# An Approach to Formal Verification of Real Time Concurrent Ada Programs

Douglas J. Howe
School of Computer Science
Carleton University, Ottawa, Ontario, Canada
howe@scs.carleton.ca

Stephen Michell
National Research Council
Sydney, Nova Scotia, Canada
stephen.michell@nrc.gc.ca

## Abstract

*The SPARK system provides static analysis tools for a highly restricted sequential Ada subset, including a proof checking tool for verifying partial correctness properties. Recently, SPARK Ada has been extended to include much of the Ravenscar Tasking Profile which supports construction of high integrity real time systems. However, the verification machinery has not been changed, and can only handle purely sequential properties of the code. This paper sketches an approach to reasoning about the concurrent and real-time aspects that SPARK cannot handle. The approach involves compiling an abstract model of the Ada program that can be embedded in a general purpose theorem prover (e.g. PVS). The compilation makes heavy use of SPARK's existing static analysis tools.*

## 1 Introduction

The development of high performance real time systems, especially high integrity real time systems, has for a long time stretched the ability to adequately analyze the code to guarantee certain behaviours, such as proper timing, proper scheduling of work, and lack of deadlock. This analysis places a significant burden on these programs, resulting in heavy restrictions on the languages used, language constructs used, coding techniques, and designs.

Formal (mathematical) analysis of static program code is an approach to analyze such programs. Tools exist to reason statically about the dynamic behaviour of such code in either the sequential domain or in the concurrent domain, but very few tools exist to reason about the concurrent behaviour of a complete program and the sequential correctness of parts of the program together.

Recent developments in high integrity Ada and the development of formal verification tools which understand the syntax and semantics of concurrent programs make it feasible to do formal verification of actual concurrent real time Ada programs. This paper discusses one approach.

## 2 Background

There have been a number of developments in recent years to improve the ability to design and analyze time critical real-time systems and high integrity real time systems.

For more than a decade there have been subsets of Ada to support the development and formal analysis of sequential Ada code [6, 8]. The most successful of these is the SPARK [15] language, a heavily restricted sequential subset of Ada [1] with sufficient analysis tools to do a reasonable job of statically verifying some classes of programs.

Through the late 1980's and 1990's, work was done to show how preemptive priority concurrent systems could be used to satisfy a large classes of real-time problems using Rate Monotonic Scheduling [14] or other forms, but at that time there were no runtime systems available to permit developers to successfully use these techniques.

The Ravenscar Tasking Profile is a subset of Ada's tasking runtime and was designed at IRTAW 8 [12, 3] to support the construction of concurrent programs. The Ravenscar Tasking Profile permits preemptive, priority-based task scheduling, protected operations with subprograms and entries, delay until statements, and interrupts, but restricts each capability so that all scheduling choices made are predictable. IRTAW 8 speculated that programs written in Ravenscar could be formally verified.

The reasons for the formal analysis of a program are well documented, and there have been a number of attempts to formally verify sequential Ada programs [8, 6] and Concurrent Ada programs [11, 10, 9]. To date, these analyses either tackle the sequential portions of the code, subprograms, data elements, preconditions, postconditions, invariance; or the analyses tackle the concurrent portions to show correct scheduling, produce timing and load budgets, and analyze priority inversions.

Neither of these analyses (sequential and concurrent) are sufficient since there are many ways in which the correctness of the sequential code depends upon the concurrent behaviour, and that the correct order of execution depends upon sequential elements. For example,

- Sequential code relying on concurrent behaviour — sequential code which is called by a task is not correct if the task does not have sufficient resources to execute, or a read on a shared variable is corrupt if a concurrent task updates that variable halfway through the read, and
- Concurrent behaviour relying upon sequential constructs — protected entry guards control the release of suspended tasks but are set and cleared by sequential code fragments; no analysis of correct task-related behaviour could be complete without consideration of the sequential code which manipulates these guards.

Much of the difficulty joining the verification of sequential properties of code and the verification of concurrent properties of code lies in the type of relationships which are being expressed, and the places where these properties can be expressed. In the Ada/Ravenscar community there have been model checkers and theorem provers where one could specify the global characteristics of a program (task structure, protected objects and interrupts), and there have been sequential proof systems which let one reason about the correctness of subprograms and collections of subprograms. In order to move toward an integrated verification of a concurrent program, either sequential reasoning need to be added to concurrent proof systems, more generalized reasoning added to sequential provers, or a way to integrate sequential and concurrent reasoning in a unified set of tools.

# 3 Recent Developments

In March 2003, Praxis Critical Systems released a version of the SPARK Examiner Toolkit which includes some support for the Ravenscar Tasking Profile. This support includes

- Recognition of tasking related syntax, such as tasks, protected types and objects, protected entries, delay statements and time as defined in package Ada.Real_Time, suspension objects, infinite loops.
- Extended syntax to inform the analysis tools about protected state (in the sense of variables which can only be accessed through mechanisms which enforce sequential update for read/write access), about how this protected stated is accessed and used, about interrupt-based protected objects, and about suspension of tasks due to protected entries or suspension objects.
- Extension of the formal verification tool to permit reasoning about the sequential portions of protected objects, such as preconditions and postconditions of protected subprogram bodies to permit formal verification of the linear code in a protected subprograms.

SPARK with Ravenscar Profile successfully handles many of the formal verification issues in sequential code, and recognizes the syntax of concurrent programs, but does not attempt to deal with the issues around concurrency, such as order of access to protected objects, preemption, timing, and resource usage. It does, however, provide a framework where such reasoning could be connected to the sequential analysis and a formal verification system of the complete system could be constructed.

In concert with this, Burns and Lin [4] propose extensions to SPARK to perform timing analysis of concurrent Ada programs using a rely-guarantee paradigm of annotations. This approach is interesting, but we wish to verify more general properties of a concurrent program, such as progress, correctness of data transformations, correct order of access or update of data, and, of course, timing properties. It would be difficult to add two paradigms to a system such as SPARK, one for timing and one for other formal analysis.

## 3.1 Simplification for Analysis

Concurrent programs can be exceedlingly complicated to reason about because of the number of possible interleavings of concurrent execution. The restrictions of Ravenscar reduce this complexity somewhat, and the SPARK restrictions reduce it further. The localization of variables to internal task variables and to protected objects means that all interleavings which leave the state of variables unchanged are irrelevant, from the point of view of invariance between variables. For example, preemptions of a task cannot change internal state, hence are irrelevant to that state; and only preemptions which cause shared protected objects to be updated are relevant to the state of a protected object or to tasks which consume such state.

Hard real time systems or high integrity real time systems restrict programs further, following design patterns to help with their own analysis, ensuring that the scheduling rate of each task is known, that cpu usage for each task and each protected subprogram is bounded, and that access to common objects (shared variables) is carefully controlled and usually sequentialized.

The complexity is still significant, but overall behaviour can be stated in a tractable set of formal statements and analyzed. In the following section we discuss a method to integrate sequential and concurrent reasoning about such programs.

```
with Ada;
--# inherit Ada, PC;
package PC.Buffer_Pkg
  --# protected buffer : Buffer_Type(priority => 22);
is
  protected type Buffer_Type
  is
    pragma Priority (22);
    procedure Read( D : out PC.Data_Type );
    --# post D.Value > 17;
    procedure Write( D : in PC.Data_Type );
    --# pre  D.Value > 17;
  private
    Data : PC.Data_Type :=
           PC.Data_Type'(Value => 0, Count => 0);
    Read_Count : Integer := PC.Number_Consumers;
  end Buffer_Type;
  Buffer : Buffer_Type;
end PC.Buffer_Pkg;
```

**Figure 1. Buffer package spec**

```
package body PC.Buffer_Pkg
is
  protected body Buffer_Type
  is
    procedure  Read(  D :    out PC.Data_Type )
    --# post  D.value = Data.Value and
    --#       D.Count = Data.Count and D.Value > 17;
    is
    begin
        D.Value  := Data.Value;
        D.Count  := Data.Count;
        Read_Count := Read_Count-1;
    end Read;
    procedure Write( D : in     PC.Data_Type )
    --# pre     D.Value > 17 ;
    --# post    D.Value = Data.Value  and
    --#         D.Count = Data.Count and Data.Value > 17;
    is
    begin
       Data.Value  := D.Value;
       Data.Count  := D.Count;
       Read_Count := PC.Number_Consumers;
    end Write;
  end Buffer_Type;
end PC.Buffer_Pkg;
```

**Figure 2. Buffer package body**

# 4 The Verification Approach

## 4.1 Verifying Sequential Code in Spark

The Spark tool supports formal proof of *partial correctness* properties of Spark Ada code. Spark Ada code can be annotated with correctness assertions written in the sorted first-order logic *FDL*. Spark will generate FDL *verification conditions* whose truth implies the correctness of the annotations. Spark provides tools to formally prove the verification conditions.

There are essentially two kinds of correctness annotations in Spark. One kind is for specifying procedures (there are also analogous annotations for functions, but we will ignore them in this paper). A procedure can be given a *pre-condition* and a *post-condition*. The procedure is *partially correct* with respect to these conditions if whenever the procedure starts in a state satisfying the pre-condition and executes to completion, then the resulting state will satisfy the post-condition. The correctness is "partial" because the post-condition says nothing about the result of execution in the case where the procedure fails to terminate normally (e.g. executing an infinite loop or raising an exception).

Placing pre- and post- conditions on a procedure entails constraints on the procedure implementation and on the procedure use. Verification conditions must guarantee that the body of the procedure meets the specification given by the pre- and post-conditions, and also that the precondition is met every time the procedure is called.

The second kind of annotation are assertions placed in executable code. These annotations require that the assertion be true whenever execution reaches that point in the code.

Verification condition generation is based on the well-known *Floyd-Hoare* technique. The executable parts of the program are decomposed into a set of paths where each path essentially consists only of assignment statements and procedure calls. The paths correspond to segments of straightline code between *cutpoints*, which include procedure body beginnings and endings and user-specified *assertions* in loops. Conditional statements (if-then-else) give rise to multiple paths between cutpoints.

Each end of a path has a condition associated with it. For example, the path starting at the beginning of the body of a procedure $P$ and ending at the call of a procedure $Q$ will associate $P$'s pre-condition with the beginning of the path, and $Q$'s precondition with the end. In general, it is required that if the beginning condition is true in some state, and the code in the path is executed, then the condition at the end is true. This requirement is transformed into a logical statement in FDL by computing the *weakest precondition* of the path with respect to the end condition, which is an FDL expression that characterizes all states where executing the path code will end up in a state satisfying the end condition. The verification condition for this path is then the logical statement that the beginning condition implies this weakest precondition.

In order for this path analysis to work, loops must have cutpoints given by user-specified assertions. The assertion becomes a *loop invariant*. In the simplest kinds of loops, there will be a path from the cutpoint back to the cutpoint, and the beginning and ending conditions will both be the loop invariant.

## 4.2 Sequential Verification in the Ravenscar Profile

Spark Ada has been extended to include the real-time features of Ravenscar. However, the verification machinery

has not been changed. As the current machinery can only handle sequential code, care has been taken not to allow annotations that would require reasoning about concurrent aspects of the program.

Consider the example Ravenspark package in Figures 1 and 2. This package implements a simple protected object for a buffer and is part of an example concurrent program with a single data producer and several data consumers. As can be seen in the Figures, a buffer is implemented as two variables, one containing the data, and the other a count of how many readers have not yet read the current value. The protected object has two procedures, one which writes data to the buffer, initializing the count, and the other which reads the buffer, decrementing the count. For the purposes of this example, we take as a data-type invariant that the `value` field is always greater than *17*. The Figures show pre- and post-conditions, but omit dataflow annotations.

The key thing to observe about this example is the difference in annotations between the package specification and the package body. In the body, the postconditions specify what effect the procedures have on the variable `Data`. Verification conditions will be generated to show the procedure bodies meet these conditions.

However, the post-conditions in the specification do not say anything at about the resulting buffer values. This is because these values may be modified by concurrently running tasks. If a postcondition, say of `write`, specified a property of the buffer, then the verification condition generator would make use of the assumption that after `write` is called in a task, the buffer property holds. This may be false, since there may be an intervening modification of the buffer by a concurrently executing task.

Thus in Ravenspark, postconditions (and preconditions) of specifications of operations on concurrently accessed data cannot mention that data. In our example (Figures 1 and 2) we see that the conditions only mention the procedure parameters.

### 4.3 Verification of Concurrent and Real-Time Properties

Ravenspark can be used to do useful kinds of flow analysis of Ravenscar programs, and we believe that the existing sequential verification machinery can be used to prove certain useful properties of the program, e.g. local data type invariants. However, properties involving concurrency cannot be handled at all. In the remainder of this section we sketch our approach to solving this problem.

Our approach is to compile an abstract state-transition system from Ravenspark programs and to use an external theorem-proving tool to specify and reason about the concurrent and real-time aspects of the program. Currently we are using PVS [13] as the external tool for reasoning about these state transition systems. However, there are a number of other tools we could use for this purpose: any general-purpose theorem-proving system would be possible. We chose PVS because of the extensive support for practical formal reasoning about models such as ours.

The states in this new system represent the state of the executing Ravenspark program, and include all values of variables (both local and global), an indication of the next instruction to execute in each of the tasks and interrupt handlers in the program, and the values of any clocks. Formalizing the set of states of a Ravenspark program in PVS is straightforward.

Formalizing the transitions in PVS is not as easy. The transitions need to account for the purely local action of sequential code within tasks, and also the effect on global data of operations on protected objects. Correspondingly, there are two kinds of transitions we need to deal with. One kind corresponds to the segments of sequential code like the program paths produced by the path analysis described above, except that in addition to the cutpoints described above, we also need to consider protected object operations to act as additional, "virtual" cutpoints. The other kind of transition is the action performed by the operations on protected objects. In either case, a transition $t$ can be modeled as the set $S_t$ of all pairs of states $(s, s')$ such that executing the sequential code associated with $t$ starting in state $s$ ends up in state $s'$.

To formalize the sets $S_t$ in PVS, we need some formula to represent them, e.g. a predicate $P_t(s, s')$ which is true if and only if $(s, s') \in S_t$. If we can find an FDL expression for $P_t$ we are done because FDL is a simple first order logic that can be cast as a subtheory of the logics of most general purpose theorem provers, PVS included.

In the case of the second kind of transition, a logical characterization of the transition is already given to us: it is the pre- and post-conditions given in the package *body* for the protected object, as in Figure 2. Ravenspark cannot "export" these conditions from the package, but this restriction is unnecessary in our setting.

Representing the first kind of transition, however, requires some effort. This is because the sets $S_t$ are defined in terms of the execution of the fragments of sequential Ada code determined by the cutpoint and virtual cutpoints. Of course, we would like to avoid duplicating the work that Spark does in its path analysis. Fortunately, there is a fairly easy way to get a logical characterization of these code fragments.

Write the code fragment as $C = c_1; c_2; \ldots; c_k$ for atomic commands $c_i$, and let $\overline{x} = x_1, \ldots, x_n$ be the local variables occurring in the task containing the code fragment. For simplicity, assume the commands $c_i$ are just assignments (i.e. not procedure calls). If we insert a "dummy" assertion $P(\overline{x})$ in the Ravenspark code in a position corresponding to the

end of the fragment, then Spark will generate weakest preconditions, for paths ending at the assertion, roughly of the form

$$(b_1[\overline{x}] \wedge \ldots \wedge b_m[\overline{x}]) \Rightarrow P(e_1[\overline{x}], \ldots, e_n[\overline{x}])$$

for some FDL expressions $b_i[\overline{x}]$ and $e_j[\overline{x}]$.[1]. The expressions $b_i[\overline{x}]$ correspond to the conditions of conditional statements on the path. From this weakest precondition we can read off a logical specification of a transition: in states where the values $\overline{u}$ of the variables $\overline{x}$ satisfy each $b_i[\overline{u}]$, there is a transition to a state where each $x_i$ has the value $e_i[\overline{u}]$.

To help see this, take

$$P[\overline{x}] \equiv c_1 = x_1 \wedge \ldots \wedge c_n = x_n$$

for some new constants $c_i$. By definition of weakest precondition, if

$$(b_1[\overline{x}] \wedge \ldots \wedge b_m[\overline{x}]) \Rightarrow P(e_1[\overline{x}], \ldots, e_n[\overline{x}])$$

is true in a state where $\overline{x} = \overline{u}$, then after executing the code fragment $C$, if we let $\overline{v}$ be the new values of the variables $\overline{x}$, then we will have $P(\overline{v})$. Substituting, we have that if

$$(b_1[\overline{u}] \wedge \ldots \wedge b_m[\overline{u}]) \Rightarrow c_1 = e_1[\overline{u}] \wedge \ldots, c_n = e_n[\overline{u}]$$

then

$$c_1 = v_1 \wedge \ldots \wedge c_n = v_n.$$

If we know that $b_i[\overline{u}]$ is true for each $i$, then the previous statement is equivalent to

$$v_1 = e_1[\overline{u}] \wedge \ldots \wedge v_n = e_n[\overline{u}].$$

Sequential properties of the program that are proved in Spark can be incorporated into the state transition model. In particular, in any sequence of legal transitions that starts from a valid initial state, every time a transition is made to a cutpoint (actual, not virtual), the property that was proved in Spark to hold at that point can be taken to hold in the state transition model as well.

The state-transition model, once formalized in PVS, provides a general semantic account of the concurrent real-time behaviour of the Ravenspark program. Because of this, it is straightforward to formalize virtually any program specification one might have. What is unknown at this point, however, is how feasible it will be to formally prove that the system meets the specification. We are currently working on implementing this approach.

[1] We use the notation $e[\overline{x}]$ to indicate that the expressions free variables are contained in $\overline{x}$

## 5 Conclusions and Future Work

We have shown an approach which we expect will lead to the combined formal sequential and concurrent verification of real time programs. Work is underway to implement the approach and then apply it in some case studies.

The framework provided by SPARK Ravenscar profile provides a significant amount of the infrastructure needed to combine sequential and concurrent analysis of Ravenscar compliant programs. More work is required, however. This work includes ways to extend SPARK annotations to improve reasoning about sequential code, ways to improve the integration of sequential and concurrent properties of the system, and ways to help developers improve design and implementation to support formal analysis of their systems.

*Invariance.* At present, SPARK does not have a notion of invariance. Invariance is very useful for reasoning about package state but is fundamental in reasoning about concurrency. Syntax for invariance in SPARK would be very useful to aid in capturing knowledge about program state to extract into the concurrent reasoning framework.

*Applying other tools.* As mentioned above, the state-transition model compiled from a SPARK program is suitable for formalizing in a variety of theorem proving tools. It may also be possible to adapt the model to a form suitable for input to a model-checking tool (e.g. [10]).

*Programming Patterns.* Research is needed into the ways that protected objects are used to collect and manage shared state and the ways that this affects performance and reasoning about invariance. A single protected object containing all state of significance makes analysis straightforward, but has a high cost in terms of the effects on the sequentialization of code, number of protected operations needed, priority, and effects on modularity. On the other hand, distribution of program state in many protected objects improves the system from the software engineering and concurrency perspectives but increases the number of paths where invariants must be reasoned about. More work is needed to develop patterns which provide adequate modularity and support for concurrency but permit effective verification.

## References


[1] *The Consolidated Ada Reference Manual*. Springer-Verlag, Berlin, 2001. LNCS 2219.

[2] Amey P. and Dobbings B. High Integrity Ravenscar, in *Proceedings of Reliable Software Technologies - Ada Europe 2003*. Springer-Verlag, Berlin, 2003. LNCS 2655.

[3] Burns J. The Ravenscar Tasking Profile, in *Ada Letters*. ACM Press, New York, December 1999.

[4] Burns A. and Lun T. Adding Temporal Annotations and Associated Annotations to the Ravenscar Profile, in *Proceedings of Reliable Software Technologies - Ada Europe 2003*. Springer-Verlag, Berlin, 2003. LNCS 2655.

[5] Bruneton E. and Pradat-Peyre J.F. Automated Verification of Concurrent Ada Programs, in *Proceedings of Reliable Software Technologies - Ada Europe '99*. Springer-Verlag, Berlin, 1999. LNCS 1622.

[6] Carre B. and Jennings T.J. The Spade Ada Kernel, University of Southhampton, 1988.

[7] Gonzalez A. and Crespo A. Environment for the Development and Specification of Real-Time Ada Programs, *Proceedings of Reliable Software Technologies - Ada Europe '99*. Springer-Verlag, Berlin, 1999. LNCS 1622.

[8] Guaspari, Marceau and Polak. The Formal Verification of Ada Programs", in *IEEE Transactions on Software Engineering*. IEEE Publishing, New York, Sept 1990.

[9] Fowler S. and Wellings A. Formal Development of a Real-Time Kernel", in *Proceedings of the 19th IEEE Real-Time Systems Symposium*. IEEE Publishing, New York, Dec. 1997.

[10] Lundqvist K. and Asplund L. A Formal Model of the Ada Ravenscar Tasking Profile: Delay Until, in *Proceedings of SIGAda 99*. ACM Press, 1999.

[11] Lundqvist K., Asplund L. and Michell S.G. A Formal Model of the Ada Ravenscar Tasking Profile: Protected Objects, in *Proceedings of Reliable Software Technologies - Ada Europe '99*. Springer-Verlag, Berlin, 1999. LNCS 1622.

[12] *Proceedings of the 8th International Real Time Ada Workshop*. ACM Press, New York, 1997. Ada Letters.

[13] Owre S., Rushby J., Shankar N. and von Henke F. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS, in *IEEE Transactions on Software Engineering*. IEEE Publishing, New York, Feb. 1995. 21(2):107-125.

[14] Sha L., Klein M.H. and Goodenough J.B. Rate Monotonic Analysis for Real-Time Systems. CMU/SEI Technical report 6, 1991.

[15] Praxis Critical Systems. *SPARK Language Documentation Set*, 2002.