A *Circus* semantics for Ravenscar protected objects

D. Atiya[1], S. King[1], and J. C. P. Woodcock[2]

[1] Department of Computer Science, University of York
[2] Computing Laboratory, University of Kent

Abstract. The Ravenscar profile is a subset of the Ada 95 tasking model: it is certifiable, deterministic, supports schedulability analysis, and meets tight memory constraints and performance requirements. A central feature of Ravenscar is the use of protected objects to ensure mutually exclusive access to shared data. We give a semantics to protected objects using *Circus*, a combination of Z and CSP, and prove several important properties; this is the first time that these properties have been verified. Interestingly, all the proofs are conducted in Z, even the ones concerning reactive behaviour.

Keywords: Ravenscar, Ada Protected Objects, Formal Semantics, Z, *Circus*.

1 Introduction

Ada [6, 3] is a high-level programming language designed specifically for large systems, where issues like reliability and safety are major concerns. The language has proved to be successful in the area of high-integrity industrial systems; for example, over 99% of the avionics software in the Boeing 777 is in Ada. Ada's model for concurrent programming is powerful and extensive, but it is also complex, making it difficult to reason about the properties of real-time systems. The Ravenscar profile has been proposed as a greatly simplified subset of the Ada tasking model [4, 5]. The profile has already been accepted [1] for inclusion in the next revision of the Ada language standard. It places an emphasis on predictability and verifiability, and is certifiable and deterministic, supports schedulability analysis, and meets tight memory constraints and performance requirements.

The Ravenscar profile does not allow Ada's rendezvous construct for communication between tasks. Instead, tasks in Ravenscar communicate through shared variables, usually encapsulated inside *protected objects*. This makes protected objects central and fundamental building blocks in Ravenscar programs, as they provide a safe mechanism for accessing the data shared between various tasks. We provide a formal semantics for Ravenscar protected objects, and prove that they enjoy several important properties. We are currently working on a cost-effective technique [2] for verifying Ravenscar programs and this formal semantics is a central part of our work.

The rest of this paper is organised as follows. Section 2 provides a brief, informal account of Ravenscar protected objects; Section 3 then gives a description of *Circus*, which is used to provide their formal model. Sections 4 and 5 contain the semantics and proofs of desired properties. Finally, Section 6 draws conclusions and discusses related work. We show that, although the motivation for providing a formal semantics for Ravenscar protected objects is a specialised one, the results are of general interest in the area of verifying concurrent programs.

2 Ravenscar protected objects

Ada protected objects [3, 6, 16] provide a mechanism for asynchronous communication between tasks over shared variables. The declaration of an Ada protected object comprises two main parts: the private data to be shared between the communicating tasks, and an interface of operations for accessing that data. The data encapsulated within a protected object can be accessed only through the interface provided by that object. It is guaranteed that the operations are executed in a manner that ensures mutual exclusion while the data is updated [6].

There are three kinds of interface operations: *protected functions*, *protected procedures*, and *protected entries*. Whilst protected functions provide concurrent read-only access, protected procedures and protected entries provide mutually exclusive read/write access to the data encapsulated in a protected object. Also, calls to protected functions are executed in mutual exclusion with calls to protected procedures/entries. Thus, at any moment, the protected object is in exactly one of the following states:

1. No calls are executing.
2. Only protected function calls are executing.
3. Only one call is executing, either a protected procedure or a protected entry.

An entry call is guarded by a boolean-valued *barrier*. If a protected entry call is made when the barrier is false, then the call is suspended, even if there are no function or procedure calls currently executing inside the protected object. The suspended task goes into the *entry queue*, to wait until the barrier is true and there are no calls are currently executing. All barriers get re-evaluated after the execution of any procedure or entry call [6, Chapter 7]. Thus, entry barriers can be used to provide conditional synchronisation between tasks accessing the protected object. For example, if the data encapsulated inside the protected object is an array, then one can use entry barriers to state that a read (write) cannot be performed when the array is empty (full).

To help provide a concrete view of the discussion above, Figure 1 presents an Ada protected object that encapsulates an integer variable. The encapsulated variable can be accessed only through the protected function *Read*, which returns the current value, and the protected procedure *Write*, which assigns a new value. Many tasks could be concurrently executing *Read* calls; however, if one task is actively executing a *Write* call, then no other task can concurrently execute either a *Read* or a *Write* call until the current task has finished.

```
– – A protected object containing an integer variable
protected pInteger is

    function Read return INTEGER;

    procedure Write (data : in INTEGER);

    private
        d : INTEGER;

end pInteger;

protected body pInteger is

    function Read return INTEGER is
    begin
        return d;
    end Read;

    procedure Write (data : in INTEGER) is
    begin
        d := data;
    end Write;

end pInteger;
```

Fig. 1. An example of a protected integer variable

The designers of the Ravenscar profile chose protected objects as the only mechanism for communication between tasks, in order to improve schedulability analysis [5]; they also imposed a number of other restrictions, in order to meet various design requirements, such as determinism. Many of these restrictions are syntactic; for example, Ravenscar does not permit declaration of protected objects local to subprograms, tasks, or other protected objects. The discussion of such restrictions is not relevant in this paper; rather, we are interested in the restrictions imposed on the functional aspects of protected objects, which can be summarised as follows:

- **R1:** A protected object can have at most one entry.
- **R2:** No more than one task may queue on an entry at any time.
- **R3:** The barrier must be either static or the value of a component.
- **R4:** Like in Ada, potentially blocking *operations* are not allowed.

An application could further restrict **R2** so that only one task is able to call each protected entry [5]. We adopt the strong version of **R2**, as a static check could be provided for it. Thus, at most one task can be associated with the protected entry of a protected object and this task can then be determined statically at compile time. In **R4**, the profile prohibits the presence of entry call statements inside the body of any protected operation, as the execution of that operation could then block.

Section 4 provides a formal model in *Circus* of Ravenscar protected objects; for those not familiar with the language, the next section describes *Circus*.

3 Circus

Circus is a unified programming language that combines Z [13, 25] and CSP [11, 19] constructs, together with specification statements [17] and guarded commands [9]. With Z and CSP integrated into the language, *Circus* can be used to describe both the state-oriented and the behavioural aspects of concurrent systems. Though there are several other examples of combining Z and CSP in the literature (see, for example, the survey in [10]), *Circus* distinguishes itself by a theory of refinement [7, 8, 21] for the derivation of programs from their specifications in a calculational style like Morgan's [18]. The formal semantics of *Circus* [24] is based on unifying theories of programming (UTP) [12], as well as various laws for refining [8] specifications into designs and code.

A *Circus* program is a sequence of:

- ***Z paragraphs:*** declaring the types, global constants, and other data structures used by the processes defined in the program.
- ***Channel definitions:*** declaring typed channels through which processes can communicate or synchronise.
- ***Process definitions:*** declaring encapsulated state and reactive behaviour.

In its simplest form, a process definition is a sequence of Z paragraphs describing the internal state, and a sequence of actions (defined in terms of Z schemas, CSP operators, and guarded commands) describing the possible interaction between the process and its environment. In more sophisticated forms, a process may be defined in terms of combinations of other processes using the operators of CSP.

Example: We use *Circus* to model a simple bank account that stores the account balance and provides four ways of interacting with the outside world:

- Initialise the balance and overdraft facility.
- Credit the account.
- Debit the account.
- Request the account balance and funds available.

Figure 2 contains the *Circus* program. Interaction with the outside world is through the four channels: *init*, *cred*, *deb*, and *bal*. The encapsulated state has three components: the account *balance* (which may be negative), the permitted overdraft (a non-negative value), and the funds available (invariantly, the sum of the balance and the overdraft). A further state invariant requires that the funds be non-negative too, so that the balance hasn't exceeded the overdraft facility. Thus, a legitimate state may have a *balance* of -£450, and an *overdraft* of £1,000; this implies that *funds* is set to £550, which satisfies the required constraint.

$$
\begin{array}{l}
\textbf{channel}\ init, bal : \mathbb{Z} \times \mathbb{N};\ cred, deb : \mathbb{N} \\
\textbf{process}\ BankAccount \mathrel{\widehat=}\ \textbf{begin} \\
\quad AccountState \mathrel{\widehat=} [\, balance, funds : \mathbb{Z};\ overdraft : \mathbb{N} \mid funds = overdraft + balance \geq 0 \,] \\
\quad InitAccountState \mathrel{\widehat=} [\, AccountState';\ d?, o? : \mathbb{N} \mid balance' = d? \land overdraft' = o? \,] \\
\quad Credit \mathrel{\widehat=} cred\,?\,value \rightarrow balance := balance + value \\
\quad Deduct \mathrel{\widehat=} (\, amount : \mathbb{N} \bullet amount \leq funds \ \&\ balance := balance - amount \,) \\
\quad Debit \mathrel{\widehat=} deb\,?\,value \rightarrow Deduct(value) \\
\quad CheckBalance \mathrel{\widehat=} bal\,!\,(balance, funds) \rightarrow Skip \\
\quad \bullet\ (\, init\,?\,(d, o) \rightarrow InitAccountState;\ \ (\, \mu\, X \bullet (\, Credit \,\Box\, Debit \,\Box\, CheckBalance \,);\ X \,) \,) \\
\textbf{end}
\end{array}
$$

Fig. 2. A *Circus* *BankAccount*

The external behaviour of this process is given by the *main action*

$$
\begin{array}{l}
init\,?\,(d, o) \rightarrow InitAccountState; \\
\mu\, X \bullet (\, Credit \,\Box\, Debit \,\Box\, CheckBalance \,);\ X
\end{array}
$$

This depends on the definition of four auxiliary actions, each defined within the body of *BankAccount*. First the account is initialised with a communication of the variables d and o through the channel *init*. Following the communication, the schema action *InitAccountState* is executed; this specifies that *balance* is assigned the value of d, and *overdraft* the value of o. This is followed by a non-terminating loop that repeatedly offers the external choice between three actions. The *Credit* action inputs a natural number *value* through the *cred* channel and then adds this to the *balance*. The *Debit* action inputs a natural number *value* through the *deb* channel and then behaves as specified by $Deduct(value)$. The parametrised action *Deduct* subtracts its argument from the *balance*, provided that there are enough *funds* to do this. Finally, *CheckBalance* outputs both the *balance* and *funds* available as a pair on the *bal* channel. □

4 Protected objects: a *Circus* model

4.1 Global definitions

Protected objects are used for interaction between tasks in a Ravenscar program. Tasks are drawn from the given set *TaskId*; by convention, there is a

distinguished identifier that is never used by any task.

$$[\, TaskId \,]$$

$$null\_task : TaskId;\ ValidTaskId : \mathbb{P}\ TaskId$$
$$\hline$$
$$ValidTaskId = TaskId \setminus \{null\_task\}$$

We model a protected object as a *Circus* process with nine channels, each corresponding to some interaction with its environment. The channel *read* (*write*) is used to communicate the events where a task issues a call to a protected function (procedure). If the entry task issues a call to the protected-object entry, and the barrier is true and no other task is accessing the object, then the entry task can gain access; this is modelled by a communication over the *enter* channel. Otherwise, the entry task must *wait*. If at some later point, the barrier becomes true and there are no tasks accessing the object, then the waiting entry task may *start*. An event on the channel *leave* corresponds to a task leaving the object. Changes in the state of the barrier are signalled through the *update_bar* channel, after the execution of a protected procedure or the protected entry. Finally, the channels *get* and *put* are used for accessing and updating the protected object's data.

$$\textbf{channel}\ read, write, enter, wait, start, leave : ValidTaskId$$
$$\textbf{channel}\ [\, T \,]\ update\_bar : T \times Boolean;\ get, put : ValidTaskId \times T$$

Every communication between the protected object and a task requires the task's identity as part of the event; in each case, the *null_task* is excluded. Our model of the protected object is generic, in that we parametrise the type of the data being encapsulated. As a consequence, the channels *get* and *put* are also defined generically.

$$\textbf{process}\ PO\ \widehat{=}\ [\, T \,]\ \textbf{begin}$$

In the next section, we describe the state of a protected object.

4.2 Process state

There are six components in the state of a protected object.

- The data encapsulated, *data*, of type T.
- The entry task's identifier, *entry_task*. If this is the *null_task*, then no entry call is possible.
- The current value of the boolean entry barrier, *barrier*.
- A boolean flag that is true exactly when the entry task is *waiting*.
- The set of *readers*, those tasks currently actively executing a function call.
- The set of *writers*, those tasks currently actively executing a procedure or an entry call.

Both sets must be finite and contain only valid task identifiers. As usual, we use *Circus*'s boolean values as though they were predicates.

There are three further state invariants.

- Reading and writing are mutually exclusive.
- There must be no more than one writer.
- If the entry task is waiting, then it can neither be the *null_task* nor a reader or writer.

The declaration and invariants are collected into a schema describing the state of the process.

$$
\begin{array}{|l}
\hline
POState \\
\hline
data : T \\
entry\_task : TaskId \\
barrier, waiting : Boolean \\
readers, writers : \mathbb{F}\ ValidTaskId \\
\hline
readers \neq \emptyset \Rightarrow writers = \emptyset \\
\#writers \leq 1 \\
waiting \Rightarrow entry\_task \neq null\_task \land entry\_task \notin readers \cup writers \\
\hline
\end{array}
$$

The invariant that readers and writers be mutually exclusive is captured in our model by requiring that, if there are any *readers*, then there must be no *writers*. Note that this also requires as a consequence (its contrapositive), that, if there are any *writers*, then there must be no *readers*.

4.3 Process actions

A protected object in its initial state has no waiting entry task and no readers or writers; its data, entry task identifier, and barrier must be given initial values.

$$
\begin{array}{|l}
\hline
InitPOState \\
\hline
POState' \\
d? : T \\
t? : TaskId \\
b? : Boolean \\
\hline
data' = d? \land barrier' = b? \land entry\_task' = t? \\
\neg\ waiting' \land readers' = writers' = \emptyset \\
\hline
\end{array}
$$

1. When a task issues a function call, it may become a reader within the protected object; this is signalled by the communication of the task's identifier over the *read* channel. This event is permitted if there are no writers, and no waiting entry task with an open barrier.

$$
\begin{array}{l}
BecomeReader \mathrel{\widehat=} \\
\quad writers = \emptyset \land \neg\ (barrier \land waiting)\ \& \\
\quad\quad read\,?\,t : ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \rightarrow \\
\quad\quad\quad readers := readers \cup \{t\}
\end{array}
$$

Only valid task identifiers are candidates for becoming a reader; moreover, if the entry task is waiting, then it cannot also become a reader.

2. When a task issues a procedure call, it may become a writer within the protected object; this is signalled by the communication of the task's identifier over the *write* channel. This event is permitted if there are no readers or writers, and no waiting entry task with an open barrier.

$$
\begin{array}{l}
BecomeWriter \mathrel{\widehat=} \\
\quad readers \cup writers = \emptyset \land \neg\, (barrier \land waiting)\ \& \\
\qquad write\,?\,t : ValidTaskId \setminus (\{entry\_task\} \lhd waiting \rhd \emptyset) \rightarrow \\
\qquad\quad writers := \{t\}
\end{array}
$$

Only valid task identifiers are candidates for becoming a writer; moreover, if the entry task is waiting, then it cannot also become a writer.

3. When the entry task issues the protected entry call, it may become a writer or it may have to wait, depending on the *barrier*. In both cases, there must be no readers or writers, and the entry task must not be already waiting.
   (a) If the barrier is open, then the entry task may enter the object; this is signalled by the event $enter.entry\_task$.

$$
\begin{array}{l}
ETEnter \mathrel{\widehat=} \\
\quad readers \cup writers = \emptyset \land barrier \land \neg\, waiting\ \& \\
\qquad enter.entry\_task \rightarrow \\
\qquad\quad writers := \{entry\_task\}
\end{array}
$$

   The entry task becomes the sole writer.
   (b) If the barrier is closed, then the entry task must wait on the entry queue; this is signalled by the event $wait.entry\_task$.

$$
\begin{array}{l}
ETWait \mathrel{\widehat=} \\
\quad readers \cup writers = \emptyset \land \neg\, barrier \land \neg\, waiting\ \& \\
\qquad wait.entry\_task \rightarrow \\
\qquad\quad waiting := \mathsf{True}
\end{array}
$$

   The next action describes how the waiting entry task can proceed.

4. If the barrier is open, there are no readers or writers, and there is a waiting entry task, then it may become a writer.

$$
\begin{array}{l}
ETStart \mathrel{\widehat=} \\
\quad readers \cup writers = \emptyset \land barrier \land waiting\ \& \\
\qquad start.entry\_task \rightarrow \\
\qquad\quad writers, waiting := \{entry\_task\}, \mathsf{False}
\end{array}
$$

When the waiting task starts, it leaves the entry queue.

5. When an actively reading task completes its function call, it leaves the protected object; this is signalled by the communication of the task's identifier over the *leave* channel.

$$ReaderLeave \mathrel{\widehat=} leave\,?\,t : readers \rightarrow readers := readers \setminus \{t\}$$

6. When an actively writing task completes its procedure or entry call, it also leaves the protected object; this is signalled by the communication of the task's identifier over the *leave* channel.

$$WriterLeave \mathrel{\widehat=} leave\,?\,t : writers \rightarrow update\_bar\,!\,data\,?\,b \rightarrow writers, barrier := \emptyset, b$$

   The barrier may have changed as a result of the actions of the writer, so it must be updated.
7. Any of the tasks currently reading or writing may read the protected data; this is signalled by a communication on the *get* channel.

$$GetData \mathrel{\widehat=} get\,?\,t : (readers \cup writers)\,!\,data \rightarrow Skip$$

   The state invariant ensures that, if there are tasks reading, then there are no tasks writing, and *vice versa*.
8. Any of the tasks currently writing may write to the protected object; this is signalled by a communication on the *put* channel.

$$PutData \mathrel{\widehat=} put\,?\,t : writers\,?\,d : T \rightarrow data := d$$

   The state invariant ensures that, if a task is writing, then it is the sole writer.

The choice between actions (1–8) is offered repeatedly.

$$\begin{aligned} ReactiveBehaviour \mathrel{\widehat=} \mu\, X \bullet (\ & BecomeReader \\ \Box\ & BecomeWriter \\ \Box\ & ETEnter \\ \Box\ & ETWait \\ \Box\ & ETStart \\ \Box\ & ReaderLeave \\ \Box\ & WriterLeave \\ \Box\ & GetData \\ \Box\ & PutData\ );\ X \end{aligned}$$

The extensional behaviour of the process is given by its main action.

$$\bullet\ InitPOState;\ ReactiveBehaviour$$

end

A useful check on the consistency of our model is that the initial state exists.

Theorem 1 (Consistency of protected object initial state).

$$\exists\, POState' \bullet InitPOState$$

Proof *Each state component is fixed by an equality in InitPOState; these expression trivially satisfy POState's invariant (by the one-point rule and properties of propositional calculus and set theory).* □

5 The model exhibits the expected properties

In this section we prove that our *Circus* model of Ravenscar protected objects is free from the risk of deadlock or divergence, and that its state invariants are preserved by its actions. To do this, we define a deadlock-free, livelock-free abstraction that has the same structure, and then prove that our model refines the abstraction.

Our abstract model has the same structure as before. The abstract model has the same state components, with the same types and invariants. It uses the same channels. It has similar actions, except that there are no guards, and state changes are unconstrained, provided the state invariant is maintained. Its main action is the repeated nondeterministic choice between its actions, following the initialisation of the state.

process $APO \mathrel{\widehat=} [\, T \,]$ **begin**

$$
\begin{array}{|l}
\hline
APOState \\
\hline
data : T \\
entry\_task : TaskId \\
barrier, waiting : Boolean \\
readers, writers : \mathbb{F}\ ValidTaskId \\
\hline
readers \neq \emptyset \Rightarrow writers = \emptyset \\
\#writers \leq 1 \\
waiting \Rightarrow entry\_task \neq null\_task \land entry\_task \notin readers \cup writers \\
\hline
\end{array}
$$

$$InitAPOState \mathrel{\widehat=} [\, APOState'; \; d? : T; \; t? : TaskId; \; b? : Boolean \,]$$

$$ABecomeReader \mathrel{\widehat=} \prod t : ValidTaskId \bullet read.t \rightarrow \Delta APOState$$

$$ABecomeWriter \mathrel{\widehat=} \prod t : ValidTaskId \bullet write.t \rightarrow \Delta APOState$$

$$AETEnter \mathrel{\widehat=} \prod t : ValidTaskId \bullet enter.t \rightarrow \Delta APOState$$

$$AETWait \mathrel{\widehat=} \prod t : ValidTaskId \bullet wait.t \rightarrow \Delta APOState$$

$$AETStart \mathrel{\widehat=} \prod t : ValidTaskId \bullet start.t \rightarrow \Delta APOState$$

$$AReaderLeave \mathrel{\widehat=} \prod t : ValidTaskId \bullet leave.t \rightarrow \Delta APOState$$

$$AWriterLeave \mathrel{\widehat=} \prod t : ValidTaskId \bullet leave.t \rightarrow (\prod d : T; \; b : Boolean \bullet update\_bar.d.b \rightarrow \Delta APOState)$$

$$AGetData \mathrel{\widehat=} \prod t : ValidTaskId; \; d : T \bullet get.t.d \rightarrow \Delta APOState$$

$$APutData \mathrel{\widehat=} \prod t : ValidTaskId; \; d : T \bullet put.t.d \rightarrow \Delta APOState$$

$$
\begin{aligned}
AReactiveBehaviour \mathrel{\widehat=} \mu\, X \bullet (\ & ABecomeReader \\
\sqcap\ & ABecomeWriter \\
\sqcap\ & AETEnter \\
\sqcap\ & AETWait \\
\sqcap\ & AETStart \\
\sqcap\ & AReaderLeave \\
\sqcap\ & AWriterLeave \\
\sqcap\ & AGetData \\
\sqcap\ & APutData\ );\ X
\end{aligned}
$$

$\bullet$ *InitAPOState*; *AReactiveBehaviour*

end

Theorem 2 (Abstraction total and non-stopping). *If both ValidTaskId and T are nonempty, then the abstraction APO is both deadlock and livelock-free.*

Proof *There are eight conditions that are* sufficient *for a Circus process to be both deadlock and divergence-free:*

1. *It is sequential.*
2. *It is free from hiding.*
3. *It doesn't mention Stop or Chaos.*
4. *All internal and external choices are over non-empty sets.*
5. *Its channel types are non-empty.*
6. *It local definitions are satisfiable.*
7. *Its main action's initial state exists.*
8. *Its actions are all total on the state.*

Conditions (1)–(3) are satisfied syntactically. Conditions (4) and (5) are guaranteed by the provisos of the theorem. Condition (6) is trivially satisfied, since there are no local definitions. Condition (7) may be stated as

$$\forall\, d? : T;\ t? : TaskId;\ b? : Boolean \bullet \exists\, APOState' \bullet InitAPOState$$

Expanding the schemas, we must prove that

$$
\begin{aligned}
&\forall\, d? : T;\ t? : TaskId;\ b? : Boolean \bullet \\
&\quad \exists\, data' : T;\ entry\_task' : TaskId;\ barrier', waiting' : Boolean; \\
&\qquad readers', writers' : \mathbb{F}\ ValidTaskId \bullet \\
&\qquad\quad (readers' \neq \emptyset \Rightarrow writers' = \emptyset) \land \\
&\qquad\quad \#writers' \leq 1 \land \\
&\qquad\quad (waiting' \Rightarrow entry\_task' \neq null\_task \land entry\_task' \notin readers' \cup writers')
\end{aligned}
$$

which is true, since both T and TaskId are non-empty. Condition (8) follows trivially from the construction of the actions from the total, but arbitrary state change ΔAPOState: all actions have true guards and never abort. □

Thus, if we can prove that PO is a refinement of APO, then we are sure that PO is also deadlock-free and divergence-free. Moreover, the main action of PO shall preserve the state invariants, otherwise the process would not be divergence-free. We state and prove that PO is a refinement of APO in Theorem 3, which will make use of the following three laws.

Law 1 is about the action refinement ($\sqsubseteq_{\mathcal{A}}$, see [7]) of internal choices over a number of prefixed actions. Using this law, the internal choice can be transformed to an external choice over a number of guarded actions.

Law 1 (Refine nondeterministic prefixed actions) *Suppose, for $i \in I$, that c_i is a channel, that S_i and T_i are subsets of the communicable values over c_i, that T_i is non-empty, that A_i and B_i are actions over a common state, that g_i is a boolean-valued expression over the state, and that pre is an assertion about the state.*

$$\{pre\}\ \prod i : I \bullet (\prod x : T_i \bullet c_i.x \rightarrow A_i\ ) \quad \sqsubseteq_{\mathcal{A}} \quad \Box\, i : I \bullet g_i\ \&\ c_i\,?\,x : S_i \rightarrow B_i$$

provided

1. $pre \Rightarrow \bigvee i : I \bullet g_i \wedge S_i \neq \emptyset$
2. $\forall\, i : I \bullet S_i \subseteq T_i$
3. $A_i \sqsubseteq_{\mathcal{A}} B_i$, *for all* $i : I$

There are two sources of nondeterminism in the abstract action: the choice between actions, and the choice between the value communicated; both of these become external choices, with certain alternatives excluded by the introduction of the guard and restricted range of input. The assertion $\{pre\}$ is used to record the abstract action's precondition. □

Law 2 applies to guarded prefixed actions. Simply, the law states that if the action does engage in a communication with its environment, then the guard (g) and the communicated value (x) are in scope for the that part of the action which follows the communication.

Law 2 (Guarded, prefixed action assumption) *Suppose that A is an action, g is a guard over A's state, c is a channel, and S is a subset of c's communicable values.*

$$g\ \&\ c\,?\,x : S \rightarrow A \quad = \quad g\ \&\ c\,?\,x : S \rightarrow \{g \wedge x \in S\}\ A$$

Although state is encapsulated in processes, it is not encapsulated in actions; however, if there are parallel actions, then partitioning the state ensures that the assumption is safe. □

Law 3 states the necessary conditions for the refinement of a schema operation into an assignment statement.

Law 3 (Refine schema action to assignment) *Suppose that Op is a schema action over a state with variables x and w, that e is an expression with the same type as x, and that pre is an assertion over the variables in scope.*

$$Op \quad \sqsubseteq_{\mathcal{A}} \quad \{pre\}\ x := e$$

provided

$$pre \wedge \text{pre}\ Op \Rightarrow Op\,[\,x', w' := e, w\,]$$

The notation $S\,[\,y := f\,]$ denotes the predicate S, with f systematically substituted for y. □

Now, we will show that our model of Ravenscar protected object PO is deadlock-free and divergence-free. To prove these properties for APO it was necessary to have $T \neq \emptyset$ and $ValidTaskId \neq \emptyset$, see Theorem 2 above. For PO, however, we will need a slightly stronger proviso: $T \neq \emptyset$, and $ValidTaskId \setminus \{entry\_task\} \neq \emptyset$. That is, T is not empty, and there exist at least one valid task which is not the entry task.

Theorem 3 (Protected object total and non-stopping). *Provided that $TaskId \setminus \{entry\_task\}$ has at least one element, and that PO is instantiated by a non-empty actual parameter, then PO is deadlock and livelock-free.*

Proof *It is sufficient to show that $APO \sqsubseteq_{\mathcal{P}} PO$. From [8], and since APO and PO have the same state, this refinement holds provided that*

(a) $InitAPOState \sqsubseteq_{\mathcal{A}} InitPOState$
(b) $AReactiveBehaviour \sqsubseteq_{\mathcal{A}} ReactiveBehaviour$

Proviso (a) follows from Theorem 1. We also know that $\sqsubseteq_{\mathcal{A}}$ distributes through recursion. Thus, to prove Proviso (b), it is sufficient to show that

$$(\ ABecomeReader \sqcap \ldots \sqcap APutData\ ) \sqsubseteq_{\mathcal{A}} (\ BecomeReader \,\Box\, \ldots \,\Box\, PutData\ )$$

This, in turn, is a direct consequence of applying Law 1 to the nondeterministic choice over APO actions.
Thus, all we have to do now is prove that provisos 1–3, of Law 1, hold for APO and PO actions.
Provisos 1–2 are proven in Lemma 1, Appendix A
Proviso 3 follows from the following proof obligations:

1. *BecomeReader*

$$[\,\Delta APOState;\ t : ValidTaskId\,] \quad \sqsubseteq_{\mathcal{A}} \quad \left\{ \begin{array}{c} writers = \emptyset \wedge \neg\,(barrier \wedge waiting) \\ t? \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \end{array} \right\} \quad readers := readers \cup \{t\}$$

2. *BecomeWriter*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \left\{ \begin{array}{l} readers \cup writers = \emptyset \land \neg\,(barrier \land waiting) \\ t \in ValidTaskId \setminus (\{entry\_task\} \lhd waiting \rhd \emptyset) \end{array} \right\}\ writers := \{t\}$$

3. *ETEnter*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{readers \cup writers = \emptyset \land barrier \land \neg\ waiting\}\ writers := \{entry\_task\}$$

4. *ETWait*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{readers \cup writers = \emptyset \land \neg\ barrier \land \neg\ waiting\}\ waiting := \mathsf{True}$$

5. *ETStart*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{\, readers \cup writers = \emptyset \land barrier \land waiting \,\}\ writers, waiting := \{entry\_task\}, \mathsf{False}$$

6. *ReaderLeave*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{t \in readers\}\ readers := readers \setminus \{t\}$$

7. *WriterLeave*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{t \in writers\}\ update\_bar\,!\,data\,?\,b \rightarrow writers, barrier := \emptyset, b$$

8. *GetData*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{t \in readers \cup writers\}\ Skip$$

9. *PutData*

$$[\,\Delta APOState;\ t? : ValidTaskId\,] \sqsubseteq_{\mathcal{A}} \{t \in writers\}\ data := d$$

Proof obligation (BecomeReader) follows directly from Lemma 2, Appendix A. The other obligations have similar proofs. □

As a direct consequence of Theorem 3, *PO* must preserve the state invariants; otherwise the process is cannot be divergence-free. Thus, like protected objects, *PO* provides concurrent read-only access to the encapsulated data, and ensures mutual exclusion when that data could be updated. Also, by definition, *PO* complies with restriction R1–R4 in Section 2. Therefore, *PO* satisfies the expected properties of Ravenscar protected objects.

One important remark before finishing that section is that although the statement of Theorem 3 is about reactive behaviour properties, the proof is conducted entirely in Z. This means that Z tools (e.g. [14, 20, 22]), usually used for verifying sequential systems, can be used to provide proofs about concurrent programs as well. Also, we now have one language in which we can reason about both the state-oriented and behavioural properties of a given system. In turn, this means less overhead in reasoning about different properties of the system in different notations.

6 Conclusions and related work

The Ravenscar profile is a restricted tasking model of Ada—designed for verifiability, certifiability, and predictability. The Ravenscar profile provides a shared-variable asynchronous tasking model for communications between tasks. This means that in Ravenscar, protected objects are important as the only mechanism for: encapsulating the data shared between the tasks, granting mutually exclusive access to that data, and providing condition synchronisation between the various tasks. In this report, we provided a formal model, in *Circus*, for Ravenscar protected objects. This is a novel contribution where the functional properties of Ravenscar protected objects have been completely formalised and verified.

Another formal model, presented in UPPAAL, of Ravenscar protected objects is provided in [15]. However, the UPPAAL model is mainly concerned with the timing of calls to protected objects. Also, being based on a model checking approach, the UPPAAL model of protected object was only verified for three tasks; no statement could be made by the authors about the validity of the model for a larger number of tasks. Unlike the work in [15] the proofs about our model are independent of the environment, i.e. the number of calling tasks and the details of which task is a reader and which task is a writer. Indeed, the proof technique presented in this report stand as an interesting result on its own. This is because, despite the fact that some of the properties verified are about the behavioural aspect of the model (e.g., freedom from deadlock), our proof-by-refinement approach enabled us to conduct all the proofs in Z—we believe that this approach can be easily adopted for reasoning about other *Circus* specifications.

The benefits of using Z to conduct proofs about concurrency are manifold. For example, we can hide the complicated details of the UTP semantics of *Circus* [24] away from the program verifier. Also, we can employ current tools for Z (e.g., CADiZ [22, 23], Z-Eves [20], or ProofPower [14]) and use them for reasoning about concurrent programs.

We are currently using the *Circus* model, presented here, as a basis for implementing CSP channels in Ravenscar. This is an essential part of a larger project [2] where we aim at a cost-effective technique for verifying Ravenscar programs against their *Circus* specifications.

7 Acknowledgements

This work is partially supported by the QinetiQ company. Thanks are also due to Alan Burns and Ana Cavalcanti for their insightful comments and useful discussions.

References

1. P. Amey and B. Dobbing. High Integrity Ravenscar. In *8th International Conference on Reliable Software Technologies — Ada-Europe 2003 (AE03)*, Toulouse, France, 2003. To appear.
2. D. M. Atiya and S. King. A compliance notation for verifying concurrent systems. In *ICSE02 – International Conference on Software Engineering*, pages 731–732, Orlando, USA, 2002.
3. J. Barnes. *Programming in Ada 95*. Addison-Wesley, 2nd edition, 1998.
4. A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In L. Asplund, editor, *Ada-Europe 98*, volume 1411 of *Lecture Notes in Computer Science*, pages 263–275. Springer-Verlag, 1998.
5. A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, Department of Computer Science, University of York, January 2003.
6. A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 2nd edition, 1998.
7. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in *Circus*. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002.
8. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *to appear in Formal Aspects of Computing*, 2003.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.
10. C. Fischer. How to Combine Z with a Process Algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *Proceedings of the 11th International Conference of Z Users (ZUM'98)*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23, Germany, 1998. SpringerVerlag.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
12. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall, 1998.
13. ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.
14. Lemma 1 Ltd. *ProofPower Compliance Tool: User Guide*. 2000.
15. K. Lundqvist, L. Asplund, and S. Michell. A Formal Model of the Ada Ravenscar Tasking Profile; Protected Objects. In M. G. Harbour and J. A. de la Puente, editors, *Reliable Software Technologies, Proceedings of the Ada Europe Conference.*, volume 1622 of *Lecture Notes in Computer Science*, pages 12–25, Santander, 1999. Springer-Verlag.
16. MITRE Corporation. Ada Reference Manual, ISO/IEC 8652:1995(E) with Technical Corrigendum 1, 2000.
17. Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.
18. Carroll Morgan. *Programming from Specifications*. Prentice Hall International, 2nd ed. edition, 1994.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice Hall, 1998.
20. M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.

21. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002 — Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.
22. I. Toyn. Formal reasoning in the Z notation using CADiZ. In N. A. Merriam, editor, *2nd International Workshop on User Interface Design for Theorem Proving Systems*, 1996.
23. I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software Practice and Experience*, 25(3):305–330, 1995.
24. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.
25. Jim Woodcock and Jim Davies. *Using Z—Specification, Refinement, and Proof*. Prentice Hall, 1996.

A Lemmas

This appendix presents the lemmas, used inside the proof of Theorem 3, and their proofs.

Lemma 1 (Protected object refinement, Law 1 provisos (1) and (2)). *In applying Law 1 in our proof that APO is refined by PO, provisos (1–2) hold.*

Proof *Proviso (2) is trivially satisfied, since the abstract sets are all types. Proviso (1) requires that at least one branch in PO has a true guard and non-empty range of input. That is,*

$$
\begin{array}{l}
(\ readers = \emptyset \land \neg\ (barrier \land waiting) \land \\
\quad ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \neq \emptyset\ ) \\
\lor \\
(\ readers \cup writers = \emptyset \land \neg\ (barrier \land waiting) \land \\
\quad ValidTaskId \setminus (\{entry\_task\} \lhd waiting \rhd \emptyset) \neq \emptyset\ ) \\
\lor \\
readers \cup writers = \emptyset \land barrier \land \neg\ waiting \land \{entry\_task\} \neq \emptyset \\
\lor \\
readers \cup writers = \emptyset \land \neg\ barrier \land \neg\ waiting \land \{entry\_task\} \neq \emptyset \\
\lor \\
readers \cup writers = \emptyset \land barrier \land waiting \land \{entry\_task\} \neq \emptyset \\
\lor \\
readers \neq \emptyset \\
\lor \\
writers \neq \emptyset \\
\lor \\
readers \cup writers \neq \emptyset \\
\lor \\
writers \neq \emptyset \land T \neq \emptyset
\end{array}
$$

This may be simplified in the propositional calculus to

$$readers \cup writers = \emptyset \land \neg\ barrier \land waiting \Rightarrow \{entry\_task\} \neq ValidTaskId$$

providing that the assumptions of Theorem 3 hold: that $ValidTaskId \setminus \{entry\_task\}$ *and* T *are non-empty. Thus, if no tasks are currently reading or writing, and the barrier is closed, and the entry task is waiting, then PO will deadlock if the entry task is the only valid task. In this case, the entry task can make progress only when the barrier opens; but this depends on another task completing its writing, and there is no other task. Deadlock is avoided if* $ValidTaskId \setminus \{entry\_task\} \neq \emptyset$. □

Lemma 2 (Correctness of action *BecomeReader*). *Applying Law 1 to prove that APO is refined by PO, then the correctness of BecomeReader requires us to prove an instance of proviso (3):*

$$[\, \Delta APOState;\ t? : ValidTaskId \,] \quad \sqsubseteq_{\mathcal{A}} \quad readers := readers \cup \{t\}$$

Proof *Unfortunately, this is simply not true: we cannot prove it, because we have lost the guard and restrictions on* t. *Instead, we must first use Law 2 to preserve this information.*

$$\begin{array}{l} BecomeReader \\ = \{ \text{ by definition } \} \\ writers = \emptyset \land \neg\ (barrier \land waiting)\ \& \\ \quad read\,?\,t : ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \rightarrow \\ \quad\quad readers := readers \cup \{t\} \\ = \{ \text{ by Law 2 } \} \\ writers = \emptyset \land \neg\ (barrier \land waiting)\ \& \\ \quad read\,?\,t : ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \rightarrow \\ \quad\quad \left\{ \begin{array}{c} writers = \emptyset \land \neg\ (barrier \land waiting) \\ t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \end{array} \right\} \\ \quad\quad readers := readers \cup \{t\} \end{array}$$

This means that we should prove that

$$\begin{array}{l} [\, \Delta APOState;\ t? : ValidTaskId \,] \\ \quad \sqsubseteq_{\mathcal{A}} \\ \quad\quad \left\{ \begin{array}{c} writers = \emptyset \land \neg\ (barrier \land waiting) \\ t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \end{array} \right\} \\ \quad\quad readers := readers \cup \{t\} \end{array}$$

Applying Law 3, and noting that $\theta APOState = \theta POState$, *we should prove*

$$\begin{array}{l} writers = \emptyset \land (\neg\ barrier \lor \neg\ waiting) \land \\ t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \land \\ POState \Rightarrow \\ \quad POState' \,[\, readers' := readers \cup \{t\}; \\ \quad\quad\quad data', barrier', waiting', writers' := data, barrier, waiting, writers \,] \end{array}$$

$= \{$ by definition of $POState' \}$

$$
\begin{array}{l}
writers = \emptyset \land (\neg\ barrier \lor \neg\ waiting) \land \\
t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \land \\
POState \Rightarrow \\
\quad (\ readers' \in \mathbb{F}\ ValidTaskId \land writers' \in \mathbb{F}\ ValidTaskId \land \\
\quad\ (readers' \neq \emptyset \Rightarrow writers' = \emptyset) \land \\
\quad\ \#writers' \leq 1 \land \\
\quad\ (waiting' \Rightarrow entry\_task' \neq null\_task \land entry\_task' \notin readers' \cup writers')\ ) \\
\quad [\ readers' := readers \cup \{t\}; \\
\quad\ data', barrier', waiting', writers' := data, barrier, waiting, writers\ ]
\end{array}
$$

$= \{$ by substitution $\}$

$$
\begin{array}{l}
writers = \emptyset \land (\neg\ barrier \lor \neg\ waiting) \land \\
t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \land \\
POState \Rightarrow \\
\quad (\ readers \cup \{t\} \in \mathbb{F}\ ValidTaskId \land writers \in \mathbb{F}\ ValidTaskId \land \\
\quad\ (readers \cup \{t\} \neq \emptyset \Rightarrow writers = \emptyset) \land \\
\quad\ \#writers \leq 1 \land \\
\quad\ (waiting \Rightarrow entry\_task \neq null\_task \land entry\_task \notin readers \cup \{t\} \cup writers)\ )
\end{array}
$$

$= \{$ by assumption and from $POState$, $readers \cup \{t\}$ and $writers \in ValidTaskId \}$

$$
\begin{array}{l}
writers = \emptyset \land (\neg\ barrier \lor \neg\ waiting) \land \\
t \in ValidTaskId \setminus ((\{entry\_task\} \lhd waiting \rhd \emptyset) \cup readers) \land \\
POState \Rightarrow \\
\quad (\ (readers \cup \{t\} \neq \emptyset \Rightarrow writers = \emptyset) \land \\
\quad\ \#writers \leq 1 \land \\
\quad\ (waiting \Rightarrow entry\_task \neq null\_task \land entry\_task \notin readers \cup \{t\} \cup writers)\ )
\end{array}
$$

$= \{$ by the propositional calculus, and using $writers = \emptyset \}$

$$
\begin{array}{l}
writers = \emptyset \land \neg\ barrier \land waiting \land \\
t \in ValidTaskId \setminus (\{entry\_task\} \cup readers) \land \\
POState \Rightarrow \\
\quad entry\_task \neq null\_task \land entry\_task \notin readers \cup \{t\}
\end{array}
$$

$= \{$ by set theory $\}$

$$
\begin{array}{l}
writers = \emptyset \land \neg\ barrier \land waiting \land \\
t \in ValidTaskId \land t \neq entry\_task \land t \notin readers \land \\
POState \Rightarrow \\
\quad entry\_task \neq null\_task \land entry\_task \notin readers \land entry\_task \neq t
\end{array}
$$

The first and second consequents follow from POState's invariant and the antecedent that waiting is true; the third consequent is also an antecedent. □