A Quality Design Solution for Object Synchronization

António Rito Silva

INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000 Lisboa, PORTUGAL

Abstract. This paper presents a quality design solution, the Customizable Object Synchronization pattern, for object synchronization which decouples object synchronization from object concurrency and object functionality (sequential part). The solution described by this pattern provides encapsulation, extensibility, modularity and reuse of synchronization policies.

1 Introduction

This paper describes a quality object-oriented design solution for object synchronization by identifying the components and interactions which provide the desired characteristics: synchronization is enforced at the object side and it is encapsulated by an interface object; generic class specialization provides several synchronization policies; and the synchronization-specific classes are decoupled from sequential classes.

The solution is described as an object-oriented design pattern which emphasizes its qualities and shows how they are achieved. This description is free of implementation details. Particular pattern implementations can be applied in the construction of object-oriented frameworks or included in composition systems as reflective systems.

2 Customizable Object Synchronization Pattern

The design pattern is described using an extension to the format in [1]. The extension, Objectives and Assessment, emphasizes the design quality. Objectives defines the goals to be achieved by the design, e.g. design qualities or efficiency. Assessment describes how the pattern objectives are accomplished by the solution.

Intent. The Customizable Object Synchronization pattern abstracts several object synchronization policies. It decouples object synchronization from object concurrency and object functionality (sequential part).

Objectives. An object-oriented solution for the object synchronization problem must have the qualities: **encapsulation** requires the synchronization part of an object to be placed within the object itself rather than spread out among its clients; **extensibility** requires abstraction of synchronization policies; **modularity** requires separation of object synchronization from object concurrency and object functionality (sequential part); and **reusability** requires separate reuse of sequential and synchronization code.

Structure and Participants. The Booch class diagram in Figure 1a illustrates the structure of the Customizable Object Synchronization pattern.

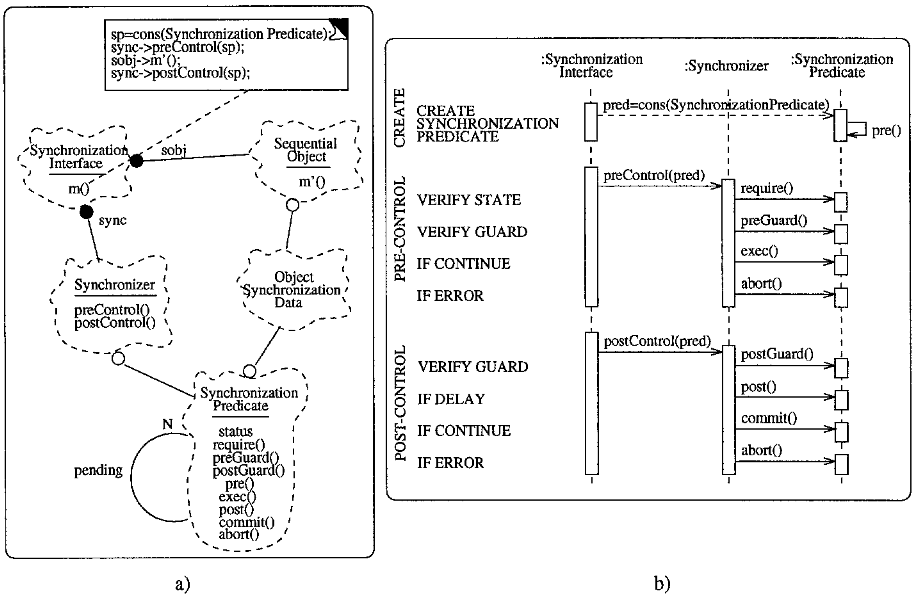

Fig. 1. a) Structure. b) Collaborations

The main participants in the Customizable Object Synchronization pattern are:

- `Sequential Object`. Contains the sequential code and data, where accesses should be synchronized.
- `Synchronization Interface`. Is responsible for the synchronization of invocations to the `Sequential Object` using the services provided by the `Synchronizer`. It invokes `preControl` before invocation proceeds on the `Sequential Object` and `postControl` after.
- `Synchronizer`. It decides whether an invocation may continue, stop or should be delayed (returns values `CONTINUE`, `ERROR` and `DELAY`). Operations `preControl` and `postControl` control the order of invocations. The former enforces pessimistic policies while the latter enforces optimistic policies.

- Synchronization Predicate. Identifies the invocation and contains its current status which can be: pre-pending, executing, post-pending, committed and aborted (attribute status with values PRE, EXEC, POST, COMMIT and ABORT). Contains a queue of pending invocations pending. Defines the synchronization semantics of an invocation through operations require, preGuard and postGuard. Operations pre, exec, post, commit and abort update synchronization data.
- Object Synchronization Data. Provides the global object synchronization data. It may use the Sequential Object to get the synchronization data, e.g. the number of items in a bounded buffer.

Collaborations. When method m is invoked, a Synchronization Predicate object is created by the Synchronization Interface (first phase in Figure 1b). Afterwards, it invokes preControl (second phase in Figure 1b) and postControl (third phase in Figure 1b) on the Synchronizer, respectively, before and after method m′ is executed on the Sequential Object. preControl synchronizes an invocation before execution and an error may be returned, preventing invocation execution, otherwise access is delayed or resumed. postControl verifies if an invocation already done is correctly synchronized with terminated invocations. To evaluate synchronization conditions (operations preControl and postControl) the Synchronizer interacts with Synchronization Predicate. The Synchronization Predicate may use the synchronization data in other pending invocations or in the Object Synchronization Data.

Sample Code. This section presents the implementation of a pessimistic readers/writers policy. Two categories of methods are considered: read and write. Invocations to read methods are delayed whenever a write method is executing, while invocations to write methods are delayed whenever read or write methods are executing.

Below it is shown the implementation of a pessimistic synchronizer. Note that post control phase always return CONTINUE since invocations are synchronized before access by preControl method. Identifier names have been shortened.

```
// Pessimistic Synchronizer preControl
X_Status Pess_Synchronizer::
preControl(Sync_Pred *pred)
{
  X_Status xStatus;
  // compatible state
  xStatus = pred->require();
  // conflicting invocations
  if (xStatus == CONTINUE)
    xStatus = pred->preGuard();
  // update predicate sync data
  if (xStatus == CONTINUE)
    pred->exec();
  else if (xStatus == ERROR)
    pred->abort();
  // returns result
  return xStatus;
}
```

```
// Pessimistic Synchronizer postControl
X_Status Pess_Synchronizer::
postControl(Sync_Pred *pred)
{
  // update predicate sync data
  pred->commit();
  // returns result
  return CONTINUE;
}
```

Two synchronization predicates, Read_Pred and Write_Pred, are defined for, respectively, read and write invocations. Their super-class, RW_Sync_Pred, defines the two categories, READ and WRITE. Read and write predicates are shown below. They verify the status of predicates in the pending_queue. Since synchronization does not depend on object state, **require** operations return CONTINUE.

```
// conflicts with executing writes
X_Status Read_Pred::preGuard()
{
  // iterator for pending predicates
  Iterator iter(pending_);
  RW_Sync_Pred *pred;
  while (pred = iter.next(), pred != 0)
    // there are write invocations executing
    if ((pred->getStatus() == EXEC) &&
        (pred->getCat() == WRITE))
      // conflict
      return DELAY;
  // no conflict
  return CONTINUE;
}
```

```
// conflicts with executing reads and writes
X_Status Write_Pred::preGuard()
{
  // iterator for pending predicates
  Iterator iter(pending_);
  RW_Sync_Pred *pred;
  while (pred = iter.next(), pred != 0)
    // there are invocations executing
    if (pred->getStatus() == EXEC)
      // conflict
      return DELAY;
  // no conflict
  return CONTINUE;
}
```

Assessment. In this section it is shown how the design pattern achieves the objectives previously stated.

Encapsulation is achieved by placing the synchronization code within the synchronized object such that client objects can invoke **Synchronization Interface** ignoring synchronization issues.

Extensibility is achieved by specializing the abstract classes, **Synchronization Predicate** and **Object Synchronization Data**. Above a pessimistic readers/writers policy was described. The design pattern also supports several optimistic policies by defining postControl operation and producer/consumer policies by defining **require** operation since synchronization depends on the object state.

Modularity is achieved by decoupling synchronization, concurrency and sequential code. The synchronization code is kept in classes **Synchronization Predicate** and **Object Synchronization Data**. Two policies of object concurrency are considered: active object and passive object. Object synchronization requires, from concurrency policies, mutual exclusion and activity delay/awake services. Both implementation are orthogonal to synchronization code.

Reusability is partially achieved. Composition reusability is completely achieved but inheritance reusability has some limitations. Due to the decouple of synchronization from sequential code it is possible to have separate composition reuse. Concerning inheritance reuse, the presented solution allows extension of synchronization code independently of sequential code, but the opposite is not necessarily true. For instance, the introduction of a new sequential method may require the redefinition of synchronization predicates.

3 Related Work and Conclusions

This paper discusses a design pattern, the Customizable Object Synchronization pattern, for object synchronization. Contrarily to the common description of design patterns which emphasizes the pattern known uses, this description emphasizes the quality aspects associated with the solution.

Several proposals tie object synchronization with concurrency and distribution, e.g. [2]. The design pattern presented in [3] also associates synchronization with active objects. The Object Synchronization approach decouples object synchronization from concurrency and distribution.

The Object Synchronization pattern supports several of the features of McHale's work on declarative object synchronization mechanisms [4]: it completely separates object synchronization data from sequential data, and it has the expressive power of scheduling predicates. Moreover, it supports optimistic policies by considering new cases where synchronization data can be changed, (`post` method of `Synchronization Predicate`), and can easily be integrated with object recovery, (`abort` method of `Synchronization Predicate`).

The inheritance anomaly problem [5] is not completely solved by the design pattern. However, the supported separation of synchronization from functionality solves some of the problems. Other approaches to this problem stressed this separation, e.g. [6]. However, the pattern approach does not require new languages neither a new inheritance mechanism.

The Object Synchronization design pattern is implemented in a framework for heterogeneous concurrency, synchronization and recovery in distributed applications. This framework is publicly available from the WWW page `http://albertina.inesc.pt/~ars/dasco.html`.

Acknowledgments. Thanks to my colleagues Francisco Rosa and Teresa Gonçalves for reading this document.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
2. Murat Karaorman and John Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*, 36(9):103–116, September 1993.
3. R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley, 1996.
4. Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, Department of Computer Science, Trinity College, Dublin, 1994.
5. Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
6. Cristina Videira Lopes and Karl Lieberherr. Abstracting Process-to-Function Relations in Object-Oriented Applications. In *ECOOP '94*, pages 81–99, Bologna, Italy, July 1994.