Transforming Ada Serving Tasks Into Protected Objects

Bangqing Li Baowen Xu
Department of Computer Science & Engineering
Southeast University
Nanjing 210096, P. R. China
{AdaLab, bwxu}@seu.edu.cn

Huiming Yu
Department of Computer Science
North Carolina A&T State University
Greensboro, NC27411
cshmyu@ncat.edu

1. Abstract

Protected objects are new features of Ada 95 that overcome the limitations of tasks in Ada 83, support more efficient communication, and provide mutually exclusive access to shared data. Transforming large concurrent software systems, that were written in Ada 83, to software systems using protected objects would make maintenance easier and improve system performance dramatically. In this paper the semantics of rendezvous and protected objects is examined, a group of hypotheses is developed, an algorithm that transforms serving tasks written in Ada 83 into protected objects in Ada 95 is presented, and finally an example is given to demonstrate applications of this algorithm.

1.1 Keywords
Ada, serving tasks, protected objects, transformation

2. Introduction

The innovative rendezvous of Ada 83 provides a good overall model for task (process) communication, through which tasks are mutually excluded automatically and avoid difficulties by using low-level mechanisms such as semaphores, where a user of shared data must remember to lock and unlock semaphores. However, it had not proved adequate for the problems of shared data access. A serving task is often needed to control access to shared data or shared resources, which makes systems have a relatively high overhead and increases possibilities of race conditions[2,9].

Protected types are introduced in the revised standard, also called Ada 95, to overcome these difficulties. Protected objects provide synchronization and mutually exclusive access to shared data. They have the same powerful expressiveness as rendezvous, while minimize unnecessary context switches[8,9].

In this paper, we will discuss the semantics of rendezvous and protected objects, present a transforming algorithm, and give an example to illustrate applications of the algorithm.

3. Serving Tasks vs. Protected Objects

Transforming large concurrent software systems written in Ada 83 to software systems using protected objects in Ada 95 would make maintenance of these systems easier and improve system behaviors. Many people had done valued work on this area. In the paper [4] the author presents a wide set of Ada tasking idioms for which there is more efficient implementation than the straightforward assignment of a thread of control to each task. In the paper [3] it was proposed to use a variable in the transformed program to record what the current point of execution would be in the original program. Jackson uses the same technique for the Jackson Structured Programming transformation known as "inversion" in the book [6]. And in the paper [7] replacing passive tasks with protected records was proposed. In the following section we will discuss semantics of rendezvous and protected objects, and requirements for tasks that will be transformed.

3.1 Semantics of rendezvous and protected objects

Many tasks exist concurrently at run-time and each of them proceeds independently. When a task needs another task's state information or its service(s) communication between tasks will start. Communication has two forms that are synchronous and asynchronous. Rendezvous in Ada is a mechanism for synchronous communication. It does not naturally map to asynchronous communication[1,2,5]. According to [2], rendezvous is the kind of hand-in-hand communication mechanism. It means that both tasks must reach synchronous points when rendezvous begins, and then proceed independently after the rendezvous completes. Mutual exclusion of different tasks is automatic and shared data is thus guaranteed to be consistent.

However, when managing a group of shared data, a serving task is often needed. In its scope, the serving task is always running. As a result, the system has a relatively high overhead and race conditions may arise. The extra

serving tasks are used because Ada 83 lacks a lightweight data synchronization primitive, the omission of which is criticized by some researchers[7].

Ada 95 solved this problem by introducing a low overhead synchronization mechanism based on the concept of protected objects. A protected object encapsulates a group of data and operations. Protected operations are automatically synchronized to allow only one writer or multiple readers at the same time, which makes protected objects safer than lower-level mechanisms such as conditional critical regions[5,8,9]. Programs implemented with protected objects can have higher readability and safety than those with serving tasks.

3.2 Hypotheses

Serving tasks to be transformed must be checked before transforming because not all serving tasks can be transformed by our algorithm. To begin with, five hypotheses are made for tasks that will be transformed.

Hypothesis 1: There are no entry call statements (direct or indirect) in the task body, as an entry call statement is thought of active here.

Hypothesis 2: The first statement of a selective alternative is an accept statement(may be guarded), except the terminate alternative.

Hypothesis 3: The last out-most statement of the task body is a basic loop statement. Before the loop begins there may be an initialization part followed by a sequence of accept or select statements. The initialization part is a sequence of zero or more assignments, if, loop, case or procedure call statements. This hypothesis is made because a rendezvous is an asymmetric communication mechanism and the called task has no information about calling tasks, therefore serving tasks should always exist in the scope.

Hypothesis 4: Each statement of the task body, except accept, select, the basic loop statement and the initialization part, is contained by an accept statement or a selective alternative, and is a sequence of zero or more assignments, if, loop, case or procedure call statements. The hypothesis is made because serving tasks are passive, they only proceed when one of the entries is called or one of the selective alternatives is selected.

Hypothesis 5: There are no task declarations contained by the task body. The hypothesis is made because task declarations are hard to represent in a protected body.

Based on these hypotheses a transforming algorithm has been developed to transform Ada serving tasks to protected objects. The algorithm will be presented in the next section.

4. A Transforming Algorithm

As there exist control flow relationships among accept statements in a task body and protected operations are called randomly, we must construct and add necessary variables and statements to proper places to keep control flows invariant. Control flow relationships can be obtained through simple syntax analysis, and are assumed that are known here.

4.1 Specification transformations

The main work of specification transformation is to transform task entry specifications. Task entries can be transformed to protected entries or protected subprograms. There are two methods that can be applied. The first method is a direct one. It transforms guarded entries to protected entries and unguarded entries to protected procedures. However, when it is needed to add new Boolean variable(s) to protected subprograms in the following steps (in section 3.2 step 1 or 2), the protected subprograms must be transformed to protected entries. The second is a general method. It transforms task entries to protected entries with barriers being the corresponding guards(TRUE if there are no guards). Just before the transformation finishes, transform protected entries with TRUE is used as barriers for protected procedures to improve run-time efficiency. We prefer the second method.

Correspondingly, some syntax transformations must be taken, such as transforming **task** to **protected**, **entry** to **procedure** or vice verse, and adding a reserved word **private**, etc.

4.2 Body transformations

A serving task body may contain a declaration part, which is private to the task. After transforming, the corresponding part in the protected object it should remain private. In our algorithm subprogram declarations and bodies in the task body are copied to the protected body, and other declarations are copied to the private part(the part following the reserved word private) of the protected specification. Other parts of the task body may be very complex, but they can be possessed recursively. Six steps are given in the following. Four main steps implement transformations of a sequence of accept statements, nesting accept statements, select statements and one task entry with multiple accept statements. In order not to influence calling tasks, all identifiers are kept unchanged.

For simplicity some details are ignored.

1. Assume there is a sequence of accept statements in a task body such as

accept A_1(_) **do**

_

end A_1;
accept A_2(_) **do**

_

end A_2;

accept A_n(_) **do**

_

end A_n;

where n>0, A_i is an entry designator. These accept statements will be transformed to protected entries. Statements contained in an accept statement are transformed to corresponding protected entry bodies. Each protected entry has the same parameters (if any) as the corresponding task entry, and the guards are transformed to entry barriers (TRUE if there is no guard). For each protected entry a new Boolean variable Var_A_i_S is created and called "control-flow variable". The control-flow variable will "And" with the entry barrier to form a new barrier, e.g., an entry barrier is transformed from R_i to (R_i) ***and*** *Var_A_i_S*. The control-flow variables are used to keep control flow relationships invariant. Two assignment statements are added to the end of each protected entry body(e.g., Var_A_i_S := FALSE; Var_$A_{i \bmod n + 1}$_S := TRUE;). The variable declarations are added to the private part of the protected object. Each of them is assigned a default value.

The principle of assignment to control-flow variables is that in each protected entry body FALSE is assigned to the variable generated for it (disables itself) and TRUE is assigned to the variable generated for the next protected entry under control flow (enables next). Default values are assigned somewhat alike: the variable generated for the first protected entry under control flow is assigned TRUE, others are assigned FALSE. The same principle applies in the following.

Some syntax changes are needed accordingly as in the following example. Statements

when Cond1=>
accept A_i(_) **do**

_

end A_i;

are translated to

entry A_i(_) **when** (Cond1) **and** Var_A_i_S **is**
begin

_

end A_i;

2. If there is a nesting accept statement in a task body, it is possible that the task can be in rendezvous with different tasks according to outer and inner accept statements. In this case the outer rendezvous cannot complete until all inner rendezvous complete. However, only one protected entry or protected subprogram can be called at the same time. To keep semantics invariant, requeue mechanism is used to simulate a nesting accept statement. The outer accept statement is divided into m (1<m<n+1) protected entries (assume it contains a sequence of n accept statements), a Boolean variable for each protected entry is created, and requeue statements are used to relate them.

Assume there is a nesting accept statement as shown in figure 1.

accept A(_) **do**
S_1;
accept B_1(_) **do**

_

end B_1;
S_2;
accept B_2(_) **do**

_

end B_2;

_

S_n;
accept B_n(_) **do**

_

end B_n;
S_{n+1};
end A;

Figure 1. A Nesting Accept Statement

In figure 1 n>0, A, B_i are entry designators; S_i is a sequences of zero or more assignments, if, loop, case or procedure call statements. A new protected entry NewEntry_A_E_i is constructed for each sequence S_i that is not empty. The generated entries have the same profile as A and are private to the protected object. They are executed implicitly to simulate S_i. Private entry NewEntry_A_E_{n+1} must be constructed even if S_{n+1} is empty. S_1 is added to the end of A before the assignment statements of generated variables. Construct a new Boolean variable Var_A_N_i for each generated private entry. Note that a different naming pattern is used to distinguish variables generated here with those generated in step 1. Variable Var_A_N_i has the same effect as Var_A_i_S, and is called "control-flow variable" too. The generated entries combine with A and B_i that are generated in a previous to form a new control flow. Transformation of a nesting accept statement is the same as transformation of a sequence A; B_1; NewEntry_A_E_2; B_n; NewEntry_A_E_{n+1};(for simplicity we use entry designators to represent the corresponding accept statements), if the corresponding S_i is not empty. Note that, as S_1 has been appended to A, there is no NewEntry_A_E_1 here. But for simplicity, we may use NewEntry_A_E_1 to represent A. Thus the nesting accept statement can be transformed using methods presented in step 1. The only difference is that variables generated for each NewEntry_A_E_i is Var_A_N_i, not Var_NewEntry_A_E_i_S. To simulate the nesting accept statement, a requeue statement is added to the end of A and each NewEntry_A_E_i(0<i<n+1):

requeue NewEntry_A_E_j;

where 0<i<j<=n+1, _ k, i<k<j, S_k is empty. Entry NewEntry_A_E_{n+1} contains no requeue statement, but includes S_{n+1} and assignment statements to generated Boolean variables. The declarations of generated variables and entries are added to the private part of the protected object and default values are assigned to these variables.

Parameters of outer accept statements can be used or referenced in inner ones. To simulate this, some variables are constructed in the protected object to substitute the parameters. These variables are private. The construction rules are: a) for each parameter $Para_i$ of entry A a new variable A_Para_i is created that has same type as $Para_i$; b) for each **in** and **in out** parameter $Para_j$, a statement $A\_Para_j := Para_j$ is added to the beginning of protected entry A; c) for each **out** and **in out** parameter $Para_k$, a statement $Para_k := A\_Para_k$ is added to the end of entry $NewEntry\_A\_E_{n+1}$, before the assignment statements for generated Boolean variables. All other appearance of $Para_i$ is replaced by A_Para_i. Data consistency is thus guaranteed.

3. One of selective alternatives may be a terminate alternative, which is used to terminate a serving task. In this case the terminate alternative can simply be ignored during transformation.

If there is only one accept statement in a selective alternative, the corresponding protected entry should contain all statements following the accept statement in the same selective alternative and keep the order of statements.

If a selective alternative is a sequence of A_1; S_1; _ A_n; S_n;(here n>0, A_i and S_i are same as in step 1, and entry designators are used to represent accept statements), then the selective alternative is transformed using step 1. The only difference is that each S_i should be added to A_i before assignment statements for generated Boolean variables.

If a select statement is contained in a sequence of accept statements, a Boolean variable is constructed using the first entry designator of the first selective alternative. The variable is set by the previous protected entry under control flow, tested and cleared by the protected entries corresponding to every first accept statement of selective alternatives. The Boolean variable of the next protected entry(under control flow) is also set by all these entries.

4. The discussions in step 1 to step 3 is for one entry having one accept statement. When an entry includes multiple accept statements (assume its designator is A) some changes must be taken. Assign a natural to the accept statements having the same designator from 1 in an ascending order. For the i-th accept statement, the control-flow variables Var_A_S, $Var\_A\_N_j$ and variables A_Para_i generated earlier are changed into Var_A_S_i, $Var\_A\_N_j\_i$ and $A\_Para_i\_i$. Other parts that depend on these variables are changed accordingly. These accept statements are composed into one protected entry and an if statement is used. The entry barrier is constructed as follows: "and" the guard with the corresponding generated Boolean variables to form a variable group, "or" these variable groups together to form the entry barrier. Each branch of the if statement is made up of corresponding statements of accept statements and assignment statements for generated Boolean variables, and the condition is the corresponding variable group. If all accept statements have an identical body, then this body can be moved ahead of the if statement to improve readability.

5. The task body may contain an initialization part that is executed only once when the task begins. To simulate this in the protected object, the initialization part is added to the beginning of the first protected entry (under control flow), and a new if statement is formed. A Boolean variable Is_The_First_Time is created as the condition of the if statement. The initialization part becomes the "then" part of the if statement. When the protected entry is called at the first time, and only at this time, the initialization part is executed.

6. Using steps 1-4 iteratively until there are no accept, select or basic loop statements.

5. An Example

A concrete example is given to demonstrate how to use the transforming algorithm. The example is a mailbox which does not separate collecting of mails from depositing. Figure 2 is the code that implement the mailbox using a serving task. Figure 3 shows the transforming result using a protected object.

```
task MailBox is
  entry Deposit(X: in Item);
  entry Collect(X: out Item);
end MailBox;
task body MailBox is
begin
  loop
    accept Deposit(X: in Item) do
      accept Collect(X: out Item) do
       Collect.X := Deposit.X;
      end Collect;
    end Deposit;
  end loop;
end MailBox;
```

Figure 2. Implementation of Mailbox Using a Serving Task

```
protected MailBox is
  entry Deposit(X: in Item);
  entry Collect(X: out Item);
 private
  Var_Deposit_S: Boolean := TRUE;
  Var_Collect_S: Boolean := FALSE;
  Var_Deposit_N2: Boolean := FALSE;
  Deposit_X: Item;
   entry NewEntry_Deposit_E2(X: in Item);
end MailBox;
protected body MailBox is
 entry Deposit(X: in Item)  when Var_Deposit_S
is
  begin
    Deposit_X := X;
    Var_Deposit_S := FALSE;
    Var_Collect_S := TRUE;
     requeue NewEntry_Deposit_E2;
```

```
        entry Collect(X:out Item) when Var_Collect_S
is
         begin
           Collect.X := Deposit_X;
           Var_Collect_S := FALSE;
           Var_Deposit_N2 := TRUE;
         end Collect;
         entry NewEntry_Deposit_E2(X: in Item) when
         Var_Deposit_N2 is
         begin
          Var_Deposit_N2 := FALSE;
          Var_Deposit_S := TRUE;
         end NewEntry_Deposit_E2;
       end MailBox;
```

Figure 3. The Transforming Result of the Mailbox

6. Discussion

The developed transforming algorithm can be applied to serving tasks conversion based on hypotheses described in the section 2.2. However, some serving tasks do not satisfy these hypotheses but they still can be transformed to protected objects. How to recognize and transform these serving tasks in a canonical and correct way is our future work. In hypothesis 4 a selective statement that has waits with else parts is excluded because it's difficult to be represent in a protected object. How to transform it is also our future work.

There may exist name clashes in the generated program, i.e., some names generated by the algorithm conflict with those in the original task. Several methods can be used to solve the problem. One of them is to add a prefix like New_ to generated variables. As this problem does not interfere with the transforming algorithm greatly, solutions are not presented in this paper.

After the transformation finishes some generated variables may be renamed manually to make the logic of the program more obvious. For example variable Var_Deposit_N2 in the mailbox can be renamed to Has_Been_Collected. However, the renaming pattern is dependent on specific domains and is not discussed in the paper.

While transforming, a control flow of the transformed task is needed. The control flow may change during the course of transforming, but the changes are simple and a control flow diagram can be competent. Protected objects transformed using our algorithm can also be optimized. In some circumstances some variables are redundant, they can be removed with the corresponding assignment statements. In other circumstances, some accept statements may be transformed to protected functions to improve concurrency. However, as shared data are used as constants in protected functions, more control flow and data flow analysis are needed.

As protected objects can synchronize and mutual exclude calling tasks automatically, transforming serving tasks to protected objects can improve systems' performance sharply. From the perspective of methodology, rendezvous is control-oriented and protected objects are data-oriented. Protecting shared data using protected objects coincides with object-oriented programming and has been proved that is a good methodology[1].

7. Acknowledgments

We would like to thank Xiaoyu Zhou, Yuming Zhou and Lujun Zhang for their discussions and suggestions.

8. References

[1] Andrews, G. R. Concurrent Programming: Principles and Practice. The Benjamin/Cummings Publishing Company, Inc. 1991

[2] ANSI/MIL-STD-1815A-1983(ISO 8652-1987), Reference Manual for the Ada Programming Language, 1983

[3] Bohm, C., and Jacopini, G. Flow Diagrams, Turing Machines, And Languages With Only Two Formation Rules. Communications of the ACM 9, No.5, 1966: 366-371

[4] Hilfinger, P. N. Implementation Strategies For Ada Tasking Idioms. Proceedings of the AdaTEC Conference on Ada, Arlington, Virginia, October 6-8, 1982: 26-30

[5] ISO/IEC 9652: 1995(E), Ada Reference Manual Language and Standard Libraries, 1995

[6] Jackson, M. A. Principles of Program Design. Academic Press, 1975

[7] Locke, D., Mester, T. J., and Vogel, D. R. Replacing Passive Tasks with Ada 9X Protected Records. ACM Ada Letters, 1993, 13(2): 91-96

[8] Naiditch, D. J. Rendezvous with Ada 95. John Wiley & Sons, Inc. 1995

[9] Xu, B. Ada 95 Protected Objects And Data-Oriented Synchronization. Computer Research and Development, 1997, 34(1): 72-77