Introducing Synchronization into an Object-Oriented Reuse Library

Yunyau Shih

Department of Computer Science
Thomas J. Watson School of Engineering and Applied Science
Binghamton University, Binghamton NY 13902-6000
Phone: (607)777-4802, Fax: (607)777-4822
Email: yunyau@bingsuns.cc.binghamton.edu

Les Lander

Department of Computer Science
Thomas J. Watson School of Engineering and Applied Science
Binghamton University, Binghamton NY 13902-6000
Phone: (607)777-2309, Fax: (607)777-4822
Email: lander@bingsuns.cc.binghamton.edu

Abstract

We discuss how to build an object-oriented software reuse library emphasizing the inclusion of synchronization constructs and seeking to overcome the conflict between synchronization and inheritance. Several synchronization primitives and concurrency control policies which permit automatic synthesis are presented. A preliminary scheme for component classification, based on synchronization properties, is also proposed.

Keywords: Concurrency, Object-oriented, Reuse library, Synchronization.

Workshop Goals: Discuss approaches to including synchronization in object-oriented reusable components.

Working Groups: Reuse and OO Methods and Tools and Environments

1 Background

Software reuse has been advocated as a means for overcoming the software crisis [1]. Software reuse can be divided into two categories–horizontal reuse and vertical reuse [2]: horizontal reuse refers to building and using a component so that it can itself continue to evolve for other applications, by parameterization or otherwise. On the other hand, when reuse is effected using inheritance and without changing existing classes, thereby creating class hierarchies, it is called a vertical reuse.

There are many reuse-based systems available commercially or in research environments [3]. However most of them target code for sequential execution. In this position paper we focus on vertical reuse, especially for an object- oriented software reuse library, which emphasizes concurrency. We believe that vertical reuse is better than horizontal reuse because existing components are not modified and remain validated as the system evolves.

When a reuse library for a concurrent object-oriented software is established, inheritance has to be considered. Nevertheless, synchronization is not orthogonal to inheritance [4]. In a concurrent object-oriented language, synchronization mechanisms can be divided into two categories [5]: *object-level* and *in-code*. "Object-level" synchronization regulates the invocations of methods (operations) based on the current state of an object. That is, the object-level synchronization mechanism is like a gatekeeper which is global to the entire object. "In-code" synchronization is the one where the traditional synchronization mechanisms (semaphore, fork, etc) are used and the synchronization is scattered in the code of the methods. In-code synchronization is error prone and we will not pursue it. Although object-level mechanisms can also be divided into several approaches, most of them produce some kind of conflict with inheritance [6]. Frequently, methods inherited from a superclass have to be re-coded in the subclass in order to achieve synchronization between the methods from the superclass and the new methods of the inherited class. Such problems are counterproductive in software development.

The basic problem can be described as follows. When a class P is defined, the synchronization specification is based on the methods existing in class P. When a subclass, Q, is defined and a new method m is added, the synchronization in the superclass P may not be compatible with the new method m because the original relationships may conflict with the new method. An example, which is used quite often in the literature [7, 6, 8, 9], is described as follows. Suppose a direct superclass implements a queue with the standard operations "Get" and "Put." Now, if we wish to use inheritance to add a new method called "Get2," which fetches two elements atomically, the synchronization defined in the parent is not quite right because "Get2" is not defined there. Thus, when "Get2" is invoked, what is the relationship between "Get2" and the other methods in the direct superclass P? No synchronization information has been given (or can be given) in the superclass because "Get2" does not exist there. Also, a programmer cannot simply add synchronization information in the inherited class because it will have to involve modifications to the synchronization constructs of the superclass.

We will discuss how to build a concurrent object-oriented software reuse library systematically from the synchronization point of view in the remainder of the paper. We will also introduce a model COORL to illustrate the discussion.

2 Position

2.1 Basic Structure of the COORL Model

To enhance software reuse we propose that an *abstract* component, i.e. an abstract class, be the basic unit for software reuse. The reason is twofold. First, it is not easy for people to become familiar with a great variety of programming languages and a simple specification language, free of implementation detail should be easy to learn. Second, it is intended to generate the corresponding implementation classes automatically using a translation tool. This kind of tool for translation is quite common, e.g. [10]. A high level structure for the two classes is shown in Figure 1. Thus, in COORL, there are two kinds of classes: *abstract* and *implementation*. An abstract class uses a high level language such as a program design language (PDL) or a specification language to describe its functionality abstractly while an implementation class contains the specific programming language source code for the corresponding abstract class which can be compiled and executed.

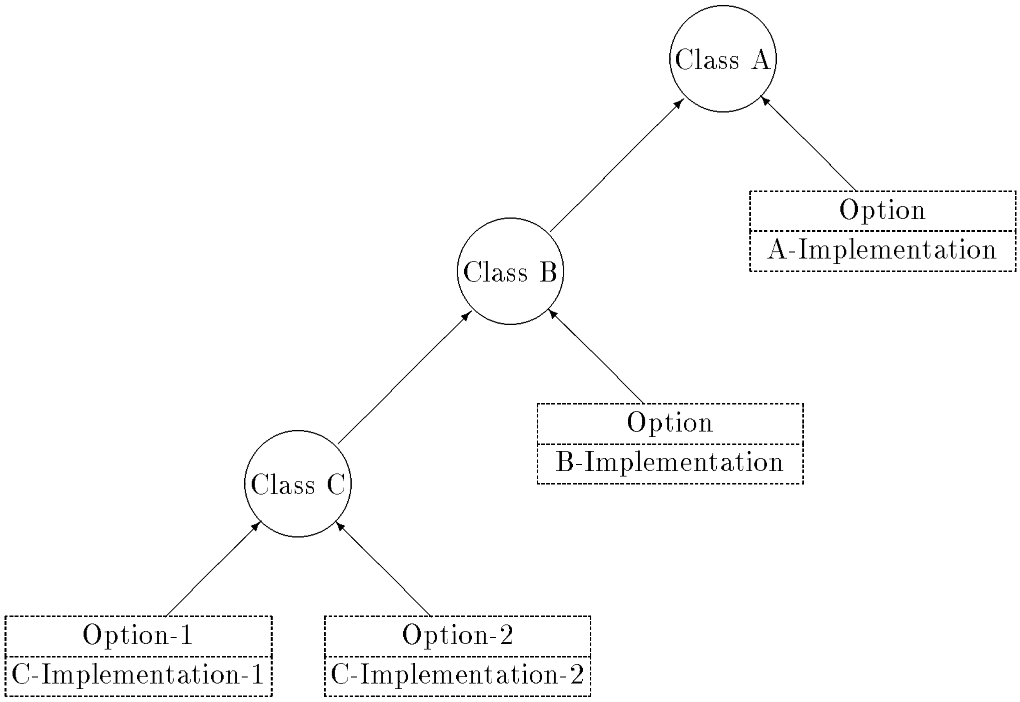

Figure 1

In Figure 1, Class X is an abstract class while X-Implementation-N is its corresponding implementation class. N is an optional number or other identifier and is used to enumerate different implementations of Class X. There is an Option part accompanying each implementation class. It may be used to store relevant information such as a data structure. For example, Class A and Class B in Figure 1 are abstract classes while A-Implementation, and B-Implementation are their corresponding single implementation classes. C-Implementation-1 and C-Implementation-2 are two different implementation classes for Class C, which is an abstract class. Option-1 and Option-2 could be two different kinds of data structures for implementing Class C. For instance, Class C might be a bounded buffer and then C-Implementation-1 might use an array to implement Class C while C-Implementation-2 might use a linked list. It is common for a reuse library to have a browser or other query access system. When the browser or a query language is used to skim a library built according to COORL, users primarily access abstract classes. A sophisticated user can also request the different options used to implement a specific abstract class but that is not encouraged because such low level information should be shielded from a user in general to promote encapsulation, not to mention protecting the implementation code itself.

This paper focuses on a high level view on the synchronization issue and ignores other aspects. An earlier version of COORL can be found in [11].

2.2 Synchronization Mechanism for a Reuse Library

2.2.1 General Structure

An ideal, if not the best, synchronization mechanism for a concurrent software reuse library, based on our experience, is a declarative mechanism. Synchronization primitives for a class should be placed in a predefined segment (or section) so that it is easy to modify them for reuse. Thus, we use an object-level synchronization mechanism in COORL. For example, see the Control segments in Figure 2 (ignore the syntax for the time being). Of course, when a declarative synchronization mechanism is used, the primitives will inevitably lose some flexibility compared with the traditional synchronization mechanisms, such as semaphore, fork, etc. But overall, reusability increases, especially in our model when an abstract class is the basic reuse unit.

There exist many declarative mechanisms for concurrent object-oriented languages but many of them have the same basic constructs, i.e. synchronization counters, such as req(op), wait(op), start(op), exec(op), etc., where "op" represents an invocation of a method. These kinds of primitives can be found in Guide [7], DRAGOON [12], and Scheduling Predicates [13], etc.

In COORL, we can synthesize those synchronization counters once a subclass has been defined. As long as the counters are consistent, our primitives can be used to manipulate them and, at the same time, to avoid the conflict between the synchronization and inheritance. In order to enhance the expressive power for a declarative synchronization mechanism, we merge two other constructs, namely the *atomic block* and *transaction* concepts [14], in the current version of COORL.

2.2.2 Basic Synchronization Primitives

In the first version of COORL [11], there are four basic synchronization primitives used to construct a concurrent object-oriented software reuse library.

- $\#\{X\}$ Operator $\#\{Y\}$: X and Y are methods. $\#\{X\}$ denotes the number of invocation of X. "Operator" represents the relation between the numbers of invocations of X and Y. "Operator" might be $=, >, <, \geq, \leq$, etc. The symbol "#" is simply illustrative; finer levels of control can be used, such as exec (X), term (X), act (X) used in Guide and DRAGOON.

The following three synchronization clauses are used to reflect a new relationship when a new method is defined based on an existing class.

- $\{X\} = +\{Y\}$: the symbol "+" means that method X and method Y have the same effect on the class in terms of the number of invocations of X and Y. "+" is the default and can be omitted, see Figure 2(b).

- {X} = - {Y}: on the other hand, "-" (i.e., minus) indicates that X and Y will have opposite effects. Conceptually, Get and Put can be denoted as {Put} = - {Get}, although this is not actually used in the example in Figure 2.
- {X} = n{Y}: "n" is a number representing how many invocations of method Y equate to the number of invocations of X. For example, {Get2} = 2 {Get} indicates that method "Get2" has the effect of two invocations of method "Get."

<table>
<tr><td>Class P:
Control:
  #Put > #Get
Code:
 Public:
  Get{...}
  Put{...}

(a) Class P</td><td rowspan="2">Class P:
Control:
  #Put > #Get
Code:
 Public:
  Get{...}
  Put{...}

Class Q: Inherit P
Control:
  #Put > #Get + #Pop
Code:
 Public:
  Get
  Put
  Pop{...}

(c) Library after Class Q is generated</td></tr>
<tr><td>Class Q: Inherit P
Control:
  {Put} = + {Get}
Code:
 Public:
  Pop{...}

(b) Define Class Q</td></tr>
</table>

Figure 2

The underlying system will generate a set of synchronization clauses for a new class by interpreting the user's instructions. The synchronization in Figure 2(c) provides an example; it is a combination of the synchronization clauses in Figure 2(a) and Figure 2(b). That is, "#{Put} > #{Get} + #{Pop}" can be obtained by merging "#{Put} > #{Get}" and "{Pop} = +{Get}."

In COORL the synchronization would be visible to users by means of a browser or a keyword search (classification issues will be addressed in the next section) while the implementation of the methods will be shielded from users because of encapsulation. It is useful that the synchronization clauses of a class should be part of its interface to (or contract with) its users. If only those synchronization clauses which are added to a subclass Q of P were visible (which corresponds to the traditional inheritance approach), then it is possible that the full suite of active synchronization clauses would become scattered throughout a class hierarchy and be extremely difficult to understand as a whole. Therefore, it should not be considered a redundancy to make all the synchronization explicit in each subclass. Besides, there is not much burden on the user because the underlying system will generate the synchronization clauses automatically as each new subclass is entered into the library.

In COORL a second level of synchronization construct is available. It is similar to the guard statement. Thus, apart from the synchronization clauses described above, which certainly provide an object level synchronization mechanism and handle the overall relationships between methods, a guard statement may be attached to a method, if necessary. Though a guard is somewhat at the method level, it is not scattered inside the code of a method and can be managed easily. The most significant difference between these two mechanisms is that state variables are involved in this second kind of synchronization mechanism. For instance, method "Put" in a class defining a queue should not be invoked unless there is an empty space left. This information will be reflected in the guard statement for method "Put," not in the object-level synchronization clauses which are enforced by a Control segment. For a detailed description and examples for the second level synchronization construct, please refer to [11]. This distinction is adopted to enforce the encapsulation principle for object-orientation.

2.2.3 Concurrency Policy

Synchronization and concurrency are different faces of the same coin. Basically there are two points of view concerning concurrency. One point of view is that concurrency should be identified implicitly, e.g., a compiler might be used to detect any parallelism available for a program in a specific hardware environment. Though it is quite successful in some specific applications, this approach is a great challenge for a compiler in general applications. The other point of view is that programmer annotations are needed to exploit possible concurrency in general programs, at least the programmer must be provided with constructs that express concurrency in order to add optional variants, extensions or improvements to what a compiler might provide automatically.

We are investigating what concurrency policies are appropriate to be indicated explicitly, while we include atomic block and transaction policies in our current version of COORL. The keyword *Atomic* and *Transaction* are used in the control segment to indicate the execution sequence. When no concurrency policy is indicated, the methods in an object can be executed concurrently. Of course, that also depends on the underlying hardware environments. When a method, which is declared to be "atomic," is executed, other methods in the same object will be suspended. "Transaction" is used by one object to inform other objects to block other invocations until it returns the results from a method which is identified as a transaction method. "Transaction" is a very strict policy.

We notice that some existing concurrency policies are too rigid. We would like to merge data flow and atomic concepts to increase parallelism. Thus, an atomic method might suspend some methods based on the data flow relationship, but not block all the others.

2.3 Classification Scheme for Locating and Retrieving

Our classification scheme follows Prieto-Diaz's approach [15] but in COORL we define another "facet" for synchronization. The introduction of the new facet is justified as follows. It is possible for a set of the same methods to be regulated by different synchronization requirements. Though it is possible to be distinguished by the Medium facet [15], it is desirable to add a synchronization facet because synchronization is an important characteristic for a concurrent software library. If we could define a good domain for the facet, search-time in the library would be cut down dramatically.

It is possible for the synchronization facet to be activated by two approaches. One is by some predefined keywords; the other is by the synchronization primitives described in the previous section. The latter is not encouraged because it violates encapsulation. Some predefined keywords could be as follows: Space, Time, Order, Priority by X (X could be time, length, ...), FIFO, LIFO, Descending, and Ascending. We are in the process of studying a more structured keyword representation.

3 Comparison

We discussed how to build a concurrent object-oriented software reuse library and a scheme to classify a reuse component from the synchronization point of view. We have explored these issues using the COORL model. We believe the approach is better than only developing a new language as in [12, 7, 6, 8] because of the conflict between inheritance and synchronization. With our approach a concurrent object-oriented software reuse library can be built without the conflict.

With respect to concurrency policy, even though similar concepts are adopted in Meld [14], reuse is not a concern there. Therefore, code tracing and code modification are often needed when new methods are added to an existing class to obtain a subclass. This approach violates encapsulation. We argue for putting the concurrency policies in the control segment, thus conforming our placement of synchronization primitives.

Our classification scheme for a reuse library follows Prieto-Diaz's approach [15] but we argue for defining another "facet" for synchronization in subsection 2.4.

Our approach will enhance reusability by being easy to access, understand, and modify. With synchronization primitives and concurrency policies expressed in the control segment, software reuse can be improved.

References

[1] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[2] Y. Shih, "Synchronization in a reusable concurrent object-oriented software library," tech. rep., Department of Computer Science, Binghamton University, 1993. In preparation.

[3] T. Levine, "Reusable software components," *ACM Ada Letters*, pp. 62–73, May/Jun 1993.

[4] P. Wegner, "Dimensions of object-based language design," *OOPSLA'87*, pp. 168–182, 1987.

[5] Y. Shih, "Survey of the synchronization in concurrent object-oriented programming languages," tech. rep., Department of Computer Science, Binghamton University, 1992.

[6] D. Kafura and K. Lee, "Inheritance in actor based concurrent object-oriented languages," *ECOOP'89*, pp. 131–145, 1989.

[7] D. Decouchant *et al.*, "A synchronization mechanism for typed objects in a distributed system," *SIGPLAN Notices*, vol. 24, pp. 105–107, April 1989.

[8] C. Neusius, "Synchronization actions," *ECOOP'91*, pp. 118–132, 1991. Also appeared in Lecture Notes in Computer Science, Vol. 512.

[9] C. Tomlinson and V. Singh, "Inheritance and synchronization with enabled-sets," *OOPSLA'89*, pp. 103–112, 1989.

[10] Luqi, "Software evolution through rapid prototyping," *IEEE Computer*, pp. 13–25, May 1989.

[11] Y. Shih and L. Lander, "RCOOL: a model for reusable concurrent object-oriented software library," 1993. Submitted to Software Engineering Research Forum 1993.

[12] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley, 1991.

[13] C. McHale *et al.*, "Scheduling predicates," tech. rep., Department of Computer Science, Trinity College, University of Dublin, 1991.

[14] G. Kaiser *et al.*, "Multiple concurrency control policies in an object-oriented programming system," in *Proceedings, Second IEEE Symposium on Parallel and Distributed Processing*, pp. 623–626, 1990.

[15] R. Prieto-Diaz and P. Freeman, "Classifying software for reusability," *IEEE Software*, vol. 4, no. 1, pp. 6–16, 1987.

Biography

Yunyau Shih is a Ph.D. student in the Department of Computer Science at Binghamton University in New York. He graduated from the Tunghai University in Taiwan and also has an MS from Binghamton. For his Dissertation he is working on object-oriented reuse libraries with an emphasis on the inclusion of constructs for synchronization.

Les Lander is a faculty member of the Department of Computer Science at Binghamton University. He obtained his BA at the University of Cambridge and his MS and PhD in mathematics from the University of Liverpool, U.K. His interests include programming languages and software engineering and he also works on expert sytems and system-level fault location. He chairs the ACM SIGAda Local Chapter for the Binghamton/Owego area.