Using Ada's Visibility Rules and Static Analysis to Enforce Segregation of Safety Critical Components

J-P. Rosen

Adalog, 2 rue du Docteur Lombard, 92441 Issy les Moulineaux CEDEX, France;
email: rosen@adalog.fr

J-C. Van-Den-Hende

ALSTOM Transport, 48, rue Albert Dhalenne,93482 SAINT-OUEN CEDEX, France;
email: jean-christophe.van-den-hende@transport.alstom.com

Abstract

Segregation of components is required in mixed criticality systems, where different safety integrity levels apply to various components. This paper presents a solution where appropriate organization of the project into child units and proper usage of Ada's visibility rules complemented with simple static analysis are sufficient to ensure that all violations of segregation rules will be rejected at compile time.

This paper provides some explanations about the Ada mechanisms used to that effect, in order to make it understandable by those who are not familiar with the Ada language.

The need for segregation

ALSTOM Transport is a leading provider of ground and embedded railway systems. In order to minimize costs as well as to maximize safety, it is developing a new, components based, architecture in Ada that would maximize the possibility of reusing components between various systems.

Railway safety is highly dependent on software; although it is true that a train can stop in an emergency situation (unlike planes), stopping a high speed train (such as the French TGV) with emergency breaking requires three minutes and 3300 meter distance. This is far too much to avoid an accident that would be caused by a software failure, and no manual action of the driver can compensate for a software fault. Therefore, railway systems are subject to very strict rules ensuring correctness of the software.

Railway software is governed by the safety standard EN-50128 [1], which defines five *Safety Integrity Levels* (SIL), ranging from SIL0 (lowest criticality) to SIL4 (highest criticality). This is similar to the "levels" E to A of DO178C [2] for avionics systems. As can be expected, the cost of developing, checking, and certifying SIL4 software is much higher than the one of lower SILs. The necessity of reducing development costs implies that only truly critical parts be subject to the highest criticality checks.

Mixed criticality systems

In a complex system such as those that ensure safety and correct operation of trains, only a relatively small subset of the functions (and hence associated components) is of a SIL4 level. However, the lower criticality components (considered SIL0 for short) run on the same computer and are part of the same main program as the SIL4 components.

Such systems where components with different safety requirements are running together are called *mixed criticality systems*, whether the components are several applications running on the same computer, or a single application that mixes various software components.

Of course, the difficulty with mixed criticality systems is that a defect in a SIL0 component could adversely affect the behaviour of a SIL4 component. The traditional approach to addressing this issue is to submit all components to the same safety process as required by the highest criticality component in the system - in practice the SIL4 process. While this has the benefit of ensuring the highest confidence in the system as a whole, it has an enormous cost, since the vast majority of components must suffer a costly validation and certification process that goes far beyond what is required for their own criticality.

Segregation

This cost can be dramatically reduced through *segregation*, i.e. if it can be proven that SIL0 components are independent from SIL4 ones, and that the behaviour of no SIL4 component depends on a SIL0 component. Such a segregation can be achieved through hardware or software control.

For example, in avionics systems (which have similar issues), the ARINC-653 [3] standard has been designed to ensure hardware segregation of components of different levels: the standard ensures that components of different criticalities have different address spaces, and a MMU ensures that each component can access only its own address space. Communications between components are performed through a dedicated bus, etc. Note however that hardware segregation prevents corruption by an incorrect low criticality component at execution time, but does not ensure that the software is free from such errors.

On the other hand, software proofs and other static verification techniques can be used to demonstrate that by design, no low criticality component performs dangerous or incorrect actions that could jeopardize the safety of high criticality components. Of course, to be effective and

economical, such proof systems have to be much cheaper than the usual SIL4 validation process.

The study and its requirements

ALSTOM wanted to evaluate various solutions to ensure segregation of components, and asked Novasys [4] (part of the Pacte-Novation group) to conduct two studies on solutions using hardware and software segregation respectively. The hardware solution was studied directly by Novasys, while the software solution, which is the purpose of this paper, was conducted by Adalog [5], a subsidiary of Novasys specialized in Ada consultancy, expertise, and training.

Requirements

A SIL4 component is one which is responsible for actions that can compromise safety, like setting the speed of the train, controlling the opening of the doors, etc. Such components must not only be checked for their own correctness; it is also important to check that they do not use unsafe operations, that their provided operations are not called in an incorrect manner and that they do not operate on incorrect data.

Therefore, the following rules were established as a basis for the software segregation study:

- Data passed from SIL0 to SIL4 components are deemed unreliable; it is up to the SIL4 component to assess the validity of the data.
- Except for the dedicated zones for data exchange, no SIL0 component is allowed to access SIL4 data.
- Some utility components that do not perform any safety critical function can be called by SIL0 as well as SIL4 components; however, since they are used by SIL4 components, they are classified as SIL4.
- If a SIL0 component needs to be called by a SIL4 component, this can be done only through a dedicated SIL4 component that will perform all required checking.
- Except for the special cases above, no SIL4 component or functionality can be used by a SIL0 component.

In addition, low level features of Ada, unchecked programming, and removal of language checks are not allowed in SIL0 components, in order to guarantee memory integrity of the system (see below).

A software architecture for statically checking segregation rules

The software study goal was to find a convincing (and economical) way of enforcing the above rules. The study proposed an architecture of the software that would allow checking of the segregation rules by the compiler. In other words, a program that would not obey by the rules would simply not compile. This was made possible by using Ada's visibility rules related to packages and child packages.

Ada packages and visibility rules

In Ada a *package* is a logical module that gathers a set of logically related elements (types, constants, subprograms…). Like all Ada units, a package has a *specification* and a *body*. The specification exposes the elements that are usable outside of the package, while the body contains the implementation of the services announced in the specification. The specification is furthermore divided into a *visible part* and a *private part*; actually, only elements from the visible part are made available to the outside units. This part can contain *private types* that are announced without revealing their internal structure. The private part of the package serves to give the compiler the full declaration of these types, without making it visible to the users. This allows the definition of *abstract data types*, where only the type name and its operations are made visible, all implementation details being hidden in the private part and in the body. Of course, the body of a package sees the private part, including the full declaration of abstract data types.

The typical structure of a package is shown in the following example:

```
package Example is -- specification
  type T is private;    -- a private type
  procedure P (X : T);  -- operation
private     -- beginning of private part
  type T is -- full declaration of T
    record
      Compo: Compo_Type;   -- Components…
    end record;
end Example;

package body Example is    -- body
  procedure P (X : T) is  -- body of P
    …
  end P;
end Example;
```

Figure 1 Structure of a package

Packages can be organized as a hierarchy of parent/child units. A child package is simply a package whose name is prefixed by the name of its parent. A child package can be either public or private.

- A public child can be accessed normally by the rest of the system; however its visible part has only access to the visible part of its parent[1]. For implementation purposes, its own private part and its body see the private part of the parent.
- A private child is available only to the bodies of its parent and siblings (and descendants). A parent, together with its private children, defines a subsystem, where only the parent interface is available outside the subsystem.

[1] Consequently, a public child cannot reveal declarations hidden in the private part of its parent.

The following example illustrates the declaration (specification) of public and private child packages:

```
-- public child package
package Parent.Pack1 is
  …
end Parent.Pack1;

-- private child package
private package Parent.Pack2 is
  …
end Parent.Pack2;
```

Figure 2 Child packages

The architecture

As exposed above, Ada features a sophisticated system for controlling visibilities, and therefore the allowed calls between separately compiled modules. The idea of the study was to use these features to provide compile-time enforcement of the segregation rules.

The proposed structure followed the overall general framework exemplified by the following figure:

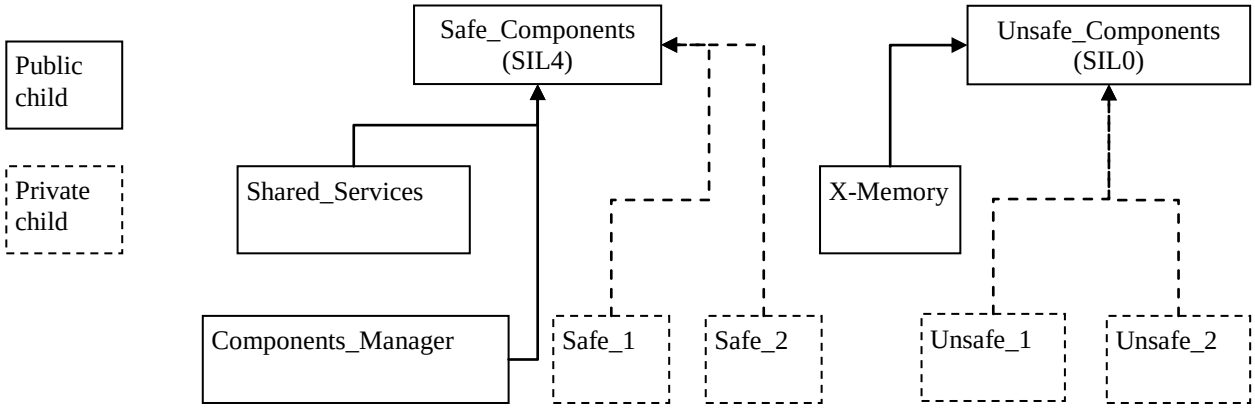

Figure 3 Architecture of the application

In this example, "Safe_Components" and "Unsafe_Components" are empty packages that serve as roots to the SIL4 and SIL0 hierarchies, respectively. "Shared_Services" and "Components_Manager", which are callable from SIL0 components, are public children of "Safe_Components" (thus visible and callable by all components), while SIL4 components are private children (therefore visible and callable only from within the SIL4 hierarchy): with this structure, it is impossible for SIL0 components to call SIL4 components, except for the dedicated and easily identifiable shared components.

Similarly, SIL0 components are private children of "Unsafe_Components", thus preventing them from being called by SIL4 components. On the other hand, the dedicated area for exchange of data ("X-Memory"), which is classified as SIL0 but usable from SIL4 components, is declared as a public child of "Unsafe_Components".

In the few cases where a SIL0 component would need to call a functionality from a SIL4 component, it would do so through an exported service of "Shared_Services", that would either perform the required validation of data, or, if there is no safety issue, simply be a renaming of the underlying (hidden) SIL4 service that remains private.

As far as data are concerned, except for the exchange area ("X-Memory"), no SIL0 variable should be accessible from SIL4 components, and conversely. This is easily obtained by forbidding the declaration of any variable in the visible part of packages (which is, in addition, a generally accepted coding rule, independently of any segregation issue). Possible data shared between components of the same level are placed in private children of "Safe_Components" and "Unsafe_Components".

Tracing the integrity level of components

In a mixed criticality system, it is important to trace the integrity level of each element, in order to perform checks appropriate to each level. This requires generally extra documentation, check lists, special comments, etc.

Another benefit of this structure is that the classification (SIL4 or SIL0) of components shows directly from the structure of the software; there is no need of maintaining manually a list of components with their assigned safety level. The level of the component appears directly from its Ada name; for example, the full name of the "Safe_1" component, the one given in its declaration, would be "Safe_Components.Safe_1", thus immediately showing that it is a SIL4 component. The list of SIL0 components is simply obtained by filtering all components whose name start with "Unsafe_Components.".[2]

Conversely, the simple fact that a component's name starts with "Safe_Components." or "Unsafe_Components." will automatically enforce the corresponding segregation rules.

[2] A common convention is to name a file containing a unit with the name of the unit (with some substitutions, like replacing "." with "-"). Some popular compilers enforce this convention. In such a case, obtaining the list of *files* containing SIL0 units is as simple as using the Unix command "ls unsafe_components-*".

Alternative possible architectures

The above described architecture was optimized according to the requirements of Alstom. But many variations on this basic principle of architecture are possible, depending on the constraints of the project. For example, shared component could constitute a hierarchy of their own rather than being under the "Safe_Components" tree[3].

In summary, the basic principles used for achieving segregation, and that Ada rules can enforce, are:

- Every segregated subsystem constitutes a single tree, with an empty root and where every module (except for communication modules) are private child units.
- Communication between modules of different criticality is achieved through public child units. Every communication module needs to be certified at the highest integrity level among its own level and the level of all possible callers.

Other necessary checks

Because it is sometimes necessary to escape from common programming rules, often in connection with low level programming such as direct management of hardware, Ada provides so-called *unsafe programming* features. These features include special packages to overcome normal type checking and provide direct access to memory, and *pragmas* for the removal of mandatory compiler checks (such as array overflow control). Malicious use of these features could be used to defeat the controls provided by the above structure, therefore their use is not allowed in SIL0 components[4].

In a safety critical system, it is not sufficient to have a programming standard that forbids such features; it must be *proven* that they are effectively not used. In Ada, any compilation unit that requires the use of a package must name it in a special clause (a *with clause*), therefore ensuring that any dependency between units is explicitly stated – and this applies to predefined packages as well. Removal of language checks requires the use of special pragmas. Therefore, it is sufficient to make sure that there is no **with** clause naming one of the unsafe programming packages and no use of the special pragmas to ensure that the safety features of the language are effective.

Checking these rules is easily achieved with static analysis tools. One of these tools is Adalog's AdaControl tool [6][7][8], a free static rule checking tool whose rich set of rules covers all the necessary restrictions.

Finally, some constructs that are normally allowed by the language were forbidden by the constraints of the project, such as the declaration of variables in the visible part of packages. This can be checked by manual inspection; however AdaControl is also able to check these automatically, which is always preferable to human (and therefore fallible) inspection.

In addition, the study analyzed (existing) ALSTOM's coding standard to determine which SIL4 rules were applicable to SIL0 components in order to allow cohabitation, and all applicable rules were also found checkable with AdaControl.

Conclusion

In conclusion, the appropriate use of visibility rules related to public and private children allowed the definition of a structure where segregation rules are enforced by the compiler.

The remaining safety constraints were checked automatically by a static analysis tool (AdaControl), thus allowing cohabitation of SIL4 and SIL0 components without loss of safety, and with a considerable economic gain compared to solutions that involve hardware segregation, or full certification at SIL4 level of SIL0 components.

As an additional benefit, the structure allows easy tracing of the integrity level of each component.

References

[1] CENELEC (2011), *EN50128:2011 Railway Applications -Communications, signaling and processing systems.*

[2] DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1992.

[3] ARINC 653 - Avionics Application Software Standartd Interface, November 2010.

[4] http://www.novasys-ingenierie.com/

[5] http://www.adalog.fr/en/

[6] J-P. Rosen: *On the benefits for industrials of sponsoring free software development*, Ada User Journal, Volume 26, n° 4, december 2005.

[7] J-P. Rosen: *AdaControl: a free ASIS based tool*, presentation at FOSDEM, Brussels, Belgium, February 2006.

[8] M. Jemli and J-P. Rosen: *A Methodology for Avoiding Known Compiler Problems Using Static Analysis*, proceedings of the ACM SIGAda Annual International Conference (SIGAda 2010), ACM Press, ACM order number 825100, Fairfax, USA, October 24-28, 2010.

[3] This possibility was not retained because Alstom wanted to have all units requiring SIL4 verification under the same root.

[4] they are allowed in SIL4 components, since those are subject to extensive reviews to make sure that the features are used only appropriately.