language that computer systems can directly understand and execute. The script is called a process model or a process program. Process programs embed the knowledge of software experts in computer systems so that software developers can get assistance from the computer systems as if they are assisted by software experts.

One of the key elements of a process program is that it manipulates software objects as well as regular data objects. By software objects, I mean software artifacts, like source programs and executable programs, that can be directly manipulated by computer systems. Software objects behave quite differently from regular data objects. As a result, process programming environments that manipulate software objects need to satisfy various special requirements. Among the special requirements, I particularly focus on the following two issues.

Producing software objects: Software objects are typically produced or built from other software objects by a sequence of transformation. An executable file is produced by linking several object files that are obtained by compiling their source files. Even the source files are often produced from other source files by some tools, like Lex and Yacc. As a result, to produce the final object correctly, all the transformation steps in the software production process must be executed in the right order, and each transformation step must be applied to the right intermediate objects. In addition, the dependencies among the software objects, that are established by the transformation at the creation time, must be recorded and maintained for future enhancements and/or bug fixes to the objects.

Sharing software objects: Software development is a group activity. Software objects are often shared by two or more software developers. Some activities need to be synchronized to keep the integrity of the objects. Simultaneous modification of a shared object by two or more software developers, for example, often conflicts with each other. One might overwrite the valid changes made by other people. As a result, a clear synchronization protocol to access shared objects needs to be specified.

1) This research is in part supported by the Grant-in-Aid for Scientific Research by the Education Ministry of Japan

There are tools and environments that support the production and management of software objects. Make [4] is a tool that supports software production processes. SCCS [11] and RCS [18] are version management tools that implement the reserve/release mechanism so that only one software developer can reserve or check-out a single version at a time for modification purpose. DSEE [8] is an environment that integrates its version control mechanism with its software production support mechanism. Process programs may interact with these tools and/or environments to produce and share software objects. However, it is not always the case that the software processes can get the most desirable support from the tools and environments, because the support is hard-coded within the tools and/or the environments.

About this paper

This paper is an attempt to report my experience on applying the process programming environment OPM [15] to the production and management processes of software objects. OPM includes the ability to model software objects as well as the production processes of these software objects in its process programming language *Galois* [17]. One can define his/her own production and management mechanism of software objects in *Galois*. Process programs written in Galois define software objects, and directly perform transformation to produce and manage the software objects. They keep track of the dependencies among the software objects, and implement a protocol to share objects. As a result, OPM's support for software objects is highly configurable to meet the diversified requirements of various software processes. The goal of this paper is to prove the effectiveness of OPM's approach.

This paper is organized as follows. The second section gives a brief overview of OPM. The third section explains how software objects are defined in OPM. It also illustrates OPM's mechanism, called *derivation*, to model software production processes. Derivation is one of the distinctive features [16] of *Galois* that are specifically designed to support software processes, although *Galois* is syntactically a super set of C++ [12]. The fourth section illustrates how access to shared software objects is defined and controlled in OPM. Derivation is again used to implement the mechanism for shared software object. The fifth section evaluates the effectiveness of OPM's approach, and the last section concludes the paper.

2. A brief overview of OPM

It is a common practice that software developers establish their personal work space in their software development environment before they start their software development activities. They create a working directory and copy necessary files onto it. They set environment variables, and write a shell script so that commands names are bound to the right version of the desired tools. This practice implies the necessity of a *personal environment* [13] to support the development efforts of individual developers. The development of a software system involves a wide variety of software processes. It is quite essential to execute each software process in a personal environment, which is specifically designed for the process, to improve the accuracy, efficiency, and security of the process execution.

OPM is a meta-environment that is able to generate a wide variety of personal environments depending on the nature of the software process model to execute from a high level process description in *Galois*. Each software developer is supported by his/her own personal environment that provides a work space for him/her. The work space includes software objects and activities that are required to accomplish his/her task. The personal environment offers a user interface through which the software developer can manipulate the software objects and carry out the activities in the process. Although software processes are primarily independent entities, the activities of these personal processes are synchronized into larger coherent group processes to support team efforts. Synchronization is achieved by exchanging messages, communicating through shared resources, and/or other methods. As a result, the software development environment, as a while, is a collection of heterogeneous personal environments.

One of the significant features of a personal environment in OPM is that it has embedded knowledge to support the software process that it executes. The process program describes the goal of the software process, such as the documents to be produced and/or the conditions to be satisfied. It also describes the rules and constraint that must be satisfied to carry out its activities, such as "who executes the activity", "when the activity needs to be carried out", "how the activity is carried out", and "where the activity is carried out". The personal environment navigates and/or assists the human activities so that they conform to the description of the process program. The software developer will receive assistance, guidelines, and/or supervision from the personal environment. The personal environment may be an instructor who teaches how to carry out the

activities, a counselor who suggests next activities, a co-worker who performs some of the activities, and/or a supervisor who collects the information on the process execution to improve the process.

OPM supports a window-based graphical user interface, as its standard, in which a personal environment is denoted by a window. Software objects and activities of the personal environment are represented by icons and menus in the window respectively. Software developers will request the process to execute its activity by clicking the menu. On the other hand, personal environments will give assistance and/or supervision to the software developers using dialog boxes and/or other methods. Dialog boxes may be used to send messages to the software developers. Activation and deactivation of menus may indicate the appropriateness of executing individual activities. The menu color may specify the degree of urgency of executing each activity.

3. Deriving software objects

3.1. Software objects in OPM

In OPM, every software object is an instance of its class which is written in *Galois*. Classes serve as a template to determine the internal structure of objects that are their instances. All the instances of a class have the same internal structure. By internal structure, I mean the constituent elements and/or attributes of an object. A software object may be a collection of other software objects, and may have various attributes that explain the nature of the object. Typical attributes include the creation date and the owner name of objects.

A class also determines the operations that can be applied to its instances. Each class assures that only the operations defined in the class can be applied to its instances. Typical operations include those to create, edit, and destroy the instances of the class. Operations to manage versions of objects, and to synchronize multiple accesses to objects are also important operations. Implementation of each operation is hidden from the outside of the class. Operations defined on each object can be invoked by sending a request to the object.

3.2. Derivation

OPM relies on *Galois' derivation* to represent software production processes. Derivation is a mechanism to produce objects from existing objects by transformation. Objects that are created by derivation are called *derived objects*. Figure 1 illustrates a typical derivation to

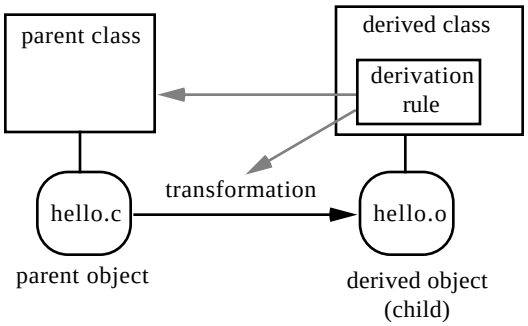

Figure 1: Derivation

produce an object file `hello.o` from its source file `hello.c` by compilation. Derivation is characterized by (i) a derived object to be created, (ii) the *parent objects* from which the derived object is built, and (iii) the operation to transform the parent objects to the derived object. The derived object is also called a *child* of the parent objects. The operation may perform complex transformation, like compilation, but it is not necessary. The operation may be a simple copy operation that copies the value of the parent objects to the derived object.

3.3. Derived classes

OPM assures that each derived object is built with the right derivation by means of classes. Classes whose instances are built by derivation are called *derived classes*. Some people [12] use "derived classes" as a synonym of "sub classes." In *Galois*, however, derived classes and sub classes are related but different concepts. *Galois* has both derived classes and sub classes.

Each derived class specifies a *derivation rule* to produce its instances. A derivation rule consists of (i) the operation to be used for the transformation, and (ii) the classes of the parent objects of the derivation. The classes of the parent objects are called the *parent classes* of the derived class. In Figure 1, the dashed arrows indicate that the derivation rule in the derived class specifies the parent class and the transformation to be used in the derivation. A single derived class may have two or more parent classes so that two or more parent objects can be used to produce a single derived object. Each derived class assures that the transformation to produce its instances is carried out by the operation specified in its derivation rule, and only the objects belonging to the specified parent classes are used as input to the transformation. Examples of derived classes can be seen in [17].

3.4. Backward chaining of derivation

Each class includes the derivation rule of its instances, but it does not include the derivation rules of their parent objects. The derivation rules of the parent objects are defined in the parent classes.

The parent classes assure that the parent objects are ready for use before the derivation is carried out. When a derived class receives a request to build its instance, it sends a request to its parent classes to make the parent objects ready for use. The parent classes create the parent objects, if they are not available, or update the existing parent objects, if they are out of date. The parent classes, in their turns, need right parent objects to produce their instances. They will send a request to their parent classes to obtain the parent objects for their instances. As a result, a backward chain of derivation is established until the requests reach the classes with no parent class.

The mechanism described above gives a default action for backward chaining of derivation. OPM has a mechanism to control the backward chaining of derivation. Each operation can be associated with preconditions, which assure that the operation can execute only when the preconditions are satisfied. Invocation requests to the operations, in which the associated preconditions are not met, can be either denied or delayed for future execution, depending on the class designer's decision.

Backward chaining of derivation can be controlled by specifying appropriate preconditions associated with the operation of the derivation. OPM rebuilds derived objects, as its default, when they are older than their parent objects. In this case, OPM uses the time stamp of the objects in the precondition. However, any preconditions, which may involve human's authorization or comparison of the contents of the objects, can be specified to define more complicated rules for chaining.

3.5. Derivation history

Derivation establishes *derived-from* relationships between the derived and parent objects, which are recorded and maintained even after the object creation is completed. OPM uses the derived-from relationships to identify the parent objects of a given derived object as well as the transformational operation that produced the derived object.

Furthermore, the parent objects, that are identified by the derived-from relationships, can be manipulated as if they are parts of the derived object. Operations on the parent objects can be invoked as if they are the operations of the derived object. When the derived object receives a request to invoke one of the operations on its parent objects, the derived object forwards the request to the appropriate parent object. The mechanism to forward the request is called delegation [9]. If an executable file is given, for instance, the source files and the compilation and link operations to produce the executable file can be instantaneously identified. If any bugs are found in the executable file, one can even apply the edit operations to the source files to fix the bugs through the executable file.

3.6. Forward chaining of derivation

OPM also uses the derived-from relationships to identify the children of a given object. When an object is modified, its children are notified by the object because the children might need to be rebuilt using the new parent object. Similarly, if the children are rebuilt, the objects that are derived from the children, called *grand children*, are notified by the children because they might need to be rebuilt. As a result, a forward chain of derivation is established.

Some changes in an object do not require the reproduction of its children. The decision if each child is rebuilt or not is made by the precondition associated with the operation to rebuild the child. OPM uses the time stamp of objects, as its default, in the precondition. However, one can specify any preconditions, as one does to control backward chaining of derivation. In some cases, the modification of an object results in the creation of new versions of the object. In such cases, its children keep looking at the existence of new versions, instead of looking at its time stamp.

3.7. Derivation and Instantiation

Instantiation is a very common object creation mechanism that can be found in most programming languages with classes or types. Derivation is a generalization of instantiation. More precisely, instantiation is a special case of derivation in which no parent object is involved.

Like derivation, instantiation creates an object based upon its class or its type that is a template to determine the structure of the object. The value of the object is initially determined by the instantiation operation. The instantiation operation may take parameters to determine the value of the object to be created, as derivation takes the value of the parent objects to determine the value of the derived object. As a result, instantiation can partially

simulate the effect of derivation by supplying the value of the parent objects to the instantiation operation as its invocation parameters. However, instantiation does not record the parent objects that are used to produce the object, unless the object explicitly records them. This results in the following major differences between derivation and instantiation: (i) instantiation does not establish derived-from relationships; (ii) the parent objects of a given derived object cannot be identified through the derived object; (iii) the parent objects of a given derived object cannot be accessed through the derived object; (iv) instantiation of an object does not trigger the instantiation of its parent objects, unless the parent objects are constituent elements of the object; (v) modification of an object does not rebuild its children.

3.8. Supporting mechanisms for derivation

OPM provides an object browser that is a multiple-paned window showing information of software objects in the environment. The object browser allows software developers to examine the attributes, the parent objects, the child objects, and other information of the objects.

In the personal environments that OPM offers, as I mentioned earlier, software objects are shown as icons that have a pop-up menu of its operations. The pop-up menus can be hierarchical to illustrate the derivation history of objects. The top level menu of an object includes its own operations as well as the names of its parent objects. Moving the mouse cursor onto one of the parent object names pops up a second level menu that includes the operations of the parent object. The operations of the parent object can be directly invoked from this menu. The second level menu also includes the parent object names of the parent object that is selected at the first level. Moving the mouse cursor onto one of them pops up a third level menu.

4. Shared objects and access control

In the previous section, I explained how software objects are defined and produced in OPM. The goal of this section is to clarify how software objects are accessed in a multi-agent environment. In the *OPM* environment, all the software objects are initially stored in an object database which is maintained by the object manager. Processes are not allowed to access software objects directly. Processes will issue a request to the object manager when they need to access some software objects. Then each process will receive a child of the software object from the object manager. When two or more processes need to access the same software object, they will receive different children of the single software object from the object manager, but a single child cannot be shared by two or more processes.

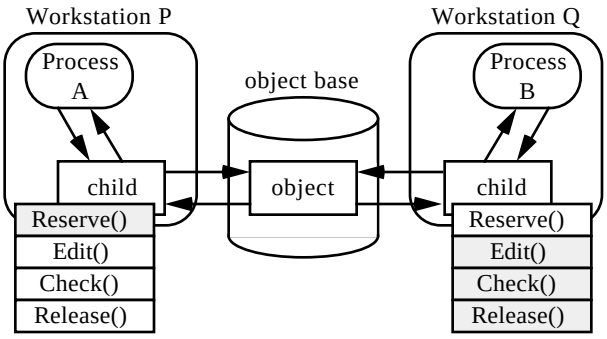

Figure 2: Mutual exclusive access to a shared software object

Once a process receives a child from the object manager, the process is allowed to directly access the child. Children of an object accept requests to access the object from software processes, and forward the requests to the object, as Figure 2 illustrates. This guarantees distributed access to the software object. However, the children are not "pointers", like symbolic links and alias, that simply forward the requests to the object. Symbolic links and/or alias are a handy way to share files among multiple software engineers in current operating systems and environments. By creating symbolic links or alias to the software objects in their concern in their own working directory, software engineers can access the files as if the files exist in their own directory regardless of the actual location of the files. However, symbolic links and alias do not allow fine grain access control to the files that they point to. One cannot define operations on symbolic links and alias to coordinate simultaneous access to a single file.

Children of an object, on the other hand, are themselves objects that can have their own operations. Fundamentally, each process can access a software object through a child of the software object independently of other processes which keep children of the same software object. Some access to the software object, however, must be carried out synchronously. Operations on the children will be used to control the simultaneous access to the object.

4.1. Reserve/Release mechanism

One of such synchronization technique is *mutual exclusive* access control which guarantees that only one process is allowed to access a software object at a time. The software object in Figure 2 provides a

reserve/release mechanism like SCCS and RCS to implement the mutual exclusive access control. Each child of the software object provides four operations: `Reserve()`, `Edit()`, `Check()`, and `Release()`. A process must execute the `Reserve()` operation first before it executes any other operations. Successful execution of the `Reserve()` operation guarantees mutual exclusive access to the software object for the process. Successive execution of the `Reserve()` operation by other processes will be denied or delayed until the first process executes the `Release()` operation that allows another process to get an access right to the software object.

Figure 3 shows how this reserve/release mechanism can be implemented. Each child of the software object has an instance variable called `locked`. The `Edit()`, `Check()`, and `Release()` operations are associated with a precondition so that they can execute only when `locked` is **`True`**. The instance variable `locked` is originally set to **`False`** when each child is created, and is set to **`True`** by the `Reserve()` operation. As a result, the `Edit()`, `Check()`, and `Release()` operations can execute only after the `Reserve()` operation completes. Notice that each child has its own instance variable `locked`. Therefore `Edit()`, `Check()`, and `Release()` can execute only on the child which successfully completes the `Reserve()` operation.

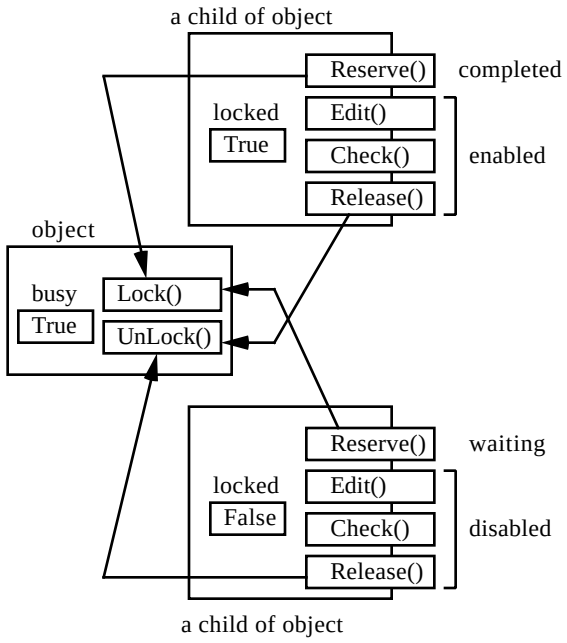

Figure 3: Synchronization mechanism

4.2. Mutual exclusive access control

One of the most important issues here is that the execution of the `Reserve()` and `Release()` operations requires synchronization among all the children of the software object to assure that only one child could succeed in executing the `Reserve()` operation. Otherwise, two or more processes are allowed to execute the `Edit()` operation simultaneously. Generally speaking, when children of a software object receive requests which need synchronization, the children will process these requests synchronously by communicating to each other through the software object, their parent. The software object will work as a communication center among its children.

As Figure 3 illustrates, the `Reserve()` operation on the children invokes the `Lock()` operation in the software object. Although *Galois* provides a concurrent environment in which multiple objects can work simultaneously, each object is essentially a sequential process like monitors [3][6]. Each object will carry out only one of its operations at a time. As a result, even if two or more children try to execute the `Lock()` operation simultaneously, these requests are processed sequentially. Furthermore, the software object has an instance variable `busy`, and the `Lock()` operation is associated with a precondition so that it can execute only when `busy` is **`False.`** The instance variable `busy` is originally set to **`False`** when the software object is created, and is set to **`True`** by the `Lock()` operation. As a result, only the first request to execute the `Lock()` operation is granted, and successive requests to the `Lock()` operation will be delayed.

The `Release()` operation invokes the `UnLock()` operation on the software object which sets the instance variable `busy` to **`False`** so that the requests to execute the `Lock()` operation by other children can be granted. As a result, even if two or more processes try to reserve the software object, only the first process will succeed in reserving the software object, and all other requests are delayed until the first process releases the software object.

4.3. Scheduling Software Objects

The question arises here of how to schedule the delayed requests to execute the `Lock()` operation, when two or more processes try to reserve the software object. *Galois'* approach to support scheduling is totally different from existing languages that support concurrency, such as CSP [7], Ada [1], and Concurrent C++ [5]. *Galois* does not require low level primitives, such as semaphors, to express concurrency.

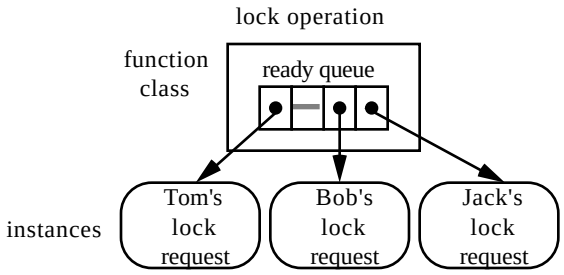

Figure 4: Scheduling multiple invocation

In *Galois*, delayed requests are processed later based upon the scheduling algorithm associated with the class of the operation. *Galois* treats operations as classes. Whenever an operation is invoked, its instance is created. Multiple invocation requests of a single function class will result in the creation of multiple instances of the operation class. The function class serves as a manager and a coordinator of the multiple instances. The class of the function class determines how multiple instances are scheduled.

Figure 4 illustrates a typical implementation of the `Lock()` operation that processes multiple invocation requests in a FIFO order. The `Lock()` operation maintains a FIFO queue of its instances which are waiting for being executed. When the `Lock()` operation is invoked, a new function instance is created. If no other function instance is executing and the associated precondition is met, the newly created instance will be immediately executed. However, if some other instance is executing or the associated precondition is not met, the newly created instance will be placed into the FIFO ready queue, and its execution will be delayed. When the executing instance completes and the associated precondition is met, the instance on top of the ready queue will be selected and executed. As a result, multiple requests to execute the `Lock()` operation are processed in the FIFO order.

4.4 Supporting Mechanism

In the personal environments that OPM generates, children of software objects are also denoted as icons that have a pop-up menu of their operations. In Figure 2, the children have a pop-up menu containing the four operations. OPM visualizes the state of each operation, to indicate if its associated precondition is met or not, by enabling or disabling the corresponding menu item. A menu item is shaded when its precondition is not met. Shaded menu items cannot be selected to avoid a run-time error and/or delay of execution. In Figure 2, on one child, all the operations except `Reserve()` are disabled because `Reserve()` operation is not executed yet. On the other child, only the `Reserve()` operation is disabled because `Reserve()` operation is already executed, and it is not necessary to reserve the same object twice.

OPM also offers a mechanism to examine the state of the delayed invocation requests to operations. When an invocation request of an operation is queued, OPM generates a window that includes a list of delayed requests to the operation. Through this window, software developers can examine the location of their requests in the queue. They may remove their delayed requests if they wish, or they may postpone the execution of their requests until midnight or some other time so that their requests can be processed more quickly.

5. Evaluation

To verify the effectiveness of OPM's approach, I tried to implement the features of existing tools such as Make and SCCS in *Galois*. I used the key mechanisms of OPM, which I presented in this paper, to implement these tools. This attempt was quite successful. This shows that the *Galois'* mechanism is quite useful and essential to implement these tasks effectively.

Furthermore, the use of classes in OPM showed several advantages over existing tools. Classes allow one to easily reuse software objects defined by some other people. One can localize the object creation rule within the class. One can reuse objects without knowing how the objects are created. One just needs to create an object as an instance of the class that has the desired derivation rule. In case of Make, a typical Makefile includes the creation rules of two or more components of a system. One needs to extract the necessary creation rule from the Makefile to reuse one of the components. As a result, it is not so easy to reuse a particular component of a software system. In OPM, on the other hand, such extraction is not necessary because each derived class has its own derivation rule. I have implemented a new tool, called Object Make [14], that simulates the use of classes within the framework of Make, to feed back the lessons learned from this experience to the original tool.

Another advantage of OPM is that the support for software production and management processes can be easily integrated with other software object management mechanisms, like a version control mechanism. DSEE integrates its version control mechanism with its software production support mechanism. However, since the version control and the software production support

mechanisms are hard-coded, it is not possible to add a new version control mechanism and integrate it with the original software production support mechanism. OPM, on the other hand, implements version control mechanisms in terms of operations of objects. Two or more version control mechanisms can be easily mixed in a single environment without interfering to each other by defining them in different classes. I do not intend to describe the version control mechanisms of OPM within this small section. The point is that the software production support can be easily integrated with any of the version control mechanisms without worrying about interface problems between different tools.

Before concluding the paper, I would like to make some quantitative evaluation of OPM's approach of including derivation rules of objects in the classes of the objects. I will show how OPM can reduce the effort to design and produce the creation rules of a single software system compared with Make.

I will use the COCOMO model [2] to compute the effort. Creation rules are also software. In case of Make, creation rules in Makefiles are written in a shell scripting language. In case of OPM, derivation rules in classes are written in *Galois*. As a result, the effort to build them can be computed as we compute the effort to build software systems. COCOMO calculates the effort (*MM*) to build a software system based upon *DSI* (the number of Delivered Source Instructions). Here, I will use the following COCOMO Semidetached Mode Effort Equation:

$$MM = 3.0\left(\frac{DSI}{1000}\right)^{1.12}$$

However, when some of the components are adapted from existing software, one cannot directly use *ADSI* (the number of delivered source instructions adapted from existing software) in place of *DSI*. To handle the effects of adapted software, COCOMO calculates an *EDSI* (Equivalent number of delivered source instructions) from the *ADSI* which is defined as follows:

$$EDSI = ADSI\,\frac{AAF}{100} + DSI_{new}$$

where DSI_{new} is the *DSI* of the components to be created from scratch, and *AAF* (adaptation adjustment factor) is defined as follows:

$$AAF = 0.40\,DM + 0.30\,CM + 0.30\,IM$$

where DM (percent Design Modified) is the percent of the adapted software's design which is modified in order to adapt it to the new system, CM (percent Code Modified) is the percent of the adapted software's code which is modified in order to adapt it to the new system, and IM (percent of Integration required for Modified software) is the percentage of the effort required to integrate the adapted software into the new system and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

In order to show how OPM can reduce the effort compared with Make, I will compute:

$$ratio = \frac{MM_{make}}{MM_{opm}}$$

where MM_{make} is the effort when I use Make, while MM_{opm} is the effort when I use OPM. Notice that:

$$\frac{MM_{make}}{MM_{opm}} = \frac{3.0\left(\frac{EDSI_{make}}{1000}\right)^{1.12}}{3.0\left(\frac{EDSI_{opm}}{1000}\right)^{1.12}} = \left(\frac{EDSI_{make}}{EDSI_{opm}}\right)^{1.12}$$

As a result, I just need to compute the ratio $\frac{EDSI_{make}}{EDSI_{opm}}$.

Consider a system consisting of *n* components. Among them, *p* components are adapted from existing systems, while the remaining *n-p* components are newly built from scratch. Assume the average size of the derivation rules of the components is *b*.

To write a Makefile of the new system, I have to look at the Makefiles of the existing systems that include the components adapted by the new system. Assuming that the existing systems contain *k* components, I have to examine the Makefiles of the existing components consisting of *kb* lines to extract the necessary *pb* lines. Then I obtain:

$$IM_{make} = \frac{kb}{pb} \times 100 = \frac{k}{p} \times 100 = \frac{10^4}{u}$$

where

$$u = \frac{p}{k} \times 100$$

denotes the percent of the adapted components in the original systems. To make the story simple, I assume that once the creation rules of the adapted components

are extracted from the containing Makefiles, they can be reused without modification. This implies:

$$DM_{make} = 0 \text{ and } CM_{make} = 0$$

I obtain:

$$AAF_{make} = 0.4 \times 0 + 0.3 \times 0 + 0.3 \times \frac{10^4}{u} = \frac{3 \times 10^3}{u}$$

As a result:

$$EDSI_{make} = pb \times \frac{30}{u} + (n - p)b = \frac{30\,pb}{u} + \frac{n}{100}(100 - t)b$$

where

$$t = \frac{p}{n} \times 100$$

denotes the percent of the adapted components in the new system.

On the other hand, when I use OPM:

$$DM_{opm} = 0 \text{ and } CM_{opm} = 0 \text{ and } IM_{opm} = 0$$

because no extra work is necessary for adaptation. As a result, I obtain:

$$EDSI_{opm} = (n - p)b = \frac{n}{100}(100 - t)b$$

Therefore:

$$\frac{EDSI_{make}}{EDSI_{opm}} = \frac{\frac{30\,pb}{u} + \frac{nb}{100}(100 - t)}{\frac{nb}{100}(100 - t)} = \frac{30t}{u(100 - t)} + 1$$

Finally I obtain:

$$ratio = \frac{MM_{make}}{MM_{opm}} = \left(\frac{30t}{u(100 - t)} + 1\right)^{1.12}$$

Figure 5 shows *ratio* as a function of *t.* From this graph, I can observe several facts. First of all, the value of *ratio* is always bigger than 1. This implies that Make requires more effort than OPM does. When *u* is fixed, *ratio* increases as *t* increases. This implies that Make requires more effort when one adapts a large number of components, than it does when one adapts a small

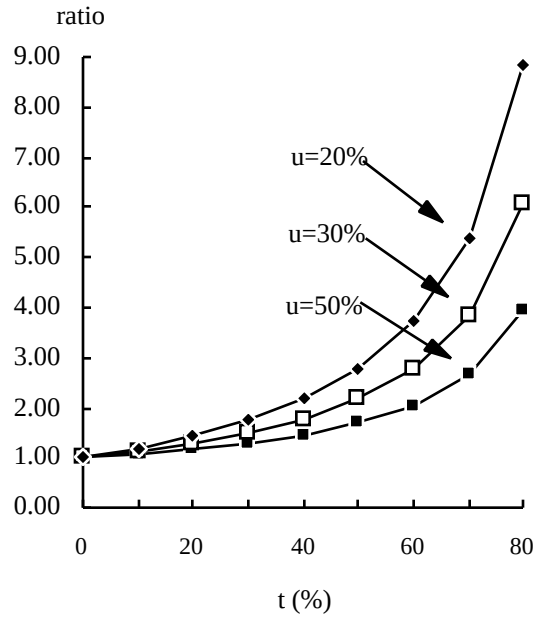

Figure 5: Effort comparison of Make and OPM

number of components in the new system. When *t* is fixed, *ratio* increases as *u* decreases. This implies that Make requires more effort when one adapts components from large systems, than it does when one adapts components from small systems.

6. Conclusion

I believe that I have clarified several key features of OPM that prove the effectiveness of its software production and management support, although I was able to present only a small portion of my experience due to the space limitation. Derivation is one of such important and new features of *Galois*.

The object production and management mechanisms themselves, which I presented in this paper, may not be conceptually new. The important issue, however, is that they are implemented in a new way. They are not hard-coded in OPM. They are programmed by process programs written in *Galois*. I have presented that I can implement the object production and management mechanisms easily and effectively if I use the key features that OPM and *Galois* offer. I also presented the advantages of OPM's approach over the traditional approach. This implies that *Galois* is an effective vehicle to implement the mechanisms to produce and manage

objects. Furthermore, it will make us possible to build our own object production and management mechanisms that have not been implemented by any other tools and environments.

OPM is in its prototype stage and it keeps evolving. OPM itself is written in *Galois*. Any new features can be defined in *Galois* at any time, and they can be easily added to OPM without interfering with existing features. I am currently engaged in the research on the management of software processes. In OPM, software processes are also software objects that need to be produced and maintained. I am verifying the effectiveness of derivation to support changes in software processes.

Bibliography

[1] *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983. United States Department of Defense, 1983.

[2] Boehm, B. W. *Software Engineering Economics*. Prentice Hall, 1981.

[3] Brinch-Hansen, P. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[4] Feldman, S. I. MAKE - A Program for Maintaining Computer Programs. *Software-Practice and Experience*, vol.9, pages 255-265, 1979.

[5] Gehani, N. H. and W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software-Practice and Experience*, vol.18, no.12, pages 1157-1177, John Wiley & Sons, Ltd., December, 1988.

[6] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, vol.17, no.10, pages 549-557, October, 1974.

[7] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM*, vol.21, no.8, pages 666-677, August, 1978.

[8] Leblang, D. B., R. P. Chase Jr. and G. D. McLean Jr. The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. in Proceedings of the *IEEE Conference on Workstations*, pages 266-280, San Jose, California, IEEE, November, 1985.

[9] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. in Proceedings of the *ACM Symposium on Object-Oriented Programming Systems, Languages and Applications*, pages 214-223, Portland, Oregon, ACM, September, 1986.

[10] Osterweil, L. J. Software Processes are Software Too. in Proceedings of the *9th International Conference on Software Engineering*, pages 2-13, Monterey, California, ACM-IEEE, March, 1987.

[11] Rochkind, M. J. The Source Code Control System. *IEEE Transactions on Software Engineering*, vol.SE-1, no.4, pages 364-370, December, 1975.

[12] Stroustrup, B. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.

[13] Sugiyama, Y. Incorporating Your Process Support into Your Software Development Environment. in Proceedings of the *8th International Software Process Workshop*, Walden, Germany, IEEE, March, 1993.

[14] Sugiyama, Y. Configuration Management with Object Make. *Software Engineering Notes*, vol.94, No.18, SIGSE, Information Processing Society of Japan, 1994, in Japanese.

[15] Sugiyama, Y. and E. Horowitz. OPM: An Object Process Modeling Environment. in Proceedings of the *5th International Software Process Workshop*, pages 134-136, Kennebunkport, Maine, ACM-IEEE, October, 1989.

[16] Sugiyama, Y. and E. Horowitz. Language Support for Object Process Modeling. in Proceedings of the *6th International Software Process Workshop*, Hokkaido, Japan, October, 1990.

[17] Sugiyama, Y. and E. Horowitz. Building Your Own Software Development Environments. *Software Engineering Journal, Special Issue on Software process and its support*, vol.6, no.5, pages 317-331, IEE, September, 1991.

[18] Tichy, W. F. RCS - A system for Version Control. *Software - Practice and Experience*, vol.15, no.7, pages 637-654, July, 1985.