Formal Modeling of Synchronization Methods for Concurrent Objects in Ada 95

Ravi K. Gedela
Dept. of EECS
Concurrent Software Lab
The University of Illinois at Chicago
Tel: +1-303-473-6722
R.Gedela@ericsson.com

Sol M. Shatz
Dept. of EECS
Concurrent Software Lab
The University of Illinois at Chicago
Tel: +1-312-996-5488
shatz@eecs.uic.edu

Haiping Xu
Dept. of EECS
Concurrent Software Lab
The University of Illinois at Chicago
Tel: +1-312-666-8588
hxu1@eecs.uic.edu

1. ABSTRACT

One important role for Ada programming is to aid engineering of concurrent and distributed software. In a concurrent and distributed environment, objects may execute concurrently and need to be synchronized to serve a common goal. Three basic methods by which objects in a concurrent environment can be constructed and synchronized have been identified [1]. To formalize the semantics of these methods and to provide a formal model of their core behavior, we provide some graphic models based on the Petri net formalism. The purpose of this formal modeling is to illustrate the possibility of automatic program analysis for object-oriented features in Ada-95. Models for the three distributed-object synchronization methods are discussed, and a potential deadlock situation for one of the methods/models is illustrated. We conclude with some comparison of the three methods in terms of the model abstractions.

1.1 Keywords

Ada-95, concurrent objects, distributed software, synchronization methods, Petri net formalism

2. INTRODUCTION

With the growing interest in concurrent and distributed computing applications, there is significant value in new capabilities to support the engineering of distributed software. One of the principle objectives of concurrent and distributed programming is to coordinate the behavior of concurrent tasks. This is aided by using object-oriented techniques in combination with concurrent programming. The resultant software systems consist of objects that execute concurrently and need to be synchronized to serve a common goal. There might be situations where access to an object is required by more than one other object or task. In such cases, it is vital to enforce synchronization. For example, consider a printer as an object. The services of this printer object (server) might be required by multiple tasks (clients). This would require synchronization among the client tasks so that only one task at a time can gain access to the printer object. There are many such practical situations where synchronization is very important. Thus, synchronization among tasks accessing an object is a critical issue.

In the specific context of Ada-95 [2], Burns and Wellings have identified three methods to introduce synchronization among objects in a concurrent environment [1]. These methods are listed as follows:

1. Synchronization is added if and when it is required, by extending the object.

2. Synchronization is provided by the base (root) object type.

3. Synchronization is provided as a separate protected type and the data is passed as a discriminant.

Unfortunately, these methods can be difficult to understand due to the lack of an abstraction formalism. We have used Petri nets [3] to formalize the behavior of these methods. Because Petri nets are graphically based, the models provide a visualization result with well-defined dynamic behavior. Petri nets provide a graphic model that supports the fundamental concepts of concurrency, synchronization, and nondeterminism. In

Petri net models, conditions are represented by "place nodes," depicted as circles, and events are represented by "transition nodes," depicted by bars. Although we are not yet ready to discuss Figure 1, it provides an example of a Petri net graph. Note that directed arcs connect the place and transition nodes and thus provide a logical connection between the holding of conditions and the occurrence of events. The other key component of a Petri net is its marking, which is a distribution of *tokens* to place nodes. Tokens are represented by black dots, as can be seen in the place labeled L in Figure 1. As events occur (i.e., transitions fire), tokens are consumed from input places and deposited into output places. Due to lack of space in this paper, we are not able to provide a more complete introduction to Petri nets. Full details on this model, including associated analysis techniques, can be found in other references (e.g., [3]). There does exist some other published work on Petri net modeling for Ada. Among the earliest work is that of Mandrioli, et al [4], which focused on using Petri nets to provide a formal semantic for the basic tasking mechanisms of Ada-83. Earlier work by our own research group investigated the use of Petri nets for development of tools and techniques for automated concurrency analysis of software based on Ada tasking [5-6]. Again the focus was on Ada-83. More recent work provided a formal description of Ada-95 tasking constructs, such as the asynchronous transfer of control and requeue statement [7].

In the remainder of this paper, we will explain the various object synchronization methods and present associated net models for illustrative examples using these methods. We also discuss a potential deadlock situation and compare the three synchronization methods at the model level.

3. SYNCHRONIZATION FOR CONCURRENT OBJECTS

Among the three synchronization methods for concurrent objects in Ada-95, the first method, whose synchronization is added by extending the object, is the simplest one. In the original example for the first method by Burns and Wellings [1], procedures Op1 and Op2, defined in package Object, are redefined in its child package Object.Synchronized and its grandchild package Object.Synchronized.Extended (as shown in Section 4.1). Since this kind of polymorphism is resolved at compile time, to make the formalism more obvious, we eliminate the polymorphism by changing the corresponding procedure name to Op1_syn, Op1_ext and so on. On the other hand, for both the second and third synchronization methods, a common procedure is used to dynamically dispatch the correct procedure for different parameters, and the polymorphism is resolved at execution time. In such cases, the dynamic behavior of those original examples are well-captured by the Petri net models.

3.1 Synchronization Added by Extending the Object

Let us first consider the method where synchronization is added if and when it is required by extending the object. This is the most general approach that could be followed to construct objects in a concurrent environment. The Obj_Type is defined as tagged and can be extended to facilitate synchronization. We remind the reader that this technique was introduced in [1]. Consider the following simple example:

```
package Object is
   procedure Op1 (O : in out Obj_Type);
   procedure Op2 (O : in out Obj_Type);
private
   type Obj_Type is tagged limited record .end record;
end Object;

package Object.Synchronized is
   type Protected_Type is new Obj_Type with private;
   procedure Op1_syn (O : in out Protected_Type);
   procedure Op2_syn (O : in out Protected_Type);
private
   type Protected_Type is new Obj_Type with
      record
         L : Mutex;
      end record;
end Object.Synchronized;
```

Type Mutex provides a simple mutual exclusion lock, and is defined by a protected type as follows:

```
protected type Mutex is
    entry Lock;
    entry Unlock;
 end Mutex;
```

In the body of package Object.Synchronized, procedures Op1_syn and Op2_syn can be defined to call the procedures Op1 and Op2, defined in package Object, and to include the synchronization facilities:

```
procedure Op1_syn (O : in out Protected_Type) is
```

```
begin
   O.L.Lock;
   Op1(Obj_Type(O));
   O.L.Unlock;
end Op1_syn;

procedure Op2_syn (O : in out Protected_Type) is
begin
   O.L.Lock;
   Op2(Obj_Type(O));
   O.L.Unlock;
end Op2_syn;
```

In this example, procedure Op1_syn takes O, a parameter of type Protected_Type, and makes an attempt to gain the lock. Then the procedure Op1, defined in Object, is called by passing Obj_Type(O) as a parameter. After the execution of the procedure Op1, entry Unlock is called to release the Lock. Procedure Op2_syn has behavior similar to procedure Op1_syn.

Now consider the following extension of the package Object.Synchronized:

```
package Object.Synchronized.Extended is
   type Extended_Protected_Type is new Protected_Type
        with private;
   procedure Op1_ext (O : in out Extended_Protected_Type);
   procedure Op2_ext (O : in out Extended_Protected_Type);
   procedure Op3_ext (O : in out Extended_Protected_Type);
private
   type Extended_Protected_Type is new Protected_Type
         with record .end record;
end Object.Synchronized.Extended;

procedure Op1_ext (O : in out Extended_Protected_Type) is
begin
   O.L.Lock;
   Op1(Obj_Type(O));
   O.L.Unlock;
end Op1_ext;

procedure Op2_ext (O : in out Extended_Protected_Type) is
begin
   O.L.Lock;
   Op2(Obj_Type(O));
   O.L.Unlock;
end Op2_ext;
procedure Op3_ext (O : in out Extended_Protected_Type) is
begin
   O.L.Lock;
   -- data processing
   O.L.Unlock;
end Op3_ext;
```

The use of package Object.Synchronized.Extended can be demonstrated by calling the procedure Op1_ext as follows:

```
O : Extended_Protected_Type;
Op1_ext(O);
```

In package Object.Synchronized.Extended, procedures Op1_ext and Op2_ext are defined to have the same functional behavior as the procedures Op1_syn and Op2_syn, defined in package Object.Synchronized. The only difference is the formal parameter type, which is defined as Extended_Protected_Type. Procedure Op3_ext is a new procedure that is added to package Object.Synchronized.Extended.

The Object.Synchronized.Extended object can be modeled by a single Petri net as shown in Figure 1. In this model, place *L* provides for mutual exclusion on the execution of procedure Op1_ext, Op2_ext and Op3_ext, which are represented as transitions *op1_ext*, *op2_ext* and *op3_ext* respectively. Note that we have ignored modeling the execution of procedure Op1_syn and Op2_syn, which are defined in package Object.Synchronized. This is because procedure Op1_syn and Op2_syn have the same functionality as those procedures Op1_ext and Op2_ext, which are defined in package Object.Synchronized.Extended, and they can be modeled in exactly the same way as shown in Figure 1. To understand the model behavior, assume that task *A* calls Op1_ext and task *B* calls Op2_ext simultaneously. Both transitions *op1_ext* and *op2_ext* can fire, resulting in a token in both places *a* and *e*. Under this condition, both transitions *lock_op1_ext* and *lock_op2_ext* will be enabled at the same time. However, due to the conflict firing these two transitions (i.e., the competition for the token in place *L*), only one of them will succeed. The other transition must wait until the procedure defined in package Object completes and the lock is released. The synchronization involving calls to Op3_ext is similar.

3.2 Synchronization Provided by the Base Object Type

The second method of providing synchronization to objects in a concurrent environment is synchronization provided by the base object type. In this method the base object incorporates the synchronization mechanism. Consider a similar example as discussed in the first method. In this case the mutual exclusion lock is declared in the base object. The Ada code for this would be as follows:

```
package Protected_Object is
  type Protected_Type is abstract tagged limited private;
  procedure Class_Wide_Op1(O: in out Protected_Type'Class);
  procedure Class_Wide_Op2(O: in out Protected_Type'Class);
private
  type Protected_Type is abstract tagged limited
      record
         L : Mutex;
      end record;
  procedure Op1 (O : in out Protected_Type) is abstract;
  procedure Op2 (O : in out Protected_Type) is abstract;
end Protected_Object;
```

In this example, we declare the procedures Op1 and Op2 to be abstract and private, and the class-wide operations, *Class_Wide_Op1* and *Class_wide_Op2*, are being made the only interfaces that are exported from the package. Extensions of the base object must implement the procedures Op1 and Op2. The class-wide operations can now be defined in the package body as follows:

```
procedure Class_Wide_Op1(O: in out Protected_Type'Class) is
begin
   O.L.Lock;
   Op1(O);                    -- dispatch to correct operation
   O.L.Unlock;
end Class_Wide_Op1;
```

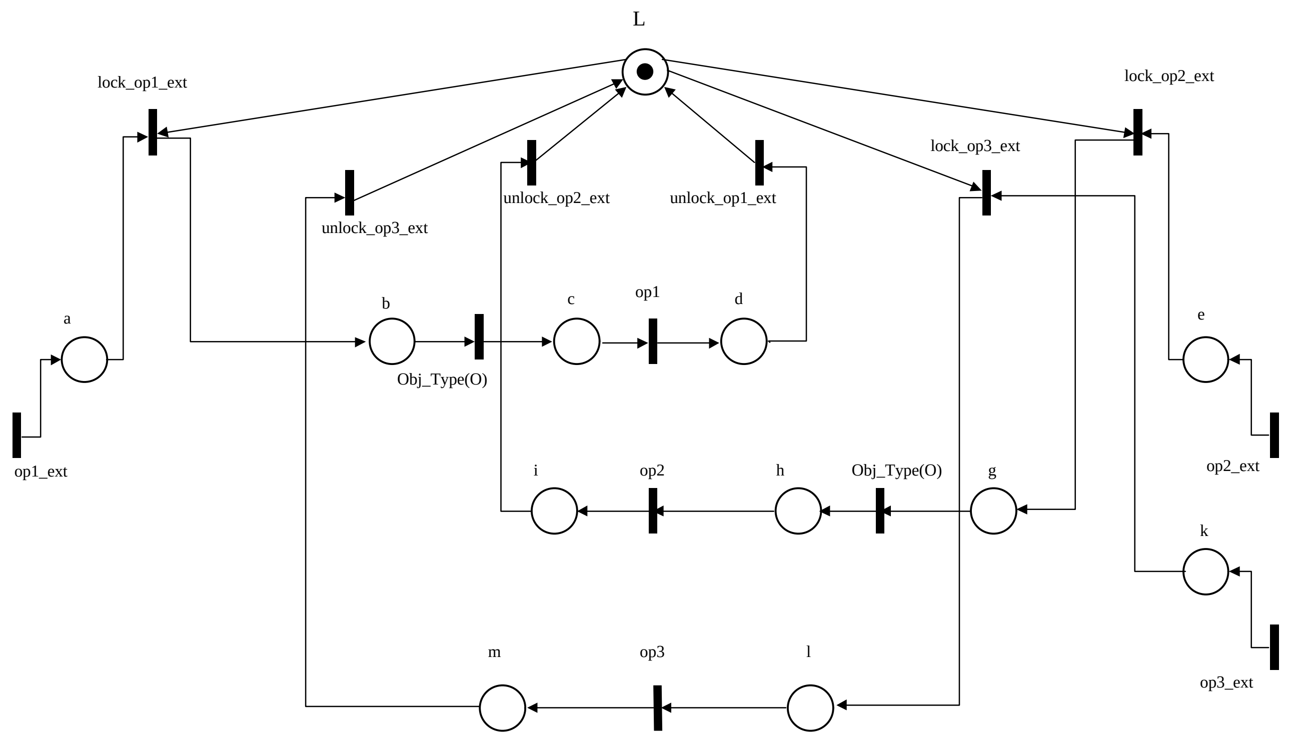

Figure 1. Object.Synchronized.Extended model

```
procedure Class_Wide_Op2(O: in out Protected_Type'Class) is
begin
   O.L.Lock;
   Op2(O);                        -- dispatch to correct operation
   O.L.Unlock;
end Class_Wide_Op2;
```

The following code demonstrates the usage of this Protected_Object:

```
package Protected_Object.My_Object is
   type My_Object_Type is new Protected_Type
          with record …end record;
 private
   procedure Op1 (O : in out My_Object_Type);
   procedure Op2 (O : in out My_Object_Type);
 end Protected_Object.My_Object;
```

This method uses the dynamic dispatching mechanism. Depending on the object type of the object that is passed as a parameter in the call to the procedure, the runtime system directs the call to the appropriate code defined for that object type. The type can be extended, retaining the mutual exclusion, as long as the operations are called with the class-wide operator. For example, one extension of the above is as follows:

```
package Protected_Object.My_Object.Extended is
   type Extended_Protected_Type is new My_Object_Type
          with private;
   procedure Class_Wide_Op3 (O: in out
                 Extended_Protected_Type'Class);
private
   type Extended_Protected_Type is new My_Object_Type
          with record .end record;
   procedure Op1 (O : in out Extended_Protected_Type);
   procedure Op2 (O : in out Extended_Protected_Type);
   procedure Op3 (O : in out Extended_Protected_Type);
end Protected_Object.My_Object.Extended;
```

The use of package Protected_Object.My_Object can be demonstrated by calling the procedure Class_Wide_Op1, which is defined in package Protected_Object, with parameters of different types My_Object_Type and Extended_Protected_Type as follows:

```
MO: My_Object_Type;           -- represented as color M in the
                              -- following Petri net model
Class_Wide_Op1(MO);
  …
EP: Extended_Protected_Type; -- represented as color E in the
                              --following Petri net model
Class_Wide_Op1(EP);
…
```

For procedure Op3, if mutual exclusion is also required, a new procedure Class_Wide_Op3 should be defined:

```
procedure Class_Wide_Op3 (O : in out
            Extended_Protected_Type'Class) is
begin
   O.L.Lock;
   Op3(O);             -- dispatch to correct operation
   O.L.Unlock;
end Class_Wide_Op3;
```

The object Protected_Object.My_Object.Extended can be modeled by the Petri net in Figure 2.

When a call to Class_Wide_Op1 is made with an object of type Extended_Protected_Type, the transition *class_wide_op1* fires. An output token with identity *E* (called the "color" of the token) is put in the place *a.*[1] Color *E* signifies that the call to *Class_Wide_Op1* uses an object of type Extended_Protected_Type. The firing of *lock_class_wide_op1* models the gaining of the lock and this transition firing puts an output token of color *E* in place *b*. Note that the firing of transition *op1* or *op1_ext* is dependent on the color of the token in place *b*. If the token color is *M*, meaning that the object type is My_Object_Type, then *op1* would fire. But if the color is *E*, as in the present case, then the transition *op1_ext* will fire. This choice of transition models the dynamic dispatching mechanism. When the transition *op1_ext* fires, an output token of color *E* is put in place *c*. When the procedure execution is complete, the transition *unlock_class_wide_op1* is enabled. Note that this

[1] This type of Petri net model that uses "colored"tokens (or tokens with attributes) is called a colored Petri net [8]. In colored Petri nets, a transition becomes enabled when its input places have tokens with attributes that match the inscriptions on the corresponding arcs from the place to the transition.

transition can fire with an input token of color *M* or *E*. The lock is released when the transition *unlock_class_wide_op1* fires. Similarly, Class_Wide_Op2 is modeled in the same way. Although the additional procedure Class_Wide_Op3 is also modeled as the transition *class_wide_op3*, this transition can fire only when the input token color is *E*. This is necessary since Class_Wide_Op3 is defined only on Extended_Protected_Type and not on My_Object_Type.

3.3 Synchronization Using a Protected Type with Data Parameters

The third method of providing synchronization among concurrent objects is by using a protected type with the data passed as a discriminant. First, the base type is defined as protected and all of its protected operations are defined as abstract, which means extension of this base type must implement these operations. Then a protected type can be constructed, which has a class-wide access discriminant as a formal parameter and has operations used to dispatch the appropriate operations according to the discriminant. Consider the following example:

```
package Object is
   type Obj_Type is abstract tagged null record;
   protected type Controller (O : access Obj_Type'Class) is
      procedure Op1;
      procedure Op2;
   end Controller;
private
   procedure Op1 (O : in out Obj_Type) is abstract;
   procedure Op2 (O : in out Obj_Type) is abstract;
end Object;

procedure Op1 is
begin
   Op1(O.all);
end Op1;

procedure Op2 is
begin
   Op2(O.all);
end Op2;
```

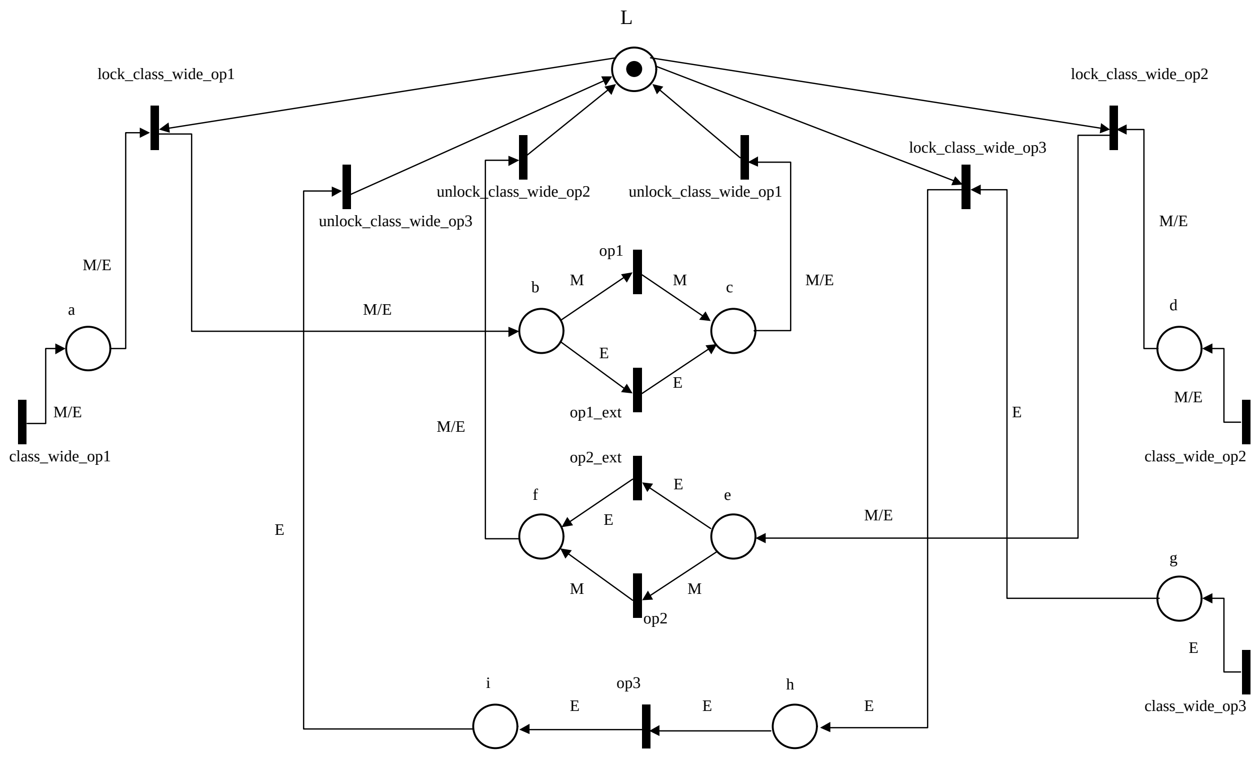

Figure 2. Protected_Object.My_Object.Extended model

In Object, the base type Controller and its operations Op1 and Op2 are defined as a protected type. The data type Obj_Type'Class is passed to the protected type Controller as a discriminant.

The usage of Object is demonstrated by the following segment of code:

```
package Object.My_Object is
   type My_Obj is private;
private
   type My_Obj is new Obj_Type with record ... end record;
   procedure Op1(O : in out My_Obj);
   procedure Op2(O : in out My_Obj);
end Object.My_Object;
...
O : aliased My_Obj;
contr : Controller(O'Access);
contr.Op1;
```

Package Object.My_Object extends Obj_Type and defines Op1 and Op2, with parameters of type My_Obj. Mutual exclusion over Op1 and Op2 is ensured by the definition of protected declaration of the Controller. This method has the similar mechanism as the second method. But, one disadvantage of this method is that new operations cannot be added into the Controller, because a protected type cannot be extended.

Now consider an extension of Object.My_Object:

```
package Object.My_Object.Extended is
   type My_Obj_Ext is private;
private
   type My_Obj_Ext is new My_Obj with record .end record;
   procedure Op1 (O : in out My_Obj_Ext);
   procedure Op2 (O : in out My_Obj_Ext);
end Object.My_Object.Extended;
```

The use of package Object.My_Object.Extended can be demonstrated by calling the protected object Controller with different parameters as follows:

```
OM : aliased My_Obj;          -- represented as color O in the
                               -- following Petri net model
contr1 : Controller (OM'Access);
contr1.Op1;
…
OE: aliased My_Obj_Ext;     -- represented as color E in the
                              -- following Petri net model
contr2: Controller (OE'Access);
contr2.Op1;
```

The above extension defines a new type, My_Obj_Ext, an extension of My_Obj, and defines procedures Op1 and Op2 for parameters of type My_Obj_Ext. A Petri net model of this situation is shown in Figure 3.

When a call to Op1 is made by an object of type Controller with a data discriminant of type My_Obj_Ext, the transition *op1_protected_start* fires, and an output token of color *E* is put in place *a.* Color *E* signifies that the data discriminant of the object of type Controller is of the type My_Obj_Ext. Firing of the transition *op1_protected_start* removes the token in *controller* place, thus disabling any other calls to the protected procedures Op1 and Op2. Now, the firing of transition *op1* or *op1_ext* is dependent on the color of the token in place *a.* If the token color is *O*, meaning that the data discriminant of the calling object of type Controller is of the type My_Obj, then transition, *op1* would fire. But, if the token color is *E*, which is the present case, the transition *op1_ext* would fire. Again, this choice of firing of transitions, based on the color of the token in the input place, models the dynamic dispatching mechanism. Now, the transition *op1_protected_end* can fire, taking away a token of color *E* from place *b* and putting an output token in *controller* place. This models the end of the protected operation associated with executing Op1, so other objects waiting to gain access can now proceed.

4. DISCUSSION

4.1 Potential Deadlock Problem

One disadvantage with the first method is that when the type Object.Synchronized is further extended, there are circumstances that can cause a potential for deadlock. To make the deadlock situation not so obvious, we rewrite both the child package Object.Synchronized and the grandchild package Object.Synchronized.Extended as follows, which are the same as those in Burns and Wellings's original example [1]:

```
package Object.Synchronized is
   type Protected_Type is new Obj_Type with private;
   procedure Op1 (O : in out Protected_Type);
   procedure Op2 (O : in out Protected_Type);
private
   type Protected_Type is new Obj_Type with
```

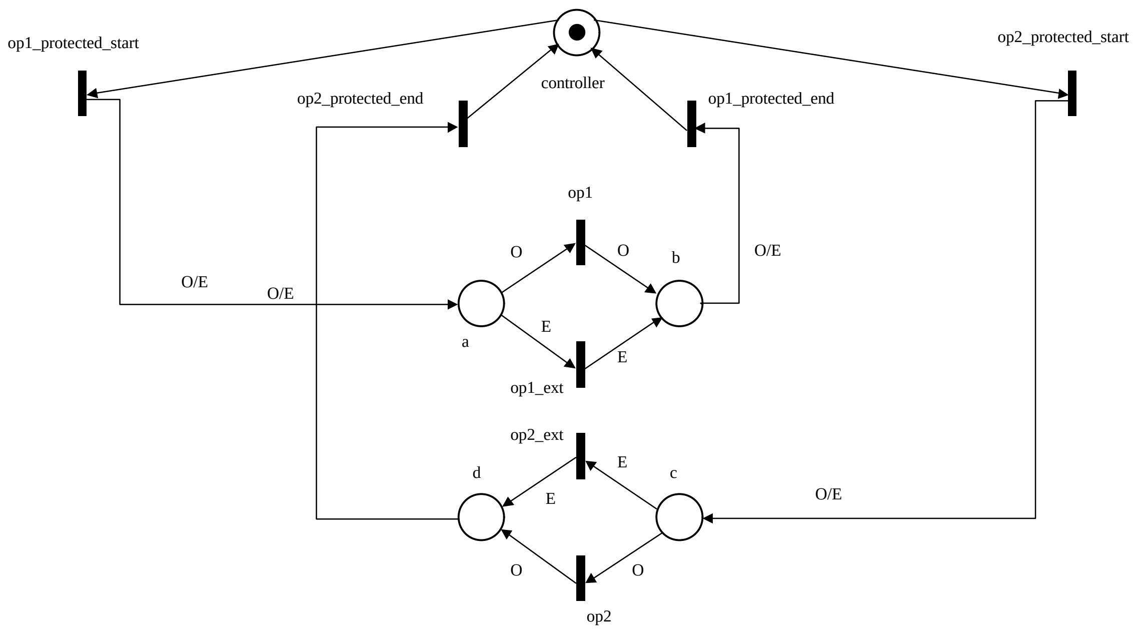

Figure 3. Object.My_Object.Extended model

```
    record
      L : Mutex;
     end record;
end Object.Synchronized;

package body Object.Synchronized is
  procedure Op1 (O : in out Protected_Type) is
  begin
    O.L.Lock;
    Op1(Obj_Type(O));
    O.L.Unlock;
  end Op1;
  …
end Object.Synchronized;

package Object.Synchronized.Extended is
  type Extended_Protected_Type is new Protected_Type
      with private;
  procedure Op1 (O : in out Extended_Protected_Type);
  procedure Op2 (O : in out Extended_Protected_Type);
  procedure Op3 (O : in out Extended_Protected_Type);
 private
  type Extended_Protected_Type is new Protected_Type
      with record .end record;
end Object.Synchronized.Extended;

package body Object.Synchronized.Extended is
  procedure Op1 (O : in out Extended_Protected_Type) is
  begin
    O.L.Lock;
    -- pre_processing;
    Op1(Protected_Type(O));
    -- post_processing;
    O.L.Unlock;
  end Op1;
  …
end Object.Synchronized.Extended;
```

The use of package Object.Synchronized.Extended can be demonstrated by calling the procedure Op1 with a parameter of type Extended_Protected_Type as follows:

```
O : Extended_Protected_Type;
Op1(O);
```

In this example, the procedures Op1 and Op2, which are defined in package Object, are redefined in both of its child package Object.Synchronized and grandchild package Object.Synchronized.Extended. The resolution of this polymorphism is made according to the parameter types at compile time. So, in our Petri net model, they are represented as different transitions. Similar as before, the procedures Op1 defined in package Object, its child package Object.Synchronized and its grandchild package Object.Synchronized.Extended are represented as transitions *Op1*, *Op1_syn* and *Op1_ext* respectively. The most significant difference between this example and the one in Section 3.1 is that here procedure Op1, defined in package Object.Synchronized.Extended, calls the procedure Op1 with a parameter of type Protected_Type rather than Obj_Type as before. At compile time, it is determined that the actual called procedure Op1 is the one which is defined in package Object.Synchronized. The Petri net model for this revised version of Object.Synchronized.Extended is shown in Figure 4. To make the deadlock detection more obvious, we ignore modeling the execution of procedures Op2 and Op3, which are defined in grandchild package Object.Synchronized.Extended. They can be modeled exactly the same as in Figure 1.

In the model of Figure 4, when a call to Op1, defined in package Object.Synchronized.Extended, is made with an object of type Extended_Protected_Type, the transition *op1_ext* fires and a token is deposited into place *a.* The firing of the enabled transition *lock_op1_ext* takes the token from place *L* and deposits an output token in place *b*. This models the acquisition of the lock by the object that called procedure Op1, defined in Object.Synchronized.Extended. The firing of transition *pre_processing* models the starting of some arbitrary data processing. Upon the completion of this data processing, a call is made to Op1, defined in package Object.Synchronized, with type Protected_Type. The conversion of type is modeled by the firing of the transition *Protected_Type(O)*, and the call is modeled by the firing of the transition *op1_syn*, as was used in the previous model of Figure 1. But now, the procedure Op1, defined in package Object.Synchronized, tries to again obtain the lock. In this state, where there is a token in place *e* but no token in place *L*, the transition *lock_op1_syn* is not enabled. The resulting deadlock is naturally captured and can be visualized in the model by the lack of any enabled transition.

4.2 Some Comparison Comments on the Net Models

The study of the Petri net models for different methods of providing synchronization among concurrent objects provides us an opportunity to compare them to identify model relationships at the code level. The model in Figure 1 is comparable to the models in Figure 2 and Figure 3. One significant difference among these is the use of colored Petri nets for the later two methods, whereas an uncolored Petri net is sufficient to model the first method. A study of the models reveals that the model in Figure 2 actually contains the models used in Figures 1 and 3. For example, in Figure 2 the firing sequence *class_wide_op1*, *lock_class_wide_op1*, *op1_ext*, *unlock_class_wide_op1* is comparable to the firing sequence *op1_ext*, *lock_ext_op1*, *Obj_Type(O)*, *Op1*, *unlock_ext_op1* in Figure 1 and the firing sequence *op1_protected_start, op1_ext, op1_protected_end* in Figure 3. A similar situation exists for the firing sequence for *class_wide_op2*. However, consider the firing sequence in Figure 2 for *class_wide_op3*, namely *class_wide_op3*, *lock_class_wide_op3*, *op3_ext*, *unlock_class_wide_op3*. This has a comparable firing

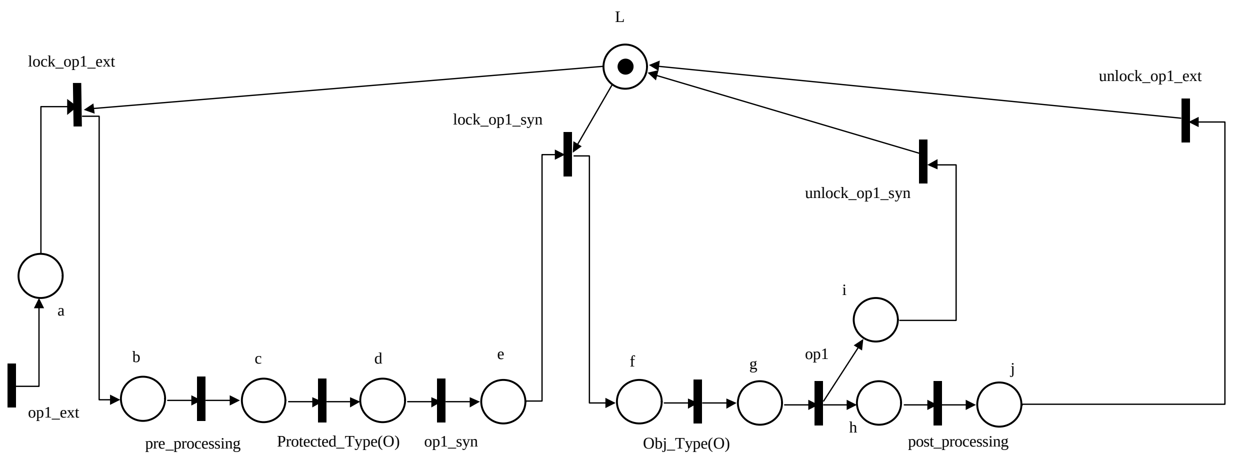

Figure 4. Object.Synchronized.Extended model

sequence *op3_ext*, *lock_ext_op3*, *op3*, *unlock_ext_op3* in Figure 1, but there is no such firing sequence in Figure 3. The reason for this is that in the third method we cannot extend the protected type with respect to adding new protected procedures.

Note that the model in Figure 1 transition *op1_ext* can only accept calls with a parameter of type Extended_Protected_Type, but not of type Protected_Type. On the other hand, both models in Figure 2 and Figure 3 have the capability of handling all the extensions of the base class with a common transition, say *class_wide_op1* or *op1_protected_start*. This advantage is due to the dynamic dispatching mechanism in these methods.

4.3 Future Work

A well-known advantage of Petri nets is their potential for formal analysis. By translating Ada code into Petri nets, we may achieve the goal of automated analysis of Ada programs, such as automated deadlock analysis of Ada programs as we had done before [6]. The critical issue here is how to automatically translate Ada code into Petri nets, since constructing a model manually is not only error-prone but also unrealistic for a large program. In our previous work, we have successfully developed a tool kit called TOTAL (the tasking-oriented tool kit for the Ada language) [9] to build correct models for Ada tasking by using compiler techniques. Our future work will further this technique and enhance this tool kit to include the object-oriented features in Ada-95 programming. The work done in this paper indicates that automated translation is possible and has potential to aid validation of object interactions.

5. CONCLUSION

Formal modeling of the synchronization constructs for concurrent objects in Ada is difficult due to the need to properly capture many behaviors that are interdependent. In this paper, we have presented and discussed a means to model these constructs using the Petri net modeling formalism. Petri nets have been chosen because they tend to provide a visual, and thus easy to understand, model. Also, Petri nets are well matched to the problem due to their inherent support for modeling concurrency, nondeterminism, synchronization, and mutual exclusion. We developed models for three different object synchronization methods. These methods were defined in terms of Ada-95 and presented in [1]. As a simple example, we described how a deadlock situation caused by improper object synchronization design can be observed in terms of a standard Petri net deadlock. Also we provided some comparison comments on the synchronization methods in terms of their formal model attributes. The value of this type of modeling is revealed by our plans for future work.

6. ACKNOWLEDGMENTS

This material is based upon work supported by, or in part by, the U.S. National Science Foundation under grant CCR-9321743 and the U.S. Army Research Office under grant number DAAG55-98-1-0470.

7. REFERENCES

[1] A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge Press, 1995.

[2] J. Barnes, *Programming in Ada 95*. Addison-Wesley, Inc., 1996.

[3] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE*, 77(4):541-580, April 1989.

[4] D. Mandrioli, R. Zicari, C. Ghezzi and F. Tisato, “Modeling the Ada Task System by Petri Nets,” *Computer Languages*, 10(1):43-61, 1985.

[5] S. M. Shatz, S. Tu, T. Murata, and S. Duri, “An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 12, Dec. 1996, pp. 1307-1322.

[6] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, “Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada,” *ACM Transactions on Software Engineering Methodology*, Vol. 3, No. 4, Oct. 1994, pp. 340-380.

[7] R. Gedela and S. M. Shatz, “Formal Modeling of Advanced Tasking in Ada: A Petri Net Perspective,” *2nd International Workshop on Software Engineering for Parallel and Distributed Systems* (PDSE-97), Boston, May, 1997, pp. 4-14.

[8] K. Jensen, “Coloured Petri Nets: A High Level Language for System Design and Analysis,” *Advances in Petri Nets 1990*, G. Rozenberg (Editor), in *Lecture Notes in Computer Science*, 483, Springer-Verlag, 1990.

[9] S. M. Shatz, et al, “Design and Implementation of a Petri Net Based Tool kit for Ada Tasking Analysis,” *IEEE Transactions On Parallel and Distributed Systems*, Oct. 1990, pp.424-441.