

Transformation of the Ravenscar Profile based Ada real-time application to the verification-ready statecharts : Reverse engineering and Statemate approach

Chang Jin Kim, Jin-Young Choi

Formal Methods Lab. Korea University

242 A-San Science Building, Anam-dong 5-1, Sungbuk-gu, Seoul, Korea (136-713)

{cjkim, choi}@formal.korea.ac.kr

Tel: 082-02-929-8681, Fax: 082-02-953-0771

Abstract

The Ravenscar Profile is a subset of Ada95 tasking model which removes the Ada's unsafe real-time characteristics and allows high-integrity of system. By the Ravenscar Profile, Ada95 can meet the determinism on system behavior. It also allows schedulability analysis and formal verification on the concurrent model of system. But the formal verification may be additional hard works to improve value of the Ravenscar Profile. In this paper, we present the transformation of Ravenscar programs to statecharts model by reverse engineering techniques, which allows the statecharts to be used for program analysis and formal verification. The advantage of these works is transforming the Ada application code into the verification-ready statecharts with ease and simple way than general formal methods approach. We used STI's reverse engineering tool, Understand for Ada version 1.4 for code analysis and I-Logix's Statemate Magnum 4.1 for modeling and model checking purpose.

Key Words: Ada, Ravenscar Profile, Statemate, Verification, Reverse Engineering

1. Introduction

Ada was developed by US DoD as one of the solution against 'software crisis'. It has been used for the design and development of various systems in military, avionics and space area. But in spite of the excellence of language features, Ada has a weakness that language itself can not guarantee the quality and integrity of system just like as other programming languages. Ada contains the various real-time characteristics but the most of real-time systems using Ada83 adopted 'non tasking' model which excluded some of the Ada83's real-time primitives such as 'task' or 'rendezvous' [1].

After Ada95 introduced its object-oriented features and protected object, Ada real-time issues are investigated in earnest regarding shared resource problem of rendezvous, unbounded priority inversion etc.

Traditional methods for the design and development of complex applications, which concentrate primarily on functionality, are increasingly inadequate for hard real-time systems.

This is because non-functional requirements such as safety, reliability, timeliness, memory usage and dynamic change management are left until too late in the development cycle [2]. And so far, the most of approach for formal verification is an entirely separated process from the main project and it is apt to be performed by external formal methods expert group.

Regarding above issues, the Ravenscar Profile is considered to provide the integrated guidelines which accommodate design, analysis and verification. Not only defining highly restricted tasking model but also it requires static analysis and formal verification on concurrent model.

Up to now, there are not much practical implementations fully applying the Ravenscar Profile. It might be very challenging for industries or commercial vendors to achieve formal certification satisfying the Ravenscar Profile. Formal certification requires verification evidence of safety and reliability on the implementation. But those are hard to guarantee because the complexity of Ada concurrent model's dynamic semantics and the scale of system's states represented by dynamic data structures.

Those are why the Ravenscar Profile has been applied for embedded systems which are comparably small and have strict verification bound.

In this paper, we present the possibility of applying the Ravenscar Profile in practical development by showing the Statemate can represent the Ravenscar program and it can be verified by model checking. Statemate is not a perfect formal specification language and we did not aim the verification of entire system. But we focus on the familiar and easy way to facilitate the Ravenscar Profile on development process. Most developers are get used to make programs first, and then analyze them for testing purpose. Our approach of reverse engineering for verification is due to that reason.

2. Values of the Ravenscar Profile

Ravenscar Profile defines the tasking model for real-time systems such as aircraft's embedded systems or reactor system in nuclear plants which require highest-integrity, safety and reliability. This restricted subset of Ada95 tasking model satisfies determinism, schedulability analysis and memory boundedness required in hard real-time systems.

It's been increasingly known that development or operating of traditional Ada run-time system causes huge costs or risky safety problems. To remove the unsafe properties of Ada language, many researches had been undertaken and the efforts achieved success in 8th IRTAW (1997) which adopted the Ravenscar Profile proposed by Alan Burns and Wellings.

The analysis of an Ada application that makes unrestricted use of Ada run-time features including tasking rendezvous, select and abort statements are not currently feasible. In addition, the nondeterministic and potentially unbounded behaviour of many tasking and other run-time calls may make it impossible to provide the upper bounds on execution time that are required for schedulability analysis and simulation.

Thus Ada coding style rules and subset restrictions must be followed to ensure that all code within critical tasks is statically time-bounded, and that the execution of the tasks can be defined in terms of response times, deadlines, cycle times, and blocking times due to contention for shared resources [2].

Regarding the above issues, the Ravenscar Profile restricts the Ada tasks only to exchange data through protected objects. The use of protected object allows the task behavior to be characterized and verified as sequential code.

Also the schedulability analysis on the program can be clarified by the fixed priority scheduling policy of the Ravenscar Profile.

There have been several case researches to prove the excellence of the Ravenscar Profile. In OBOSS, ICU, and AOCS cases, it is shown that the Ravenscar-like design has an enough expression capability. Also it is feasible to implement the relatively low-level development using a Ravenscar-approach, and for the simple algorithms running in the AOCS benchmark the Ravenscar code generator rules have been preliminarily tested [3].

The values of the Ravenscar Profile can be summarized as follows [3].

- It makes the design clear and simple.
- It's possible to raise the real-time issues to specification level
- Early checking that real-time requirements are consistent and in line with the design
- It can achieve the performance improvements by the

simplified runtime system

- The real time characteristics become predictable by the fixed priority analysis.
- It is portable since features that do not have formal clear or unambiguous semantics have been removed.

So, the Ravenscar Profile is suited for certification and use in safety critical system.

3. Application of the Ravenscar Profile

3.1 Ravenscar Profile Coding Style

The Ravenscar Profile does not require specific coding style for writing Ada applications. But the templates comply with the Ravenscar Profile can be useful for certain jobs such as schedulability analysis with supporting tools.

The basic considerations of the templates include cyclic tasks, sporadic tasks, interrupt handler and protected objects. To apply the Ravenscar Profile in ADA PDL (Program Design Language) level, we modify the Kjell Nielsen's ATCS model [4] and present its source codes.

The typical form of a cyclic task body for the Ravenscar Profile contains an outermost infinite loop with one or more 'delay until' statements as its last statement.

It is required to separate the sporadic tasks in certain interval to allow all the tasks in the system operate on time. It also can be resolved by using 'delay until' statement, which delays the event-triggered task not to be invoked too often exceeding the specific interval.

The codes below show the example for a cyclic task using 'delay until' statement.

```
with Ada.Real_Time; use Ada.Real_Time;
with Definitions; use Definitions;
with Track_File_Monitor;
with External_Devices;

separate (Display_Interface)

task body Interface is
  x, y : Coordinates;
  Track_Index : Track_Number;
  Next_Time : Ada.Real_Time.Time;
begin
  Track_File_Monitor.Epoch.Get_Start_Time
                                (Next_Time);
  Next_Time := Ada.Real_Time.Clock
                                + Display_Periodic;

  Periodic_Display:
  loop
    -- Delay appropriate interval
    delay until Next_Time;
    Track_File_Monitor.Initialize_Track_Reading;
```

```

Display_The_Tracks:
loop null;
  -- Use Track_File_Monitor.Next_Track
  -- and External_Devices.Display_A_Track
  -- to get and display tracks
end loop Display_The_Tracks;
end loop Periodic_Display;
end Interface;

```

Example 1. Infinite loop containing 'delay until'

If the other types of voluntary-suspension are used, the schedulability analyses may become difficult.

A cyclic task in the Ravenscar program must contain an infinite loop because task termination in the program is considered as an error.

The type of time used in the Ravenscar Profile is that of 'Ada.Real_Time.Time' which is more precise than the 'Calendar.Time'. If there are several tasks using the time, they might read system clock independently regardless of the other tasks. But it is useful to share a common epoch that can adjust each task start time considering each other. It's because the use of priority or common epoch make the precedence relations between the tasks of same priority and prevent blocking of tasks on running.

In the case of event-triggered task, the body of task has an outermost loop and the loop contains 'Ada.Synchronous_Task_Control.Suspend_Until_True' statement with statements using protected entry or suspension object [5].

The suspension object is typically used when the signaler task and waiter task just perform simple signal operation each other without data exchange. If they need to perform the additional operation other than signaling, they should use protected entry.

The below is an example code for the protected object 'Track_Index' and 'Epoch'. 'Track_Index' provides the subprogram for mutual exclusive access to the shared resource. In the codes, the 'Epoch' is used for sharing common epoch between the tasks.

```

with System;
with Ada.Real_Time;
with Definitions; use Definitions;

package Track_File_Monitor is
  protected Track_Index is
    function Get_Track return Track_Number;
    procedure Update_Track_File
      (Data : in Track_Number);

  private
    pragma Priority(5);
    Current : Track_Number;
  end Track_Index;

  protected Epoch is
    procedure Get_Start_Time

```

```

(T : out Ada.Real_Time.Time);

  private
    pragma Priority(System.Priority'Last);
    Start : Ada.Real_Time.Time;
    First : Boolean := True;
  end Epoch;

  procedure Add_Track (x, y : in Coordinates);
  Track_Store_Full : exception; -- raised by Add_Track
  procedure Initialize_Track_Reading;
  procedure Next_Track (x, y : out Coordinates;

  Track_Id : out Track_Number);
  No_More_Tracks : exception;
  -- raised by Next_Track
  procedure Update_Track (x, y : in Coordinates;
    Track_Id : in Track_Number);
  end Track_File_Monitor;

package body Track_File_Monitor is
  protected body Track_Index is
    function Get_Track return Track_Number is
    begin
      return Current;
    end Get_Track;

    procedure Update_Track_File
      (Data : in Track_Number) is
    begin
      Current := Data;
    end Update_Track_File;
  end Track_Index;

  protected body Epoch is
    procedure Get_Start_Time
      (T : out Ada.Real_Time.Time) is
    begin
      if First then
        First := False;
        Start := Ada.Real_Time.Clock;
      end if;

      T := Start;
    end Get_Start_Time;
  end Epoch;

  -- Monitor
  task Monitor is
  -- Calls
  -- Track_File_Manager.Initialize_Track_Reading
  -- Track_File_Manager.Next_Track
  -- Track_File_Manager.Add_Track
  -- Track_File_Manager.Update
  -- Track_File_Manager.Correlate
  -- Track_File_Manager.Extrapolate
  end Monitor;
  -- Extrapolation_Timer

```

```

task Extrapolation_Timer is
-- Calls
-- Monitor.Extrapolate
-- delays for appropriate extrapolation interval
-- and tells Monitor that it is time to extrapolate
end Extrapolation_Timer;

-- Transport_Coordinates
task Transport_Coordinates is
-- Calls
-- Raw_Data_Buffer_Package
-- Transports a set of coordinates from the
-- buffer to the track file Monitor
end Transport_Coordinates;

-- task bodies
task body Monitor is separate;
task body Extrapolation_Timer is separate;
task body Transport_Coordinates is separate;
end Track_File_Monitor;

```

Example 2. Spec and implementation of the protected object

The protected object used for the purpose of mutually exclusive access is apt to consist of protected subprogram without protected entry. Because the protected entry is usually used for inter-tasks synchronization purpose only. Protected procedure can change the data in protected object but protected function just reads the data in the protected object not affecting the contents of the data [6].

3.2 Transformation of the Ravenscar Program to statechart for verification

In the chapter, we present the transition steps of the Ravenscar program to the Statemate [7] model – statecharts – which is ready to be formally verified. For this, it is required to analyze the source code to identify tasks, protected objects and relationship between them. We do not deal with the schedulability analysis in the paper, but concentrate on the protected objects and its related task interactions for modeling and verification. The benefits of using Statemate as an additional design tool are the important start point of this research. First, by using the appropriate tools, the scale of the Ravenscar program can be extended, not restricted to the small program or partial implementation. Second, it is simple and clear to identify the Ravenscar artifacts such as task, protected object by the tool's graphical, intuitive expressions. Third, the errors are reduced in the design phase. The correctness and completeness checking of the Statemate derive the syntactic and logical conflicts even before the formal verification. Positively, it is effective way to reduce the schedule and costs of the program

Forth, the simulation and rapid prototyping demonstrate the behavior of the system and it can be applied as a feedback channel to facilitate the cycle of the program development.

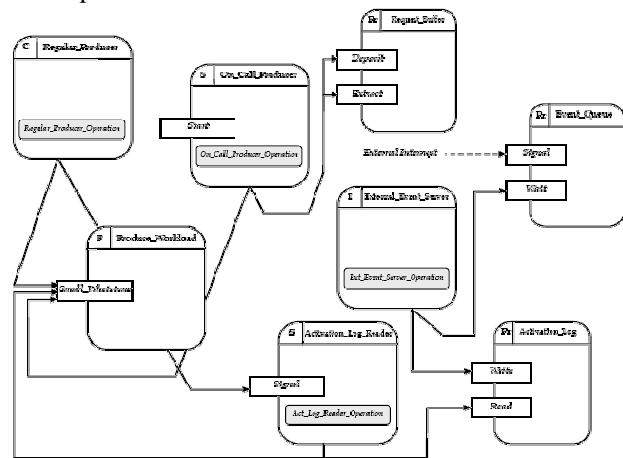


Figure 1. Structure of the example system (HRT-HOOD [8]-like expression)

Figure 1 shows HRT-HOOD-like expression for the example system referred in [2], which performs the dynamic workload with the periodic task and sporadic task.

These are the system requirements recomposed from the original.

- R1. Regular_Producer includes the periodic task and performs the certain amount of the workload provided by the Small_Whetstone program.
- R2. Regular_Producer takes over On_Call_Producer the workload exceeding its threshold (756).
- R3. On_Call_Producer has a sporadic task triggered by R2 condition.
- R4. Request by the Regular_Producer invokes the Start operation of On_Call_Producer and the Start operation deposits the request at the Request_Buffer via Deposit operation. The Request_Buffer should be defined as a protected object because it should accept the new request during the execution of the previous request.
- R5. While the system performs the workload, sporadic interrupt request from external device can arrive and the request is transformed via the Signal operation of the protected object, Event_Queue Signal.
- R6. The sporadic task External_Event_Server writes the number of interrupts and each occurrence time via the Write operation of the Activation_Log.
- R7. The Regular_Producer reads the records of the Activation_Log at a certain condition via the Read operation of the Activation_Log_Reader to monitor the service requests from the external devices.

R8. Regular_Producer performs the task synchronization using the suspension object.

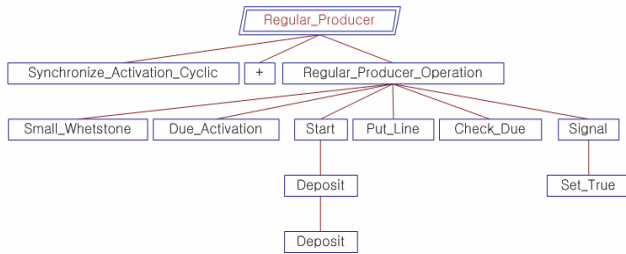
Also, [2] presents the written Ada source codes fulfilling the above requirements but it's very hard to verify the behavior of the system with analyzing the source codes only. So we attempt to translate the program written by Ravenscar Profile guideline into the formal verification-ready design model.

First of all, we analyzed the written codes using the reverse engineering tool, STI's UFA (Understand for Ada) V.1.4 [9] to identify the components – tasks and protected objects.

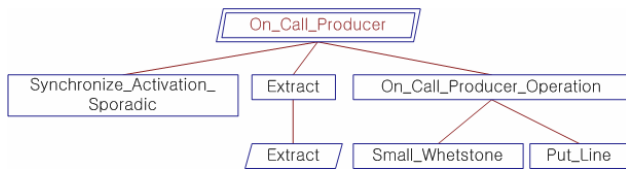
UFA analyzes the source codes and generates various graphical views such as declaration tree or invocation view.

Invocation view of a certain task shows that which subprograms or entries the task invokes, and which protected objects the task accesses to.

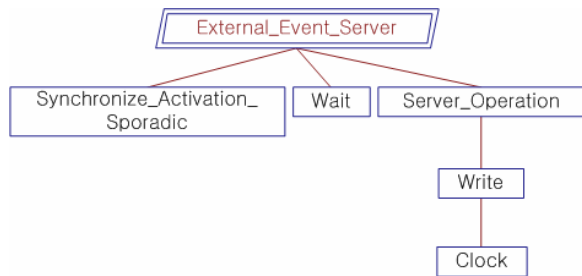
Figure 2 shows the objects of our interests in the example program. The selected tasks and protected objects are to be transformed into the matching components in the statechart.



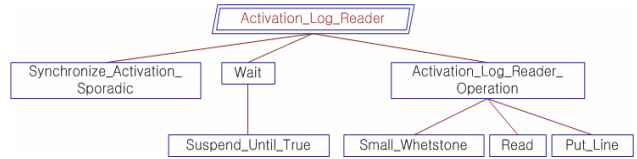
<Invocation View: Task Regular_Producer>



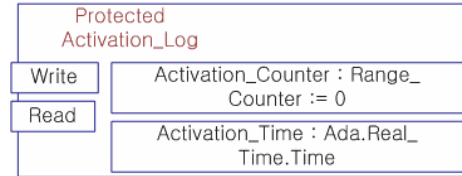
< Invocation View: Task On_Call_Producer>



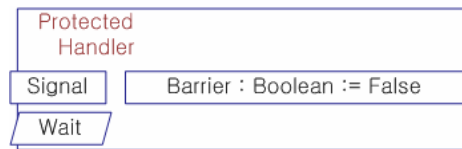
< Invocation View: Task External_Event_Server>



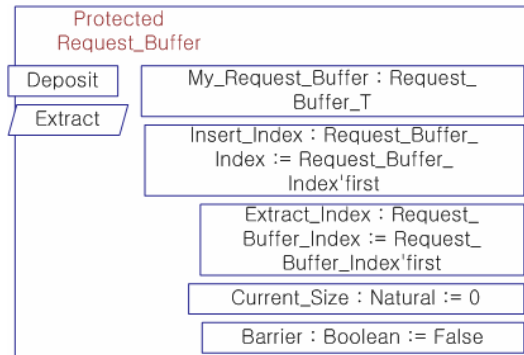
< Invocation View: Task Activation_Log_Reader>



<Declaration: Protected Object Activation_Log>



< Declaration: Protected Object Activation_Log>



< Declaration: Protected Object Activation_Log>

Figure 2.Examples of the UFA's graphical views

The first step of transformation is to define the "Object_ID" table which contains the tasks and protected objects identified by the invocation or declaration views.

Name	Type	Symbol
Regular_Producer	Task	A
On_Call_Producer	Task	B
External_Event_Server	Task	C
Activation_Log_Reader	Task	D
Activation_Log	P.O	E
Handler	P.O	F
Request_Buffer	P.O	G

Table 1. Object ID table to symbolize the tasks and protected object identified.

As shown in the Table 1, each named entity has a unique symbol such as A, B, C and so on, for convenience of identification. We selected 4 tasks and 3 protected objects from the program and assigned each object a matching alphabet symbol.

The next step is to construct the “Method_ID” table which identifies the subprogram called by tasks, protected subprogram and entries of the protected objects that shown in the UFA’s views.

To simplify the problems in the paper, we just consider the relationship between tasks and protected objects, not the timing constraint for scheduling policy. And even the methods listed in the table may not be shown in the final step – the statechart. The reason will be explained in the later part.

The identified entities in the table are considered as the methods to interact between each task and protected object. In the table, the caller task/protected object is designated in the column “Fm (From)” and the callee task or the destination of the data/signal is designated in the column “To”. For example, the procedure 1.Regular_Producer_Operation is called by the task Regular_Producer (A) and affects the task Activation_Log_Reader (D) or the protected object Request_Buffer (G). These are used later as foundations to define relations between the states of the system.

No	Name	Tp	Fm	To
1	Regular_Producer_Operation	P	A	D/G
2	Start	F	A	G
3	Deposit	P	A	G
4	Deposit	F	A	G
5	Signal	F	A	D
6	Set_True	P	A	D
7	On_Call_Producer_Operation	P	B	G
8	Extract	F	B	G
9	Extract	E	B	G
10	Server_Operation	P	C	E
11	Wait	P	C	C
12	Write	P	C	E
13	Activation_Log_Reader_Op'	P	D	E
14	Wait	P	D	D
15	Read	P	D	E

(Tp (Type) - P: Procedure, F: Function, E: Entry)

Table 2. Method ID table to identify the subprograms or entries

By these steps, the procedures, functions or the protected entries needed to construct the re-design of the system are identified so that will be utilized for modeling the system’s behavior.

In the next step, the states are defined considering tasks and protected objects in this table, and the detailed

operation of each method should be refined to be used in defining the transition relations between the states. Once the components representing the functionality of the system are identified, we could design the top-level activity chart of Statestate shown in Figure 3

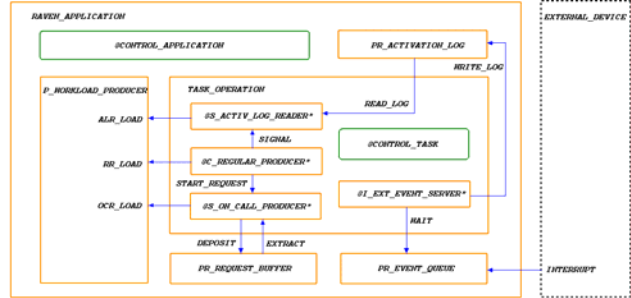


Figure 3.Statestate top-level activity chart for the example system

Now we determine the states of the system and depict them into the “State_ID” table considering the table1 and table2.

As shown in the Table3, each task’s running state is defined as one of the states in the system. For the protected object, its reading state and writing state are separated and the protected object of synchronization purpose only is not represented as a state, e.g. Event_Queue in the Table1. The tasks which access to the Event_Queue just check the signal from the Event_Queue, so it simply provides the condition of invoking the task Interrupt_Service. The following table “State_ID” is the result of the steps to define the system’s state, which consists of 8 states.

No	State Name	Related Task/P.O	Remarks
1	Periodic_Processing	Regular_Producer	A
2	On_Call_Processing	On_Call_Producer	B
3	Interrupt_Service	Ext_Event_Server	C
4	Log_Access	Active_Log_Reader	D
5	Reading_Act_Log	Activation_Log	E
6	Writing_Act_Log	Activation_Log	E
7	Deposit_Rqst	Request_Buffer	G
8	Extract_Rqst	Request_Buffer	G

Table 3. State ID table

The final step is to define the state transition relations according to the Method_ID table and the State_ID table. In this step, the parameters of each subprogram or the entry are considered as artifacts to compose the transitions. But if the subprogram transmits the parameters via two or more other subprogram calls, those whole sequences are considered as one transition condition. For example, 3. Deposit and 4. Deposit in the Table 2 are merged into one trigger condition which performs the transition from PERIODIC_PROCESSING to DEPOSIT_RQST.

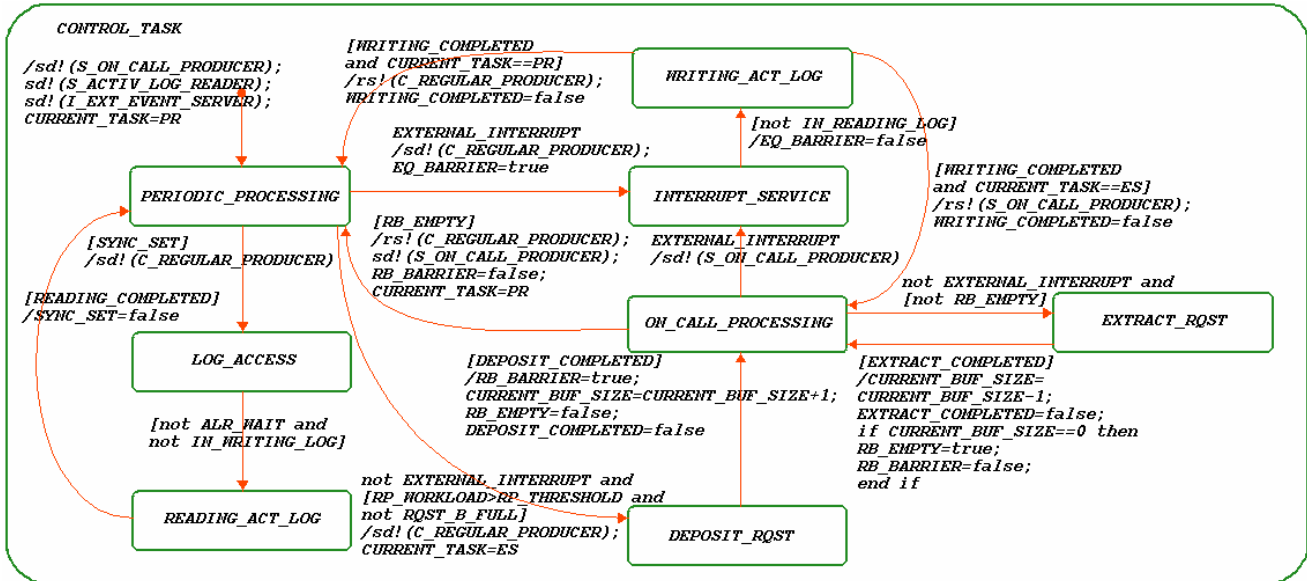


Figure 4. Statechart transformed from the example system : tasks and protected objects interactions

The statechart represents the task interactions with the protected objects are shown in the Feature 3.

The model in the statechart is now ready to be checked with the ‘correctness’ and ‘completeness’ by the embedded functionality of the Statemate. If there’s no errors, the model can be model-checked by ‘ModelChecker’, the model checking tool of the Statemate. In this paper, we don’t treat the detailed steps of model checking, but ModelChecker can perform non-determinism, race condition, range violation and reachability check etc. If the entered property is not fulfilled, the ModelChecker automatically generates the SCP (Simulation Control Program) which represents the counter example.

4. Conclusion

The Ravenscar Profile is the subset of Ada 95 tasking model for hard real-time systems.

Inter-task communication via protected object improves the deterministic operation of the program and guarantees the mutual exclusion of the shared resource access. Also, the priority based preemptive scheduling policy allows the schedulability analysis and the Ravenscar programs are fit for the formal verification.

We performed the analysis on the written Ravenscar program through the reverse engineering technique using UFA and the re-design of the current program by Statemate and ModelChecker tool. All these are to improve the efficiency of the work, and ultimately, to perform the formal verification on the Ravenscar program.

Completing the application of the Ravenscar Profile to actual development process, the key point is to integrate formal verification process into the main project stream. Because of the difficulty derived from the formalism, its related works tend to be treated as a other new quest. So, to make an effect on actual work process, the tool support in the project level which facilitates the formalism-related activities is a mandatory.

5. References

- [1] Kristina Lundqvist, Lars Asplund, "A Formal Model of a Run-Time Kernel for Ravenscar", Uppsala Univ., Information Technology, 1999
- [2] Alan Burns, et al., "Guide for the use of the Ada Ravenscar Profile in high integrity systems", University of York Technical Report YCS-2003-348, 2003
- [3] Morten Rytter Nielsen, "(Ada) Ravenscar Evaluation Workshop", 29th Nov., 2001
- [4] Kjell Nielsen, "Designing Large Real-Time Systems with Ada", Hughes Aircraft Company, 1987
- [5] Ada95 Reference Manual, International Standard ANSI/ISO/IEC-8652:1995, 1999
- [6] Michael A. Smith, "Object-oriented Software in Ada95 Second Edition", Mcgraw-Hill, 2001
- [7] David Harel, Michal Politi, "Modeling Reactive Systems with Statecharts: The Statemate Approach", I-Logix, 1999
- [8] Alan Burns, Welling, "HRT-HOOD : A design method for hard real-time Ada, Real-Time Systems, 1994
- [9] <http://scitools.com/products/understand/ada>