

Smart Certification Of Mixed Criticality Systems

Peter Amey, Rod Chapman, Neil White

Praxis High Integrity Systems, 20 Manvers St., Bath BA1 1PX, UK
{peter.amey, rod.chapman, neil.white}@praxis-his.com

Abstract. High integrity applications, such as those performing safety or security critical functions, are usually built to conform to standards such as RTCA DO-178B [1] or UK Def Stan 00-55 [2]. Typically such standards define ascending levels of criticality each of which requires a different and increasingly onerous level of verification. It is very common to find that real systems contain code of multiple criticality levels. For example, a critical control system may generate a non-critical usage log. Unless segregation can be demonstrated to a very high degree of confidence, there is usually no alternative to verifying all the software components to the standard required by the most critical element, leading to an increase in overall cost. This paper describes the novel use of static analysis to provide a robust segregation of differing criticality levels, thus allowing appropriate verification techniques to be applied at the subprogram level. We call this fine-grained matching of verification level to subprogram criticality *smart certification*.

1 Introduction

Many systems are composed of mixed criticality code. A military system being developed to DEF-STAN 00-55 [2] may contain SIL0, SIL2 and a small amount of SIL4 code. A civil aviation application being developed to EUROCAE ED12B/DO-178B [1] may contain Level C and Level A code. The same applies to almost any system you pick, compliant with almost any standard, from any sector of industry. Even if your application is not business-, mission- or safety-critical, there are almost certainly functions that are considered “core”, and thus worthy of a more rigorous validation phase.

The premise behind Smart Certification is to allow a mathematically rigorous partitioning of the source code into the different criticalities within an application. Criticalities could mean SIL level, DO-178B level, or a user-defined distinction between core and non-core.

This partitioning, which operates at a subprogram granularity, allows verification and validation techniques to be focused on specific areas, rather than spread over large swathes of the system. For example, EUROCAE ED12B/DO-178B sets the goal of full MCDC (Multiple Condition Multiple Decision) coverage for all Level A code. By partitioning the code we can ensure we do not spend time and money achieving this goal for non-Level A subprograms.

This focussed approach reduces costs and development time without impacting code quality or integrity.

2 Requirements for segregation

Showing that two software components, running on the same processor, in shared memory space, do not interfere with one another is a non-trivial task. The principal causes of interference between two software components are as follows:

- unintended data and information flow from one component to another;
- misbehaviour by one component, such as a writing outside an array boundary, such that data or code used by the other component is corrupted;
- preventing one component from running because the other has halted the processor by, for example, dividing by zero;
- resource hogging by one component such as consumption of all free memory or all available clock cycles; and,
- in concurrent systems, one component blocking the timely scheduling of another.

Furthermore, demonstrating that there are no malign effects present by use of source code static analysis requires an unambiguous mapping from source code to its compiled behaviour.

3 The role of SPARK

SPARK [3, 4] is an annotated subset of Ada with some specific properties that are designed to make static analysis both deep and fast. The annotations take the form of special comments which are ignored by an Ada compiler but have semantic meaning for SPARK's support tool, the SPARK Examiner.

For the purposes of this paper, the key properties of SPARK are:

Lack of ambiguity. The language rules of SPARK, enhanced by its annotations, ensure that a source text can only be interpreted in one way by a legal Ada compiler. Compiler implementation freedoms such as sub-expression evaluation order, cannot affect the way object code generated from a SPARK source behaves. For example, a complete detection of parameter and global variable aliasing ensures that SPARK parameters have pass-by-copy semantics even if the compiler actually passes them by reference.

Bounded resources. SPARK language rules prohibit both direct and indirect recursion. They also prevent direct use of dynamically allocated heap memory and avoid constructs, such as functions returning unconstrained arrays, where implicit heap use may be required. The result of these rules is that it is possible to know statically, prior to execution, what the worst case memory usage for a SPARK program will be.

Freedom from run-time exceptions. SPARK programs are amenable to proof-based forms of formal verification. The Examiner toolset includes provisions for the automatic generation of proof obligations that correspond to each predefined run-time exception check defined by the Ada language. Discharging these proof obligations is sufficient to guarantee that the associated exception can never be raised. The process is described in [5].

Well-behaved concurrency. SPARK has recently been extended to include a subset of Ada 95 concurrency constructs compatible with the Ravenscar profile, see [6]. A combination of language rules and additional annotations allows the static elimination of exceptional behaviour defined by the Ravenscar profile. A RavenSPARK program can therefore be shown to be free from problems of blocking or priority inversion.

Facilitates information flow analysis. Many of SPARK's language rules were motivated by the desire to provide an analytical framework in which mathematically rigorous data and information flow analyses can be conducted. The principles of such analyses are described in the seminal ACM paper [7]. Flow analysis is important for two reasons:

1. Elimination of undefined variable values by data flow analysis is an essential step in providing a sound environment for program proof. Clearly such proofs are complicated if we have to allow for unknown and potentially invalid data items.
2. Information flow analysis, which establishes the *influence* of variability of one data item on another, provides the main foundation on which we can build segregation arguments.

We can see that these properties have a significant bearing on the requirements for segregation described in Section 2.

Lack of ambiguity is a straightforward prerequisite for any reliance on source code static analysis. We cannot *trust* such analysis if there is the possibility that our analysis tool and compiler may be making different interpretations of the source code. (We may of course still *use* static analysis in such circumstances but only in the hope of finding some errors rather than proving their absence).

Bounded resource usage is a necessary precondition for ensuring that low-criticality code cannot prevent operation of high-criticality code by, for example, consuming all the system's memory. There remains an outstanding issue here, proof of loop termination, which is discussed in Section 5.3.

The ability, by static means, to prove that a SPARK program will never raise a predefined exception provides the support necessary to ensure that low-criticality code cannot corrupt the memory space or unexpectedly halt processing.

Well-behaved, Ravenscar-compliant, concurrency ensures that critical tasks are not blocked by less critical ones.

That leaves only the flow of data and information from one subprogram to another as a potential source of influence of less critical on more critical code. Our final SPARK property, a detailed and precise data and information flow analysis, provides a starting point for addressing this remaining source of potential corruption. The rest of this paper is concerned with improved methods for showing freedom from incorrect data coupling between units.

4 Case studies

Before we go on to address incorrect data coupling we present two historical case studies. These are real systems where software segregation has been demonstrated manu-

ally. Both systems would have benefited from the tool-supported segregation we are describing.

4.1 SHOLIS

The SHOLIS system, described in [8], provides safety-critical guidance on the safe operating of ship-borne helicopters. In particular, it defines safe operating envelopes for prevailing wind conditions and sea states. SHOLIS was the first project ever to attempt to meet the requirements of UK Defence Standard 00-55 [2] which placed a significant emphasis on formal methods and program proof.

A novel aspect of the SHOLIS implementation was the mixing of different criticality levels of software in a single memory space and running on a single processor. The primary application was fully safety critical, SIL 4 in 00-55 terms, but other software components concerned with data logging for example, were much less critical. The Standard requires the generation of a safety case and as part of this, a rigorous justification for the mixing of SIL level was required.

We did a *manual* allocation of criticality to package-level state, and then did a *manual* verification of the derives annotation on every procedure to show that that no lower-criticality state was used to update higher-criticality state. Obviously this justification needs to be laboriously maintained for every small code change.

4.2 Engine Monitoring Unit

The Engine Monitoring Unit (EMU) is a health monitoring system for a large civil jet engine. The system was developed to DO-178B [1], with some system functions assessed to be Level C, and the remaining functionality assessed to be Level E.

The application software was a multi-tasked SPARK95 program running on the GreenHills INTEGRITY real-time operating system. The INTEGRITY RTOS is designed for use in mission-critical embedded systems. Task groups run in hardware isolated memory spaces to minimise the impact of errors, and resource requirements can be analysed to guarantee availability. Extra INTEGRITY modules build on the basic RTOS functionality, providing Ethernet drivers, file systems, and much more. The core INTEGRITY RTOS can be certified to DO-178B Level A, but some of the additional INTEGRITY modules (for example, the Ethernet driver) can only be certified to lower levels.

In project EMU, the option to build and certify the entire system to DO-178B Level C - the highest common denominator - was not available. The software implementing one of the Level E functions used an INTEGRITY module which could only be certified to Level E. A re-implementation of the module in question was prohibitively expensive.

The solution was to partition the functionality so that individual tasks were either level C or Level E. No task had mixed level functionality. Communication between tasks was only allowed via semaphore protected, uni-directional shared data. The term uni-directional is simply used to mean that one task produced data, and one task consumed the data. This enabled us to use SPARK information flow to show that no level E task could influence a level C task. We were then able to certify the code for distinct tasks

to the level applicable for that task. The troublesome Level E module, used only by a Level E task, was separated from all the Level C code.

The disadvantage of this design-time architecture constraint was that we ended up with a task structure that was not optimal and not particularly intuitive. This caused an increase in implementation cost, and looms over all future maintenance work. (Since certification there have been zero reported application software faults, but we assume that somebody will want to maintain the software at some point in the future.)

Using the techniques described in this paper, we would not have required the architecture constraint, and would have been in a position to produce a system with a more optimal assignment of system functionality to tasks. We would also have reduced implementation costs, and been able to provide an easily repeatable, tool supported argument of the partitioning.

5 Extensions to SPARK

From the case studies it is clear that information flow analysis provides an approach to demonstrating code segregation but that better support is required to both simplify and automate the arguments used so as to make them easily repeatable.

5.1 Foundation Work

The first steps to exploiting information flow analysis to show separation of concerns exploits the SPARK *own variable* annotation. An own variable annotation provides a name for one or more items of static, package-level data or for a point where the program interacts with its external environment. The rationale for and use of own variables is described in [9]. The foundation work for the developments described in this paper is to allocate criticality levels to each own variable by means of a new annotation. Information flow analysis is then used to show that no critical output is dependent on a non-critical input or a non-critical intermediate state variable. The process is described fully in [10] and is applicable to both safety and security critical applications.

Clearly this approach moves us closer to our goal of demonstrable separation of criticality levels at the subprogram level but does not get us there for reasons described in the next section.

5.2 New Developments

The SPARK language extensions outlined in the previous section allow us to ensure that critical outputs are not tainted by untrusted data; however, they are not sufficient to identify each critical code *statement sequence*. Further extensions to the analysis are needed because of the parameterisation of subprograms. Where a subprogram directly updates an own variable of a particular criticality then it is clear that that subprogram is at least as critical as the data it is processing. Such global dependencies are revealed by SPARK annotations.

Consider a trivial example, a package that maintains a highly critical counter variable.

```

package Counter
--# own State (Integrity => SIL4);
--# initializes State;
is
  procedure Increment;
    --# global in out State;
    --# derives State from State;

    function Read return Integer;
      --# global in State;
end Counter;

```

From the annotation of procedure `Increment` it is clear that it *updates* own variable `State` which is marked as being of integrity level “SIL4” (a suitably-defined numeric constant). From this we can conclude that procedure `Increment` must be considered a SIL4 subprogram and verified in a manner appropriate to that level.

Note that there is no similar assumption to be made about function `Read`. Although this *references* own variable `State` that does not impose any criticality considerations on it; it is the way in which the function is *used* that matters. For example, if we only called `Read` in order to pass the current counter value to some low-level data logging routine then the integrity of the function would not be important.

Things are less clear, however, when we consider the following implementation.

```

package body Counter
is
  State : Integer := 0;

  procedure Inc (X : in out Integer)
    --# derives X from X;
  is
  begin
    X := X + 1;
  end Inc;

  procedure Increment
  is
  begin
    Inc (State);
  end Increment;

  function Read return Integer
  is
  begin
    return State;
  end Read;
end Counter;

```

Here the annotations give us no indication that local procedure `Inc` is being used to update the critical data item `Counter.State` and should therefore be regarded as critical code.

We could certainly determine the criticality of such data flows given a complete program from which we could determine the whole call tree and hence know the use to which each subprogram was being put. However, an important SPARK principle is the desire to be able to analyse programs incrementally before they are complete. Interactive analysis during development is a prime means by which SPARK has been shown to both improve quality *and* reduce cost. We can achieve the goal of detecting critical code segments while still doing partial program analysis by annotating subprograms with the level of data they are permitted to handle and performing an analysis to detect cases where there is a mismatch between subprogram and data criticality levels.

Mathematical Details

In this section we will look at the mathematical model which underpins both the rigorous partition, and the static analysis which enforces the partition. Each subprogram, S , is annotated with its proposed level of criticality, S_C . This is the developer stating the level of criticality that the subprogram will be developed to. Every own variable, G , already has an associated criticality, G_C . By analysing the global and derives annotations for S we can calculate S 's highest criticality exported own variable, which we denote S_{GC} . Condition 1 follows immediately.

Condition 1 *We need $S_{GC} \leq S_C$ for a useful subprogram.*

If condition 1 is violated, so if S_{GC} is larger than S_C , then we have a subprogram directly or in-directly updating an item of state with a higher criticality. This is not allowed. Now our subprogram is internally consistent, we can look at calls to S .

Condition 2 *Every call to S from a distinct¹ subprogram D must satisfy $S_{GC} \leq D_C$.*

To violate condition 2 would allow a lower criticality subprogram to indirectly update a higher criticality state element. Note that a low criticality subprogram is not banned from calling a higher criticality subprogram².

As we saw in the earlier example, actual parameters used in calls to S are a complication. In a call to S list the actual parameters for `out` and `in` `out` formal parameters as $P^1, P^2, \dots P^n$. These each have a criticality P_C^i .

Condition 3 *For each call to S , we require $\forall i \in 1..n \cdot P_C^i \leq S_C$.*

To violate condition 3 would allow S to update a higher integrity state element via a parameter³.

¹ Since SPARK does not allow recursion, D must be distinct from S .

² For a well formed SPARK program Condition 2 actually follows from Condition 1. If D calls S , then D must export own variables of at least the same criticality as S , so $S_{GC} \leq D_{GC} \leq D_C$. We state the condition separately for clarity.

³ The `in` parameters must be of suitable criticality for the state they are used to derive. This is handled by the existing functionality described in section 5.1 and [10].

Functions in SPARK are pure, so they update no global data and have only `in` parameters. This means that the criticality of a function is bounded only by its calling environment. Consider a call to function F , where the return result is assigned to a variable R of criticality R_C .

Condition 4 For each call to F , we require $R_C \leq F_C$.

In other words, a function must be at least as critical as the most critical state item ever used to capture the return value.

The last question we must address is the criticality of local variables. This is essential if we are to be able to check the validity of our subprogram calls. Each local variable is used to derive one or more own variables, otherwise the local variable is ineffective⁴. Local variables inherit the criticality of the *most* critical own variable they are used to derive. So for a local variable L , used to derive own variables G^1, G^2, \dots, G^n , we have $L_C = \max\{\forall i \in 1..n \cdot G_C^i\}$. This concludes our look at the underlying mathematical model.

In the previous example, `Increment` must be annotated with a criticality of *at least* SIL4 because it exports the global `State`, which is SIL4 (See condition 1). `Inc` exports no globals, so condition 1 gives us no lower bound for the criticality, allowing any subprogram to call `Inc`. `Increment` calls `Inc` with a SIL4 actual parameter for `X`. If `Inc` is annotated as anything less than SIL4, then the actual parameter is more critical than the called routine, which is a violation of condition 3. `Inc` can only be legally annotated as SIL4. Note that we could still call `Inc` from a lower SIL subprogram with a lower SIL parameter. The criticality of the function `Read` has no lower bound, and the upper bound cannot be deduced in isolation. (See condition 4.)

The only extension to the SPARK language is an annotation to allow a specific subprogram to be given a criticality by the developer. We propose the following notation, which nicely complements the notation for state criticality:

```
package Counter
--# own State (Integrity => SIL4);
--# initializes State;
is
  procedure Increment;
  --# global in out State;
  --# derives State from State;
  --# declare Integrity => SIL4;

  function Read return Integer;
  --# global in State;
  --# declare Integrity => SIL2;
end Counter;
```

⁴ The Examiner will identify any ineffective variables and issue suitable warnings.

5.3 Possible Future Work

There is a single remaining way in which interference can occur: the non-termination of a loop in non-critical code such that execution of critical code is blocked. The proof facilities of the SPARK Examiner currently allow *partial proof*, ie a proof the the code is correct *if it terminates*. Work is underway to extend the proof model to include a proof of loop termination[11].

In the interim, one option is to force a proof of loop termination through the proof of absence of run-time errors. We can declare a fresh numeric variable, initialise it, and then increment it on each trip round the loop. In order to show the variable does not exceed its upper bound, we are forced to construct a rigorous argument about the maximum number of times the loop can iterate.

6 Smart Certification Example

This section presents an example, loosely based on real life, showing the detection of the misuse of a subprogram above its authorised criticality level. The example centres around a brake-by-wire system. First we give a specification for package Brake.

```
package Brake
--# own in Pedal (Integrity => SIL4);
--# out HydraulicPressure (Integrity => SIL4);
is
  type BrakeLevel is range 0 .. 100;

  procedure Operate(Amount : out BrakeLevel);
  --# global HydraulicPressure, Pedal;
  --# derives HydraulicPressure, Amount from Pedal;
  --# declare Integrity => SIL4;
end Brake;
```

The procedure `Operate` reads the variable `Pedal` to ascertain the users current braking request. The operation updates `HydraulicPressure` accordingly, and returns the current retardation (as a percentage of the maximum retardation available) to the calling environment. Note that both state items and the procedure are all annotated as SIL4 state. We omit the body of package Brake. The package Log is a simple non-critical record of the vehicle's braking history.

```
package Log
--# own Data (Integrity => SIL0);
--# initializes Data;
is
  procedure Enter(IsBraking : in Boolean);
  --# global Data;
  --# derives Data from Data, IsBraking;
  --# declare Integrity => SIL0;
end Log;
```

```

package body Log
--# own Data is Pointer, Queue;
-- Both SIL0, inherited from abstract own.
is
  type HistoryIndex is mod 256;
  type History is array(HistoryIndex) of Boolean;

  Pointer : HistoryIndex := 0;
  Queue   : History      := History'(others => False);

  procedure Enter(IsBraking : in Boolean);
  --# global Pointer, Queue;
  --# derives Queue from Queue, Pointer, IsBraking &
  --#           Pointer from Pointer;
  -- No new SIL declaration allowed.
  is
  begin
    Queue(Pointer) := IsBraking;
    Pointer := Pointer + 1;
  end Enter;
end Log;

```

We simply build up an buffer of Booleans to indicate if we were braking at a particular time. The buffer is circular, with new overwriting old as required. All we need now is a controlling procedure to pull everything together.

```

with Brake, Log;
use type Brake.BrakeLevel;
--# inherit Brake, Log;
package Controller
is
  procedure Run;
  --# global Brake.Pedal, Brake.HydraulicPressure, Log.Data;
  --# derives Brake.HydraulicPressure from Brake.Pedal &
  --#           Log.Data               from Log.Data, Brake.Pedal;
  --# declare Integrity => SIL4;
end Controller;

package body Controller
is
  procedure Run
  is
    LogIt   : constant Brake.BrakeLevel := 20;
    CurrentAmount : Brake.BrakeLevel;
  begin
    Brake.Operate(CurrentAmount);
    Log.Enter(CurrentAmount > LogIt);
  end Run;
end Controller;

```

Controller.Run is called at a suitable frequency by the overall vehicle software scheduler (not shown). The procedure Run is SIL4 because it calls into Brake and updates SIL4 state. Just as we finish, we get a change in requirements - we need a stop light. Consider the specification of package Lamp.

```
package Lamp
--# own out Stop (Integrity => SIL3);
is
  procedure Light(On : in Boolean);
  --# global Stop;
  --# derives Stop from On;
  --# declare Integrity => SIL3;
end Lamp;
```

All we need to decide is how to call Lamp.Light. The obvious location is the body of Log.Enter, as this has the Boolean parameter IsBraking which can be used in the call to Lamp.Light. If we do this, the Examiner immediately points out that any caller of Lamp.Light must be SIL3. Log.Enter is only SIL0. We need to increase the criticality of the Log.Enter procedure to SIL3 before the system can be considered sound. But the Log package is *not* critical code. We are corrupting our design. A preferable solution is to modify the procedure Controller.Run.

```
procedure Run
is
  LogIt : constant Brake.BrakeLevel := 20;
  LampOn : constant Brake.BrakeLevel := 20;
  CurrentAmount : Brake.BrakeLevel;
begin
  Brake.Operate(CurrentAmount);
  Lamp.Light(CurrentAmount > LampOn);
  Log.Enter(CurrentAmount > LogIt);
end Run;
```

This option is sound, as Run is SIL4 already. Furthermore, this option provides greater future proofing as the “lamp on” and “log it” conditions are de-coupled. We could in future choose to log braking at a different retardation percentage than that which triggers the lamp.

This example is small, and the punch line can be seen coming from a significant distance, but it provides an indication of how the process works. With 150 packages, 10 levels of subprogram calling, and 5 distinct criticality levels, it becomes a distinctly non-trivial activity to “see the punch line coming”.

7 Conclusions

We have shown that a simple static analysis extension to the SPARK Examiner allows the confident development of mixed criticality code in a cost effective manner. With

the mathematically rigorous partitioning of criticalities we can construct simple, tool-supported, arguments about the scope of required certification. By applying verification and validation techniques to specific areas we reduce both development cost and development time without impacting quality.

During the maintenance phase we can easily see the impact of changes, and construct equally simple arguments about the scope of re-certification. This allows us to minimise the cost of re-certification by doing only what is necessary.

This approach is equally suitable for security critical systems. The rigorous partitioning allows us to show that the data we wish to remain secure is not used to derive insecure outputs. Furthermore, we can easily see the sequences of code that are manipulating secure data.

References

1. *RTCA-EUROCAE: Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B. 1992.
2. Ministry of Defence: *Requirements for Safety Related Software in Defence Equipment, Defence Standard 00-55*. August 1997.
3. Barnes, John: *High Integrity Software - the SPARK Approach to Safety and Security*. Addison Wesley Longman, ISBN 0-321-13616-0. 2003.
4. Finnie, Gavin et al: *SPARK 95 - The SPADE Ada 95 Kernel — Edition 3.1*. 2002, Praxis Critical Systems¹.
5. Chapman, Rod; Amey, Peter: *Industrial Strength Exception Freedom*. Proceedings of ACM SIGAda 2002².
6. Amey, Peter: *High Integrity Ravenscar*. Proceedings of Reliable Software Technologies - Ada Europe 2003. Lecture Notes in Computer Science Volume 2655².
7. Bergeretti, Jean-Francois; Carré, Bernard: *Information-flow and data-flow analysis of while-programs*. ACM Transactions on Programming Languages and Systems 1985¹, pp37-61.
8. King, Steve; Hammond, Jonathan; Chapman, Rod; Pryor, Andy: *Is Proof More Cost Effective Than Testing?*. IEEE Transactions on Software Engineering Vol 26, No 8, August 2000, pp 675-686².
9. Amey, Peter: *A Language for Systems not just Software*. Proceedings of ACM SIGAda 2001².
10. Chapman, Rod; Hilton, Adrian: *Enforcing Security and Safety Models with an Information Flow Analysis Tool*. Proceedings of ACM SIGAda 2004.
11. Hammond, Jonathan: *Specification of SPARK Total Correctness Proofs*. Praxis HIS, S.P0468.41.5. October 2004.

¹ Also available from Praxis High Integrity Systems

² Also downloadable from www.sparkada.com