Extending Ravenscar with CSP Channels

Diyaa-Addein Atiya and Steve King

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
{diyaa, king}@cs.york.ac.uk

Abstract. The Ravenscar Profile is a restricted subset of the Ada tasking model, designed to meet the requirements of producing analysable and deterministic code. A central feature of Ravenscar is the use of protected objects to ensure mutually exclusive access to shared data. This paper uses Ravenscar protected objects to implement CSP channels in Ada – the proposed implementation is formally verified using model checking. The advantage of these *Ravenscar channels* is transforming the data-oriented asynchronous tasking model of Ravenscar into the cleaner message-passing synchronous model of CSP. Thus, formal proofs and techniques for model-checking CSP specifications can be applied to Ravenscar programs. In turn, this increases confidence in these programs and their reliability. Indeed, elsewhere, we use the proposed Ravenscar channels as the basis for a cost-effective technique for verifying concurrent safety-critical system.

1 Introduction

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. Indeed, analysis of programs that make unrestricted use of Ada run-time features like rendezvous, select statements, and abort is currently infeasible [4]. With predictability and verifiability as design objectives, the Ravenscar Profile [3, 4] has been proposed as a greatly simplified subset of the Ada tasking model.

In Ravenscar, there is no task-to-task communication as that of Ada's rendezvous constructs. Instead, data communications between tasks are indirect, through the use of *protected objects*. This makes protected objects central and fundamental building blocks in Ravenscar programs. We use Ravenscar protected objects to implement CSP [6] channels in Ada. This allows us to transform the data-oriented asynchronous tasking model of Ravenscar into the synchronous message-passing model of CSP. The advantages of doing this are manifold. For example, the CSP model eliminates the need for the programmer to worry about synchronization and physical data transfer between communicating tasks; all of these are now embedded into the channel construct. Also, synchronous Ravenscar programs are more amenable to formal proof and model checking. In turn, this contributes to the production of more reliable and trustworthy systems.

The rest of this paper is organised as follows. Section 2 provides a brief account of Ravenscar protected objects. Section 3 presents an implementation

T. Vardanega and A. Wellings (Eds.): Ada-Europe 2005, LNCS 3555, pp. 79–90, 2005.

of CSP channels in Ravenscar. Section 4 describes a CSP model for the Ravenscar implementation. Using that CSP model, Section 5 shows that the implemented Ravenscar channels have the same semantics as CSP channels. Finally, Section 6 draws conclusions and discusses related work.

2 Ravenscar Protected Objects

Simply, Ravenscar protected objects are Ada protected objects with extra restrictions imposed to meet various design requirements such as determinism and schedulability analysis. Thus, as in Ada, a Ravenscar protected object ensures mutually exclusive access to its encapsulated data, through a provided interface of *protected functions*, *protected procedures*, and/or *protected entries*. However, to meet the design requirements, the Profile imposes a number of restrictions on protected objects. Many of these restrictions are syntactic; for example, the Ravenscar Profile does not permit declaration of protected objects local to subprograms, tasks, or other protected objects. The discussion of such restrictions is not relevant to this work; 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 state variable.
R4 As in Ada, potentially blocking operations are not allowed inside the body of a protected object.

An application could further restrict **R2** so that only one task is able to call each protected entry. This paper adopts the stronger version of **R2**, as a static check could be provided for it.

Figure 1 represents a Ravenscar protected object, *Data*, which comprises: a protected entry *Get*, a protected procedure *Put*, and two variables. The first variable, d, represents the encapulated data, which can be of any valid Ada type T. The second variable, *ReadyToRead*, is of type *Boolean* and is used as a barrier for *Get*. The definition of *Data* guarantees that every call to the protected entry *Get* has to be followed by a call to the protected procedure *Put* before the next entry call can execute. This protected object is used as part of the implementation of CSP channels in Ravenscar, see Section 3.

3 Implementing CSP Channels in Ravenscar

In CSP, communication between concurrent processes occurs by passing values on channels. Two types of events can happen on a channel: input and output. An input receives a value from a channel and assigns that value to a variable. An output event, on the other hand, sends out a value to a channel. Respectively, the CSP notation for input and output is $c?x : T$ and $c!v$ – where c is the channel name, x is the variable to which the input value is assigned, v is the

```
protected Data is
   entry Get(var: out T);
   procedure Put(var: in T);
Private
   d : T;
   ReadyToRead : Boolean := False;
end Data;

protected body  Data is
   entry Get(var: out T) when ReadyToRead is
   begin
     var := d;
     ReadyToRead := False;
   end Get;

   procedure Put(var: in T) is
   begin
     d := var;
     ReadyToRead := True;
   end Put;
end Data;
```

Fig. 1. Protected object *Data*

value output through the channel, and T is the type of values communicated on the channel c.

CSP channels are synchronous; that is, both the input and the output processes have to be ready for a communication to proceed, and whoever gets to the communication point first has to wait for the other party. In this section we consider two Ada tasks, *Producer* and *Consumer*, and provide an implementation of a CSP channel for communicating values from *Producer* to *Consumer*.

We need two protected objects:

1. **Data** (Figure 1): encapsulates the data communicated between the tasks, and ensures that every *read* by *Consumer* is followed with a *write* by *Producer*. The protected object comprises a protected entry *Get*, a protected procedure *Put*, and two variables. Only the task *Consumer* can call *Get* and only the task *Producer* can call *Put*.
2. **Sync** (Figure 2): ensures that whoever reads/writes first has to wait for synchronisation with the other party before leaving the channel. The protected object comprises a protected entry *Stay*, a protected procedure *Proceed*, and a boolean variable *HasRead* which is used as a barrier for *Stay*. Only the task *Producer* can call *Stay* and only the task *Consumer* can call *Proceed*.

The two protected objects conform to the restrictions of Ravenscar. However, *Data* and *Sync* can simulate the behaviour of a CSP channel only if calls by *Producer* and *Consumer* are made according to the following protocols. To write

```
protected Sync is
   entry Stay;
   procedure Proceed;
Private
   HasRead : Boolean := False;
end Sync;

protected body Sync is
   entry Stay when HasRead is
   begin
     HasRead := False;
   end Stay;

   procedure Proceed is
   begin
     HasRead := True;
   end Proceed;
end Sync;
```

Fig. 2. Protected object *Sync*

a value v to the channel, the *Producer* task has to make the following two calls: `Data.Put(v); Sync.Stay;`
And, to read a value v from the channel, the *Consumer* task has to make the following two calls: `Data.Get(v); Sync.Proceed;`
The *Read* and *Write* protocols are depicted in Figure 3, and are implemented as the interface procedures to the Ada package representing Ravenscar channels, see [1–Appendix C].

To show how the two protected objects can simulate a CSP channel, consider the initial state. When *Producer* and *Consumer* reach the protected object *Data*, only *Producer* can execute the procedure *Data.Put*(v) – if *Consumer* gets there first, it will wait at the entry barrier until *Producer* finishes writing the data to the protected object. When *Producer* finishes writing the data to *Data* it changes

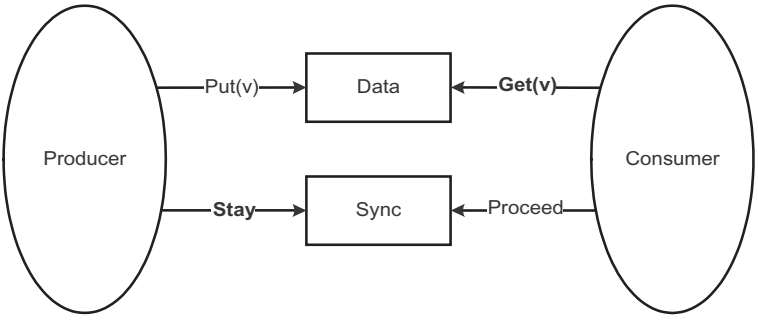

Fig. 3. Write/Read protocols; entry calls are shown in **bold**

the entry barrier *ReadyToRead* into True, giving *Consumer* the chance to proceed and read the data just written. However, leaving *Data*, the *Producer* will wait at the entry barrier of *Sync* until *Consumer* finishes its reading operation. Now, when *Consumer* reaches the protected object *Sync*, it changes the entry barrier *HasRead* to True, enabling *Producer* to proceed. Thus, the two tasks move on to carry their own computations.

4 The Formal Model of Ravenscar Channels

From the discussion above, one can see that the correctness of our implementation depends on which task calls which protected entry/procedure. Nonetheless, the behaviour of the two protected objects and the "read" and "write" protocols is independent of the communicated data, which could be of any type. Thus, the correctness of Ravenscar channels depends on the communicating tasks, and *not* on the communicated data – this remark will become more evident in the formal model presented below. As in occam [7] our implementation provides a One-To-One CSP channel; that is, only two tasks can communicate over a single channel. This small number of tasks suggests that model-checking is a good approach for verification. Therefore, we provide a CSP model of the implementation and verify it using the FDR [5] tool.

4.1 Ravenscar Protected Objects in CSP

Let PO be a Ravenscar protected object that has an entry task[1]. Also, let the tasks communicating on PO be drawn from the non-empty[2] set $ValidTaskId$.

There are six components that determine the state of PO.

- The data encapsulated, $data$, of type T.
- The entry task's identifier, $entry\_task$, which determines the one task that can call the protected entry of the object.
- The current value of the boolean entry barrier, $barrier$.
- A boolean flag ($waiting$) that is true exactly when the entry task is waiting on the entry queue.
- The set of $writers$ ($\subseteq ValidTaskId$), those tasks currently actively executing a procedure or an entry call.
- The set of $readers$ ($\subseteq ValidTaskId$), those tasks currently actively executing a function call.

We model PO as a CSP process with nine channels, each corresponding to some interaction between the protected object and its environment.

[1] Since both *Data* and *Sync* has an entry task, we here limit the CSP model to that restricted form of protected objects. A full formal model of Ravenscar protected objects is presented and verified in [1, 2].

[2] $ValidTaskId$ is not empty since it contains the entry task associated with PO.

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

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 protected 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.

There are nine processes that control the external behaviour of PO.

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

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

   If the entry task is waiting, then it cannot also become a reader[3].
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 writers, and no waiting entry task with an open barrier. Also, if the entry task is waiting, then it cannot become a writer.

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

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 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 $enter.entry\_task$.

[3] $A \lhd B \rhd C$ = *If* B *then* A *else* C.

$$ETEnter(data, entry\_task, barrier, waiting, writers, readers) =$$
$$writers = \emptyset \wedge barrier \wedge \neg\, waiting \;\&$$
$$enter.entry\_task \rightarrow$$
$$PO(data, entry\_task, barrier, waiting, \{entry\_task\}, readers)$$

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 $wait.entry\_task$.

$$ETWait(data, entry\_task, barrier, waiting, writers, readers) =$$
$$writers = \emptyset \wedge \neg\, barrier \wedge \neg\, waiting \;\&$$
$$wait.entry\_task \rightarrow$$
$$PO(data, entry\_task, barrier, \mathsf{True}, writers, readers)$$

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

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

$$ETStart(data, entry\_task, barrier, waiting, writers, readers) =$$
$$writers = \emptyset \wedge barrier \wedge waiting \;\&$$
$$start.entry\_task \rightarrow$$
$$PO(data, entry\_task, barrier, \mathsf{False}, \{entry\_task\}, readers)$$

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

5. When a 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.

$$ReaderLeave(data, entry\_task, barrier, waiting, writers, readers) =$$
$$leave?t : readers \rightarrow$$
$$ReaderLeave(data, entry\_task, barrier, waiting, writers, readers \setminus \{t\})$$

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

$$WriterLeave(data, entry\_task, barrier, waiting, writers, readers) =$$
$$leave?t : writers \rightarrow$$
$$update\_bar.t?b \rightarrow PO(data, entry\_task, b, waiting, \emptyset, readers)$$

The barrier may have changed as a result of the actions of the writer, so it must be updated – the leaving task updates the barrier.

7. Any of the tasks currently reading or writing may read the protected data; this is signalled by a communication on the *get* channel.

$$GetData(data, entry\_task, barrier, waiting, writers, readers) =$$
$$get?t : (readers \cup writers)!data \rightarrow$$
$$GetData(data, entry\_task, barrier, waiting, writers, readers)$$

8. Any of the tasks currently writing may write to the protected object; this is signalled by a communication on the *put* channel.

$$
\begin{aligned}
&PutData(data, entry\_task, barrier, waiting, writers, readers) = \\
&\quad put?t : writers?d : T \rightarrow \\
&\qquad PO(d, entry\_task, barrier, waiting, writers, readers)
\end{aligned}
$$

The process PO repeatedly offers the choice between the above processes.

$$
\begin{aligned}
&PO(data, entry\_task, barrier, waiting, writers, readers) = \\
&\qquad (\ BecomeReader(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ BecomeWriter(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ ETEnter(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ ETWait(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ ETStart(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ ReaderLeave(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ WriterLeave(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ GetData(data, entry\_task, barrier, waiting, writers, readers) \\
&\qquad \Box\ PutData(data, entry\_task, barrier, waiting, writers, readers) \quad)
\end{aligned}
$$

Now, we will use this CSP model of Ravenscar protected objects to provide a formal semantics for Ravenscar channels.

4.2 The Two Protected Objects

The two protected objects can be defined as instantiations of the above PO process through *renaming*. Let the set *channels*, and the two bijective relations D and S be defined as follows:

$$channels ::= read \mid write \mid enter \mid wait \mid start \mid leave \mid update\_bar \mid get \mid put$$

$$
\begin{aligned}
D &= \{x : channels \bullet (x, D\_x)\} \\
S &= \{x : channels \bullet (x, S\_x)\}
\end{aligned}
$$

That is, the relation D (S) add the suffix $D\_$ ($S\_$) to each channel of the process PO. Now, using the CSP renaming operator, the protected objects *Data* and *Sync* can be defined as:

$$
\begin{aligned}
Data &= PO[\![D]\!] \\
Sync &= PO[\![S]\!]
\end{aligned}
$$

Data is the process that can perform the event $D(e)$ whenever PO can perform the event e. Similarly, *Sync* is the one that can perform the event $S(e)$ whenever PO can perform the event e. Since they are both defined in terms of the renaming operator, both *Data* and *Sync* are guaranteed to preserve the properties of Ravenscar protected object exhibited by PO. In particular, the behaviour of *Data* and *Sync* is independent of the data they encapsulate.

4.3 The Protocols

As well as the two protected objects, the implementation of the Ravenscar channel requires two protocols to regulate how the tasks can write to or read from the protected objects. To write a *value* to the channel, the *Producer* task has to make the following two calls: `Data.Put(v); Sync.Stay;`
We model this protocol by the following CSP process, $REC(\_)$.

$$\textbf{channel}\ obtain : ValidTaskId \times T;\ ack : ValidTaskId$$
$$REC(t) = obtain.t?value \rightarrow Write(t, value);\ ack.t \rightarrow REC(t)$$
$$Write(t, value) = PutData(t, value);\ WSynchronise(t)$$

That is, the process $REC(\_)$ repeatedly waits to receive an event (comprising the identification of writing task and the value to be written) on the channel *obtain*. When the value to be written is obtained, the process then executes the $Write(\_,\_)$ protocol to write the received value to the Ravenscar channel. Finally, a successful *Write* is acknowledged by an event on the channel *ack*.

The *PutData* process stands for the call *Data.Put*. First, the task has to gain a write access to the protected object *Data*. Then, the task executes the procedure *Put*; this is signalled by communicating the task's identifier and the *value* over the channel *D_put*. Finally, the task leaves the protected object, updating the entry barrier as it leaves.

$$PutData(t, value) =$$
$$D\_write!t \rightarrow D\_put!t!value \rightarrow D\_leave!t \rightarrow D\_update\_bar.t!\mathsf{True} \rightarrow Skip$$

The *WSynchronise* process stands for the call *Sync.Stay*. First, the task has to gain write access as the entry task of the protected object *Sync*. Then, the task executes the entry *Stay*; this is signalled by communicating the task's identifier over the channel *S_put*. Finally, the task leaves the protected object, updating the entry barrier as it leaves.

$$WSynchronise(t) =$$
$$(\ S\_enter.t \rightarrow Skip\ \Box\ S\_wait.t \rightarrow S\_start.t \rightarrow Skip\ );$$
$$S\_put!t \rightarrow S\_leave!t \rightarrow S\_update\_bar.t!\mathsf{False} \rightarrow Skip$$

An important remark here is that the communicated *value* does not determine the subsequent behaviour of the $REC(\_)$ process. The reading protocol, $SEND(\_)$, is defined similarly [1–Appendix C]. As $REC(\_)$, the behaviour of $SEND(\_)$ is independent of the communicated data.

4.4 Ravenscar Channels

The process representing a Ravenscar One-To-One channel is the parallel composition of the two protected objects, the write protocol, and the read protocol.

$$\begin{aligned} RavenChannel(t_1, t_2) = {} & REC(t_1) \\ & [\![WriterEvents]\!] \\ & (\ Data(0, t_2, \mathsf{False}, \mathsf{False}, \emptyset, \emptyset)\ |||\ \\ & \ \ Sync(\mathsf{False}, t_1, \mathsf{False}, \mathsf{False}, \emptyset, \emptyset)\ ) \\ & [\![ReaderEvents]\!]\ SEND(t_2) \end{aligned}$$

The two protected objects run independently of each other, hence the interleaving operator. The protocols for writing and reading synchronise with the two protected objects on the events described by the sets *WriterEvents* and *ReaderEvents*, respectively.

$$\begin{aligned} \mathit{WriterEvents} = \{\!| \; & \mathit{D\_read}.t_1, \mathit{D\_write}.t_1, \mathit{D\_enter}.t_1, \mathit{D\_wait}.t_1, \mathit{D\_start}.t_1, \\ & \mathit{D\_leave}.t_1, \mathit{D\_update\_bar}.t_1.\mathit{true}, \mathit{D\_put}.t_1, \mathit{D\_get}.t_1, \\ & \mathit{S\_read}.t_1, \mathit{S\_write}.t_1, \mathit{S\_enter}.t_1, \mathit{S\_wait}.t_1, \mathit{S\_start}.t_1, \\ & \mathit{S\_leave}.t_1, \mathit{S\_update\_bar}.t_1.\mathit{false}, \mathit{S\_put}.t_1, \mathit{S\_get}.t_1 \; |\!\} \end{aligned}$$

$$\begin{aligned} \mathit{ReaderEvents} = \{\!| \; & \mathit{D\_read}.t_2, \mathit{D\_write}.t_2, \mathit{D\_enter}.t_2, \mathit{D\_wait}.t_2, \mathit{D\_start}.t_2, \\ & \mathit{D\_leave}.t_2, \mathit{D\_update\_bar}.t_2.\mathit{false}, \mathit{D\_put}.t_2, \mathit{D\_get}.t_2, \\ & \mathit{S\_read}.t_2, \mathit{S\_write}.t_2, \mathit{S\_enter}.t_2, \mathit{S\_wait}.t_2, \mathit{S\_start}.t_2, \\ & \mathit{S\_leave}.t_2, \mathit{S\_update\_bar}.t_2.\mathit{true}, \mathit{S\_put}.t_2, \mathit{S\_get}.t_2 \; |\!\} \end{aligned}$$

The behaviour of *Data*, *Sync*, *REC*, and *SEND* does not depend on the data communicated. Thus, the behaviour of $\mathit{RavenChannel}(\_)$ is also independent of the communicated data.

5 Correctness of Ravenscar Channels

Consider a CSP network of parallel processes ($\mathit{Net} = P_1 \parallel P_2 \parallel .. \parallel P_n$). If our Ravenscar channel is correct, we should be able to replace all CSP channels in *Net* with the *RavenChannel* processes, without affecting the external behaviour of the network. We will show that this is possible if *Net* satisfies the following two conditions:

1. *Net* is free from the external choice operator ($\Box$).
2. The channels used by *Net* are One-To-One; that is, each channel c is used by exactly one process P_i for input and exactly one process P_j $(i \neq j)$ for output.

Consider the following CSP process

channel *transmit* : T

$$\mathit{Left}(t) = \mathit{obtain}.t?\mathit{value} \rightarrow \mathit{transmit}!\mathit{value} \rightarrow \mathit{ack}.t \rightarrow \mathit{Left}(t)$$
$$\mathit{Right}(t) = \mathit{ready}.t \rightarrow \mathit{transmit}?\mathit{value} \rightarrow \mathit{deliver}.t!\mathit{value} \rightarrow \mathit{Right}(t)$$

$$\mathit{JCSPCHANNEL}(t_1, t_2) = \mathit{Left}(t_1) \; |[\, \{\!|\, \mathit{transmit} \,|\!\}\,]| \; \mathit{Right}(t_2) \setminus \{\!|\, \mathit{transmit} \,|\!\}$$

In their work [8] on implementing CSP channels for Java, P. Welsh and J. Martin have given a proof-by-hand that *JCSPCHANNEL*, shown in figure 4, can be used as CSP channels for any network *Net* satisfying the two conditions above. The replacement of the CSP channels with the *JCSPCHANNEL* happens by transforming each process P_i as follows:

- replace all occurrences of "$c!x \rightarrow$" by "$\mathit{obtain}.i!x \rightarrow \mathit{ack}.i \rightarrow$"
- replace all occurrences of "$c?x \rightarrow$" by "$\mathit{ready}.i \rightarrow \mathit{deliver}?x \rightarrow$"

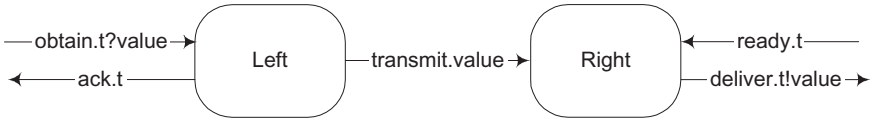

Fig. 4. JCSP Channel

This result is of special interest to us as now the problem of proving the correctness of Ravenscar channel can be reduced to proving that our process *RavenChannel* is equivalent to *JCSPCHANNEL*.

Theorem 1 (Implementation is Correct). *Let* $Net = P_1 \parallel P_2 \parallel .. \parallel P_n$ *be a network of parallel processes. Assume that Net satisfies the following two conditions:*

- *Net is free from the external choice operator* ($\Box$).
- *Each channel c in Net is One-To-One, i.e. c is used by exactly one process* P_i *for input and exactly one process* P_j $(i \neq j)$ *for output.*

Then, we can replace the channels in Net with RavenChannel processes while preserving the external behaviour of Net.

Proof. *It is sufficient to prove that the processes JCSPCHANNEL and RavenChannel are equivalent. We used FDR to prove this equivalence and successfully discharged the two assertions:*

1. $SimpleRavenChannel(prod, cons) \sqsubseteq JCSPCHANNEL(prod, cons)$
2. $JCSPCHANNEL(prod, cons) \sqsubseteq SimpleRavenChannel(prod, cons)$

Where

$$ValidTaskId = \{prod, cons\}$$
$$SimpleRavenChannel(t_1, t_2) = RavenChannel(t_1, t_2) \setminus Internal$$
$$Internal = \{| \; D\_read, D\_write, D\_enter, D\_wait, D\_start, D\_leave, D\_update\_bar, D\_put, D\_get, S\_read, S\_write, S\_enter, S\_wait, S\_start, S\_leave, S\_update\_bar, S\_put, S\_get \; |\}$$

□

Actually, this proof-by-equivalence approach gives us more than just the correctness of our implementation. Since *SimpleRavenChannel* and *JCSP CHANNEL* are equivalent, we know that our channel implementation inherits all the properties satisfied by the Java implementation of One-to-One CSP channels, as described in [8]. For example, as in Java channels, we can tell that our Ravenscar implementation works fine as long as there are at most two concurrent threads in existence (one is writing and one is reading). Indeed, a simple check with FDR (increasing the number of elements in ValidTaskId beyond two) reveals that *RavenChannel* can deadlock. This is a positive result in its own right, as now the equivalence between *SimpleRavenChannel* and *JCSPCHANNEL* not only increases the confidence about our implementation but also informs us about possible limitations.

6 Conclusions

The tasking model of Ravenscar is asynchronous. Unfortunately, this means that Ravenscar programs do not lend themselves nicely to verification techniques like model checking. In this paper we have implemented One-to-One CSP channels in Ravenscar. As a consequence, we can now transform the asynchronous tasking model of Ravenscar into the synchronous message passing model of CSP.

Like in CSP, using channels in Ravenscar programs eliminates the need to worry about issues of synchronisation and physical transfer of data between tasks. Also, synchronous Ravenscar programs are more amenable to formal proofs and techniques for model checking. For example, we can now use tools like FDR to check Ravenscar programs for properties like deadlock/livelock freedom. This all contributes to the production of more reliable and trustworthy systems.

To verify our implementation, we showed that the CSP semantics of our Ravenscar channels is equivalent to the semantics of the one-to-one channels in the JCSP library [8] for Java. This is a valuable result, as it allows arguments of correctness and proofs about properties of JCSP channels to be automatically deployed in favour of our Ravenscar channels.

Elsewhere, we have used Ravenscar channels as a key element in developing a cost-effective technique for verifying Ravenscar programs: more details of that work are available in [1].

References

1. D. Atiya. *Verification of Concurrent Safety–critical Systems: The Compliance Notation Approach.* PhD thesis, University of York. Submitted in October 2004.
2. D. M. Atiya, S. King, and J. C. P. Woodcock. A *Circus* semantics for Ravenscar protected objects. In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 617–635. Springer-Verlag, 2003.
3. 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.
4. 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, UK, January 2003.
5. Formal Systems (Europe) Ltd. *Failures-divergences refinement: FDR2 user manual.* May, 2000.
6. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, 1998.
7. SGS-THOMSON Microelectronics Limited. *occam 2.1 reference manual.* May, 1995.
8. P. H. Welch and J. M. R. Martin. A CSP Model for Java Multithreading. In P. Nixon and I. Ritchie, editors, *Software Engineering for Parallel and Distributed Systems*, pages 114–122. ICSE 2000, IEEE Computer Society Press, June 2000.