ADA USER JOURNAL

Volume 40

Number 1

March 2019

Contents

Editorial Policy for Ada User Journal

Publication

Ada User Journal — The Journal for the international Ada Community — is published by Ada-Europe. It appears four times a year, on the last days of March, June, September and December. Copy date is the last day of the month of publication.

Aims

Ada User Journal aims to inform readers of developments in the Ada programming language and its use, general Ada-related software engineering issues and Ada-related activities. The language of the journal is English.

Although the title of the Journal refers to the Ada language, related topics, such as reliable software technologies, are welcome. More information on the scope of the Journal is available on its website at *www.ada-europe.org/auj*.

The Journal publishes the following types of material:

- Refereed original articles on technical matters concerning Ada and related topics.
- Invited papers on Ada and the Ada standardization process.
- Proceedings of workshops and panels on topics relevant to the Journal.
- Reprints of articles published elsewhere that deserve a wider audience.
- News and miscellany of interest to the Ada community.
- Commentaries on matters relating to Ada and software engineering.
- Announcements and reports of conferences and workshops.
- Announcements regarding standards concerning Ada.
- Reviews of publications in the field of software engineering.

Further details on our approach to these are given below. More complete information is available in the website at *www.ada-europe.org/auj*.

Original Papers

Manuscripts should be submitted in accordance with the submission guidelines (below).

All original technical contributions are submitted to refereeing by at least two people. Names of referees will be kept confidential, but their comments will be relayed to the authors at the discretion of the Editor.

The first named author will receive a complimentary copy of the issue of the Journal in which their paper appears.

By submitting a manuscript, authors grant Ada-Europe an unlimited license to publish (and, if appropriate, republish) it, if and when the article is accepted for publication. We do not require that authors assign copyright to the Journal.

Unless the authors state explicitly otherwise, submission of an article is taken to imply that it represents original, unpublished work, not under consideration for publication elsewhere.

Proceedings and Special Issues

The *Ada User Journal* is open to consider the publication of proceedings of workshops or panels related to the Journal's aims and scope, as well as Special Issues on relevant topics.

Interested proponents are invited to contact the Editor-in-Chief.

News and Product Announcements

Ada User Journal is one of the ways in which people find out what is going on in the Ada community. Our readers need not surf the web or news groups to find out what is going on in the Ada world and in the neighbouring and/or competing communities. We will reprint or report on items that may be of interest to them.

Reprinted Articles

While original material is our first priority, we are willing to reprint (with the permission of the copyright holder) material previously submitted elsewhere if it is appropriate to give it a wider audience. This includes papers published in North America that are not easily available in Europe.

We have a reciprocal approach in granting permission for other publications to reprint papers originally published in *Ada User Journal.*

Commentaries

We publish commentaries on Ada and software engineering topics. These may represent the views either of individuals or of organisations. Such articles can be of any length – inclusion is at the discretion of the Editor.

Opinions expressed within the *Ada User Journal* do not necessarily represent the views of the Editor, Ada-Europe or its directors.

Announcements and Reports

We are happy to publicise and report on events that may be of interest to our readers.

Reviews

Inclusion of any review in the Journal is at the discretion of the Editor. A reviewer will be selected by the Editor to review any book or other publication sent to us. We are also prepared to print reviews submitted from elsewhere at the discretion of the Editor.

Submission Guidelines

All material for publication should be sent electronically. Authors are invited to contact the Editor-in-Chief by electronic mail to determine the best format for submission. The language of the journal is English.

Our refereeing process aims to be rapid. Currently, accepted papers submitted electronically are typically published 3-6 months after submission. Items of topical interest will normally appear in the next edition. There is no limitation on the length of papers, though a paper longer than 10,000 words would be regarded as exceptional.

Editorial

Is it true that life begins at forty? Looking back to the history of the Ada User Journal, I think no, it doesn't. The AUJ has a very rich and lively past, with countless contributions received from, and to, the community, actively playing an important role in the promotion of Ada and Reliable Software! Nonetheless, in the year the AUJ turns 40, we instead look to, and prepare for, the future.

An important guarantee of this future is the team in charge of the Journal. And I am glad to inform the readers of further changes to the Ada User Journal editorial team: António Casimiro, from the University of Lisbon in Portugal, joins the team as Associate Editor, and Alejandro Mosteo, from Centro Universitario de la Defensa de Zaragoza, Spain, joins as News Editor. As you all know, the Journal is put together by a set of volunteers, which dedicate time and effort to the preparation, editing, publishing and distribution of the Journal, and it is thus important to strengthen this team. Further changes are being prepared, which we will inform later in the year.

As for the contents of the issue, we conclude the publication of contributions related to the Ada-Europe 2018 conference. First, a paper derived from an industrial presentation, from a group of authors of Mälardalen University and OHB, Sweden, and Intecs, Italy, presenting the customization of the CHESS methodology and the ConcertoFLA toolset for the development of space software under the ECSS standard. Afterwards, the reader will find a paper on the use of the concurrent object concept in the context of the Rust language, which was the topic of a technical presentation at the conference, by a group of authors from the Luleå University of Technology, Sweden.

Closing the issue, we publish the first part of the Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems, which has been updated to consider Ada 2012, and is being prepared to be approved as a ISO technical report. The guide, written by Alan Burns, from the University of York, UK, Brian Dobbing, currently retired and at the time at Altran Praxis, UK, and Tullio Vardanega, from the University of Padua, Italy, includes the definition, rationale and examples of use of the Ravenscar profile, provided in this first part, and also describes the verification approach appropriate to analyse Ravenscar programs, which will be published in the next issue of the Journal.

As usual, the reader will also find the valuable information of the News and Calendar sections. I would also like to draw your attention to the advance information about the Ada-Europe 2019 conference, which, apart from the rich content of tutorials, exhibition and scientific and technical presentations will also provide a very rich networking environment.

Luís Miguel Pinho
Porto
March 2019
Email: AUJ_Editor@Ada-Europe.org

Quarterly News Digest

Kristoffer Nyborg Gregertsen

SINTEF, Email: kristoffer.gregertsen@sintef.no

Contents

Ada-related Tools

Qt5Ada

From: leonid.dulman@gmail.com
Subject: Announce: Qt5Ada version 5.12.0 release 21/12/2018 free edition
Newsgroups: comp.lang.ada
Date: Fri, 21 Dec 2018 10:56:14 -0800

Qt5Ada is Ada-2012 port to Qt5 framework (based on Qt 5.12.0 final)

Qt5ada version 5.12.0 open source and qt5c.dll, libqt5c.so(x64) built with Microsoft Visual Studio 2017 in Windows, gcc x86-64 in Linux.

Package tested with gnat gpl 2012 ada compiler in Windows 32bit and 64bit , Linux x86-64 Debian 9.4.

It supports GUI, SQL, Multimedia, Web, Network, Touch devices, Sensors, Bluetooth, Navigation and many others thinks.

Changes for new Qt5Ada release:

Added new packages: Qt.QStringView, Qt.QGraphicsCustomItem, Qt.QGLContext

My configuration script to build Qt 5.12.0 is: configure –opensource -release -nomake tests -opengl dynamic -qt-zlib -qt-libpng -qt-libjpeg -openssl-linked OPENSSL_LIBS="-lssleay32 -llibeay32" -plugin-sql-mysql -plugin-sql-odbc -plugin-sql-oci -icu -prefix "e:/Qt/5.12"

As a role Ada is used in embedded systems, but with QTADA(+VTKADA) you can build any desktop applications with powerful 2D/3D rendering and imaging (games, animations, emulations) GUI, Database connection, server/client, Internet browsing , Modbus control and many others thinks.

Qt5Ada and VTKAda for Windows, Linux (Unix) https://r3fowwcolhrzycn2yzlzzw-on.drv.tw/AdaStudio/

The full list of released classes is in "Qt5 classes to Qt5Ada packages relation table.docx" VTKAda version 8.1.0 is based on VTK 8.1.0 (OpenGL2) is fully compatible with Qt5Ada 5.12.0

I hope Qt5Ada and VTKAda will be useful for students, engineers, scientists and enthusiasts

With Qt5Ada you can build any applications and solve any problems easy and quickly.

If you have any problems or questions, tell me know.

AWS issue

From: Andrew Shvets <andrew.shvets@gmail.com>
Subject: Can't get to include AWS
Newsgroups: comp.lang.ada
Date: Thu, 27 Dec 2018 19:58:14 -0800

I installed the latest GNAT Community distribution from AdaCore in ~/GNAT and when I tried to use my *.GPR file in order to build my code, I encountered the below error:

unknown project file: "aws"

In my *.GPR file I did 'with "aws";'.

Is there some path or some other config value that needs to be set?

Thanks in advance for your replies.

From: eduardsapotski@gmail.com
Subject: Re: Can't get to include AWS
Newsgroups: comp.lang.ada
Date: Fri, 28 Dec 2018 01:23:55 -0800

Run GPS.

Open project.

Edit -> Project Properties -> Dependencies

Drag AWS to left panel.

Save.

Or in .gpr file paste: with "aws.gpr";

From: Simon Wright <simon@pushface.org>
Subject: Re: Can't get to include AWS
Newsgroups: comp.lang.ada
Date: Sat, 29 Dec 2018 20:30:30 +0000

> I installed the latest GNAT Community distribution from AdaCore in ~/GNAT and when I tried to use my *.GPR file in order to build my code, I encountered the below error:

> unknown project file: "aws"

> In my *.GPR file I did 'with "aws";'.

I have GNAT CE installed under /opt/gnat-ce-2018.

If I don't have /opt/gnat-ce-2018/bin on my PATH but say /opt/gnat-ce-2018/bin/gprbuild -P shvets.gpr where shvets.gpr contains 'with "aws";' I get the same as you.

If I do have /opt/gnat-ce-2018/bin on my PATH and say

```
gprbuild -P shvets.gpr
```

it works fine.

Protobuff for Ada

From: Per Sandberg <per.s.sandberg@bahnhof.se>
Subject: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Fri, 28 Dec 2018 19:57:40 +0100

I managed to resurrect an old master thesis work that was done by Niklas Ekendahl in 2013 and put it on

https://github.com/persan/protobuf-ada

the plan is to get it in working shape.

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Fri, 28 Dec 2018 21:53:55 -0800

Cool!

More libs, bindings, and implementations in Ada is a good thing.

Though, it should be noted that ASN.1 is *probably* the better technology in cases where ProtoBufs are being considered:

http://ttsiodras.github.io/asn1.html

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Sat, 29 Dec 2018 12:05:40 +0100

> Though, it should be noted that ASN.1 is *probably* the better technology in cases where ProtoBufs are being considered:

> http://ttsiodras.github.io/asn1.html

Sorry to disappoint you in this festive time, but this approach has the same fundamental flaw as prepared SQL statements do. You have to bind native Ada objects to protocol/serialized/persistent objects forth and back. This does not work well in practice. In fact, it barely work at all considering the overhead and hazards of type conversions.

A different approach is Ada's representation clauses which describe both objects same. Beyond simple textbook cases that does not work either.

The best practical method so far is using manually written stream attributes. Unfortunately it has shortcomings too:

1. Reuse is limited and composition is unsafe because stream attributes are non-primitive operations.

2. Introspection is almost non-existed. Only tagged types could have it.

3. No support of error handling and versioning. Though it is possible to do manually that is extremely error-prone and totally lacks static verification when the number of test cases is huge to potentially infinite. Even worse, the offending cases do not show up in a normally functioning system. So, when detected, it is always too late.

P.S. Needless to say, the problems 1-3 fully apply to other two methods as well.

P.P.S. And nothing was said about referential and recursive types...

From: Olivier Henley <olivier.henley@gmail.com>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Mon, 31 Dec 2018 08:55:40 -0800

Interesting. I do not grasp the problem in full though...

When you say "Sorry to disappoint you in this festive time", do you mean trying a solution from ASN.1 or only trying at Protobuff?

I think I get why a Protobuff could not cover "complete" transfer of Ada types around, but how does other languages do? (Almost everyone has it) Some of these languages have relatively "complex" type system..?

How do they achieve it? They express any complex types with a limited subset of primitive types(string, int32, etc)?

Can you give a more pragmatic example that exemplifies the limitations in Ada?

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Mon, 31 Dec 2018 18:59:35 +0100

>> When you say "Sorry to disappoint you in this festive time", do you mean trying a solution from ASN.1 or only trying at Protobuff?

Both. They are useless, up to harmful.

> I think I get why a Protobuff could not cover "complete" transfer of Ada types around, but how does other languages do? (Almost everyone has it) Some of these languages have relatively "complex" type system..?

The very concept of a data definition/description language (DDL) is wrong as I tried to explain. It has a very long and sad history in process automation, control, communication (e.g. CORBA), databases (e.g. SQL). Almost everybody and everyone tried it and failed. There are countless protocol describing "languages" around in process automation. I fought with them for decades, wrote several compilers for this mess. One could save huge amount of money and time if there were a law to punish people introducing this stuff... (:-))

> How do they achieve it? They express any complex types with a limited subset of primitive types (string, int32, etc)?

You cannot express a type in a DDL. Data /= Type. Type = data + operations. If you want to express complex typed objects you lose before you start with a DDL. You throw all type semantics overboard.

If you are OK without semantics then there is no need to introduce this mess. Use Ada stream attributes and simply read and write what you want and how you want. It is clean, easy, fast and 100% Ada.

> Can you give a more pragmatic example that exemplifies the limitations in Ada?

Any limitations Ada might have are unrelated to the issue of language impedance: DDL vs Ada unless you make DDL embedded like embedded SQL, which does not work either.

I believe AdaCore has a product of the sort. Though I don't think that would be much better, but I would rather trust them than anybody else...

From: G. B. <nonlegitur@nmhp.invalid>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Wed, 2 Jan 2019 06:57:14 -0000

> *If* you are OK without semantics then there is no need to introduce this mess. Use Ada stream attributes and simply read and write what you want and how you want. It is clean, easy, fast and 100% Ada.

What kind of stream do you write for your partners in business? Three of them have different needs than you WRT data and none of them is using Ada.

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Wed, 2 Jan 2019 11:02:10 +0100

> [...]

> What kind of stream do you write for your partners in business?

Stream of octets.

> Three of them have different needs than you WRT data and none of them is using Ada.

They still can read and write the stream. You are confusing description of a protocol with the implementation of.

The OP suggested having descriptions in protobuff and partial implementation generated from that. It is a bad idea.

BTW, it is very easy to write things like protobuff straight in Ada with Simple Components

http://www.dmitry-kazakov.de/ada/components.htm#17.2.1

This feature is rarely used because, as I said, the concept is too limited and constraining if not wrong altogether.

Here is a small example. Consider an example in protobuff:

```
message Person {
   required string name = 1;
   required int32 id = 2;
   optional string email = 3;
}
```

This direct Ada code:

```
type Person is new State_Machine with
   Name  : String_Data_Item
                  (Max_String_Length);
   ID    : Unsigned_32_Data_Item;
   Email : String_Data_Item
                  (Max_String_Length);
end record;
```

Thanks to Ada's "introspection" that is all. It will be read or written by the connections server automatically. On the packet receipt callback, you get values like Person_Session.ID.Value. Before sending a new packet you assign Person_Session.ID.Value. Note, this is Ada 95, no fancy stuff.

I didn't show here alternation for using optional fields because the transport level representation would be different anyway. Which is the point actually. Such key details are all left unspecified in the protobuff "description" above along with endianness and other encoding issues. Yet exactly these details are essential in practice where the protocol is already defined. Present or not bits might kept combined in the message header, special values of integers are reserved to indicate exceptional states and so on and so forth. And, again, no semantics whatsoever, just buckets of bits.

From: Per Sandberg <per.s.sandberg@bahnhof.se>
Subject: Re: protobuff for Ada
Newsgroups: comp.lang.ada
Date: Tue, 1 Jan 2019 09:05:38 +0100

From my perspective absolutely biggest flaw with technologies like protobuff is:

* Its backed by a large corporation.

* The technology is well known.

* 99.9% of the programming population think that they are the salvation to serialization.

* The licensing is open.

And on top.

* There are significantly more than one project where the lack of protobuff support has ruled out Ada as implementation technology.

And my intent was to eliminate at least the last points even if the technology is inferior.

AdaControl

From: "J-P. Rosen" <rosen@adalog.fr>
Subject: [Ann] AdaControl V1.20r7 released
Newsgroups: comp.lang.ada
Date: Thu, 3 Jan 2019 14:03:30 +0100

Adalog is pleased to announce the release of a new version of AdaControl. Thanks to the support of several sponsors, there are several interesting new controls (see file HISTORY), with a grand total of 70 rules and 565 possible tests! The automatic fixes feature has been extended too.

More details, download, etc. from http://adacontrol.fr. The executable version is now provided for Gnat Community edition 2018.

Reminder: If you have any issue with AdaControl, please report it using

http://sourceforge.net/p/adacontrol/ticket

And if you use it for an industrial project, commercial support is available from Adalog, don't hesitate to ask for information at info@adalog.fr

GNU ELPA

From: Stephen Leake <stephen_leake@stephe-leake.org>
Subject: GNU ELPA package ada-ref-man version 2012.4 is now available
Newsgroups: comp.lang.ada
Date: Sat, 5 Jan 2019 10:26:23 -0800

GNU ELPA package ada-ref-man version 2012.4 is now available. This version adds '<' '>' annotation to indicate italics in syntax element names:

```
generic_instantiation ::=
    package defining_program_unit_name is
        new <generic_package_>name
            [generic_actual_part]
              [aspect_specification];
```

Simple Components

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: ANN: Simple Components for Ada v4.36
Newsgroups: comp.lang.ada
Date: Tue, 8 Jan 2019 12:50:31 +0100

The current version provides implementations of smart pointers, directed graphs, sets, maps, B-trees, stacks, tables, string editing, unbounded arrays, expression analyzers, lock-free data structures, synchronization primitives (events, race condition free pulse events, arrays of events, reentrant mutexes, deadlock-free arrays of mutexes), pseudo-random non-repeating numbers, symmetric encoding and decoding, IEEE 754 representations support, streams, multiple connections server/client designing tools and protocols implementations. The library is kept conform to the Ada 95, Ada 2005, Ada 2012 language standards.

http://www.dmitry-kazakov.de/ada/components.htm

Changes to the previous version:

- The package GNAT.Sockets.Server.Blocking was added to provide connection servers handling blocking I/O;
- Procedures Send_Socket and Receive_Socket were added to the package GNAT.Sockets.Server;
- Procedures Reconnect and Request_Disconnect were added to the package GNAT.Sockets.Server;
- The functions Is_Configured, Is_In, Has_Device_Configuration were added GNAT.Sockets.Connection_State_Machine.ELV_MAX_Cube_Client;
- Airing time decoding/encoding error in GNAT.Sockets.Connection_State_Machine.ELV_MAX_Cube_Client.

SparForte

From: koburtch@gmail.com
Subject: Ann: SparForte 2.2
Newsgroups: comp.lang.ada
Date: Tue, 8 Jan 2019 20:15:29 -0800

SparForte version 2.2 was released over the holidays.

It is available for download from the SparForte website:

https://www.sparforte.com/

This version brings preliminary programming-by-contract, side-effect detection and additional shell features. An overview can be found on my blog:

https://www.pegasoft.ca/coder/coder_december_2018.html

There are also several recent blog articles on the design of SparForte, as requested by the mailing list subscribers.

SparForte is a shell, scripting language and web template engine with a core feature set based on Ada. I hope you will find it useful.

Note: I do not regularly read this newsgroup. Please direct questions to the SparForte mailing list.

VTKAda

From: leonid.dulman@gmail.com
Subject: VTKAda 8.2.0
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 11:19:09 -0800

I'm pleased to announce VTKAda version 8.2.0 free edition release 01/02/2019.

VTKAda is Ada-2012 port to VTK (Visualization Toolkit by Kitware, Inc) and Qt5 application and UI framework by Nokia VTK version 8.2.0, Qt version 5.12.0 open source and vtkc.dll, vtkc2.dll, qt5c.dll (libvtkc.so, libvtkc2.so, libqt5c.so) were built with Microsoft Visual Studio 2017 (15.9) in Windows (WIN32) and gcc in Linux x86-64

Package was tested with gnat gpl 2017 ada compiler in Windows 10 64bit, Debian 9.4 x86-64

As a role ADA is used in embedded systems, but with VTKADA(+QTADA) you can build any desktop applications with powerful 2D/3D rendering and imaging (games, animations, emulations) GUI, Database connection, server/client, Internet browsing and many others thinks.

VTKADA you can be used without QTADA subsystem

Qt5Ada and VTKAda for Windows, Linux (Unix) https://r3fowwcolhrzycn2yzlzzw-on.drv.tw/AdaStudio/

Florist

From: "J-P. Rosen" <rosen@adalog.fr>
Subject: Florist is in Ada !
Newsgroups: comp.lang.ada
Date: Tue, 19 Feb 2019 17:10:08 +0100

See: https://www.carolslaneflorist.com/about-us

(found this while browsing for Florist, the Ada interface to Posix) :-)

OpenGLAda

From: Felix Krause <contact@flyx.org>
Subject: ANN: OpenGLAda 0.7.0
Newsgroups: comp.lang.ada
Date: Sat, 9 Mar 2019 19:18:49 +0100

This release includes some additions to the API, but primarily adds GNAT Community 2018 support. It is also the first release with a Windows installer. This installer includes the optional dependencies (GLFW and Freetype) and installs OpenGLAda on top of an existing GNAT installation.

The dependency on the 3rd party library Strings_Edit has been removed and UTF-8 decoding is now part of the project. This hopefully reduces confusion.

Release and further information is available here:

https://github.com/flyx/OpenGLAda/releases

Ada-related Products

SPARK

From: addaon@gmail.com
Subject: New to Spark, working an example
Newsgroups: comp.lang.ada
Date: Sat, 15 Dec 2018 21:43:50 -0800

Folks, new to this list, so not quite sure on etiquette.

I've been trying to understand Spark-2014 well enough to work through an example, and understand the capabilities and workflow of the tools. The example I chose was an example of floor(lg(n)) for n positive.

Rather than put a long post here, I'll refer to my (long) post at stackoverflow:

https://stackoverflow.com/questions/53752715/proving-floor-log2-in-spark. (If this is bad etiquette here, let me know, and I'll fix -- but it does seem a bit silly to duplicate the content in two locations)

Since SO seems to have a very limited Ada/Spark community, I'm hoping someone here can point me in the right direction. Basically, trying to understand what tools I should be trying to understand at this point. :-) Should I be looking at proving this with just a better understanding of how to write loop invariants; through appropriate lemmas; through an external prover like Coq; or something else?

From: Simon Wright <simon@pushface.org>
Subject: Re: New to Spark, working an example
Newsgroups: comp.lang.ada
Date: Sun, 16 Dec 2018 09:48:17 +0000

I don't think there's anything wrong with trying to attract attention (what gets my goat a bit is people posting the same question in both places at the same time).

I have to confess that I hadn't set up my SO account to watch the tags [spark-2014] or [spark-ada] (why both?), or even [gnat] or [ada2012] - rectified. You would have got more views if you'd included [ada] (but not necessarily any (more) answers :)

Your problems are an indication of why I, as a person who has no access to professional SPARK support, haven't invested any effort to speak of in SPARK (my difficulties were with tasking/time rather than mathematical loops, which tend to be rare in control systems).

That said, it looks to me as though the version of gnatprove in GNAT CE 2018 may not fully understand exponentiation:

util.ads:3:14: medium: postcondition might fail, cannot prove
2 ** Floor_Log2'Result <= X

util.ads:3:16: medium: overflow check might fail (e.g. when Floor_Log2'Result = 0)

From: Brad Moore <bmoore.ada@gmail.com>
Subject: Re: New to Spark, working an example
Newsgroups: comp.lang.ada
Date: Tue, 18 Dec 2018 18:41:59 -0800

I am by no means a SPARK expert, but I am also interested in exploring SPARK capabilities.

My approach led me to the following solution using just the SPARK 2018 GPL download from Adacore.... (no extra provers were needed here, other than the ones that come with GNAT CE 2018)

As an aside, it appears the version of gnatprove in GNAT CE 2018 does have a pretty good understanding of exponentiation, given that I was able to get the following proven.

```
package Util with SPARK_Mode is
  Max_Log2 : constant := Positive'Size - 1;
  subtype Log_Result is Natural
           range 0 .. Max_Log2;

  function Floor_Log2 (X : Positive) return
           Log_Result with
    Global  => null,
    Depends => (Floor_Log2'Result => X),
    Post    => X >= 2**Floor_Log2'Result
        and then X / 2 < 2**Floor_Log2'Result;
end Util;

pragma Ada_2012;
package body Util with SPARK_Mode is
  function Floor_Log2 (X : Positive) return
           Log_Result is
  begin -- Floor_Log2
    Log_Loop :
    for I in Log_Result loop
      pragma Loop_Invariant
           (for all J in 0 .. I => X >= 2**J);
      pragma Assert
           (X / 2 < 2**Log_Result'Last);
      if X / 2 < 2**I then
        pragma Assert (X >= 2**I);
        pragma Assert (X / 2 < 2**I);
        return I;
      end if;
      pragma Assume(I /= Log_Result'Last);
    end loop Log_Loop;
    return Log_Result'Last;
  end Floor_Log2;
end Util;
```

I technically didn't need to use the Global aspect or the Depends Aspect to prove this function, but I think it is a good idea to provide a more detailed contract using additional SPARK and Ada features, when possible.

The approach I took is to first of all make use of Ada 2012 contracts to constrain the results to only allow valid values. The Log_Result subtype only includes valid result values.

I think this is an important goal in general to eliminate bugs, whether writing code for regular Ada as well as SPARK.

My view is that in general, types such as Integer and Float should not be used since they are types that describe memory storage, not types that describe values of interest in the application domain.

By creating types that more accurately represent the application domain, I believe it makes the job of writing proofs in SPARK much easier, since the prover can reason that the values assigned to such values have specific value ranges and properties.

Another point, is to try to write an implementation that is easier to prove. For that reason, I wrote this is a for loop rather than a while loop, because the compiler can reason statically about how many iterations are performed, and what the values of the loop parameters can be.

The prover was able to prove all the assertions in the implementation.

I had to leave in one assumption, (the pragma assume),

```
  pragma Assume(I /= Log_Result'Last);
```

Without that, the prover complains that the post condition,

```
  X / 2 < 2**Floor_Log2'Result
```

cannot be proven. It appears that the prover is not able to prove that the loop exited by the return statement, rather than iterating the full loop and exiting the loop without entering the if statement.

However, I think this can be visually inspected and confirmed to be true, since the assert for the if statment,

```
pragma Assert(X / 2 < 2**Log_Result'Last);
```

just prior to the if statement was proven.

It follows that if the assertion is true, then the if statement would have to be entered on the following line, and that the return would exit the loop.

Thus, the reader should be able to visually tell that it is impossible to get by the if statement when I = Log_Result'Last, and thus the pragma Assume is true.

The return at the end of the function should never get executed, as the only way to exit the function is via the return inside the loop.

I didn't need to have the return inside the loop for the purpose of proving the function. I just did that to eliminate the need of extra variable declarations.

Probably the prover could be improved so that such an assume could be eliminated while still proving the overall function.

There may be a way to add additional asserts or pragmas to eliminate the need for the pragma Assume. So far I haven't found any, but perhaps someone else might come up with a way. Otherwise, I'm pretty happy with the solution I ended up with, given that the one assume in the code can be visually checked easily for correctness.

I am sure that other SPARK solutions exist. I think when it comes to proving something, it is better to start with something simple, and to have in mind choosing an implementation that is easier to prove. This should make it easier to arrive at a proof.

From: Simon Wright <simon@pushface.org>
Subject: Re: New to Spark, working an example
Newsgroups: comp.lang.ada
Date: Wed, 19 Dec 2018 16:58:41 +0000

>> util.ads:3:16: medium: overflow check might fail (e.g. when >> Floor_Log2'Result = 0)

> As an aside, it appears the version of gnatprove in GNAT CE 2018 does have a pretty good understanding of exponentiation, given that I was able to get the following proven.

Apparently so. But the part of gnatprove that gives examples of when the assertion might fail is quite misleading: for example,

util.ads:7:14: medium: postcondition might fail, cannot prove
2 ** Floor_Log2'Result <= X
(e.g. when Floor_Log2'Result = 0
and X = 0) *when X is Positive* !!
and util.adb:19:15: medium: overflow check might fail (e.g. when I = 0)

l.18 for I in 1 .. Log_Result'Last loop

l.19 if 2 ** I > X then

From: Brad Moore <bmoore.ada@gmail.com>
Subject: Re: New to Spark, working an example
Newsgroups: comp.lang.ada
Date: Wed, 19 Dec 2018 20:34:13 -0800

I agree that the error messages are misleading, as I was getting similar messages when I was working on this. While the values "0" mentioned in the error messages were confusing to me, I think the messages were helpful at least in suggesting the sort of tests the prover was trying to prove, which ultimately helped me figure out the assertions that were needed to get this to pass. The values given can be a bit of a red herring sometimes, but I think the underlying test described by the message is more helpful. This is my second problem that I attempted to prove in SPARK, so I didn't know if I would succeed, or know much about how to approach this. It's kind of a rewarding feeling when you get the prover to pass.

One suggestion I have to prove post conditions, is to state the post condition as an assert before returning from the subprogram, and work backwards from there.

References to Publications

Ravenscar References

From: lyttlec <lyttlec@removegmail.com>
Subject: Ravenscar References
Newsgroups: comp.lang.ada
Date: Wed, 16 Jan 2019 12:48:28 -0500

Can anyone suggest a good reference on using the ravenscar profile? In the Ada books I have, it only gets a one or two page mention. A reference with an extended case study would be great.

From: Simon Wright <simon@pushface.org>
Subject: Re: Ravenscar References
Newsgroups: comp.lang.ada
Date: Wed, 16 Jan 2019 18:15:03 +0000

You might find something useful at http://cubesatlab.org e.g. http://www.cubesatlab.org:430/PUBLIC/brandon-chapin-HILT-2016.pdf

From: lyttlec <lyttlec@removegmail.com>
Subject: Re: Ravenscar References
Newsgroups: comp.lang.ada
Date: Fri, 18 Jan 2019 14:18:10 -0500

Thanks all for the links. They are a help. However, I'm looking for something along the lines of porting legacy code to be ravenscar "safe".

As an illustration, consider making Dmitry A Kazakov's code meet Ravenscar. I need to port lots of existing more or less standard components to meet Ravenscar. This is to satisfy some regulatory authorities.

From: "Jeffrey R. Carter" <spam.jrcarter.not@spam.not.acm.org>
Subject: Re: Ravenscar References
Newsgroups: comp.lang.ada
Date: Sun, 20 Jan 2019 18:12:11 +0100

I don't know that "port" is a good word for this activity. I once looked at implementing Sandén's FMS problem using Ravenscar. Starting from the requirements, I first had to find a Ravenscar-suitable design. The standard design has a dynamic task per job, and is clearly not possible using Ravenscar. An alternative design using a task per workstation had to be used.

From that choice, Ravenscar drove a proliferation of protected objects and helper tasks. Things that were simple in full Ada became much more complex to meet the restrictions of the profile.

Presumably you would need to apply a similar process to each of the components you need to convert.

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Ravenscar References
Newsgroups: comp.lang.ada
Date: Mon, 21 Jan 2019 17:19:43 -0600

Note that the less strict profile Jorvik, defined in Ada 2020 (and already implemented in GNAT) would simplify this process.

I don't think it is possible to "convert" regular Ada code into Ravenscar (unless, of course, it doesn't use any tasks ;-). You pretty much have to completely rewrite it with Ravenscar in mind. (In this way, it is very much like using SPARK.)

From: "J-P. Rosen" <rosen@adalog.fr>
Subject: Re: Ravenscar References
Newsgroups: comp.lang.ada
Date: Tue, 22 Jan 2019 10:25:08 +0100

I don't fully agree with that statement; it all depends where you start from.

I recently helped one of my clients who wanted to move to Ravenscar. The original structure was all Ada83, communicating with rendezvous.

However, it was already safety critical, therefore based on cyclic, never ending tasks, and limited communications. It was reasonably easy to define patterns for matching the existing structure into Ravenscar patterns.

Ada Inside

Compilation Issues

From: alexander@junivörs.com
Subject: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 03:46:02 -0800

I've read some threads on here regarding the licensing situation of AdaCore's Libre compiler. For my upcoming project, I'm going to need (= very strong desire) to use Ada and I'm also going to need to be able to license the executable produced thereof in any way I desire.

In regards to the aforementioned, I have two questions. I realize I come forth as somewhat paranoid in the upcoming paragraphs (which undoubtedly I am). The licensing situation worries me a great deal.

1. ```As for the compiler build provided by (the GetAdaNow Mac OS X section's link to Sourceforge)[1]; which parts of that GCC build for compiling Ada can you safely use and still be covered by the "GCC Runtime Library Exception"? I can see it states you can use `GNATCOLL` and `XMLAda`. I'm assuming the standard library is included as well. Can you on the other hand use all console commands? `gnat <command>`? `gprbuild`? Or would these inject "non-runtime library exception'd" GPL code into the executable?```2. ```I've been attempting to compile and link some code through the use of the `gcc` command solely, but haven't been successful in doing so. I have, on the other hand, been able to successfully generate an

executable by utilizing the `gnatbind` and `gnatlink` commands consecutively after compiling with `gcc -c <file>`. Is it possible to use only the `gcc` command for the matter, or do you need to also throw in a few calls to the `gnat` commands?

When executing the following commands...

$ gcc -c src/main.adb -o obj/main.o

$ gcc -o main obj/main.o

I wind up with the following error (on the second command, which should be a GCC link):

Undefined symbols for architecture x86_64:

"_main", referenced from:

implicit entry/start for main executable

(maybe you meant: __ada_main)

ld: symbol(s) not found for architecture x86_64

collect2: error: ld returned 1 exit status

A similar error occurs when I attempt to create `.so` libraries manually using the `-shared` compiler switch. With all that being said, is it simply not possible to do these things through solely `gcc`, or am I missing something?```

It may be worth noticing that I've fallen in love with Ada to the utmost degree over the past year. As such, I'm planning on, at the very least, stalking "comp.lang.ada" like some creepy figure. You'll probably see more from me beyond these first two questions, is what I'm saying.

[1] https://sourceforge.net/projects/gnuada/files/GNAT_GCC 20Mac OS X/8.1.0/native-2017/

From: Simon Wright <simon@pushface.org>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 16:11:48

Let me start by saying that I'm not a lawyer.

> 1. ```As for the compiler build provided by (the GetAdaNow Mac OS X section's link to Sourceforge)[1]; which parts of that GCC build for compiling Ada can you safely use and still be covered by the "GCC Runtime Library Exception"? I can see it states you can use `GNATCOLL` and `XMLAda`. I'm assuming the standard library is included as well. Can you on the other hand use all console commands? `gnat <command>`? `gprbuild`? Or would these inject "non-runtime library exception'd" GPL code into the executable?```

They may (do) *generate* source code that gets included in the executable (gnatbind does this). But that isn't code that's provided with the compiler and might have a copyright issue; it's no different in principle from object code generated directly by the compiler.

> 2. ```I've been attempting to compile and link some code through the use of the `gcc` command solely, but haven't been successful in doing so. I have, on the other hand, been able to successfully generate an executable by utilizing the `gnatbind` and `gnatlink` commands consecutively after compiling with `gcc -c <file>`. Is it possible to use only the `gcc` command for the matter, or do you need to also throw in a few calls to the `gnat` commands?
>
> [...]

Building even hello_world* is sufficiently complex that you need gnatbind, gnatlink. As you've seen, you can use gcc for the actual compilation.

Building a dynamic library (do you mean .so? are you on a Mac or Linux?

You mention my darwin 8.1.0 release) is more so.

To see what gnatbind gets up to while doing its work, look at the b__* (or b~*) files it generates. Not much fun or point in generating those by hand.

* You can build a simple null program for an embedded system on an MCU without gnatbind, gnatlink. But you have to bother about storage mappings, prcessor startup, linker scripts etc instead.

From: Lucretia <laguest9000@googlemail.com>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 08:31:59 -0800

[...].

What version is that compiler on sourceforge? Is it from FSF directly, i.e. gcc.gnu.org? Or is it GNAT-GPL/CE, i.e. from AdaCore.com? If the latter, the licence is GPL-3.0 no linking exception, otherwise it's GPL-3.0 with linking exception. Basically, avoid anything GPL-3.0 no linking exception, especially Adacore's libraries.

From: G. B. <nonlegitur@nmhp.invalid>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 18:50:45 -0000

> I've read some threads on here regarding the licensing situation of AdaCore's Libre compiler. For my upcoming project, I'm going to need (= very strong desire) to use Ada and I'm also going to need to be able to license the executable produced thereof in any way I desire.

For licensing in arbitrary ways, the aforementioned Ada distribution is not the suitable one. Another compiler distribution might meet your needs, including some FSF GNAT. GPL means tit-for-tat and thus intentionally puts restrictions on licensing, no back doors.

From: Simon Wright <simon@pushface.org>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 19:21:04 +0000

> What version is that compiler on sourceforge? [...]

It's vanilla FSF with Adacore libraries, some of which have the runtime library exception, some of which don't (as noted at the link).

The Adacore sources, at https://github.com/AdaCore, are on the whole GPLv3 with the runtime exception. I've taken care to report the status:

from https://sourceforge.net/projects/gnuada/files/GNAT_GCC MacOS X/8.1.0/native-2017/

Tools included:

Full GPL:

ASIS from https://github.com/simonjwright/ASIS at [8ba68f3].

AUnit and GDB from GNAT GPL 2017.

Gprbuild from https://github.com/AdaCore/gprbuild at commit [1e551df] (note, libgpr is GPL with Runtime Library Exception[1]).

GPL with Runtime Library Exception[1:

GNATCOLL from:

https://github.com/AdaCore/gnatcoll-core at commit [a093d11].

https://github.com/AdaCore/gnatcoll-bindings at commit [2c426fe].

https://github.com/AdaCore/gnatcoll-db at commit [b66441c].

XMLAda from https://github.com/AdaCore/xmlada at commit [8a4b2bf]

From: alexander@junivörs.com
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 12:50:42 -0800

> Building a dynamic library (do you mean .so? are you on a Mac or Linux?

> You mention my darwin 8.1.0 release) is more so.

Yes. According to (this page)[1] it's accomplishable using the following command:

$ gcc -shared -o libmy_lib.so *.o

but that causes an error mentioning how there are "Undefined symbols for architecture x86_64:".

> For licensing in arbitrary ways, the aforementioned Ada distribution is not the suitable one. Another compiler distribution might meet your needs, including some FSF GNAT. GPL

means tit-for-tat and thus intentionally puts restrictions on licensing, no back doors.

GPL on its own, I must say, does serve a purpose. It's nice for the author to be able to share their source or works and still be certain nobody can (legally anyway) steal their work and distribute it for a fee themselves.

When it comes to source code licensed under GPL lacking the runtime library exception, on the other hand, I can't say I'm too fond of it. Compilers on their own, featuring a standard library, should always be free to use; whereupon the user may licence their executable in any way they want.

[1] http://beru.univ-brest.fr/~singhoff/DOC/LANG/ADA/gnat_ugn_20.html

From: Simon Wright <simon@pushface.org>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Tue, 11 Dec 2018 23:45:48 +0000

> [1] http://beru.univ-brest.fr/~singhoff/DOC/LANG/ADA/gnat_ugn_20.html

Because that page (and even the latest one at [2]) is wrong.

Almost all Ada code requires the services of the Ada runtime, and you need to reference the runtime at the link stage.

$ gcc -shared -o libmy_lib.dylib *.o -L<whereever> -lgnat -lgnarl

(<whereever>: e.g. /opt/gcc-8.1.0/lib/gcc/x86_64-apple-darwin15/8.1.0/adalib)

This is why it is *so* much easier to use gprbuild (I see that that reference talks about using gnatmake; that's because gnatmake is part of GCC Ada, and gprbuild isn't. But modern gnatmakes will delegate to gprbuild if they find one, at any rate if libraries are involved; they can't generate libraries, because it's too complicated for Adacore to maintain in two places, the GCC tree and the gprbuild tree).

If you want to see what's going on you can use -v.

[2] http://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#general-ada-libraries

>> For licensing in arbitrary ways, the aforementioned Ada distribution >> is not the suitable one. Another compiler distribution might meet >> your needs, including some FSF GNAT. GPL means tit-for-tat and thus intentionally puts restrictions on licensing, no back doors.

> GPL on its own, I must say, does serve a purpose. It's nice for the author to be able to share their source or works and still be certain nobody can (legally anyway) steal their work and distribute it for a fee themselves.

> When it comes to source code licensed under GPL lacking the runtime library exception, on the other hand, I can't say I'm too fond of it. Compilers on their own, featuring standard library, should always be free to use; whereupon the user may licence their executable in any way they want.

I don't understand. The first para says it's good, the second says it's bad.

From: alexander@junivörs.com
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Wed, 12 Dec 2018 01:34:01 -0800

> I don't understand. The first para says it's good, the second says it's bad.

Perhaps I've misunderstood something regarding the licensing situation. Is not the reason you cannot use a bunch of AdaCore developed packages due to the fact that it's licensed under GPL without the runtime library exception, ultimately meaning your executable must be licensed under GPL too?

Let's assume someone made a tool to aid people with a repetitive task in Ada. Give that the GPL license and it'd be impossible for someone to "steal" (redistribute for a fee) the original author's code, still allowing people to learn from the code that makes up the tool.

In the second situation, I'm speaking of any library package offering nigh on essential functionality to a programming language (in this case Ada), that does not contain the runtime library exception. I believe that all code developed to ship with a compiler should contain that exception.

I will make sure to await further responses before I justify my belief mentioned in the previous paragraph, should I prove to having gotten something wrong.

Whilst quickly scouring the Internet for some information that would substantiate the claim that some library package files do not contain the runtime library exception, I came across the (`GNAT.Regpat` source)[1], which does contain some form of the runtime library exception.

I presume perhaps that is an older source file than the one shipped with the compiler at this day (Copyright (c) 1996-2002)?

[1] https://www2.adacore.com/gap-static/GNAT_Book/html/rts/g-regpat__adb.htm

From: Björn Lundin <b.f.lundin@gmail.com>
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Thu, 13 Dec 2018 10:21:54 +0100

> [...]

You can always "steal" GPL code, and redistribute it for a fee as you see fit. The freedom in GPL is not free as free beer, but free as free speach. So you would need to provide the sources to the customers you sell to. And I think, a fairly easy way to reproduce an executable/library.

You code depending on GPL (linked with) will inherit the GPL license.

But you can charge your customers whatever you want.

However you likely need to provide something better that the original code for people _wanting_ to pay you, I guess.

From: alexander@junivörs.com
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Thu, 13 Dec 2018 02:30:20 -0800

> [...]

I don't know wherefrom I got my information that you can't sell a GPL application. Thank you for clarifying this!

From: alexander@junivörs.com
Subject: Re: Licensing Paranoia and Manual Compilation Issues
Newsgroups: comp.lang.ada
Date: Thu, 13 Dec 2018 02:32:47 -0800

> I don't know wherefrom I got my information that you can't sell a GPL application. Thank you for clarifying this!

Or rather, clarifying the contrary; correcting me.

Coextension Bug In GNAT

From: Jere <jhb.chat@gmail.com>
Subject: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 07:59:00 -0800

I was messing around and trying to learn coextensions and I came across some counter intuitive functionality. If I directly initialize one via an aggregate, it works fine.

However, if I initialize through a constructing function, it seems to treat the access discriminant as a normal access type and finalizes it at the end of the program instead of when the object leaves scope. I don't fully understand them yet and there isn't much on them listed in the RM but one section (at least according to the index)[1]. That one section does indicate that initialization via a function should be valid however, so maybe I am back to I am doing it wrong or potentially a GNAT bug.

I'm using GNAT 7.1.1

Here is my test program

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Finalization; use Ada.Finalization;

procedure Hello is

   type Thing_1 is new Limited_Controlled
         with null record;

   overriding
   procedure Finalize(Self : in out Thing_1)
   is
   begin
      Put_Line("Finalize Thing_1");
   end Finalize;

   type Thing_2
      (Other : not null access Thing_1)
   is limited null record;

   procedure Test_Coextension_1 is
      The_Thing : Thing_2(new Thing_1);
   begin
      Put_Line("Coextension directly
         initialized");
   end Test_Coextension_1;

   function Make_Thing_2 return Thing_2 is
   begin
      return (Other => new Thing_1);
   end Make_Thing_2;

   procedure Test_Coextension_2 is
      The_Thing : Thing_2 := Make_Thing_2;
   begin
      Put_Line("Coextension initialized
         through build in place");
   end Test_Coextension_2;

begin
   Test_Coextension_1;
   Test_Coextension_2;
   Put_Line("Test Finished");
end Hello;
```

Any thoughts?

[1] Ada 2012 tc1 RM 3.10.2(14.4/3) - http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-3-10-2.html#I2301

From: Jere <jhb.chat@gmail.com>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 08:02:27 -0800

> [...]

Sorry, forgot to put the program output:

```
$gnatmake -o hello *.adb
gcc -c hello.adb
gnatbind -x hello.ali
gnatlink hello.ali -o hello
$hello
Coextenson directly initialized
Finalize Thing_1
Coextension initialized through build in place
Test Finished
Finalize Thing_1
```

From: Simon Wright <simon@pushface.org>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 16:56:11 +0000

> [...]

Compiling with -gnatwa I see "warning: coextension will not be finalized when its associated owner is deallocated or finalized", so GNAT clearly meant to do it!

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 20:16:09 -0600

> [...]

This message is nonsense, because a coextension is effectively part of the associated object. What they presumably mean to say is that the declaration in question is *not* a coextension, thus it will not be finalized with the owner.

P.S. I hate coextensions. One of the least necessary complications of Ada.

(Janus/Ada gives you a "feature not implemented" message if you try to create one.)

From: Jere <jhb.chat@gmail.com>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Fri, 21 Dec 2018 03:24:43 -0800

> [...]

> Compiling with -gnatwa I see "warning: coextension will not be finalized when its associated owner is deallocated or finalized", so GNAT clearly meant to do it!

that's pretty interesting. The compiler I was using did not give that warning when compiled with -gnatwa. You're right, that definitely looks intentional.

From: Simon Wright <simon@pushface.org>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 17:58:11 +0000

> [...]

```
>    procedure Test_Coextension_1 is
>       The_Thing : Thing_2(new Thing_1);
```

This is a case of 14.1/3, an allocator used to define the discriminant of an object,

```
>    begin
>       Put_Line("Coextension directly initialized");
>    end Test_Coextension_1;
>    function Make_Thing_2 return Thing_2 is
>    begin
>       return (Other => new Thing_1);
```

I think GNAT thinks this is a case of 14.2/3, an allocator used to define the constraint in a subtype_indication, though I'm hard put to it to see the difference from the first case.

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Thu, 20 Dec 2018 20:25:40 -0600

> This is a case of 14.1/3, an allocator used to define the discriminant of an object,

Right, because 14.2/3 says "subtype_indication in any other context", meaning that 14.1/3 applies in an object declaration.

> I think GNAT thinks this is a case of 14.2/3, an allocator used to define the constraint in a subtype_indication, though I'm hard put to it to see the difference from the first case.

That doesn't make any sense, since 14.2/3 is talking about a syntactic subtype_indication, and there is no subtype_indication in the above aggregate. 14.2/3 would be talking about a case like:

```
   function Make_Thing_3 return Thing_2 is
      subtype Silly is Thing_2 (new Thing_1);
      Some_Thing : Silly;
  begin
      return Some_Thing;
  end Make_Thing_3;
```

This function does NOT define a coextension.

So it does look like a GNAT bug. There is the possibility that they are associating the discriminant with the temporary object associated with the allocator, and not the return object, but that seems unnecessarily unfriendly of an interpretation. And it would be wrong for any type that requires built-in-place (I didn't look at the actual declaration of the type). I think the rules are supposed to prevent that interpretation, but whether they really do is an interesting question that I have no interest in exploring.

P.S. Did I mention I hate coextensions?? They provide an endless opportunity to puzzle over rules that really don't matter in the end (and most likely aren't quite right). I suppose they've helped me keep employed running the ARG. :-)

From: Jere <jhb.chat@gmail.com>
Subject: Re: Potential Coextension Bug in GNAT
Newsgroups: comp.lang.ada
Date: Fri, 21 Dec 2018 03:32:03 -0800

> So it does look like a GNAT bug. There is the possibility that they are associating the discriminant with the

temporary object associated with the allocator, and not the return object, but that seems unnecessarily unfriendly of an interpretation. And it would be wrong for any type that requires built-in-place (I didn't look at the actual declaration of the type). I think the rules are supposed to prevent that interpretation, but whether they really do is an interesting question that I have no interest in exploring.

Ok, that makes me feel better. I was concerned I was misinterpreting the RM about the function return (for build in place). The type was limited, which I believe requires it to be built in place.

> P.S. Did I mention I hate coextensions?? They provide an endless opportunity to puzzle over rules that really don't matter in the end (and most likely aren't quite right). I suppose they've helped me keep employed running the ARG. :-)

Overall, they aren't super useful and are not very intuitive. I don't know the history for why they were added to the language though. I will say they do provide one thing to Ada that no other feature in the language seems to, so there is that. But I don't know the cost versus reward of them.

grpexec Tool

From: VM Celier <vmcelier@gmail.com>
Subject: New tool "gprexec", basically "make with project file"
Newsgroups: comp.lang.ada
Date: Fri, 11 Jan 2019 14:00:10 -0800

I am starting a new project that I have been thinking for several years: gprexec.

gprexec is a "Make build automation tool using GPR project files to describe goals, dependencies, and processes".

It uses a new package: Execution.

Here is an example of a project that can be used by gprexec:

```
project Toto is
  for Main use ("toto.adb");
  package Execution is
    for Process ("display_main") use ("cat",
          "toto.adb");
    for Dependency ("display") use
          ("display_main");
    for Process ("display") use ("cat",
          "toto.gpr");
    for Process ("date") use ("date");
    for Process ("toto") use ("gprbuild", "-f",
          "-q", "toto.gpr");
    for Dependency ("default") use
          ("display", "toto", "date");
    for Process ("default") use ("toto");
  end Execution;
end Toto;
```

Package Execution has these attributes:

- Dependency, to indicate the goals that need to be processed before the indexed goal.
- Process, to indicate the process to be invoked, with its arguments, for the indexed goal.

gprexec needs to be invoked with a single project file and an optional goal. When no goal is specified on the command line, the goal "default" is implied.

For example with the project file toto.gpr above, invoking

```
gprexec toto.gpr
```

the goal default will be used, and according to the dependencies processes will be invoked in the following order:

(goal "display_main): cat toto.adb

(goal "display"): cat toto.gpr

(goal "toto"): gprbuild -f -q toto.gpr

(goal "date"): date

(goal "default"): toto

After displaying the main toto.adb and the project file toto.gpr, toto.adb is compiled, bound and linked, the date is displayed and the executable "toto" is invoked.

gprexec uses the project file "gpr.gpr", part of the gprbuild repository.

I just created a public repository for gprexec on Github:

https://github.com/vmcelier/gprexec

Anybody interested?

-- Vincent Celier

(no longer associated with AdaCore)

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: New tool "gprexec", basically "make with project file"
Newsgroups: comp.lang.ada
Date: Mon, 14 Jan 2019 13:06:35 -0800

> [...]

Yes, but no.

Some of the ideas behind GPR are good, but if we're being honest its tendency to be "stringly-typed" is annoying given its obvious designed similarity to Ada -- and there are a lot of missed opportunities -- and the sort-of configuration purposes which don't fully support producing an Ada executable (e.g. IIRC you have to use a completely separate configuration to handle DSA.)

From: VM Celier <vmcelier@gmail.com>
Subject: Re: New tool "gprexec", basically "make with project file"
Newsgroups: comp.lang.ada
Date: Mon, 14 Jan 2019 16:49:14 -0800

> Some of the ideas behind GPR are good, but if we're being honest its tendency to be "stringly-typed" is annoying given its obvious designed similarity to Ada

It is true that the syntax of the project language is similar to the one of Ada, but there is a big difference between the two languages:

- Ada is an executable language
- the project language is a declarative language

You don't "execute" project files, you use it to describe a system for different tools. This is why there are almost no types in the project language because types are not really needed and they would complexify the language for no real benefit.

> -- and there are a lot of missed opportunities

Could you tell us one or two of these missed opportunities?

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: New tool "gprexec", basically "make with project file"
Newsgroups: comp.lang.ada
Date: Tue, 15 Jan 2019 08:41:01 -0800

> It is true that the syntax of the project language is similar to the one of Ada, but there is a big difference between the two languages:

> - Ada is an executable language

> - the project language is a declarative language

This is actually less of an issue than might be thought; though some of the "fix-ups" might be a bit stifling to some. You could, for example, impose restrictions/mandatory-structure on the configuration and have all configurations be valid Ada.

> You don't "execute" project files, you use it to describe a system for different tools. This is why there are almost no types in the project language because types are not really needed and they would complexify the language for no real benefit.

No, real enumerations (and attendant Ada-like case-coverage) would be excellent for providing bounded alternations of the configuration.

> > -- and there are a lot of missed opportunities

> Could you tell us one or two of these missed opportunities?

Given Ada's strong generic-system configurations could be described as generic parameters [esp enumerations], which the tools could use to provide bounded options in the absence of defaults.

Package PROJECT_NAME

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: New tool "gprexec", basically "make with project file"
Newsgroups: comp.lang.ada
Date: Tue, 15 Jan 2019 09:22:07 -0800

Sorry, I accidentally submitted the form while composing my example... which is here:

```
Package PROJECT_NAME is
  Type Archetectures is ( x86, x86_64, ARM,
          SPARC, MIPS_V );
  Type Node_Type is (Storage, Processing);
  Type Partition_Type is (Active, Passive);
  Type Compilation_Parameters is record
     CPUs : Natural := 0; -- Use as many
                  -- cores as available.
     Symbols : Boolean := True; -- Don't strip
                             -- symbols.
     Target  : Archetectures;
      --...
  end record;

  Type Partition( Params :
          Compilation_Parameters; Style :
          Partition_Type ) is record
   null; --... Other DSA parameters.
  end record;

  Type Node( Style : Node_type ) is record
   Archetecture : Archetectures;
   case Style is
          when Storage =>    null; --...
          when Processing => null; --...
         end case;
  end record;

  Generic
   Params : Compilation_Parameters;
  Procedure Compile;

  --- CONCEPTUAL GENERIC PACKAGE
  Generic
   Partitions : Array (Positive range <>) of
not null access Partition;
  Package Compiler is
   Procedure Execute;
  End Compiler;

  --- CONCEPTUAL BODY FOR COMPILER
  Package Body Compiler is
   Procedure Execute is
         Begin
          For P of Partitions loop
           declare
                 Procedure Make is new
                 Compile( P.Params);
                  begin
                    Make;
                  end;
          End loop;
         End Execute;
  End Compiler;

End PROJECT_NAME;
```

Now, obviously there would have to be standardization -- and it would probably work better if "Archetectures" were a parameter to PROJECT_NAME -- because if all config-packages were generic we could "nest" dependencies:

```
Generic
 Type STANDARD_PARAM is limited
         private;
 -- "Configuration standard param"
 with Package P1 is new Project_1
         (STANDARD_PARAM );
 with Package P2 is new Project_2
         (STANDARD_PARAM );
 -- P3 depends on P1&2
 with Package P3 is new Project_3
         (STANDARD_PARAM, P1, P2 );
Package Project_4 is
 -- ... STANDARD STRUCTURE.
End Project_4;
```

Now, all of that is operating with the idea of using Ada as a config-language, which is doable, but perhaps a bit ugly... It might be a bit better to sit down, think about configurations (esp. in the presence of DSA) and develop an Ada-like language for that. (Perhaps in conjunction with a new Ada IR similar to DIANA, such that this configuration-description "compiles down to" the proper generic-nodes which can then be interpreted by the compiler as the configuration[s] to use; or processed by tools to inter-operate with current tools [ie IR → (GPR_File, Gnatdist_Configuration_File) for GNAT].)

Program entry in GPR

From: Jesper Quorning <jesper.quorning@gmail.com>
Subject: Package procedure as program entry in GPR project
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 07:12:22 -0800

Hello All,

With the package specification:

```
package My_Program_Package is
  procedure Program_Entry_Procedure;
end My_Program_Package;
```

How do i make Program_Entry_Procedure as the program entry procedure in a GPR project?

I think it is possible, but cannot find out how.

I know how to use a stand-alone procedure file as program entry and how to name the executable.

From: Jere <jhb.chat@gmail.com>
Subject: Re: Package procedure as program entry in GPR project
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 09:05:24 -0800

> [...]

With that specific setup, I am not sure. But if you are willing to change a couple of things you can do:

```
-- my_program_package.ads
package My_Program_Package is
  -- Notice no declaration here for the
  -- procedure, but you can put other
  -- things if you like
end My_Program_Package;

-- my_program_package-
program_entry_procedure.adb
procedure My_Program_Package.
          Program_Entry_Procedure is
begin
  -- your main stuff
end My_Program_Package.
Program_Entry_Procedure;
```

Then you modify the GPR file to point to it as the main:

```
for Main use ("my_program_package-
program_entry_procedure.adb");
```

I do something similar for my Gnoga GUI projects so I can have program level stuff in the top package but have the main a child of that top level package.

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Package procedure as program entry in GPR project
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 15:42:12 -0600

> [...]

I realize you are asking for GPR, so by definition you don't care about portability, but:

Ada only requires Ada implementations to support library-level procedures as the main. See 10.2(29). A particular implementation can allow more, but there is no requirement.

So if you ever might want to use some other Ada compiler (I for one, hope so), use such a routine.

It's trivial to write one, after all:

```
with My_Program_Package;
procedure My_Program_Main is
begin
      My_Program_Package.
           Program_Entry_Procedure;
end My_Program_Main;
```

From: Jesper Quorning <jesper.quorning@gmail.com>
Subject: Re: Package procedure as program entry in GPR project
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 17:47:30 -0800

I just wanted a way to avoid the trivial main file.

I also considered

```
package simple is
  procedure main
end simple;

package body simple is
  procedure main is
  begin
   ...
  end main;
private
  main;
end simple;
```

But GPR would not do that either. I will stick to the simple procedure file.

From: Simon Wright <simon@pushface.org>
Subject: Re: Package procedure as program entry in GPR project
Newsgroups: comp.lang.ada
Date: Sat, 26 Jan 2019 12:05:35 +0000

> [...]

This isn't a GPR thing, it's a GNAT thing: GNAT has no extensions here beyond the requirement.

If you have a minimal bare-board project without any requirement for the Ada runtime system, it's possible to do what you ask: see Maciej Sobczak's 'Ada and SPARK on ARM Cortex-M' tutorial[1], in particular the 'First Chapter'[2].

It would be hard (and pointless) to attempt this for a program intended to run on a typical operating system.

[1] http://www.inspirel.com/articles/Ada_On_Cortex.html

[2] http://www.inspirel.com/articles/Ada_On_Cortex_First_Program.html

GNAT Bug

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 06:51:50 -0800

This will probably sound more like venting frustration. Sorry if so. But how does anybody get anything done? gnat is *the major* Ada compiler and pretty much the only one implementing the standard in full. Yet I cannot seem to get it working past really small size in any project. As soon as I try to get any basic type composition done (only 3-4 inheritance levels, with, perhaps double interface overlay), I get that dreaded gnat bug message.. This is at least the 3rd one just within past week or two..

Specifically this:

https://github.com/gerr135/wann/tree/gnat_bug01

(the bug triggering code is in a separate branch pointed to by that link).

This is still early in design phase and far from being functional in any way, so I don't really expect much comments on the code itself (thus that "venting frustration" comment above). But the pattern that seems to universally trigger these gnat bugs is something along these lines:

```
type Base_Interface is interface;
..

type Derived1_Interface is new
Base_Interface and ..;
..
```

perhaps few more layers here..

then

```
type Base_impl1 is new Base_Interface with
private;
..
type Derived1 is new Base_impl1 and
Derived1_Interface with private..
```

basically trying to stitch together functional interface hierarchy (containing algorithmic stuff) and data storage type hierarchy. Somehow gnat very often just cannot handle this type of design :(.

(and yes, I am avoiding having to lay generics on top of other generics like Dmitry suggests - keeps design and compilation times sane, but apparently overloads gnat capacity to deal with abstraction).

So, I guess my question would be - how people deal with such situations (combining algorithmic and data representation type hierarchies) in their experience? Or, whether too many child modules makes any difference? I seem to have noticed that the more hierarchical my packages are (but this one is only like 3rd level child!) the more often I trigger that gnat bug message.. (but then keeping the code in one huge module is really messy too!)

And yeah, the specific message here is:

```
gprbuild -P wann.gpr
Compile
   [Ada]          run_customnn.adb
+===GNAT BUG DETECTE===+
| Community 2018 (20180524-73) (x86_64-pc-linux-gnu) GCC error: |
| in gnat_to_gnu_entity, at ada/gcc-interface/decl.c:429 |
| Error detected at wann-nets-vectors.ads:106:5 [run_customnn.adb:23:5]    |
| Please submit a bug report by email to report@adacore.com.  |
| GAP members can alternatively use GNAT Tracker:   |
| http://www.adacore.com/ section 'send a report'.  |
| See gnatinfo.txt for full info on procedure for submitting bugs.         |
| Use a subject line meaningful to you and us to track the bug.           |
| Include the entire contents of this bug box in the report.             |
| Include the exact command that you entered.                 |
| Also include sources listed below.
| Use plain ASCII or MIME attachment(s).                    |
+======================+
```

and the "please include" list of files lists pretty much all of them in the src dir.

But as I said, this is rather a pattern I observe, not just one-off situation..

This is with the latest FSF gnat compiler (2018 release based on gcc-7.3.1 backend, Gentoo Linux, relatively fresh everything else).

Sigh, I guess another report to file with AdaCore..

Sorry for disturbance here..

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 19:47:31 +0100

> So, I guess my question would be - how people deal with such situations (combining algorithmic and data representation type hierarchies) in their experience? Or, whether too many child modules makes any difference? I seem to have noticed that the more hierarchical my packages are (but this one is only like 3rd level child!) the more often I trigger that gnat bug message.

Do not panic. In many cases the bug is triggered by an illegal program. Try an older version of GNAT compiler to find what triggers it. In other cases you can work around it using minor code variations.

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 13:32:52 -0800

> [...]

Oh, I am far from panic. It is, as I mentioned, already like 3rd project where I trigger a similar bug in the space of a week or two. Just, when you finally laid out thing just the way you wanted and then gnat explodes on that final compile attempt. Then you get such an expression of frustration :).

Thanks for the advice though! This is pretty much how I handle these. But nice to know I am not alone in this. Well, in fact not so nice - would be nicer if this never happened of course :). But at least reassuring. So thanks again.

From: Simon Wright <simon@pushface.org>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 01 Feb 2019 20:41:06 +0000

```
> gprbuild -P wann.gpr
> Compile
>    [Ada]          run_customnn.adb
> +===GNAT BUG DETECTE===+
> | Community 2018 (20180524-73) (x86_64-pc-linux-gnu) GCC error: |
> | in gnat_to_gnu_entity, at ada/gcc-interface/decl.c:429 |
> | Error detected at wann-nets-vectors.ads:106:5 [run_customnn.adb:23:5]    |
```

but I get

```
$ gprbuild -p -P wann
wann.gpr:5:32: "../../libs/ada_common/src" is not a valid directory
gprbuild: "wann" processing failed
```

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 13:26:27 -0800

Oops, that's a stale import of an extra lib I thought to use at one point but then rolled back. Apparently I forgot to remove the path, and I obviously still have that lib on my system, even if it is not withed any more.

Removed, you should be able to proceed now. Sorry about that.

One other note: at first build the compiler may complain about missing obj/dbg dir. Please just run:

```
mkdir -p obj/dbg
```

from the project dir (not src, one level above it).

I have obj/ in .gitignore to prevent it tracking generated files (and git tends to ignore the entire dir, not just its contents. At least my very short attempts to force it to ignore obj/* but not obj/ itself did not succeed. I preferred the annoyance of running once the mkdir command over spending more time trying to beat git when I set it up).

Thanks for your attempt!

From: Simon Wright <simon@pushface.org>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 01 Feb 2019 23:17:17 +0000

OK, and all the compilers I have here fail in the same way:

FSF GCC 6, 7, 8, 9

GNAT 2016, 2017, 2018

For GCC 9, the relevant code in decl.c is

```
  /* If we get here, it means we have not yet done anything with this entity. If we are not defining it, it must be a type or an entity that is defined   elsewhere or externally, otherwise we should have defined it already. */

  gcc_assert (definition
      || type_annotate_only
      || is_type
      || kind == E_Discriminant
      || kind == E_Component
      || kind == E_Label
      || (kind == E_Constant &&
         Present (Full_View (gnat_entity))
             || Is_Public (gnat_entity));
```

... and we are none the wiser.

I tried

```
  gprbuild -p -P wann.gpr -c -u -f wann-nets-vectors.adb
```

and it compiled OK except for loads of 'unimplemented' warnings.

Poking around at your main program, it seems that things go wrong at the line

```
   package PNetV  is new PNet.vectors;
```

(i.e., I deleted stuff starting at the bottom, by the time I'd deleted this line it compiled "OK".

> One other note: at first build the compiler may complain about missing
> obj/dbg dir. Please just run:
> mkdir -p obj/dbg
> from the project dir (not src, one level above it).

'gprbuild -p' will create missing directories.

Or you could add

```
  for Create_Missing_Dirs use "true";
```

to your GPR (recent ones only).

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Fri, 1 Feb 2019 23:16:32 -0800

> [...]

> I tried

>   gprbuild -p -P wann.gpr -c -u -f wann-nets-vectors.adb

> and it compiled OK except for loads of 'unimplemented' warnings.

Ok, so the file itself compiles (I gotta read up on all those switches apparently. This is a ways to quickly test stuff. Thanks for a suggestion!)

But that is quite what I expect, given the nature of the bugs I get - they clearly come from gnat getting lost in all the inheritances I throw at it.

> Poking around at your main program, it seems that things go wrong at the line

The specific offending lines are:

wann-nets-vectors.ads:104 and 106

these two full type definitions (if I comment out one it still fails on the other):

```
   type Cached_Proto_NNet is abstract new Proto_NNet and Cached_NNet_Interface with null record;

   type Cached_Checked_Proto_NNet is abstract new Proto_NNet and Cached_Checked_NNet_Interface with null record;
```

These are null record at the moment, as I did not yet get around to properly implement them. Just placeholders essentially. And this is what might be confusing gnat I suspect. I did not yet try to add any actual data inside.

> 'gprbuild -p' will create missing directories.

> Or you could add

Thanks, I'll add this too.

A small note: I will be at the Fosdem most of today and possibly tomorrow. So, I may not be able to reply in a timely manner these two days.

(But I will surely pass by the Ada dev room today!)

From: Per Sandberg <per.s.sandberg@bahnhof.se>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Sat, 2 Feb 2019 08:13:02 +0100

I did put some effort to reduce the problem and the workaround is quite simple, in file "wann-nets.ads:69" mark the procedure Del_Neuron as abstract instead of null.

Here is the small reproducible I ended up with after stripping the code:

```
pragma Warnings (Off);
generic
   type Real is digits <>;
package wann is
end Wann;
--
generic
package Wann.Neurons is
end Wann.Neurons;
---
generic
package Wann.Nets is
  type NNet_Interface is limited interface;
  procedure Del (Net : in out
            NNet_Interface) is null;
  --  Fails
  -- procedure Del (Net : in out
  -- NNet_Interface) is abstract;--  Works
  type Cached_NNet_Interface is limited
interface and NNet_Interface
end Wann.Nets;
--
generic
package wann.nets.vectors is
   type Proto_NNet is abstract new
            NNet_Interface with NULL record;
   type Cached_Proto_NNet is abstract new
      Proto_NNet and
     Cached_NNet_Interface with null record;
end wann.nets.vectors;
--
pragma Warnings (Off);
with wann.nets.vectors;
procedure run_customNN is
   package PW is new wann(Real => Float);
   package PNet   is new PW.nets;
   package PNetV  is new PNet.vectors;
begin
  null;
end Run_CustomNN;
```

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Sat, 2 Feb 2019 11:05:19 -0800

Wow, thank you for your time!

Looking at how that final code is so small and basic, and that snippet of gnat internals that was dug out on another comment above, it looks like gnat does not implement null primitives in full.. (which is a pity, as null method makes more sense there than abstract, but well..)

Once I am completely back from Fosdem I'll play with this a bit more, to see if that's package hierarchy, generics or combination thereof that is triggering it and submit a bug with final details.

Thanks again!

From: Per Sandberg <per.s.sandberg@bahnhof.se>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Sat, 2 Feb 2019 22:37:01 +0100

Well I think it's more about deeply nested generics, since that is a real nightmare to implement in its full context.

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Mon, 4 Feb 2019 04:28:45 -0800

Not exactly as far as I can tell.

I have played some more with the code and could simplify it even more - there is no need for that extra top package level. Same thing happens if the interfaces are declared at the top, and overridden in a child. Flat package structure (still generic) compiles fine. Removing generics (and instead doing "type Real is new Float" at the top) given unstable behavior - one time I got the same bug triggered, but after I renamed sources (originally names "workaround" to "alternative" to reflect better the situation) gnat started to compile it properly (giving error message about declaring vars of abstract type). Apparently it has a sense of humor - this is literally the situation of "what is written here is a lie").

Anyway, I have created a github project to keep the code producing gnat bugs I have so far encountered (only one at the moment, but there are two more I need to clean-up and report). This project shows the code triggering the bug, as well as workarounds and the status of the bug report. I think such a resource would be rather useful (given that AdaCore themselves don't really support the bug tracker, at least for the community version [1]). So, please feel free to consult or even contribute, if there are any more commonly encountered bugs.

The project can be found here:

https://github.com/gerr135/gnat_bugs

[1] I chatted with them briefly 2 days ago on Fosdem and they told me that they prefer an email report and that tracker is not really functional for a community version at least.

From: joakimds@kth.se
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Mon, 4 Feb 2019 07:30:30 -0800

George, thanks for your efforts in making detailed gnat bug reports and your input in the Ada dev room on Fosdem 2019.

From: Simon Wright <simon@pushface.org>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Mon, 04 Feb 2019 16:11:47 +0000

> I chatted with them briefly 2 days ago on FOSDEM and they told me that they prefer an email report and that tracker is not really functional for a community version at least.

Do you mean the GCC Bugzilla? I can quite understand why reports against just GNAT CE wouldn't really be appropriate there.

AdaCore do respond to reports on FSF GCC there, especially if the report is about the GCC build system or about bad code generation. However, old bugs don't really get curated as they are fixed in new releases.

This doesn't work where the sources concerned aren't publicly visible in the repository: for example, the embedded runtimes.

Personally I like to report on Bugzilla where appropriate, because reports to report@adacore.com aren't publicly visible. I don't know how annoying it'd be to report in both places.

From: George Shapovalov <gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Tue, 5 Feb 2019 11:16:51 -0800

> Do you mean the GCC Bugzilla? I can quite understand why reports against just GNAT CE wouldn't really be appropriate there.

No, I meant the tracker mentioned on the bug message:

>GAP members can alternatively use GNAT Tracker: |

>| http://www.adacore.com/ section 'send a report'.

From his reaction I took it that that tracker is not that active. Although it would not be so useful for many people anyway, if it has usage limitations.

> AdaCore do respond to reports on FSF GCC there, especially if the report is about the GCC build system or about bad code generation.

Oh, they do? Thanks for the info!

That's not something I directly thought about, as the problem is with the upstream (of FSF), so it makes sense to take it directly to upstream (the most common reaction of many projects and distributions is to first try to figure out if its them or upstream, and if its upstream, then its universally - "report it to upstream". Which is totally logical, in avoiding messy duplication of effort. In fact it is often not something they would even have control over).

So, I just took it directly to upstream, strictly following the procedure described in the bug message :).

> Personally I like to report on Bugzilla where appropriate, because reports to report@adacore.com aren't publicly visible. I don't know how annoying it'd be to report in both places.

Yes, that's indeed a concern. This is why I created that github project, as I had a few bugs lying around already. I'll populate it with more when I get around to it.

But to the credit of AdaCore, they react quickly - I already got a confirmation that they got it and will look into it..

From: Simon Wright <simon@pushface.org>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Tue, 05 Feb 2019 20:37:16 +0000

> [...]

> From his reaction I took it that that tracker is not that active. Although it would not be so useful for many people anyway, if it has usage limitations.

If you have a contract with AdaCore then Tracker is the point of contact; and the response when I worked for a company with a contract was terrific.

If not, your only direct contact is report@adacore.com (with GNAT in the subject line).

> That's not something I directly thought about, as the problem is with the upstream (of FSF), so it makes sense to take it directly to upstream (the most common reaction of many projects and distributions is to first try to figure out if its them or upstream, and if its upstream, then its universally - "report it to upstream". Which is totally logical, in avoiding messy duplication of effort. In fact it is often not something they would even have control over). So, I just took it directly to upstream, strictly following the procedure described in the bug message :).

The AdaCore people working on FSF GCC are the same people working on the 'upstream' product, which is why I've never thought of it like that; but

I see your point.

And, I've occasionally added 'same problem with GNAT CE' to Bugzilla reports where I thought it might stimulate interest.

> [...]

> But to the credit of AdaCore, they react quickly - I already got a confirmation that they got it and will look into itIt helps if they know you!

From: George Shapovalov
<gshapovalov@gmail.com>
Subject: Re: Yet another gnat bug
Newsgroups: comp.lang.ada
Date: Wed, 6 Feb 2019 02:53:18 -0800

> The AdaCore people working on FSF GCC are the same people working on the 'upstream' product, which is why I've never thought of it like that; but

> I see your point.

Oh, so they do have people working on gcc directly? Nice!

Sure, that makes total sense (for a company that essentially sells a gcc-based compiler). But unfortunately this rarely happens in reality.

AdaCore seems like a real nice company! (A bit of praise never hearts, but seriously, thanks to AdaCore people for nice work overall!)

> > But to the credit of AdaCore, they react quickly - I already got a confirmation that they got it and will look into it.

>

> It helps if they know you!

Maybe, but then I only saw them once in a person, and that likely were other people.

But more importantly, this particular issue seems to be a general omission affecting gnat universally, which would affect all kinds of users. I am just puzzled how this thing was not triggered before by at least some users? Is nobody fond of trying to lay out their types in the most abstract way possible? That *does* force better design and ends up saving quite a bit of work down the road (to the point of coding becoming really boring after the general structure is in and successfully compiled by gnat). Well, I guess people just always write "is abstract" even where "is null" would make more sense (or that not many people mix generics and OOP abstraction)..

Alignment issue

From: Simon Wright
<simon@pushface.org>
Subject: Alignment issue
Newsgroups: comp.lang.ada
Date: Sat, 16 Feb 2019 19:40:38 +0000

I have code like this (written while working on a StackOverflow question), and GNAT ignores apparent alignment requests.

```
  with System.Storage_Pools;
  with System.Storage_Elements;
  package Alignment_Issue is

     type Data_Store is new
System.Storage_Elements.Storage_Array
     with Alignment => 16;  --
Standard'Maximum_Alignment;

     type User_Pool (Size :
System.Storage_Elements.Storage_Count)
        is  record
          Flag        : Boolean;
          Data        : Data_Store (1 .. Size);
        end record
     with Alignment => 16;  --
Standard'Maximum_Alignment;

  end Alignment_Issue;
```

(Standard'Maximum_Alignment is a GNAT special) and compiling with GNAT CE 2018 (and other GNAT compilers) I see

```
 $ /opt/gnat-ce-2018/bin/gnatmake -c -u
-f -gnatR alignment_issue.ads

 gcc -c -gnatR alignment_issue.ads

 Representation information for unit
Alignment_Issue (spec)

  for Data_Store'Alignment use 16;
  for Data_Store'Component_Size use 8;

  for User_Pool'Object_Size use ??;
  for User_Pool'Value_Size use ??;
  for User_Pool'Alignment use 16;
  for User_Pool use record
     Size at 0 range  0 .. 63;
     Flag at 8 range  0 ..  7;
     Data at 9 range  0 .. ??;
  end record;
```

which means that GNAT has ignored the alignment specified for Data_Store when setting up User_Pool.Data.

Is this expected? OK?

I found a workround of sorts:

```
  type Data_Store (Size :
System.Storage_Elements.Storage_Count)
is record
     Data :
System.Storage_Elements.Storage_Array (1
.. Size);
  end record
  with Alignment => 16;  --
Standard'Maximum_Alignment;

  type User_Pool (Size :
System.Storage_Elements.Storage_Count)
     is record
       Flag  : Boolean;
       Stack : Data_Store (Size);
     end record;
```

giving

```
  Representation information for unit
Alignment_Issue (spec)

  for Data_Store'Object_Size use ??;
  for Data_Store'Value_Size use ??;
  for Data_Store'Alignment use 16;
  for Data_Store use record
     Size at 0 range  0 .. 63;
     Data at 8 range  0 .. ??;
  end record;

  for User_Pool'Object_Size use ??;
  for User_Pool'Value_Size use ??;
  for User_Pool'Alignment use 16;
  for User_Pool use record
     Size  at  0 range  0 .. 63;
     Flag  at  8 range  0 ..  7;
     Stack at 16 range  0 .. ??;
  end record;
```

(but even then I see that Stack.Data is offset by 8 bytes because of the discriminant)

From: "Randy Brukardt"
<randy@rrsoftware.com>
Subject: Re: Alignment issue
Newsgroups: comp.lang.ada
Date: Mon, 18 Feb 2019 17:01:02 -0600

>I have code like this (written while working on a StackOverflow question), and GNAT ignores apparent alignment requests.

I wouldn't have expected Alignment to cause the effect, but when you specify representation for a record type, any requirements on the components are can be ignored. Perhaps GNAT is taking that somewhat too far??

Ada in Context

Create Attributes

From: eduardsapotski@gmail.com
Subject: Create attributes.
Newsgroups: comp.lang.ada
Date: Fri, 21 Dec 2018 21:37:12 -0800

Sorry for the stupid question...

For example. I have type:

```
  type Person is record
     First_Name : Unbounded_String :=
            Null_Unbounded_String;
     Last_Name : Unbounded_String :=
            Null_Unbounded_String;
  end record;
```

There is a list:

```
  package People_Package is new
Ada.Containers.Vectors(Natural, Person);
  People : People_Package.Vector;
```

Next, I want to display this list with headers:

```
----------------------------
|   NAME     |   SURNAME     |
----------------------------
|   John     |    Smith      |
|   Ada      |   Lovelace    |
...
----------------------------
```

Can I use attributes to display headers?

For example something like this:

People'First_Name_Header

How can this be implemented?

From: Brad Moore
<bmoore.ada@gmail.com>
Subject: Re: Create attributes.
Newsgroups: comp.lang.ada
Date: Sat, 22 Dec 2018 11:13:40 -0800

You could use a class-wide type or a type with discriminants such as;

```
  type Person_Attribute_Kinds is (Name,
Surname);
     type Person_Attribute (Attribute_Name :
Person_Attribute_Kinds
                      :=
Person_Attribute_Kinds'First) is
     record
        case Attribute_Name is
          when Name | Surname =>
             Name_String : Unbounded_String
:= Null_Unbounded_String;
        end case;
     end record;

     type Person is
        record
          First_Name :
Person_Attribute(Name);
          Last_Name  :
Person_Attribute(Surname);
        end record;

   X : Person;
begin
   Put_Line ("| " &
X.First_Name.Attribute_Name'Image &
            " | " &
X.Last_Name.Attribute_Name'Image & " |");
```

Overloading operators

From: daicrkk@googlemail.com
Subject: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Fri, 11 Jan 2019 13:46:22 -0800

I am working my way through Barnes' excellent Ada book. This is a code sample for deep comparison of linked lists from section 11.7:

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
function "=" (L, R: access Cell) return
Boolean is
begin
  if L = null or R = null then    -- universal =
    return L = R;                 -- universal = (Line
A)
  elsif L.Value = R.Value then
    return L.Next = R.Next;       -- recurses OK
(Line B)
  else
    return False;
  end if;
end "=";
```

I can't seem to wrap my head around why in Line A operator "=" of the universal_access type is called (because of the preference rule), on Line B, however, the user-defined operator "=" is called (which makes recursion possible in the first place), this time with no preference for operator "=" of universal_access.

Both L and R, as well as L.Next and R.Next are of the same anonymous type "access Cell". Why the difference in "dispatching"? Does it have to do with L and R being access parameters? If so, what is the rule there?

I did my best to find anything in the AARM, especially section 4.5.2, but could not make any sense of it.

From: Simon Wright <simon@pushface.org>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Sat, 12 Jan 2019 09:50:14 +0000

Given ARM 4.5.2(9.1 ff),

At least one of the operands of an equality operator for universal_access shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package Standard, neither operand shall be of an access-to-object type whose designated type is D or D'Class, where D has a user-defined primitive equality operator such that:

* its result type is Boolean;

* it is declared immediately within the same declaration list as D or any partial or incomplete view of D; and

* at least one of its operands is an access parameter with designated type D.

I'm not at all clear why the example code is legal, or why it would be legal to call it; since 'access Cell' appears to match "neither operand shall be of an access-to-object type whose designated type is D or D'Class, where D has a user-defined primitive equality operator ..."

Might explain why compiling this example with GNAT (CE 2018) results in stack overflow.

From: Simon Wright <simon@pushface.org>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Sat, 12 Jan 2019 14:01:43 +0000

> I'm not at all clear why the example code is legal, or why it would be legal to call it; since 'access Cell' appears to match "neither operand shall be of an access-to-object type whose designated type is D or D'Class, where D has a user-defined primitive equality operator ..."

Still not clear.

Note to self: do *not* attempt to define "=" for anonymous access types!

Would have liked the AIs to have said "it is illegal to define "=" for anonymous access types".

From: daicrkk@googlemail.com
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Sat, 12 Jan 2019 07:15:38 -0800

> [...]

> I'm not at all clear why the example code is legal, or why it would be legal to call it; since 'access Cell' appears to match "neither operand shall be of an access-to-object type whose designated type is D or D'Class, where D has a user-defined primitive equality operator ..."

I second that. Access Cell is an access-to-object type whose designated type is Cell (check), Cell has a user-defined primitive equality operator (check) such that its result type is Boolean (check), it is declared immediately within the same declaration list as Cell (check), at least one of its operands is an access parameter with designated type Cell (both operands are, check).

According to 4.5.2, universal_access "=" should never be allowed to kick in at all here, not even with "L = null". Or am I missing something?

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Mon, 14 Jan 2019 17:08:32 -0600

>I second that. Access Cell is an access-to-object type whose designated type is Cell (check), Cell has a user-defined primitive equality operator (check) such that its result type is Boolean (check), it is declared immediately within the same declaration list as Cell (check), at least one of its operands is an access parameter with designated type Cell (both operands are, check).

>According to 4.5.2, universal_access "=" should never be allowed to kick in at all here, not even with "L = null". Or am I missing something?

Yup, I agree with this. My first thought when reading that example is that it is wrong, because I don't remember anywhere in Ada where the same operator with arguments of the same type means different things. I don't think the use of "null" could change that.

Dunno if John wrote that for a different version of Ada, or he was just confused by a rule that barely makes sense anyway.

As always, best avoid anonymous access types unless you have to use one of their special features (dynamic accessibility, dispatching, special discriminant accessibility, or closures [for access-to-subprograms]). And better still, lets lobby to get those special features optionally available for named access types so no one ever has to use an anonymous anything. :-)

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Mon, 14 Jan 2019 16:34:42 -0800

> As always, best avoid anonymous access types unless you have to use one of their special features (dynamic accessibility, dispatching, special discriminant accessibility, or closures [for access-to-subprograms]). And better still, lets lobby to get those special features optionally available for named access types so no one ever has to use an anonymous anything. :-)

Well, I'm all for getting rid of anonymous access types altogether -- though that might not be acceptable to the rest of the ARG as it would make previously-valid Ada non-valid, I think reducing the complexity of the language (and reduce instances of "a rule that barely makes sense anyway").

I thought there was an AI for first class subprograms / subprogram types, but I couldn't find it with a quick search... so either I'm misremembering or I'm just hitting all the wrong keywords in the search.

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Tue, 15 Jan 2019 09:38:11 +0100

> Yup, I agree with this. My first thought when reading that example is that it is wrong, because I don't remember anywhere in Ada where the same operator with arguments of the same type means different things. I don't think the use of "null" could change that.

But the types are not same. It is universal_access vs. access.

> Dunno if John wrote that for a different version of Ada, or he was just confused by a rule that barely makes sense anyway.

> As always, best avoid anonymous access types unless you have to use one of their special features (dynamic accessibility, dispatching, special discriminant accessibility, or closures [for access-to-subprograms]). And better still, lets lobby to get those special features optionally available for named access types so no one ever has to use an anonymous anything. :-)

Named or anonymous it makes little difference, IMO.

Here is a classic multi-method case. "=" is such an operation. null is universal_access (4.2). For any access type P there are 3 equality operations "=":

```
function "=" (Left, Right : universal_access)
return Boolean;
  type P is access T;
  function "=" (Left : P; Right :
universal_access) return Boolean;
  function "=" (Left : universal_access; Right
: P) return Boolean;
  function "=" (Left, Right : P) return
Boolean;
```

When the last one is overridden, what happens with the second and the third?

One of three possibilities:

1. It inherits the base operation:

```
  function "=" (Left : P; Right :
universal_access) return Boolean is
  begin
    return universal_access (Left) = Right;
  end "=";
```

2. It silently overrides:

```
  function "=" (Left : P; Right :
universal_access) return Boolean is
  begin
    return Left = P (Right);
  end "=";
```

3. It gets overridden abstract and comparison to null becomes illegal because the operation is not defined.

[The reference manual is shy to say anything about it. It claims that universal_access is kind of class-wide, which would mean, if taken seriously, that "=" overloads and must clash with the original "=". Since it does not, universal_access is more like a parent type than class-wide.]

It seems that in the OP's case as in the case with named access types #2 is in effect, which is illogical, inconsistent, unsafe, but would be expected by most people.

Barnes' code presumes rather #1, which is logical, but confusing and error-prone.

#3 would be consistent and safe:

if Ptr_Value = Ptr_Type (null) then -- Type conversion required

But it would not work with anonymous access types. So, if #3 were adopted, then overriding for anonymous types must be banished.

All this is fine and good, except that overriding

function "=" (Left, Right : access T) return Boolean;

is also a primitive of T! You cannot banish it.

P.S. And, wouldn't it be better to fix the type system, no?

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Overloading operator "=" for anonymous access types?
Newsgroups: comp.lang.ada
Date: Tue, 15 Jan 2019 15:00:31 -0600

> [The reference manual is shy to say anything about it. It claims that universal_access is kind of class-wide, which would mean, if taken seriously, that "=" overloads and must clash with the original "=".

This is what happens. However, such a clash would mean that you could never write a user-defined "=" for an anonymous access type. That would have been a good idea, but it would have to have been enforced with a Legality Rule to be sensible. Some thought that bad because of compatibility, so...

> Since it does not, universal_access is more like a parent type than class-wide.]

...there is a hack to have a preference for the user-defined one. That doesn't change the fact that universal_access is class-wide, it just make it possible to write a user-defined operator.

>P.S. And, wouldn't it be better to fix the type system, no?

This wart would be one of the things that would make "fixing the type system" so much harder. A proper solution (and the one we should have used in the first place) is to declare a "=" for every access type. I think we wanted to avoid that as anonymous access can be declared in places where declarations aren't allowed, so we came up with something worse. :-)

It's the idea of anonymous access types that destroys the type system that you have in mind. Your system keeps the types and operations together, and that makes no sense for an anonymous type (what are the operations for an anonymous type, and where are they declared? Any answer is either magical or nonsense.)

One has to get rid of nonsense things before one could regularize the type system, especially upon the lines you have been suggesting for years. It's not really possible for Ada; you would end up with an Ada-like language.

This is just another Ada

Return types

From: danielcheagle@gmail.com
Subject: ? Is ok return a type derived from ada.finalization.controlled from a "Pure_Function" ? thanks.
Newsgroups: comp.lang.ada
Date: Thu, 24 Jan 2019 15:56:10 -0800

Is ok return a type derived from ada.finalization.controlled from a function declared "Pure_Function" ?

Or yet, is ok declare a fuction returning a controlled type as "pure_function" ?

Thanks in Advance!!!

note1 : the type has a access value.

note2 : initialize, adjust and finalize overrided and working :-)

fragment example code:

```
---------------------------------
pragma Ada_2012;
pragma Detect_Blocking;

with Ada.Finalization;

package Arbitrary
 with preelaborate
is

  type Arbitrary_Type (size : Positive) is
   new Ada.Finalization.Controlled with
private;

  function To_Arbitrary (value : Integer;
          precision : Integer)
   return Arbitrary_Type
    with inline; -- Can I add "pure_function" ?

private

 type Mantissa_Type is array (Positive
          range <>) of Integer;
 type Mantissa_Pointer is access
          Mantissa_Type;

 type Arbitrary_Type (size : Positive) is
  new Ada.Finalization.Controlled with
record
    mantissa   : Mantissa_Pointer;
    exponent   : Integer;
    sign       : Integer range -1 .. 1;
    precision  : Positive := size;
  end record;

end arbitrary;

----------------------------------------------


pragma Ada_2012;
pragma Detect_Blocking;

with Ada.Unchecked_Deallocation;

package body Arbitrary is

 procedure Delete is new
          Ada.Unchecked_Deallocation
          (Mantissa_Type,
            Mantissa_Pointer);
 ---------------------------------------------------------
 -- Initialize an Arbitrary_Type
 ---------------------------------------------------------
 procedure Initialize (Object : in out
          Arbitrary_Type) is
 begin
  Object.mantissa := new Mantissa_Type
          (1 .. Object.precision);
  Object.exponent    := 0;
  Object.sign        := 1;
  -- "here" for diminish race condition from
  -- OS' s
  Object.mantissa.all := (others => 0);
 end Initialize;

 ---------------------------------------------------------
 -- Fix an Arbitrary_Type after being  --
 -- assigned a value
 ---------------------------------------------------------
 procedure Adjust (Object : in out
          Arbitrary_Type) is
 begin
  Object.mantissa := new
       Mantissa_Type'(Object.mantissa.all);
 end Adjust;

 ---------------------------------------------------------
 -- Release an Arbitrary_Type;
 ---------------------------------------------------------
 procedure Finalize (Object : in out
          Arbitrary_Type) is
 begin
  if Object.mantissa /= null then
   Delete (Object.mantissa);
  end if;
  Object.mantissa := null;
 end Finalize;

 ---------------------------------------------------------
 -- Convert an Integer type to an
 -- Arbitrary_Type
 ---------------------------------------------------------
 function To_Arbitrary (value : Integer;
          precision : Integer)
  return Arbitrary_Type is
  result   : Arbitrary_Type (precision);
 begin
  result.mantissa (result.exponent + 1) :=
          value;
  Normalize (result);
  return result;
 end To_Arbitrary;

end arbitrary;
```

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: ? Is ok return a type derived from ada.finalization.controlled from a "Pure_Function" ? thanks.
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 15:20:47 -0600

Of course it's OK, "Pure_Function" is some GNAT-specific nonsense. :-)

My recollection is that GNAT does not check if Pure_Function makes sense, so the only question is whether you can live with the possible implications. (And I don't know why you would want to use Pure_Function anyway.)

Note that in Ada 2020, you would use the Global aspect to declare the usage of globals by your subprogram, and those are checked, so either the aspect is legal or your program won't compile. But GNAT hasn't implemented that yet, so far as I know.

From: Shark8 <onewingedshark@gmail.com>
Subject: Re: ? Is ok return a type derived from ada.finalization.controlled from a "Pure_Function" ? thanks.
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 16:22:33 -0800

IIRC, Pure_Function doesn't need to be in a Pure unit to be tagged as such, and the GNAT-specific meaning is: given a call with a particular set of parameter-values always returns the same result.

As I recall GNAT doesn't actually check this is case, but rather uses it for optimization purposes.

> Or yet, is ok declare a function returning a controlled type as "pure_function" ?

See above: "Pure_Function" has nothing to do with categorization or restrictions and is just an attribute denoting allowance for certain optimizations. (Again, IIRC.)

From: Simon Wright <simon@pushface.org>
Subject: Re: ? Is ok return a type derived from ada.finalization.controlled from a "Pure_Function" ? thanks.
Newsgroups: comp.lang.ada
Date: Sat, 26 Jan 2019 11:48:46 +0000

Given that the documentation of Pure_Function[1] says

... the compiler can assume that there are no side effects, and in particular that two calls with identical arguments produce the same result

and that

... there are no static checks to try to ensure that this promise is met

it would be a Bad Idea to apply it to your function.

[1] https://gcc.gnu.org/onlinedocs/gnat_rm/Pragma-Pure_005fFunction.html

Forbid local generic instantiations

From: joakimds@kth.se
Subject: Why forbid local generic instantiations?
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 01:43:29 -0800

[...]

Consider the following code:

```
procedure Main is
  package Integer_Vectors is new
Ada.Containers.Vectors (Positive, Integer);
begin
  null;
end Main;
```

It has a generic package instantiation local to the subprogram Main and not defined on package level. Both in AdaControl and GNATCheck there are rules to forbid local generic instantiations.

For example GNATCheck:

23.7.25 Generics_In_Subprograms

Flag each declaration of a generic unit in a subprogram. Generic declarations in the bodies of generic subprograms are also flagged. A generic unit nested in another generic unit is not flagged. If a generic unit is declared in a local package that is declared in a subprogram body, the generic unit is flagged.

This rule has no parameters.

Using AdaControl one can use the following rule to detect instantiations of generic packages/subprograms:

5.10 Declarations

This rule controls usage of various kinds of declarations, possibly only those occurring at specified locations.

...

Why is it considered bad practise to use local generic instantiations? Within the C++ Community, limiting the use of templates doesn't seem an issue. On the contrary, going all in with template metaprogramming is the norm.

Does local generic instantiations have a performance penalty? Is it something that may be error-prone? Limit cross-compiler compatibility? Why does the rule exist to ban local instantiations? I've been googling/searching the web for an answer to this question but have not found an explanation. Does anybody know?

From: "Jeffrey R. Carter" <spam.jrcarter.not@spam.not.acm.org>
Subject: Re: Why forbid local generic instantiations?
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 17:36:33 +0100

> Why is it considered bad practise to use local generic instantiations? Within the C++ Community, limiting the use of templates doesn't seem an issue. On the contrary, going all in with template metaprogramming is the norm.

It isn't bad practice. Mostly such rules are premature optimization. Are there rules against regular pkgs in such places?

There's no difference.

It makes perfect sense for things to be declared in the smallest scope in which they're needed. This is true of anything, not just pkgs.

A pkg in a subprogram is elaborated every time the subprogram is called. If the elaboration of a specific pkg is expensive and timing requirements are tight, it might make sense to move that pkg to a larger scope. But a general rule against them for "efficiency" doesn't make sense. Limiting it to pkgs that are generic instantiations makes less sense.

Perhaps such people don't know that instantiation takes place during compilation and has no run-time impact.

As a 1st-order approximation, anything the "C++ Community" does should be avoided.

From: "Randy Brukardt" <randy@rrsoftware.com>
Subject: Re: Why forbid local generic instantiations?
Newsgroups: comp.lang.ada
Date: Fri, 25 Jan 2019 15:23:55 -0600

> Perhaps such people don't know that instantiation takes place during compilation and has no run-time impact.

I agree with most of what you said, but this statement is false, since the instance is elaborated at the point of the instantiation. Depending on the generic, that could be a substantial amount of execution time. (Note that is even more true for a code-shared implementation like Janus/Ada, since the elaboration of the instance creates the instantiation descriptor.)

From: "Jeffrey R. Carter" <spam.jrcarter.not@spam.not.acm.org>
Subject: Re: Why forbid local generic instantiations?
Newsgroups: comp.lang.ada
Date: Sat, 26 Jan 2019 10:56:27 +0100

> [...]

I can't tell from what you've written if what I said is wrong or if we're saying basically the same thing in different ways. I'm not familiar with the way shared-code generics are instantiated. Macro-expansion instantiation is straightforward.

The rule I learned (Ada 83) was: Instantiation happens during compilation; elaboration happens during run time.

In more detail: Instantiation is the process whereby a compiler effectively replaces an instantiation with a regular pkg (the instance). The result is no different from having written the resulting regular pkg instead of the instantiation, except for possible code sharing with other instantiations of the same generic

[ignoring the case of an instantiation in a pkg spec].

All pkgs, regular or generic instances, are elaborated during run time. That elaboration can be as complex as the developer wants. In the case of a pkg in a subprogram, that elaboration happens every time the subprogram is called.

That's what I learned back when dinosaurs ruled the earth. I gather from what you've written that a shared-code compiler may increase the amount of elaboration by some (hopefully small, fixed?) amount, so it's not technically correct unless the increase is small enough to be considered negligible. I think it's correct for compilers that do macro-expansion instantiation, and close enough for the rule to be correct as a 1st-order approximation.

If I'm wrong, I'd like to be corrected.

From: Jere <jhb.chat@gmail.com>
Subject: Private extension of a synchronized interface
Newsgroups: comp.lang.ada
Date: Fri, 15 Feb 2019 16:52:07 -0800

I'll get to my ultimate goal later, but while following various rabbit trails, I came across a situation I couldn't solve. GNAT allows you to make private extensions to synchronized interfaces and it allows you to complete those private extensions with protected types. I can't, however, figure out how it overrides the abstract procedures and functions of the synchronized interface.

If I don't specify an override and try to call the procedure, it complains that the procedure is abstract. If I try to override the abstract function, it complains that the signature doesn't match the one in the protected body. I don't know if this is a GNAT issue or something that Ada doesn't allow. Here is some test code. It compiles as is, but there are two parts that if you uncomment either one of those it fails to compile.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
   package Example is

      type An_Interface is synchronized
            interface;
      procedure p1(Self : in out
            An_Interface)  is abstract;

      type Instance is synchronized new
            An_Interface with private;

      -- The following lines give the errors:
      -- "p1" conflicts with declaration at line
      --- xxx and missing body for "p1"

      --overriding
      --procedure p1(Self : in out Instance);

   private
      -- Some hidden implementation types,
      -- constants, etc.

      -- Instance full view is a protected type
      protected type Instance is new
            An_Interface with
         procedure p1;
      private
         -- some hidden stuff;
      end Instance;

   end Example;

   package body Example is
         protected body Instance is
         procedure p1 is
         begin
            Put_Line("Did Something");
         end p1;
      end Instance;

   end Example;

   v : Example.Instance;

begin
 Put_Line("Hello, world!");
 -- The following line gives the error:
 -- call to abstract procedure must be
 -- dispatching
   --v.p1;
end Hello;
```

My ultimate goal is not having to declare a bunch of extra types and packages in the public view to only use them in the private view of the protected object. I'd prefer that all of the private stuff actually be in a private section. So I'm not tied to interfaces, but it was one attempt at getting stuff moved down to the private section. But while I went down the interfaces rabbit hole, I just found the issue I ran into odd.

Does anyone know how to create the correct overrides for the example above?

Extension of synchronized interfaces

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: Private extension of a synchronized interface
Newsgroups: comp.lang.ada
Date: Sun, 17 Feb 2019 10:50:21 +0100

> I'll get to my ultimate goal later, but while following various rabbit trails, I came across a situation I couldn't solve. GNAT allows you to make private extensions to synchronized interfaces and it allows you to complete those private extensions with protected types. I can't, however, figure out how it overrides the abstract procedures and functions of the synchronized interface.

> If I don't specify an override and try to call the procedure, it complains that the procedure is abstract. If I try to override the abstract function, it complains that the signature doesn't match the one in the protected body. I don't know if this is a GNAT issue or something that Ada doesn't allow. Here is some test code. It compiles as is, but there are two parts that if you uncomment either one of those it fails to compile.

Reading RM 9.5.2 (13.2/2) does not really help:

"if the overriding_indicator is overriding, then the entry shall implement an inherited subprogram;"

An inherited subprogram is already implemented per, well, inheritance. May be it means:

1. shall implement a primitive operation (it overrides here);

2. shall implement an overridden primitive operation (it implements overriding declared earlier).

Neither #1 nor #2 work.

But synchronized interfaces are totally bogus from the software design POV. It is a pure implementation aspect exposed. Why do you care?

Aggregate a protected object and delegate primitive operations to it.

From: Jere <jhb.chat@gmail.com>
Subject: Re: Private extension of a synchronized interface
Newsgroups: comp.lang.ada
Date: Sun, 17 Feb 2019 05:46:17 -0800

> But synchronized interfaces are totally bogus from the software design POV. It is a pure implementation aspect exposed. Why do you care?

> Aggregate a protected object and delegate primitive operations to it.

That's what I am doing as my own solution. I was intrigued with the code above as an alternate solution because it could potentially give a compile time indication that a procedure was a protected operation (as opposed to me relying on simply providing that via comments). A delegate non protected procedure has to rely on the comment. I didn't even want the interface to use as an interface, just as a means to at the API level to have a compiler enforced indication that the procedure was from a protected object. I started with a protected object in the public view but the implementation details of the private part of the protected object led to about 10 lines of code (type declarations and a couple of package specifications) that had no use to the public view but had to be there because of how protected object declarations work. I saw this as a potential means of information hiding. My actual solution is as you suggested with delegate operations that call the protected object. However, I honestly wanted to know why Ada allowed one to setup the private extension but not allow you to actually provide the functions (or if this was a GNAT issue or if I was just not using the right syntax). So the reason I care was a thirst for knowledge of how things work.

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Subject: Re: Private extension of a synchronized interface
Newsgroups: comp.lang.ada
Date: Sun, 17 Feb 2019 15:52:38 +0100

Given to who? The compiler knows already, the user should not care. It is an implementation aspect which simply does not belong here.

What could make sense is an entry interface, a primitive operation which could be queued/requeued to, used in timed entry call etc.

> A delegate non protected procedure has to rely on the comment.

There is no contract that could require it protected. It is a property of the object/task and no property of an operation. You could not do anything with a task or protected object that would not resolve into a protected action anyway.

[...]

> However, I honestly wanted to know why Ada allowed one to setup the private extension but not allow you to actually provide the functions (or if this was a GNAT issue or if I was just not using the right syntax). So the reason I care was a thirst for knowledge of how things work.

Ada 2005 stuff, most of it makes little sense to me. It was some halfhearted attempt to unite tagged types with tasks and protected objects with no desire to actually do that...

From: Jere <jhb.chat@gmail.com>
Subject: Re: Private extension of a synchronized interface
Newsgroups: comp.lang.ada
Date: Sun, 17 Feb 2019 07:36:18 -0800

The compiler cannot always tell depending on how and where you call buried protected operations. I always prefer compile time catching over run time catching.

> > A delegate non protected procedure has to rely on the comment.

> There is no contract that could require it protected. It is a property of the object/task and no property of an operation. You could not do anything with a task or protected object that would not resolve into a protected action anyway.

Protected procedures/functions/entries are particularly heavy operations.

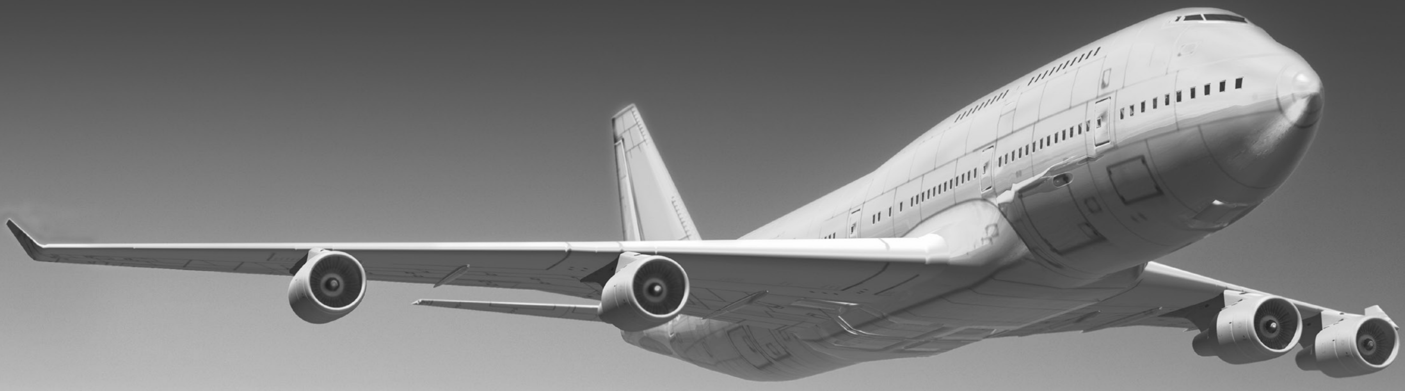
I don't know if you generally work in low level embedded environments, but being able know and plan for that can be very critical. It can change how you approach your design. When you work in systems where your system clock is 1-4MHz, timing of operations does start to matter.

> > However, I honestly wanted to know why Ada allowed one to setup the private extension but not allow you to actually provide the functions (or if this was a GNAT issue or if I was just not using the right syntax). So the reason I care was a thirst for knowledge of how things work.

> Ada 2005 stuff, most of it makes little sense to me. It was some halfhearted attempt to unite tagged types with tasks and protected objects with no desire to actually do that...

I'm just curious if or why the process was stopped half way instead of abandoned or completed (again that is assuming I didn't use the wrong syntax, in which case it's simply that I'm structuring the syntax wrong).

I don't really need to marry them with tagged types. I do appreciate the ability to dispatch over a group of related but different tasks much more easily and the interfaces give that. The way that Ada chose to implement interfaces is one of many ways (not all of which would have required tagged types).



Conference Calendar

Dirk Craeynest

KU Leuven. Email: Dirk.Craeynest@cs.kuleuven.be

This is a list of European and large, worldwide events that may be of interest to the Ada community. Further information on items marked ♦ is available in the Forthcoming Events section of the Journal. Items in larger font denote events with specific Ada focus. Items marked with ☺ denote events with close relation to Ada.

The information in this section is extracted from the on-line *Conferences and events for the international Ada community* at: http://www.cs.kuleuven.be/~dirk/ada-belgium/events/list.html on the Ada-Belgium Web site. These pages contain full announcements, calls for papers, calls for participation, programs, URLs, etc. and are updated regularly.

2019

May 01-03 8th **International Conference on Fundamentals of Software Engineering** (FSEN'2019), Tehran, Iran. Topics include: all aspects of formal methods, especially those related to advancing the application of formal methods in the software industry and promoting their integration with practical engineering techniques; software specification, validation, and verification; software architectures and their description languages; integration of formal and informal methods; component-based software systems; model checking and theorem proving; software verification; CASE tools and tool integration; industrial applications; etc.

☺ May 07-09 22nd IEEE **International Symposium On Real-Time Distributed Computing** (ISORC'2019), Valencia, Spain. Topics include: object/component/service-oriented real-time distributed computing (ORC) technology, programming and system engineering (real-time programming challenges, ORC paradigms, languages, ...), trusted and dependable systems, system software (real-time kernel/OS, middleware support for ORC, extensibility, synchronization, scheduling, fault tolerance, security, ...), applications (medical devices, intelligent transportation systems, industrial automation systems, Internet of Things and Smart Grids, embedded systems in automotive, avionics, consumer electronics, ...), system evaluation (performance analysis, monitoring & timing, dependability, fault detection and recovery time, ...), cyber-physical systems, etc.

May 07-09 11th **NASA Formal Methods Symposium** (NFM'2019), Houston, Texas, USA. Topics include: identify challenges and provide solutions for achieving assurance for critical systems; formal verification, including theorem proving, model checking, and static analysis; use of formal methods in software and system testing; run-time verification techniques and algorithms for scaling formal methods, such as abstraction and symbolic methods, compositional techniques, as well as parallel and/or distributed techniques; safety cases and system safety; formal approaches to fault tolerance; formal methods in systems engineering and model-based development; etc.

May 20-23 32nd **International Conference on Architecture of Computing Systems** (ARCS'2019), Copenhagen, Denmark. Focus: "architectures for complex real-time systems". Topics include: autonomous control systems, as well as safety and security critical systems; upcoming architectures and technologies, exploitable architectural features, languages, and tooling; architectures for real-time and mixed-criticality systems; programming models for many-core computing platforms; hypervisors and middleware for multi-/many-core computing platforms; support for safety and security; etc.

May 20-24 33rd IEEE **International Parallel and Distributed Processing Symposium** (IPDPS'2019), Rio de Janeiro, Brazil.

> ☺ May 20 24th **International Workshop on High-Level Parallel Programming Models and Supportive Environments** (HIPS'2019). Topics include: the areas of parallel applications, language design, compilers, runtime systems, and programming tools; the areas of emerging programming models for large-scale parallel systems and many-core architectures; new programming languages and constructs for exploiting parallelism/locality; experience with and improvements for existing parallel languages and run-time environments; parallel compilers, programming tools, and environments; programming environments for heterogeneous multicore systems and accelerators such as GPUs, FPGAs, and MICs; etc.

May 21-25 20th **International Conference on Agile Software Development** (XP'2019), Montréal, Québec, Canada.

May 25-26 14th IEEE/ACM **International Conference on Global Software Engineering** (ICGSE'2019), Montréal, Québec, Canada.

May 25 - Jun 01 41st **International Conference on Software Engineering** (ICSE'2019), Montréal, Québec, Canada. Theme: "The next 50 years for Software Engineering".

June 03-07 31st **International Conference on Advanced Information Systems Engineering** (CAiSE'2019), Rome, Italy. Theme: "Responsible Information Systems". Topics include: methods, models, techniques, architectures and platforms for supporting the engineering and evolution of information systems and organizations. Deadline for early registration: May 6, 2019.

☺ June 04-06 **International Conference on Reliability, Safety and Security of Railway Systems** (RSSRail'2019), Lille, France. Topics include: building critical railway applications and systems. Includes tutorials by Altran and AdaCore.

☺ June 04-06 **DAta Systems In Aerospace** (DASIA'2019), Sicily, Italy.

♦June 11-14 **Ada-Europe 24th International Conference on Reliable Software Technologies** (Ada-Europe 2019), Warsaw, Poland. Sponsored by Ada-Europe, in cooperation with ACM SIGAda, SIGBED, SIGPLAN, and the Ada Resource Association (ARA). Deadline for early registration: May 20, 2019.

> ☺ June 14 Ada-Europe'2019 - 6th **Workshop on Challenges and New Approaches for Dependable and Cyber-Physical System Engineering** (DeCPS'2019). Topics include: vehicle of the future, transport and mobility, Industry 4.0 in transportation sector, security and comfort of the end-user, human/machine interaction, safety and security, industrial experiments and case studies, integration of Internet of Things and cloud computing, evolution of standards and certification processes, impact of artificial intelligence in CPS.

June 24-28 13th ACM **International Conference on Distributed Event-Based Systems** (DEBS'2019), Darmstadt, Germany. Topics include: systems dealing with collecting, detecting, processing and responding to events through distributed middleware and applications; real-time analytics, complex event processing, distributed programming, security, reliability and resilience, Internet-of-Things, cyber-physical systems, etc.

June 26-28 18th **International Conference on Software Reuse** (ICSR'2019), Cincinnati, Ohio, USA. Topics include: approaches facilitating reuse in industry; technical debt and reuse; component-based reuse techniques; generative, systematic, and opportunistic reuse; reverse engineering of potentially reusable components; evolution and maintenance of reusable assets; dynamic aspects of reuse (e.g., post-deployment time); retrieval of reusable artifacts and knowledge in large-scale software repositories (e.g., open-source and industrial code bases); etc.

July 09-12 31st **Euromicro Conference on Real-Time Systems** (ECRTS'2019), Stuttgart, Germany. Topics include: all aspects of real-time systems, such as scheduling design and analysis, real-time operating systems, hypervizors and middleware, memory management, worst-case execution time analysis, formal models and analysis techniques for real-time systems, mixed-criticality design and assurance, programming languages and compilers, virtualization and timing isolation, etc. Event includes: CERTS - International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, WATERS - International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, WCET - International Workshop on Worst-Case Execution Time Analysis, etc.

☺ July 15-19 33rd **European Conference on Object-Oriented Programming** (ECOOP'2019), London, England. Topics include: original and unpublished results on any Programming Languages topic. Deadline for submissions: June 10, 2019 (student volunteers round 2).

July 15-19 **Software Technologies: Applications and Foundations** (STAF'2019), Eindhoven, the Netherlands. Event includes: ECMFA - 15th European Conference on Modelling Foundations and Applications, ICGT - 12th International Conference on Graph Transformation, ICMT - 12th International Conference on Model Transformations, TTC - 12th Transformation Tool Contest, STAF-JRC - 1st STAF Junior Researcher Community Event, STAF-RPS - 1st STAF Research Project Showcase Workshop.

July 15-19 43rd **Annual** IEEE **Conference on Computer Software and Applications** (COMPSAC'2019), Milwaukee, Wisconsin, USA.

July 22-29 19th IEEE **International Conference on Software Quality, Reliability and Security** (QRS'2019), Sofia, Bulgaria. Topics include: reliability, security, availability, and safety of software systems; software testing, verification, and validation; program debugging and comprehension; fault tolerance for software reliability improvement; modeling, prediction, simulation, and evaluation; metrics, measurements, and analysis; software vulnerabilities; formal methods; benchmark, tools, industrial applications, and empirical studies; etc. Deadline for submissions: May 1, 2019 (workshop papers, fast abstracts, industry track).

July 29-31 13th **International Symposium on Theoretical Aspects of Software Engineering** (TASE'2019), Guilin, China. Topics include: theoretical aspects of software engineering, such as abstract interpretation, component-based software engineering, cyber-physical systems, distributed and concurrent systems, embedded and real-time systems, formal verification and program semantics, integration of formal methods, language design, model checking and theorem proving, model-driven engineering, object-oriented systems, program analysis, reverse engineering and software maintenance, run-time verification and monitoring, software architectures and design, software testing and quality assurance, software safety, security and reliability, specification and verification, type systems, tools exploiting theoretical results, etc.

July 29 - Aug 02 38th ACM **Symposium on Principles of Distributed Computing** (PODC'2019), Toronto, Ontario, Canada.

☺ August 18-21 25th IEEE **International Conference on Embedded and Real-Time Computing Systems and Applications** (RTCSA'2019), Hangzhou, China. Topics include: real-time operating systems, real-time scheduling, timing analysis, programming languages and run-time systems, middleware systems, design and analysis tools, multi-core embedded systems, operating systems and scheduling, embedded software and compilers, fault tolerance and security, embedded systems and design methods for cyber-physical systems, applications and case studies of IoT and CPS, cyber-physical co-Design, etc.

August 26-30 27th ACM **Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering** (ESEC/FSE'2019), Tallinn, Estonia. Topics include: architecture and design; components, services, and middleware; debugging; dependability, safety, and reliability; development tools and environments; distributed, parallel, and concurrent software; education; embedded and real-time software; empirical software engineering; formal methods, including languages, methods, and tools; model-driven software engineering; processes and workflows; program analysis; program comprehension and visualization; refactoring; reverse engineering; safety-critical systems; scientific computing; security and privacy; software economics and metrics; software evolution and maintenance; software modularity and reuse; software product lines; testing and verification; traceability; etc. Deadline for submissions: May 17, 2019 (tool demos), May 24, 2019 (student research competition), May 30 - June 10, 2019 (workshop papers), May 31, 2019 (journal first papers).

August 27-29 17th **International Conference on Formal Modeling and Analysis of Timed Systems** (FORMATS'2019), Amsterdam, the Netherlands. Topics include: theoretical foundations of timed systems and languages; methods and tools (techniques, algorithms, data structures, and software tools for analyzing timed systems and resolving temporal constraints, such as scheduling, worst-case execution time analysis, optimization, model checking, testing, constraint solving, ...); adaptation and specialization of timing technology in application domains in which timing plays an important role (real-time software, problems of scheduling in manufacturing and telecommunication, ...); etc. Deadline for submissions: May 9, 2019 (abstracts), May 13, 2019 (papers).

August 27-30 30th **International Conference on Concurrency Theory** (CONCUR'2019), Amsterdam, the Netherlands. Topics include: basic models of concurrency; verification and analysis techniques for concurrent systems, such as abstract interpretation, atomicity checking, model checking, race detection, run-time verification, static analysis, theorem proving, type systems, security analysis, ...; distributed algorithms and data structures; theoretical foundations of architectures, execution environments, and software development for concurrent systems, such as multiprocessor and multi-core architectures, compilers and tools for concurrent programming, programming models such as component-based, object-oriented, ...; etc. Includes 24th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2019), 17th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'2019), etc.

August 28-30 45th **Euromicro Conference on Software Engineering and Advanced Applications** (SEAA'2019), Thessaloniki / Chalkidiki, Greece. Topics include: information technology for software-intensive systems; conference tracks on Embedded Systems & Internet of Things (ES-IoT), Software Process and Product Improvement (SPPI), etc.; special sessions on Cyber-Physical Systems (CPS), Software Engineering and Technical Debt (SEaTeD), Model-Driven Engineering and Modeling Languages (MDEML), etc.

September 01-04 **Federated Conference on Computer Science and Information Systems** (FedCSIS'2019), Leipzig, Germany. Event includes: 4th International Workshop on Language Technologies and Applications (LTA), 7th Workshop on Advances in Programming Languages (WAPL), 10th Workshop on Scalable Computing (WSC), 3rd International Conference on Lean and Agile Software Development (LASD), Joint 39th IEEE Software Engineering Workshop (SEW-39) and 6th International Workshop on Cyber-Physical Systems (IWCPS-6), etc. Deadline for submissions: May 14, 2019 (papers), June 4, 2019 (position papers).

☺ September 10-13 **International Conference on Parallel Computing** 2019 (ParCo'2019), Prague, Czech Republic. Topics include: all aspects of parallel computing, including applications, hardware and software technologies, and languages and development environments. Deadline for submissions: July 31, 2019 (full papers).

September 16-20 17th **International Conference on Software Engineering and Formal Methods** (SEFM'2019), Oslo, Norway. Topics include: software evolution, maintenance, re-engineering, and reuse; programming languages; abstraction and refinement; software testing, validation, and verification; model checking, theorem proving, and decision procedures; testing and runtime verification; other light-weight and scalable formal methods; safety-critical, fault-tolerant, and secure systems; software certification; applications and technology transfer; real-time, hybrid, and cyber-physical systems; education; case studies, best practices, and experience reports; etc. Deadline for submissions: May 3, 2019 (abstracts), May 10, 2019 (papers).

September 19-20 13th ACM/IEEE **International Symposium on Empirical Software Engineering and Measurement** (ESEM'2019), Porto de Galinhas, Brazil. Deadline for submissions: June 10, 2019 (industry papers), June 10, 2019 (emerging results and vision papers), July 1, 2019 (Journal-First submissions).

☺ Sep 30 - Oct 02 **Automotive - Safety & Security** 2019 & **SafeWare Engineering** 2019, Karlsruhe, Germany. Co-organized by Ada-Deutschland. Topics include: all aspects of reliability, safety, security, privacy, etc. in automotive systems, many of which are heavily influenced by advances in applied Software Engineering; same themes for application domain of Internet of Things (IoT). Conference (and submission) language is English.

October 01-04 38th IEEE **International Symposium on Reliable Distributed Systems** (SRDS'2019), Lyon, France. Topics include: distributed systems design, development and evaluation, with emphasis on reliability, availability, safety, dependability, security, and real-time.

October 07-11 23rd **International Symposium on Formal Methods** (FM'2019), Porto, Portugal, aka 3rd World Congress on Formal Methods. Topics include: formal methods in a wide range of domains including software, computer-based systems, systems-of-systems, cyber-physical systems, human-computer interaction, manufacturing, sustainability, energy, transport, smart cities, and healthcare; formal methods in practice (industrial applications of formal methods, experience with formal methods in industry, tool usage reports, ...); tools for formal methods (advances in automated verification, model checking, and testing with formal methods, tools integration, environments for formal methods, ...); formal methods in software and systems engineering (development processes with formal methods, usage guidelines for formal methods, ...); etc.

October 08-11 19th **International Conference on Runtime Verification** (RV'2019), Porto, Portugal. Topics include: monitoring and analysis of the runtime behaviour of software and hardware systems. Application areas include cyber-physical systems, safety/mission critical systems, enterprise and systems software, cloud systems, autonomous and reactive control systems, health management and diagnosis systems, and system security and privacy. Deadline for submissions: June 25, 2019 (abstracts), June 30, 2019 (papers, tutorials).

October 13-18 **Embedded Systems Week** 2019 (ESWEEK'2019), New York City, USA. Topics include: all aspects of embedded systems and software. Deadline for submissions: June 7, 2019 (work-in-progress track papers).

October 13-18 **International Conference on Compilers, Architecture, and Synthesis for Embedded Systems** (CASES'2019). Topics include: latest advances in compilers and architectures for high-performance, low-power embedded systems; compilers for embedded systems: multi- and many-core processors, GPU architectures, reconfigurable computing including FPGAs and CGRAs; security, reliability, and predictability: secure architectures, hardware security, and compilation for software security; architecture and compiler techniques for reliability and aging; modeling, design, analysis, and optimization for timing and predictability; validation, verification, testing & debugging of embedded software; special day on the Internet of Medical Things; etc.

October 13-18 **International Conference on Hardware/Software Codesign and System Synthesis** (CODES+ISSS'2019). Topics include: system-level design, modeling, analysis, and implementation of modern embedded, IoT, and cyber-physical systems, from system-level specification and optimization down to system synthesis of multi-processor hardware/software implementations.

October 13-18 ACM SIGBED **International Conference on Embedded Software** (EMSOFT'2019). Topics include: the science, engineering, and technology of embedded software development; research in the design and analysis of software that interacts with physical processes; results on cyber-physical systems, which compose computation, networking, and physical dynamics.

☺ October 14-20 **TOOLS 50+1: Technology of Object-Oriented Languages and Systems** (TOOLS'2019), Innopolis (Kazan), Russia. Topics include: new development in object technology; experience reports, technology transfer; challenges of developing software for embedded systems and Internet of Things; reliability and dependability; hybrid and cyber-physical systems modeling and verification; etc.

☺ October 20-25 ACM **Conference on Systems, Programming, Languages, and Applications: Software for Humanity** (SPLASH'2019), Athens, Greece. Topics include: all aspects of software construction and delivery, at the intersection of programming languages and software engineering. Deadline for submissions: May 17, 2019 (SPLASH-I), May 29, 2019 (DLS abstracts - Dynamic Languages Symposium), June 5, 2019 (DLS papers - Dynamic Languages Symposium), June 14, 2019 (GPCE abstracts - Generative Programming: Concepts & Experiences, SLE abstracts - Software Language Engineering), June 21, 2019 (GPCE papers - Generative Programming: Concepts & Experiences, SLE papers - Software Language Engineering), July 8, 2019 (MPLR - Managed Programming Languages and Runtimes), July 12, 2019 (SPLASH-E, Doctoral Symposium, Student Research Competition abstracts), August 2, 2019 (workshop papers), September 7, 2019 (Posters), end of September 2019 (student volunteer applications).

October 21-22 12th ACM SIGPLAN **International Conference on Software Language Engineering** (SLE'2019). Topics include: areas ranging from theoretical and conceptual contributions, to tools, techniques, and frameworks in the domain of software language engineering; generic aspects of software languages development rather than aspects of engineering a specific language; software language design and implementation; software language validation; software language integration and composition; software language maintenance (software language reuse, language evolution, language families and variability); domain-specific approaches for any aspects of SLE (design, implementation, validation, maintenance); empirical evaluation and experience reports of language engineering tools (user studies evaluating usability, performance benchmarks, industrial applications); etc. Deadline for submissions: June 14, 2019 (abstracts), June 21, 2019 (papers), August 16, 2019 (artifacts).

Oct 28 - Nov 01 30th IEEE **International Symposium on Software Reliability Engineering** (ISSRE'2019), Berlin, Germany. Topics include: development, analysis methods and models throughout the software development lifecycle; primary dependability attributes (i.e., security, safety, maintainability) impacting software reliability; secondary dependability attributes (i.e., survivability, resilience, robustness) impacting software reliability; reliability threats, i.e. faults (defects, bugs, etc.), errors, failures; reliability means (fault prevention, fault removal, fault tolerance, fault forecasting); reliability of open source software; etc. Deadline for submissions: May 5, 2019 (full research papers).

Oct 30 - Nov 04 16th **International Colloquium on Theoretical Aspects of Computing** (ICTAC'2019), Hammamet, Tunisia. Topics include: semantics of programming languages; theories of concurrency; theories of distributed computing; models of objects and components; timed, hybrid, embedded and cyber-physical systems; static analysis; software verification; software testing; model checking and automated theorem proving; interactive theorem proving; verified software, formalized programming theory; etc. Deadline for submissions: May 5, 2019 (abstracts), May 12, 2019 (papers).

November 10-13 24th **International Conference on Engineering of Complex Computer Systems** (ICECCS'2019), Hong Kong, China. Topics include: verification and validation, security and privacy of complex systems, model-driven development, reverse engineering and refactoring, software architecture, design by contract, agile methods, safety-critical and fault-tolerant architectures, real-time and embedded systems, systems of systems, cyber-physical systems and Internet of Things (IoT), tools and tool integration, industrial case studies, etc. Deadline for submissions: May 24, 2019 (abstracts), May 31, 2019 (full papers).

November 11-15 34th IEEE/ACM **International Conference on Automated Software Engineering** (ASE'2019), San Diego, California, USA. Topics include: foundations, techniques, and tools for automating the analysis, design, implementation, testing, and maintenance of large software systems; empirical software engineering; maintenance and evolution; model-driven development; program comprehension; reverse engineering and re-engineering; specification languages; software analysis; software architecture and design; software product line engineering; software security and trust; etc. Deadline for submissions: May 6, 2019 (research abstracts), May 13, 2019 (research papers), June 19, 2019 (other tracks), July 15, 2019 (workshop papers).

November 27-29 20th **International Conference on Product-Focused Software Process Improvement** (PROFES'2019), Barcelona, Spain. Topics include: experiences, ideas, innovations, as well as concerns related to professional software development and process improvement driven by product and service quality needs. Deadline for submissions: June 7, 2019 (abstracts for full research papers, industry papers, industry talks), June 14, 2019 (full research papers, industry papers, industry talks), August 5, 2019 (short papers), August 9, 2019 (Journal-First papers, European project space).

December 02-04 17th **Asian Symposium on Programming Languages and Systems** (APLAS'2019), Bali, Indonesia.

December 02-06 15th **International Conference on integrated Formal Methods** (iFM'2019), Bergen, Norway. Topics include: hybrid approaches to formal modelling and analysis; i.e. the combination of (formal and semi-formal) methods for system development, regarding modelling and analysis, and covering all aspects from language design through verification and analysis techniques to tools and their integration into software engineering practice.

☺ December 03-06 40th IEEE **Real-Time Systems Symposium** (RTSS'2019), Hong Kong. Topics include: all aspects of real-time systems, including theory, design, analysis, implementation, evaluation, and experience. Deadline for submissions: May 30, 2019 (papers).

December 10 Birthday of Lady Ada Lovelace, born in 1815. Happy Programmers' Day!

24th International Conference on

Reliable Software Technologies

Ada-Europe 2019

Advance Information

The 24th International Conference on Reliable Software Technologies (Ada-Europe 2019) will take place in Warsaw, Poland, 11-14 of June. This conference is the latest in a series of annual international conferences started in the early 80's, under the auspices of, and organization by, Ada-Europe, the international organization that promotes the knowledge and use of Ada and Reliable Software in general into academia, research and industry. *Ada-Europe 2019 provides a unique opportunity for dialogue and collaboration between academics and industrial practitioners interested in reliable software.*

The 2019 edition of the conference features a number of important innovations:

- Reduced fee for all authors.
- Lower registration fees for the conference and tutorials, unified for all participants.
- New, journal-based, open-access, publication model for the peer-reviewed papers
- An educational tutorial offered especially to those who wish to know more about Ada.
- More compact program: two core days (Wednesday and Thursday), and an exhibition opening in the afternoon of Tuesday, in parallel to the Ada-Europe General Assembly, followed by a welcome aperitif.

Conference Overview

	Morning	Before Lunch	After Lunch	Afternoon
Tuesday, **June 11th** Tutorials, Opening & Welcome Aperitif	**Tutorial:** P. Munts, *Controlling I/O Devices with Ada using the Remote I/O Protocol*			**Exhibition Opening** **& Ada-Europe GA** **& Welcome Aperitif**
	Tutorial: J.P. Rosen, *An introduction to Ada*			
Wednesday, **June 12th** Sessions & Exhibition	**Keynote Talk:** TO BE ANNOUNCED	**Presentation Session:** *Assurance Issues in Critical Systems*	**Presentation Session:** *Tooling Aid for Verification*	**Presentation Session:** *Best Practices for Critical Applications*
Thursday, **June 13th** Sessions & Exhibition	**Keynote Talk:** *A 2020 View of Ada* Tucker Taft *(AdaCore, USA)*	**Presentation Session:** *Uses of Ada in Challenging Environments*	**Presentation Session:** *Verification Challenges*	**Presentation Session:** *Real-Time Systems*
Friday, **June 14th** Workshop & by-invitation meetings	**Workshop:** *Challenges and new Approaches for Dependable and Cyber-Physical Systems Engineering (DeCPS)*			
	ISO WG 9 meeting	**ISO ARG meeting**		

Keynote Talks

Each day of the core program will be opened with a keynote talk delivered by eminent speakers. Currently confirmed:

- **Tucker Taft**, AdaCore, USA, *"A 2020 View of Ada"*

Tutorials

Opening the conference on Tuesday, the program includes two tutorials:

- Controlling I/O Devices with Ada using the Remote I/O Protocol, **Philip Munts**, full day
- An introduction to Ada, **Jean-Pierre Rosen**, full day

Co-Located Workshop

On Friday, June 14th, the conference program features the 6th edition of the International Workshop on Challenges and new Approaches for Dependable and Cyber-Physical Systems Engineering (DeCPS). The DeCPS workshop series aims to facilitate the exchange of ideas, research results and experience in the field of dependable and cyber-physical systems engineering, from theoretical and practical perspectives. To favour integration and interaction between the DeCPS workshop and the conference core, the full conference registration includes complimentary access to the workshop.

Vendor Presentations and Exhibition

The conference will feature an exhibition located in a central hall of the hosting site, where all the session breaks will take place. Exhibitors and vendors will also make technical presentations, scattered throughout the conference program.

Social Events

The conference program includes two coffee breaks and a seated lunch each day, with ample opportunity for technical discussions, visits to the exhibition, and social interaction. The Ada-Europe General Assembly will take place in parallel to the opening of the exhibition in the late afternoon of Tuesday, right after the tutorials. Immediately after that, the local organizers will host a Welcome Aperitif on the terrace of the Institute of Aviation, enjoying a wonderful view of the Warsaw airport and city center, accompanied by drinks and typical Polish snacks.

On Wednesday evening, the Conference Banquet will take place at the elegant old-style restaurant "Przepis na kompot", in the small Mazovian town of Zelazowa Wola, where Fryderyk Chopin was born in 1810. Chopin's family moved to Warsaw soon afterwards, returning there for summer holidays, Christmas or Easter. On summertime visits, the grand piano of the house would be taken to the garden, and Fryderyk would give concerts in the shade of firs and lindens. Zelazowa Wola now hosts concerts and musical exhibitions, "Prezentacje Muzyczne", by talented young piano players worldwide. The conference banquet will enjoy Polish cuisine, which is most delicious and renowned in Europe and the whole world, along with drinks and live piano music in the background.

Registration Fees

	Member		Non-member		Author (1)	Tutorial/ Workshop (2)
		Student		Student		
Early registration (until May 20th)	420 €	260 €	480 €	320 €	220 €	40 €
Late/on-site registration (after May 20th)	480 €	320 €	540 €	380 €		70 €
Single-day registration	270 €					

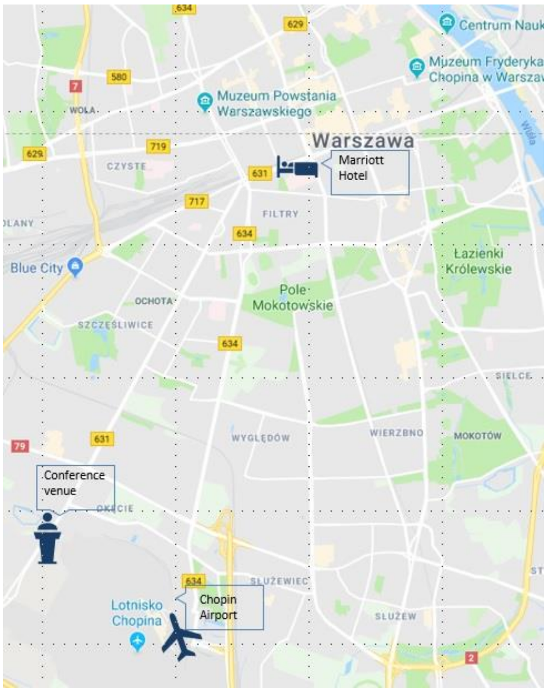
(1) One author per presented paper (peer-reviewed/industrial) is entitled to the discounted author fee

(2) Access to the workshop is included in the full conference registration.

Conference Venue

The conference takes places in Warsaw, the capital city of Poland, itself at the heart of Europe, an apt location for political, scientific, business and cultural events. Its modern architecture, user-friendly infrastructure and creative inhabitants make Warsaw the beating heart of business. Behind the hustle and bustle of the business world, beats the rhythm of city life. Try the varied delights of the city's many restaurants, take a stroll along the banks of Vistula, or just wander around and discover the fascinating reality of life in a modern European city. Warsaw is a city that wants exploring. Constantly changing and modernizing, it rapidly becomes almost unrecognizable if you do not take time to acquaint yourself with it. Yet, many aspects of its quirky character and cult places persist and just call for discovery. Whether you are visiting Warsaw on business or for pleasure, the city offers everything you need to make your stay here maximally enjoyable.

The conference venue is at the Engineering Design Center, partnership of General Electric and the Institute of Aviation, one of Europe's largest engineering institutions. Since its inception, the **Institute of Aviation** has engaged in applied research in aeronautics and astronautics, achieving significant results in the operation of aircraft, helicopters, meteorological rockets, engines and instrumentation. At present, the Institute of Aviation continues expanding its areas of research, to include CAD, new materials testing, adaptation systems, micro-/nano-technology, alternative energy sources, application of aviation technologies to medicine and health protection and local transport. Poland's membership in the European Union has created major opportunities for cooperation in all of these areas. The Institute of Aviation has joined the European research area most successfully and looks forward to working with you. ***Address**: Institute of Aviation, Al. Krakowska 110/114 St. 02-256 Warsaw. **Location**: 5XH2+G3 Warszawa.*



Conference Hotel

The local organizers recommend the Warsaw Marriott Hotel and have arranged with the hotel a special price for the dates around the conference. Details on how to obtain the discount will be provided as part of the registration process.

Sponsors

AdaCore

PTC® Developer Tools

RAPITA SYSTEMS A DANLAW COMPANY

VECTOR

The conference is supported and sponsored by

Ada europe

In Cooperation with:

Ada Resource ARA Association

acm In-Cooperation

SIGBED

SIGPLAN

SIGAda

ConcertoFLA-based Multi-concern Assurance for Space Systems

Zulqarnain Haider, Barbara Gallina

Mälardalen University, P.O. Box 883, SE- 721 23 Västerås, Sweden; zulqarnain.haider@mdh.se barbara.gallina@mdh.se

Anna Carlsson

OHB Sweden, P.O. Box 1269, SE- 16429 Kista, Sweden; anna.carlsson@ohb-sweden.se

Silvia Mazzini, Stefano Puri

Intecs, Italy; silvia.mazzini@intecs.it stefano.puri@intecs.it

Abstract

Space systems often need to be engineered in compliance with standards such as ECSS and need to ensure a certain degree of dependability. Given the multi-faceted nature of dependability (characterized by a set of concerns), assuring dependability implies multi-concern assurance, which requires the modelling of various system characteristics and their co-assessment and co-analysis, in order to enable the management of trade-offs between them. CHESS is a systems engineering methodology and an open source toolset, which includes ConcertoFLA. ConcertoFLA allows users (system architects and dependability engineers) to decorate component-based architectural models with dependability-related information, execute Failure Logic Analysis (FLA) techniques, and get the results back-propagated onto the original model. In this paper, we present the customization of the CHESS methodology and ConcertoFLA in the context of the ECSS standards to enable architects and dependability engineers to define a system and perform dependability-centered co-analysis for assuring the required non-functional properties of the system according to ECSS requirements. The proposed customization is then applied in the context of spacecraft Attitude Control Systems engineering, which is a part of satellite on-board software.

Keywords: Dependability analysis, Failure Logic Analysis, Multi-concern, Dependability assurance, ECSS standard series, CHESS toolset.

1 Introduction

Space systems such as satellites are often required to be engineered according to the standards such as European Cooperation for Space Standardization (ECSS) standards. The ECSS standards address different aspects of space project ranging from management, space system engineering and qualification. Due to the critical nature of the space systems, ECSS puts requirements on the assurance of the product and its software systems. In particular, ECSS has standards for software engineering ECSS-E-ST-40C [1], the assurance of dependability of product ECSS-Q-ST-30C [4], safety of product ECSS-Q-ST-40C [5], assurance of software ECSS-Q-ST-80 [3] and assurance of security of software ESSB-ST-E-008 [1]. To fulfil the requirements of the standards and provide assurance of dependability, safety and security, a systematic approach for co-assessment and co-analysis could have advantages on manifold. For example, modelling of various system characteristics and their co-assessment and co-analysis leads to reduction in cost as well enable the management of trade-offs between these properties.

CHESS [13] is a methodology and an open source supporting toolset based upon Papyrus UML [23]. CHESS is the result of several R&D projects, starting from the original CHESS (Composition with Guarantees for High integrity Embedded Software Components Assembly) ARTEMIS JU project [9] and continuing with CONCERTO (Guaranteed Component Assembly with Round Trip Analysis for Energy Efficient High Integrity Multicore Systems) ARTEMIS JU project [9], to provide a model based solution to address the challenges of developing critical real time and embedded systems, by adopting a component based approach, across several domains of interest, including space.

The CHESS Modelling Language (CHESSML), part of the CHESS documentation [9], is based upon UML [22], SysML [19], MARTE [20] and includes also SafeConcert [14] as its base for the dependability profile. This profile enables a support of decorating the component based architectural models with dependability related information. ConcertoFLA [6], which is a part of CHESS toolset, utilizes the decorated components and calculates the failure behaviour of the composed system, representing the assembly of these components. The CHESS design modelling capabilities along with the analysis capabilities are well supportive and compliant with the ECSS standards addressing product and software engineering and assurance.

In this paper, we extend our previous work [21] and we customize the CHESS and ConcertoFLA methodologies in the context of ECSS. The approach, resulting from the customization, enables the co-analysis of reliability, safety and security concerns. Such co-analysis has the potential to contribute in the reduction of cost, complexity and in the management of trade-offs as well as compliance with the standards for qualification purposes.

2 Background

In this section, we describe the background concepts. In particular, Section 2.1 provides the details of ECSS standards. Section 2.2 describes the ConcertoFLA analysis process.

2.1 European Cooperation for Space Standardization (ECSS) standards

ECSS standards cover all the aspects of a space system project spanning to the management of the project, engineering space system and its qualification. Assurance of different properties is an essential part of system engineering. ECSS provides standards for assurance of dependability, safety of the system and the software product as well as security of software. ECSS- E-ST-40C standard is focused on software part of the space system. The standard covers all the phases of the development of the software and puts requirements and principles for software design. For the assurance of software, the standard refers to the ECSS-Q-ST-80C.

Following are the ECSS standards related to the system and the software product assurance, in particular assurance of dependability, safety and security.

- ECSS-Q-ST-30C, defines the dependability requirements on space product assurance. In ECSS scope, the notion of dependability embraces reliability, maintainability and availability. Unlike, the academic dependability literature [12], where dependability also includes safety and security. The standard puts requirements over dependability analysis and states "*dependability analysis shall be conducted on all levels of the space system and be performed in respect of the level that is being assessed i.e., System, Subsystem and Equipment levels*".
- ECSS-Q-ST-40C, defines the requirements on space product assurance focused on Safety. The standard requires that hazard analysis shall be conducted to identify the hazards. Also, it states that "*The fault tree analysis shall be used to establish the systematic link between the system level hazard and the contributing hazardous events and subsystems, equipment or piece part failure*".
- ESSB-ST-E-008, defines the requirements for secure engineering of the space software product. The standard is focused on the security of software product and states that "*The supplier shall perform a cyber-security risk assessment of the software products in order to determine the security sensitivity of the individual software components*".
- ECSS-Q-ST-80C, lists the requirements for software product assurance with emphasise on dependability and safety. The standard state "*The supplier shall perform a software dependability and safety analysis of the software products, in accordance with the requirements of ECSS-Q-ST-30 and ECSS-Q-ST-40 and using the results of system level safety and dependability analysis, in order to determine the criticality of the individual software components*".

2.2 ConcertoFLA

ConcertoFLA is a tool-supported methodology for the compositional calculation of the failure behaviour of component-based systems, based on the failure behaviour of individual components. The failure behaviour is specified using an adaptation [8] in the CHESS context of Failure Propagation Transform Calculus (FPTC) [7] rules. Each FPTC rule defines the input/output behaviour of a specific component using a combination of the port name and the guide-word/failure mode. ConcertoFLA supports three types of failure modes with two specializations for each – the failure modes are value (coarse/subtle), timing (early, late), provision (omission, commission). Using the FPTC rules, four different behaviours of a component can be defined, which are as following:

- Propagator, a component propagates the fault it received on its input port to the output port without changing the type of the fault.
- Transformer, component transform the fault received on its input port into another type of the fault.
- Sink, component sinks the fault it receives on its input port and produces no fault on its output port.
- Source, component is the source of the fault on its output port and received no fault on its input port.

3 ECSS-compliant Multi-concern assurance approach

As recalled in Section 2.1, ECSS standards require the assurance and analysis of several non-functional properties of the system. The CHESS methodology and ConcertoFLA, recalled in Section 1, are customized for performing multi-concern assurance, focusing on three concerns, i.e., safety, security, and reliability. The overall approach, resulting from the customization, consists of five activities, as the activity diagrams, depicted in Figure 1, shows. These activities are:

1. System design- The system architecture is specified using CHESSML. First, all the components in isolation are specified and then assembled.
2. Individual component failure behavior specification using FPTC rules. As stated in Section 2.2, the failure modes used are of high abstraction. The advantage of this abstraction is the support for the assembly of heterogeneous components e.g., developed in different domains with different specialized terminology. In this paper, the above-mentioned abstraction facilitates the

interpretation of the failure modes for different concerns.

3. Behaviour injection and ConcertoFLA execution to calculate the failure behavior at system level. The analysis generates failure propagation paths, which consist of the sequences of the possible events leading to the system level failures, as a consequence of the injected behavior (including fault(s) injection, i.e., failure(s) of preceding systems feeding the system under analysis as well as normal behaviour to potentially detect components acting as sources).

4. Interpretation (conducted manually) of the analysis results for multi-concern e.g., reliability, safety and security concerns. Next, a trade-off is calculated between these properties. Base on the interpretation for multi-concern and trade off, dependability means are introduced by refactoring the system design, if the certain level of dependability is not achieved.

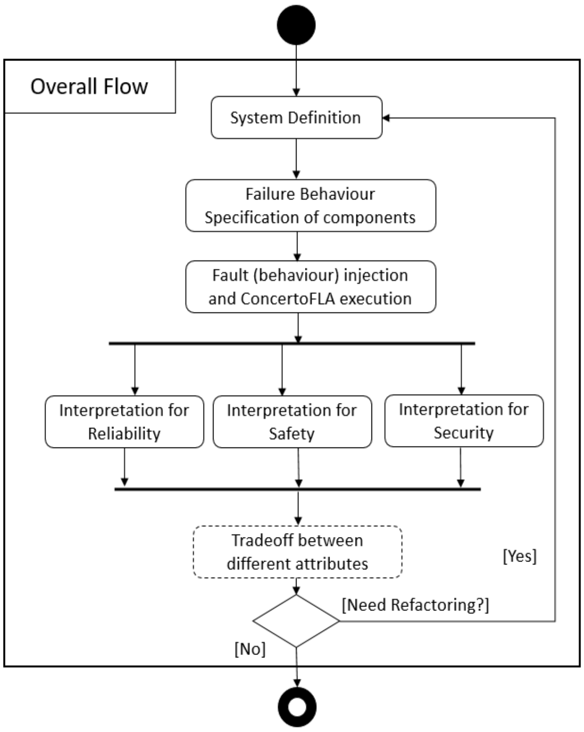

Overall Flow
System Definition
Failure Behaviour Specification of components
Fault (behaviour) injection and ConcertoFLA execution
Interpretation for Reliability
Interpretation for Safety
Interpretation for Security
Tradeoff between different attributes
[Yes]
[Need Refactoring?]
[No]

Figure 1 Multi -concern assurance approach

4 Application of Approach to Attitude Control System Engineering

In this section, first we describe the space system used for illustration purposes, then, we apply our approach to it.

4.1 Attitude Control System (ACS)

The ACS of a satellite is an on-board subsystem that controls the orientation of the satellite, relative to a reference frame, in space. For projects developed for European Space Agency (ESA), an ACS is normally developed according to ECSS standards, therefore its engineering is required to comply with the ECSS standards and a certain level of dependability, safety and security of software is assured. ACS engineering includes activities spanning performance analysis, budgets, procurement and dimensioning of sensors and actuators etc., along with the ACS development. ACS development refers to the development of ACS application software and its associated algorithms.

The ACS (application) software takes sensor data containing information about the current state of the satellite and computes the control torque to be applied to the satellite body in order to achieve its target state. To do this, ACS has three functions i.e., process unit data, state estimation and computation of the control torques to minimize the difference between current and target state. ACS has different operational modes, which involves different devices and reflects the mission requirements. For example, in Sun Acquisition and Survival Mode (SASM) it is required to control the orientation of the satellite relative to the Sun to ensure sufficient solar power to the system. The SASM normally takes inputs from sun sensors and a gyroscope to compute a torque that is applied to the satellite body e.g. using propulsion thrusters.

4.1 Application

We apply our approach to the ACS in SASM mode. We limit the scope of functions of ACS to the control function, which maintains the target state in response to the estimated state. The functional requirements of control function in SASM mode are as following.

The sun acquisition control function shall compute and output a control torque based on PD controller, gyroscopic torque compensation and deadband filter in order to point the satellite (its reference direction) at the Sun.

To design the system with above-mentioned requirement, a component based model is defined using CHESS modelling environment. Figure 2 shows the assembly of the following four components implementing the SASM control function requirement.

- PDController, computes the proportional and derivative torque to orient the satellite relative to the Sun.
- SteerController, computes the proportional torque using different gains and control law.
- FeedforwController, compensates for the gyroscopic coupling.
- TorqueSelector, selects the control torque based on the current state of satellite via choosing between two control strategy to enhance the performance and fast convergence to the target orientation.

The next step, after system definition, is to model dependability and perform ConcertoFLA analysis. In this regard, we modelled the failure behaviour of components as a propagator and injected the system with the failure of type “value”. It has been assumed that the injected failure, is due to the failure in state estimation unit of satellite and refers to the “state estimator unit provides inaccurate value” failure. Upon execution of ConcertoFLA analysis, the failure propagation paths are generated providing the failure behaviour at system level. To interpret the results

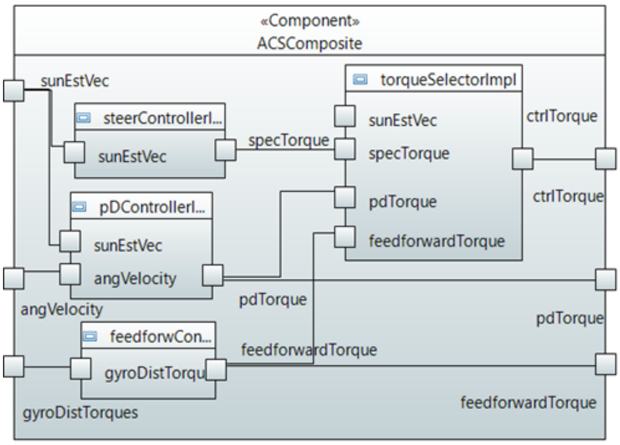


Figure 2 Component based design of ACS

for reliability, a fault tree can be constructed manually following the failure propagation paths. The system level failure, which refers to the "ACS computing inaccurate torques" is due to the value failure at "ctrlTorque" output port of ACS system. A partial manually constructed fault tree is depicted in Figure 3. To interpret the results for safety concern, the top event of the fault tree refers to a hazardous event, which is the combination of system level failure and the operational situation e.g., "ACS computing inaccurate torques in SASM mode" leads to a catastrophic consequences. To interpret the results for the security, the top event of fault tree refers to a security threat which is loss of one or more security properties i.e., confidentiality, integrity and availability.

5 Conclusion and Future work

In this paper, we presented the customization of the CHESS methodology and ConcertoFLA in the context of the ECSS standards to enable architects and dependability engineers to define a system and perform dependability-centered co-analysis for assuring the required non-functional properties of the system according to ECSS requirements. Then, we applied our customization in the context of the Attitude Control Systems engineering.

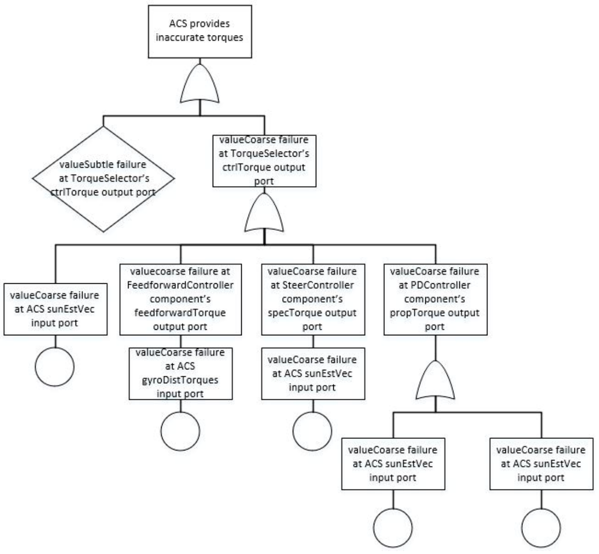

ACS provides inaccurate torques
valueSubtle failure at TorqueSelector's ctrlTorque output port
valueCoarse failure at TorqueSelector's ctrlTorque output port
valueCoarse failure at ACS sunEstVec input port
valuecoarse failure at FeedforwardController component's feedforwardTorque output port
valueCoarse failure at SteerController component's specTorque output port
valueCoarse failure at PDController component's propTorque output port
valueCoarse failure at ACS gyroDistTorques input port
valueCoarse failure at ACS sunEstVec input port
valueCoarse failure at ACS sunEstVec input port
valueCoarse failure at ACS sunEstVec input port

Figure 3 Manually constructed partial fault tree adapted from [18]

From that application it emerged that CHESSML is appropriate to design the ACS in compliance with the requirements of ECSS-E-ST-40C. More precisely, the CHESSML based design complies with Section 5.4.3 of that standard, which is focused on the software architectural design and requires the component based design. The analysis part of CHESSML i.e., ConcertoFLA supported the requirements focused on the assurance of software reliability, safety and security. Moreover, the certifiable evidences could be manually constructed to support the qualification process.

We also observed that the employment of CHESS toolset supports the end to end process, where the functional design, annotated with non-functional properties and assurance support, could shorten the feedback loop for mastering the improved design as well as reduces the complexity.

In the future, we plan to provide tool support for the manual interpretation and construction of evidences for multi concerns. In this regard, our recent work [16] automatically generates the fault tree for reliability from the ConcertoFLA results.

Acknowledgements

This work is supported by the EU and VINNOVA via the E CSEL project AMASS (No 692474) [17] [15].

References

[1] ECSS (2016), *ESSB-ST-E-008 - Secure Software Engineering Standard.*

[2] ECSS (2009), *ECSS-E-ST-40C, Space engineering – Software.*

[3] ECSS (2009), *ECSS-Q-ST-80C, Space product assurance - Software product assurance.*

[4] ECSS (2009), *ECSS-Q-ST-30C, Space product assurance - Dependability.*

[5] ECSS (2009), *ECSS-Q-ST-40C, Space product assurance - Safety.*

[6] B. Gallina, E. Sefer, A. Refsdal (2014), *Towards Safety Risk Assessment of Socio-Technical Systems via Failure Logic Analysis,* IEEE International Symposium on Software Reliability Engineering Workshops, Naples, pp. 287-292.

[7] M. Wallace (2005), *Modular architectural representation and analysis of fault propagation and transformation*, Electronic Notes in Theoretical Computer Science, volume 141 n.3, pp. 53-71.

[8] B. Gallina, M. A. Javed, F. UL Muram, S. A. Punnekkat (2012), *Model-Driven Dependability Analysis Method for Component-Based Architectures*, 38th Euromicro Conference on Software Engineering

and Advanced Applications (SEAA) Cesme, Izmir, pp. 233-240.

[9] CHESSML, https://www.polarsys.org/chess/start.html.

[10] ARTEMIS-JU-100022 – *CHESS-Composition with guarantees for High integrity Embedded Software components assembly*, http://www.chess-project.org.

[11] ARTEMIS-JU *CONCERTO - Guaranteed Component Assembly with Round Trip Analysis for Energy Efficient High-integrity Multi-core systems,* http://www.concerto-project.org

[12] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr (2004), *Basic concepts and taxonomy of dependable and secure computing,* in IEEE Trans. Dependable Sec. Comput. 1(1): 11-33.

[13] S. Mazzini, J. Favaro, S. Puri, L. Baracchi (2016), *CHESS: an open source methodology and toolset for the development of critical systems*, Third Workshop on Open Source Software for Model Driven Engineering.

[14] L. Montecchi, B. Gallina (2017), *SafeConcert: a Metamodel for a Concerted Safety Modeling of Socio-Technical Systems*, 5th International Symposium on Model-Based Safety and Assessment (IMBSA), Trento, Italy.

[15] A. Ruiz, B. Gallina, J. L. de la Vara, S. Mazzini, H. Espinoza (2016), *Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems*, Computer Safety, Reliability, and Security, SAFECOMP, LNCS, vol. 9923, Springer.

[16] Z. Haider, B. Gallina and E. M. Zornoza (2018), *FLA2FT: Automatic generation of fault tree from ConcertoFLA results*, 3rd International Conference on System Reliability and Safety (ICSRS), Barcelona.

[17] AMASS, http://www.amass-ecsel.eu

[18] B. Gallina, Z. Haider, A. Carlsson, *Towards generating ECSS compliant fault tree analysis results via ConcertoFLA*, IOP Conference Series: Materials Science and Engineering.

[19] Object Management Group (2015), *SysML v1.4 Specification Release*, http://www.omgsysml.org/specifications.htm

[20] Object Management Group, *MARTE Specification*, www.omg.org/spec/MARTE/About-MARTE/

[21] B. Gallina, Z. Haider, A. Carlsson, S. Mazzini, S. Puri (2018), *Multi-concern Dependability-centered Assurance for Space Systems via ConcertoFLA*, 23rd International Conference on Reliable Software Technologies-Industrial Presentation Track (Ada-Europe), Lisbon, Portugal.

[22] Object Management Group, *Unified Modeling Language*, www.omg.org/spec/UML/2.5.1/.

[23] Papyrus, www.eclipse.org/papyrus/.

Concurrent Reactive Objects in Rust Secure by Construction

Marcus Lindner, Jorge Aparicio, Per Lindgren

Luleå University of Technology, Sweden; email: {marcus.lindner@,jorapa-7@student.,per.lindgren@}ltu.se

Abstract

Embedded systems of the IoT era face the software developer with requirements on a mix of resource efficiency, real-time, safety, and security properties. As of today, C/C++ programming dominates the mainstream of embedded development, which leaves ensuring system wide properties mainly at the hands of the programmer. We adopt a programming model and accompanying framework implementation that leverages on the memory model, type system, and zero-cost abstractions of the Rust language. Based on the outset of reactivity, a software developer models a system in terms of Concurrent Reactive Objects (CROs) hierarchically grouped into Concurrent Reactive Components (CRCs) with communication captured in terms of time constrained synchronous and asynchronous messages. The developer declaratively defines the system, from which a static system instance can be derived and analyzed. A system designed in the proposed CRC framework has the outstanding properties of efficient, memory safe, race-, and deadlock-free preemptive (single-core) execution with predictable real-time properties. In this paper, we further explore the Rust memory model and the CRC framework towards systems being secure by construction. In particular, we show that permissions granted can be freely delegated without any risk of leakage outside the intended set of components. Moreover, the model guarantees permissions to be authentic, i.e., neither manipulated nor faked. Finally, the model guarantees permissions to be temporal, i.e., never to outlive the granted authority. We believe and argue that these properties offer the fundamental primitives for building secure by construction applications and demonstrate its feasibility on a small case study, a wireless autonomous system based on an ARM Cortex M3 target.

1 Introduction and motivation

Besides constraints set by the environment and the target platform like available memory, CPU, and energy resources in addition to other functional and extra-functional properties of the application at hand, embedded software typically operates autonomously with requirements on safety, robustness, reliability, and security. Developers commonly design embedded systems of the IoT era by taking the outset of a reactive model implemented in C/C++ either as a bare metal interrupt driven application or through the support of some threading library. Meeting the aforementioned requirements is at a large up to the programmer with little or no support for verification. Central to correctness is the management of memory resources with problems spanning from array indexing and dangling pointers all the way to race conditions and deadlocks in the concurrent setting.

In this paper, we take the outset from prior work on Concurrent Reactive Objects (CROs) [1] with a heritage to the Timber language [2] and the Real-Time For the Masses (RTFM, [3]) set of experimental languages and tools. Whereas Timber provides a high level modeling and implementation approach offering state protection in the concurrent setting, the dynamic memory model requires automatic management which precludes the deployment to lightweight targets.

With a clear motivation, we want to provide a programming model that ensures memory safety in a concurrent setting along with a concurrency model amenable to static analysis. However, developing yet a new fully fledged language with accompanying ecosystem is questionable when taking the amount of work into consideration[1]. Instead, we seek to leverage on ongoing community efforts around programming languages and ecosystems.

Among recent developments, the Rust language stands out with a memory model, which provides compile time memory safety and monomorphization, and has a tight coupling to LLVM achieving zero-cost abstractions through link time optimization. Sidestepping the compiler is explicit (`unsafe`) and can be rejected in user code, thus allowing for fearless programming to the end of memory safety and other properties within reach of the Rust compiler. In the context of embedded development, Rust applications on bare metal targets have already been shown possible [4, 5].

In this paper, we further explore the Rust memory model and the CRC framework towards systems being secure by construction. In particular, we show the following properties.

- Granted permissions can be freely delegated without any risk of leakage outside the intended set of components. Key here is the static CRC topology, where communication paths are known at compile time, together with the Rust language borrowing semantics.

[1] An observation here is that the design of any memory safe language would need to take memory aliasing into account, a property directly given by the Rust language.

- Permissions are guaranteed to be authentic, i.e., they can neither be manipulated nor faked. Key here is the underlying module system and type scoping together with the memory safety provided by the Rust language. In effect, preventing any intentional or accidental memory corruption leads up to an unauthentic permission.
- Permissions are guaranteed to be temporal, i.e., they can never outlive the granting authority. Key here is the concept of lifetimes, which the Rust language brings and the compiler enforces.

In conclusion, we believe and argue that these properties offer the fundamental primitives for building secure by construction applications and demonstrate their feasibility on a small case study, a wireless autonomous system based on an ARM Cortex M3 target.

2 Background

The Rust memory model and the Stack Resource Policy (SRP) based scheduling approach is at heart of the proposed framework.

The ambition behind Rust is to provide a systems programming language with memory safe zero-cost abstractions. In Rust, mutability is first class, distinguishing between *immutable* (`&T`) and *mutable* (`&mut T`) references with the following invariant:

> At any instance in time each value of `T` may be *mutably* referenced once or *immutably* referenced zero or arbitrarily many times.

In Figure 1, A and B denote two concurrent execution contexts while a, b, and c are references to a shared location or resource T. The invariant applies to (1) the concurrent case with accesses from a of context A and b/c of context B and (2) the sequential case with accesses from b and c of context B. For the concurrent case (1), the invariant ensures obviously race free access while for the sequential case (2), the invariant may at first glance appear too restrictive. However, the sequential restriction allows to spot and reject at compile time memory related issues such as iterator invalidation (see Section 4.9 in [6]). Moreover, the invariants are passed to the compiler back-end (LLVM) as *no alias* attributes allowing aggressive yet safe code optimization.

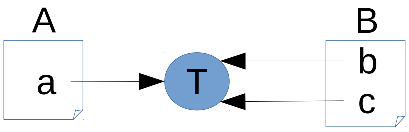


Figure 1: Illustration of the Rust memory model.

Rust implements an *affine* type system and an *ownership* model with the notion of *lifetimes*. The *borrow checker* and lifetime analysis ensures memory consistency of *safe* code.

Any access to shared mutable data ultimately boils down to explicitly stated *unsafe* code, out of reach for the Rust compiler to verify. Abstractions providing a *safe* API allow user code to access shared mutable data. Thus, given that the abstractions are sound (i.e., uphold the invariant), any program passing compilation is memory safe by construction![2]

2.0.1 Stack Resource Policy based scheduling

Stack Resource Policy (SRP) based scheduling offers a means to preemptive scheduling of tasks with shared resources on single-core processors [7]. The approach offers advantages in terms of deadlock-free execution, efficient memory utilization, single blocking, and so on and brings a plethora of readily available methods for static analysis, see, e.g., [7, 8].

SRP builds upon static analysis of the task set T to derive the ceiling value $\pi(r)$ for each resource r. $\pi(r) = max(p(t))$ with $t \in T_l(r)$, where $p(t)$ is the priority of task t and $T_l(r)$ is the set of tasks that (may) access the resource r. During execution, the (dynamic) system ceiling $\Pi = max(\pi(r))$ with $r \in L$, where L is the (global) set of currently held resources. A task $t \in T$ may preempt the currently running task t_e only if $p(t) > p(t_e)$, $p(t) > \Pi$, and $p(t) = max(p(t'))$ with $t' \in P$, where P is the set of pending tasks.

Targeting lightweight MCUs, we can exploit the underlying interrupt hardware to implement the system ceiling and perform static priority scheduling[3]in compliance to SRP, achieving performance on par with hand written bare metal code [3].

3 Model of computation

Component models are frequently used to capture the system topology and to bring the benefit of re-use. Our system model is declarative, defined in terms of nested Concurrent Reactive Components (CRCs) with Concurrent Reactive Objects (CROs) at the leafs. The system designer declares interaction inside the model and with its environment in terms of time constrained synchronous and asynchronous point-to-point messages, where ultimately the end points are methods of CRO leaf instances.

3.1 Execution semantics

The execution model builds on the notion of *time constrained messages* defined as

$$M : \{BL : Ti, dl : Ti, o : \&O, f : (\&O, D) \rightarrow R, d : D\},$$

where Ti is a time type, BL specifies the absolute release time, dl specifies the relative deadline, o indicates the target object, f indicates the method to execute, and d is the payload (i.e., the arguments for the receiver).

The execution of a message

$$E(m : M) \rightarrow R$$

returns with a value of type R. Messages execute concurrently under mutual exclusion on the object state (o) (similar to Ada's protected objects or Java's synchronized methods) and run-to-completion within their eligible timing window for any correctly scheduled system.

[2] Memory safety can in most cases be statically ensured. If not, a run-time monitor is injected to emit a `panic!` on a memory violation. Stack memory allocation errors (overruns) are assumed to be treated at the run-time system level.

[3] Eligible tasks with the same priority are scheduled in static order. While preserving invariants for correctness, we must take this into consideration during the response time analysis.

3.2 Timing semantics

The absolute release time BL along with the absolute deadline $DL = BL + dl$ define the eligible timing window for the execution of a message m. The execution $E(m : M)$ of a message m may emit additional synchronous messages

$$Sync(o' : \&O', f' : (\&O', D'), d' : D') \to m' : M,$$

which result in messages

$$m' = M\{BL = m.BL, dl = m.dl, o = o', f = f', d = d'\}$$

that *inherit* the sender's timing window. The synchronous execution $E(m' : M) \to r : R$ blocks the sender and returns the value r.

Similarly, the sender may emit asynchronous messages with a relative release time bl''

$$Async(bl'' : Ti, dl'' : Ti, o'' : \&O'', f'' : (\&O'', D''), \\ d'' : D'') \to m'' : M,$$

which result in messages

$$m'' = M\{BL = m.BL + bl'', dl = dl'', \\ o = o'', f = f'', d = d''\}$$

with a timing window *relative* to the sender's ($E(m : M)$) timing window. Emitting an asynchronous message m'' amounts to queuing the message for later execution. The emission of an asynchronous message returns a reference to that message, which allows the cancellation of the message as long as its execution is not yet scheduled[4].

3.3 Discussion

The CRO model resembles *actor* models in that the execution of asynchronous messages is decoupled from the sender. However, the notion of synchronous communication is usually not found in actor models, while here supported with resemblance to *monitors* and *protected* objects. Messages execute under mutual exclusion on the corresponding object (resource). This not only allows race-free execution by construction but also ensures sequential behavior of operations holding a resource. This is instrumental to control the order of side effects not only on object states but also for communication, i.e., synchronous calling of other objects and communication with the environment. Asynchronous messages are the units of concurrency with the execution semantics precisely defined by their resource dependencies, where mutual exclusion is the sole (necessary and sufficient) means to synchronization.

In this paper, we target lightweight MCUs and adopt an SRP based scheduling approach, where the asynchronous messages constitute SRP *tasks* and objects amount to (shared) SRP *resources*.

At the border of the system, we find the environment, which drives our reactive model, represented as event or message sources. Internal events and actions become *observable* only at the point where communication involves the environment. In the setting of embedded targets, the environment is typically represented by the hardware peripherals, where the interrupt handlers are our event or message sources. This can be generalized to APIs of external code and hosted environments [9], where the underlying operating system schedules our tasks on top of its thread model and the external code emits messages or events.

To facilitate re-use and to manage complexity, the model provides a hierarchical component based abstraction. The declarative definition allows us to statically analyze the topology of the system and derive a flat system instance without the need of dynamic bindings. As we show in the remainder of the paper, the CRC/CRO model can be implemented efficiently using zero-cost abstractions of the Rust language and rendering executables that perform on par with carefully designed bare-metal code.

4 LED runner example

Figure 2 depicts a CRC system, which autonomously controls the RGB values of an LED array. At the highest level, the system consists of two components, the `USART` CRO and the `LED` CRC wrapping the `STM` state machine and the `DMA` CROs. The `USART` receives and parses the serial stream and controls the `LED` component. The `DMA` CRO sends a frame of data to the LED array utilizing the DMA hardware. The `STM` CRO triggers on behalf of the periodically executing `transition` method the `on_update` method, which generates the frame content. The `on_command` method controls the behavior of the state machine, i.e., the direction and speed of the running lights. The `transition` method emits an asynchronous message with a baseline offset to postpone the release of the message, which implements the periodic behavior.

5 CRC framework

With the CRC framework, a developer specifies the system topology in terms of CROs and CRCs through `.cro` and `.crc` files, respectively. The developer provides the behavior of the CROs in form of standard Rust code, i.e., `.rs` files.

A *build script* (`build.rs`) is the Rust mechanism for code generation *before* compilation. Our framework uses a build script to analyze the system model, transform `.cro` and `.crc` files into actual Rust code, and inject it into the compilation process.

A `.cro` file for each CRO stores its specification. Listing 1 shows the definition of the `USART` CRO. The file specifies port signatures (`signature`) with Rust syntax. Each input port enumerates internal connections to output ports (`sync_ports`/`async_ports`) and peripheral dependencies (`peripherals`).

A `.crc` file for each CRC stores its specification. Listing 2 shows the main wrapping CRC of our example system. A CRC consists of CRO and other CRC instances, referred to

[4] We have not yet implemented this feature in the prototype Rust framework.

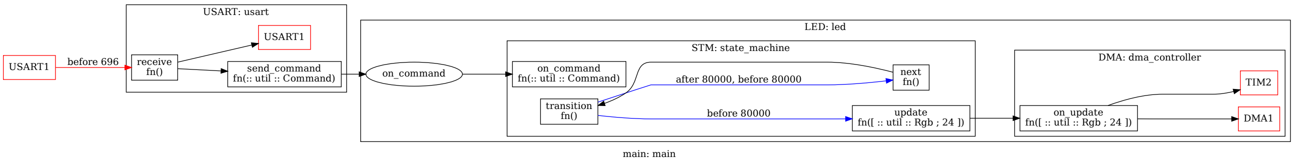


Figure 2: The LED runner CRC system autonomously controls the RGB values of an LED array. Synchronous messages are marked black and asynchronous messages are marked red for environmental messages and blue for internal messages.

```
inputs: {
    receive: {
        signature: fn(),
        sync_ports: [send_command],
        peripherals: [USART1],
    },
},

outputs: {
    send_command: fn(::util::Command),
}
```

Listing 1: File `usart.cro` (template).

as `components`. The `.crc` file specifies incoming and outgoing connections to each component on a per-component basis. Finally, the CRC itself has input and output ports. The analysis stage of the framework derives these ports from the `connections` information in the file. If a port specifies no component, it is a CRC port. The `interrupts` field defines the interrupt sources of the CRC and according component connections. The `available` field contains a list of additional interrupt sources that may be used to dispatch async messages while `device` indicates the crate (i.e., library) holding the peripheral API.

Our framework derives a system instance by spanning the top level CRC[5].

Using our CRC framework, a user implements the CRO application logic in safe Rust code. Listing 3 partly shows the state machine implementation of our example system.

A `State` struct represents the state of the CRO and defines input ports as methods on the struct. The `new` constructor initializes the state of the CRO, which the compiler evaluates at compile time (due to the `const` context).

The exact signature of each input port varies according to its `.cro` file specification. Input port methods can have as arguments a mixture of:

1. The port input (i.e., message payloads),
2. A set of output ports both synchronous and asynchronous, and
3. A set of peripherals.

The `Ports` struct provides the output ports, which are essentially normal Rust functions (see lines 19 and 23 in Listing 3).

```
components: {
    USART: {
        template: usart,
        connections: [
            self.send_command
                        -> LED.on_command,
        ],
    },

    LED: {
        template: led,
    },

},

interrupts: {
    USART1: {
        connects_to: USART.receive,
        before: 696, // 87 us
    },

    available: [EXTI0],
},

device: stm32f103xx,
```

Listing 2: File `main.crc`.

```
1  use util::{Command, Direction, Rgb};
2  cro!(); // include auto generated code
3
4  pub struct State {
5      active: bool,
6      rgb: [Rgb; 24],
7      ...
8  }
9
10 impl State {
11     pub const fn new() -> Self {
12         State { active: false, ... }
13     }
14
15     pub fn transition(
16         &mut self,
17         port: self::transition::Ports,
18     ) {
19         (port.async.next)();
20
21         if self.active {
22             ...
23             (port.async.update)(self.rgb);
24         ...
```

Listing 3: File `state_machine.rs` (abridged).

[5] Currently, our framework allows `available` and `device` fields only in the top level CRC, but we will remove this restriction in the future to allow cross crate component re-use.

```rust
// root of the crate
extern crate stm32f103xx; // target device
extern crate blue_pill; // development board

crc!(); // indicates that it is a CRC system
```

Listing 4: File `main.rs`.

```rust
1  use stm32f103xx;
2  use blue_pill::{Channel, Pwm};
3  cro!();
4
5  pub struct State {
6      buffer: ...,
7  }
8
9  impl State {
10     ...
11
12     pub fn on_update(
13         &mut self,
14         rgb: [Rgb; 24],
15         p: self::on_update::Peripherals,
16     ) {
17         p.DMA1.claim(|dma1| {
18             p.TIM2.claim(|tim2| {
19                 let pwm = Pwm(tim2);
20                 pwm.set_duties(
21                     dma1,
22                     Channel::_1,
23                     &self.buffer
24                 ).unwrap();
25             });
26             ...
27         });
28     }
29 }
```

Listing 5: File `dma_controller.rs`.

The root of the crate (see file `main.rs` in Listing 4) lists all library dependencies and indicates with a `crc!()` macro call that this is a CRC system.

5.1 Implementation

5.1.1 Peripherals

We opted to implement access to peripherals as structs with interior mutability. Thus, mutation of the registers is possible through shared (`&T`) references. The `svd2rust` [10] tool automatically generates a register-level API for peripherals from vendor based SVD files. CROs can access peripherals as if they were resources using a `claim` method and passing a closure. Nested closures allow the access to multiple peripherals (see lines 17-25 in Listing 5). Internally, the peripheral APIs use *volatile* read/write operations.

5.1.2 Analysis

The build script collects all `.cro` and `.crc` files, parses them, and combines them into a system model sys or rejects ill-formed models (see Section 7.1).

During the analysis stage, we derive for the SRP scheduling the task priorities (i.e., interrupt priorities) from the given timing constraints and the resource ceilings from the task priorities [7,9].

5.2 Code generation

If sys is well-formed, the build script proceeds to generate Rust code required for run-time execution.

5.2.1 CROs

For each CRO, our framework generates a Rust file with the definition of the `Ports` and `Peripherals` structs. The `cro!` macro injects this file into the compilation process, which allows the Rus compiler (`rustc`) to reject user code that mismatches port signatures specified in the corresponding `.cro` file.

5.2.2 Top level CRC

Our framework generates a Rust file for the whole system that contains the full application logic. The `crc!` macro injects this file into the compilation process.

This file contains a module for each CRO instance in the system, which statically allocates a `State` struct per CRO instance. The code in the system file also optimizes connections by refining synchronous messages to function calls and asynchronous messages to enqueue operations.

Synchronous messages: For every CRO instance, our framework generates a proxy function for each input port. The signature of this proxy matches the signature entered by the user in the `.cro` file. Instantiating the `Ports` struct of a sending CRO with the proxy that matches an actual connection to a receiving CRO expresses the connection between the two CRO instances.

Asynchronous messages: An asynchronous message defers the invocation of the input port (i.e., the object method) by storing the input data and the input port function pointer in a queue. An interrupt handler executes asynchronous messages at a later time. The message struct holds a `next` field forming an *in place* linked list. In order to store differently typed objects because asynchronous messages differ in the payload field type, we enforce a static layout of all message types (`#[repr(C)]`) ensuring the `next` field to have a known offset.

6 Security

Embedded systems often access and process sensitive data. For this reason, security is an important factor that may not be disregarded when designing and developing embedded software. Ravi et al. [11] argue it is wrong to establish security by solely adding features like encryption to a system. Instead, we have to take all aspects of embedded system design into account together with existing resource constraints, e.g., performance and power limitations.

In the context of lightweight embedded systems, resource constraints come into play, thus memory and CPU efficiency

```
mod trusted {
    pub struct Auth {
        level: u8,
    }

    pub fn auth(k: &str) -> Option<&Auth> {
        if k == "abc" {
            Some(&Auth { level: 1 })
        } else {
            None
        }
    }
    ...
```

Listing 6: File `trusted_base.rs` (abridged).

become an issue. The emerging security enabled microcontrollers exemplify this. They range from hardware AES encryption like the ARM Cortex-M3 based stm32l162vc to more elaborate solutions like the Cortex-M4 based CEC1702 offering hardware encryption, authentication, and public key capabilities. Hardware cryptographic ciphering may offer speedup and increase energy savings by orders of magnitude over software solutions. Moreover, pre-boot authentication of system firmware offering a root of trust, firmware update authentication, authentication of system critical commands, and protection of secrets with encryption improves system integrity.

In this paper, we focus on software in security mechanisms from the outset of the *platform agnostic* CRC framework and leverage properties of the Rust language to establish security mechanisms, which are guaranteed by the Rust semantics and statically ensured by the Rust compiler. While being complementary to security mechanisms offered by the underlying hardware, we argue that a higher degree of trust and reliability can be achieved by also offering compile time guarantees to the embedded software.

6.1 Authentication and authorization

At device level, we are concerned with the permissions to access data and perform operations. The trusted base has the authority to grant such device level permissions based on a-priory knowledge or external authentication. For this presentation, we focus on device level authorization mechanisms and discuss system level authorization as future work.

The notion of *opaque* structures in the Rust language allows us to define data types that a user can neither construct, nor inspect, nor manipulate, merely pass on as parameters. This perfectly fits the need and purpose of device level authorization, where the trusted base grants permission to the user.

Listing 6 demonstrates an implementation of an authorization ticket providing a range `u8` of permission levels. Note, the `Auth` structure is public but its `level` data field is private to the module, i.e., code from other modules can hold a reference to an `Auth` structure but not create it or access the containing data. Therefore, the public `auth` function returns an authorization ticket in case the input matches the defined a-priory knowledge (here `"abc"`).

```
1 // user code in `safe` Rust
2 fn user1(d: &Sec<u32>, e: &Enc<u32>) {
3     let a = auth("abc").unwrap();
4     user2(&sec_add_u32(d, &e.get(a)));
5     user3(d, &e.get(a));
6     user4(d, e, a);
7 }
```

Listing 7: File `user1.rs`.

By default, `structs` in Rust do not implement the `Copy` trait, thus user code cannot duplicate the ticket but a reference thereof. Moreover, tickets are temporal with a lifetime limited to the sequential execution context of the call to the granting authority. This follows from the Rust lifetime semantics.

Listing 7 depicts user code written in “safe” Rust. The user requests authorization in line 3 and uses the given permission `a` locally in lines 4 and 5. In line 6, the user passes the permission ticket on to another user function. For brevity, the example illustrates the concept with plain Rust code but permission delegation is also possible through synchronous messages. However, asynchronous messages cannot store the received ticket due to the lifetime bound and consequently the temporal property of the authorization holds.

6.2 Secure data container

Dealing with sensitive data sets a number of restrictions and requirements regarding integrity, use, and visibility. Specifically, *integrity* restricts primitive operations on sensitive data to be limited to the trusted base, an authority limits the *use* of sensitive data, and the designers intention limits its *visibility*. Also to this end, we can ensure the desired behavior through an opaque representation of secure data.

Listing 8 demonstrates an implementation of a generic (i.e., polymorphic to the type `T`) secure data container `Sec<T>`. Only the trusted base code can instantiate and delegate this type. The function `sec_add_u32` (lines 18 to 20) exemplifies how to declare primitive operations on arbitrary instances of secure data containers in the trusted base. While the function internally uses `unsafe` code, the API is *safe*, i.e., *safe* Rust code can call the function[6]. Notice here, user code has never access to the inner *secure* data or can disclose it because the return type is also a secure data container `Sec<u32>`.

6.3 Encryption and decryption

While cryptography as such is not the focus of this work, we discuss the topic from the framework perspective and highlight outsets for efficient, reliable, and secure management of sensitive information. To this end, we leverage on the Rust language zero-cost abstractions and type system with static guarantees offered by the compiler.

[6] An alternative to `unsafe` is to use the `pub(crate)` modifier and use visibility as a fence for usage violations.

```rust
...
// opaque representation of secure data
#[derive(Debug)]
pub struct Sec<T> {
    data: T,
}

impl<T> Sec<T> {
    pub unsafe fn new(d: T) -> Self {
        Sec { data: d }
    }
    pub unsafe fn get(&self) -> &T {
        &self.data
    }
}

// safe API for operating on Sec<u32>
pub fn sec_add_u32(s1: &Sec<u32>,
                   s2: &Sec<u32>)
    -> Sec<u32> {
    unsafe { Sec::new(s1.get() + s2.get()) }
}
...
```

Listing 8: File `trusted_base.rs` (continued).

```rust
    ...
    // in place transformation
    // by a cipher closure f
    fn cipher<T, F>(s: &mut T, mut f: F)
    where
        T: Sized,
        F: FnMut(&mut u8),
    {
        let ptr = s as *mut T as *mut u8;
        for i in 0..size_of::<T>() {
            f( unsafe {
                &mut *ptr.offset(i as isize)
            });
        }
    }
    ...
```

Listing 9: File `trusted_base.rs` (continued).

6.3.1 *Cipher*

A fully fledged cryptography crate (rust-crypto = "0.2.36") is readily available providing implementations for popular ciphers (AES, RC4, and others). While strong encryption by software is likely resource consuming and may thus be out of range for light-weight targets, a microcontroller may defer the actual encryption and decryption to a capable encryption hardware if supported by the target.

To the purpose of this presentation, Listing 9 demonstrates an in place transformation of raw data. The generic function `cipher<T, F>` iterates the closure `f:F` over the byte array representation of the data `s` and transforms it. With a suitable cipher closure `f`, the function encrypts or decrypts the data.

6.3.2 *Encrypted data container*

As we have already seen in Section 6.1, authorization tickets are secure against faking and manipulation in user code. We can use this approach to delegate secure information keyed with an authorization ticket. Also here, we take the outset of an opaque type definition[7].

```rust
1      ...
2      // opaque representation of
3      // encrypted data
4      pub struct Enc<T> {
5          data: T,
6      }
7
8      impl<T> Enc<T>
9      where
10         T: Copy,
11     {
12         pub unsafe fn new(d: &T) -> Self {
13             let mut c = d.clone();
14             cipher(&mut c, |i| { *i += 1; });
15             Enc { data: c }
16         }
17
18         pub unsafe fn get_unsafe(&self)
19           -> Sec<T> {
20             let mut c = self.data.clone();
21             cipher(&mut c, |i| { *i -= 1; });
22             Sec::new(c)
23         }
24
25         pub fn get(&self, _: &Auth)
26           -> Sec<T> {
27             unsafe { self.get_unsafe() }
28         }
29     }
30 }
```

Listing 10: File `trusted_base.rs` (end).

Listing 10 demonstrates an implementation of a generic encrypted data container `Enc<T>`. The signature of the `new` constructor specifies the `unsafe` modifier, and as a consequence, solely the trusted base code can call the function and create a new encrypted data container. When calling `new`, the constructor applies the `cipher` function to a copy of the data (lines 13 and 14) and returns the encrypted data in an `Enc<T>` container (line 15). The closure `|i| { *i += 1; }` (line 14) increments each byte of the data by 1, which essentially is the classical *Caesar cipher* [12]. User code has access to a *safe* API function (`get(...)` in lines 25 to 28), which internally uses an `unsafe` function to return a `Sec<T>` secure container that holds the decrypted information.

6.4 Example

Listing 11 demonstrates the application of our proposed security system. The trusted base instantiates a secure container `Sec<u32>` (line 3) and an encrypted container `End<u32>` (line 4) Following this, it calls the user code function `user1` and passes on references to the containers (line 5). The user code is free to delegate the references but has never access to the actual content of the containers. Note also, the authorization ticket `a` is temporal with a lifetime limited to the sequential execution context of `user1`, even when delegated to `user4`.

[7] While we can indeed allow the user to read encrypted data, we do not want the user to create or manipulate encrypted data outside the control of the trusted base.

```
// inside trusted base
fn main() {
    let d = unsafe { Sec::new(10u32) };
    let e = unsafe { Enc::new(&32u32) };
    user1(&d, &e);
}

// user code in `safe` Rust
fn user1(d: &Sec<u32>, e: &Enc<u32>) {
    let a = auth("abc").unwrap();
    user2(&sec_add_u32(d, &e.get(a)));
    user3(d, &e.get(a));
    user4(d, e, a);
    user5(d, e);
}

fn user2(d: &Sec<u32>) {...}

fn user3(d1: &Sec<u32>, d2: &Sec<u32>) {
    let d = sec_add_u32(d1, d2));
    ...
}

fn user4(d: &Sec<u32>, e: &Enc<u32>,
  a: &Auth) {
    let d = sec_add_u32(d, &e.get(a)));
    ...
}

fn user5(d: &Sec<u32>, e: &Enc<u32>) {...}
```

Listing 11: File `example.rs`.

```
    ...
    impl<T> !Send for Sec<T> {}
    ...
```

Listing 12: File `trusted_base.rs`.

6.5 Discussion

Our intention here is not to provide a fully fledged security framework but rather to demonstrate that security by construction is indeed feasible with our approach. The reader may notice that unwrapping encrypted information stores the decrypted data in plain form. This is perfectly secure from the perspective of the embedded software as the plain data is still wrapped in a secure container `Sec<T>` and thus not directly exposed to the user code. However, side channel attacks may exploit plain (decrypted) data that is stored in persistent memory.

With the proposed design, we allow persistent storage of decrypted data beyond the lifetime of the authorization ticket. I.e., a CRC component may store a `Sec<T>` container in its state when using a delegated authentication ticket to unwrap data from an encrypted container. If we want to ensure that this cannot happen, we need to apply only a small change to the trusted base (see Listing 12).

By default, Rust structs are `Send`, but we may override the default implementation and explicitly declare `Sec<T>` **not** to be `Send`. CRO states require `Send` and consequently the Rust compiler rejects all attempts to store a `Sec<T>` (e.g., `e.get(a)`) at compile time. The same applies to asynchronous messages, and thus when using this approach, decrypted data cannot live longer than the authorization ticket.

Looking further at Listing 8, we find that the `sec_add_u32` function operates on the `Sec<T>` type and requires encrypted data to be decrypted before passing on. With trait objects in Rust, we could implement `sec_add_u32` for any type that allows access to `T` with an additional `Auth` parameter. The advantage is that the decryption only takes place at the instant of the function execution and limits the exposure to side channel attacks. However, a drawback is the increased complexity and the impeded ability for the compiler to generate zero-cost abstractions, because the Rust compiler introduces dynamic dispatch only for trait objects.

Another possible extension is to associate each CRC component with an authorization level. This allows us to statically differentiate between partitions of the system at design time and give a base authorization that can be temporarily raised. Moreover, we may associate each `Sec<T>`/`Enc<T>` with an `Auth` level providing precise control over the data access. Note, secure software implementations do not require any of these extensions, they just provide additional means to manage granularity.

In the setting of mixed critical systems, our framework allows design time analysis of security aspects. The topology of the system statically defines the delegation of authorization, and thus our framework effectively mitigates the need for run-time monitoring of security breaches. In effect, we can fearlessly introduce untrusted code for low critical subsystems with jeopardizing neither system safety nor security.

6.6 Comparison to C/C++

We exploit the borrow semantics, the lifetime semantics, and the possibility to prohibit the execution of unsafe user code in Rust programs to establish a statically verifiable security architecture. Following our approach, the Rust compiler ensures in a system built on such a proposed trusted base that no stealing (borrow semantics) or faking (no unsafe user code) of authorization can occur as well as an authorization has a guaranteed temporal validity with well-defined life span (lifetime semantics).

When it comes to the system level languages C/C++, there is no concept like borrow semantics. Memory can freely be aliased because the compiler is not rejecting multiple references to the same memory location. In effect, it is impossible for the compiler to statically deduce a lifetime for a memory location and thus eventually drop the reference and free the memory. In contrast to Rust, where we utilize the lifetime semantics for a guaranteed temporal validity of authorization tokens, this is not possible in plain C/C++. On the other hand, the C++ Standard Library includes *smart pointers* with the special pointer type `unique_ptr`. It essentially provides the same functionality as the Rust ownership model and supports the RAII (Resource Acquisition Is Initialization, [13]) programming principle. Such pointers indicate unique ownership of the memory they reference to and the memory is automatically freed when the pointer goes out of scope. However, the big difference to Rust is the validation point. While Rust incorporates the ownership model into the language, it can be statically verified during compilation. On violation, smart pointers in C++ cause a run-time error.

C/C++ does not support the segmentation of code into safe and unsafe partitions. We utilize this functionality of Rust to assure at compile time that no user code is able to generate, copy, or store authorization tokens. In C/C++, the same assurance can only be achieved by either applying static code analysis (e.g., formal methods) or verifying the authenticity of all authorization tokens by the trusted base on each usage during run-time. But the run-time token verification generates computational overhead and requires to carry additional information along with an authorization token, e.g., a private key signature of the token from the trusted base.

7 Memory safety of the Rust CRC framework

The pillar of the Rust memory model is **avoiding mutable aliasing** (referred to as the *invariant* in the following). As we provide a safe API, user code does not contain any `unsafe` fragments, and hence the `rustc` compiler grants memory safety. CRO connections, which the build script generates with `unsafe` code, are outside the knowledge of the compiler. Consequently, we have to ensure that the `unsafe` fragments preserve the invariant.

7.1 Synchronous messages

Each method receives a `&mut self`, a mutable reference to its state. Any synchronous message chain, for which an object o appears more than once, generates a mutable alias to the state of o and hence the build script has to reject it at compile time.

7.2 Asynchronous messages

The current implementation statically allocates a single element buffer for each asynchronous connection per CRO instance. A static mutable variable, which is hidden from the user, passes the message payload by value. A data race may occur if the sender (writer) preempts the dispatcher (reader). We handle this case by *panicking* the sender. An alternative option is to use an SRP resource for the buffer that ensures race free access[8].

7.3 Peripheral access

Let us assume a system with two objects A and B, which have access to the same peripheral P and a connection between output port op of object A and input port ip of object B. Let us further assume the method associated to the input port ip of object B claims the peripheral P. If in this system a method of object A claims the peripheral P and sends a synchronous message within this claim block through the output port op to the input port ip, we end up aliasing the reference to the register block of P. This is, however, **not** a problem because a `claim` returns an immutable reference (`&T`) to the register block, which upholds the invariant.

7.4 Leaking of references

Passing data by reference in Rust is memory safe by construction. The borrow checker, one of the `rustc` compiler passes, is in charge of rejecting the use of invalid references at compile time It does this by tracking the *lifetime* of each memory location. In Rust, lifetime refers to the lexical scope for which access to a memory location is valid. The special lifetime identifier `'static` indicates in Rust that the memory location is valid for the entire program.

In our CRC framework, it is possible to pass data by reference in a synchronous message but not in an asynchronous message. The compiler can trace the lifetime of data across synchronous messages because they run in the same execution context. On the other hand, asynchronous messages run in different execution contexts. Semantically, a reference passed in an asynchronous message has to be valid for the span of both execution contexts. This cannot be verified at compile time and thus the compiler rejects it.

7.4.1 Leaking of peripherals

Peripherals provide a `claim` interface, which grants access to the peripheral register block only within the closure passed to it. The borrow checker does not allow references to escape from the closure.

7.4.2 Leaking through static variables

The Rust compiler prohibits to pass references between objects outside the message passing mechanism of the CRC framework. Such an operation requires to store the reference in a global (i.e., `static mut`) variable and load it from there. The compiler rejects this because static variables con only store *values* with `'static` lifetime and *references to values* with `'static` lifetime. E.g., the compiler rejects to store a reference to a stack allocated variable in a static variable. Apart from the lifetime problem, it is also `unsafe` to read, write, or modify `static mut` variables because the accesses to them are not synchronized. In conclusion, our CRC framework upholds the Rust memory invariants if we reject systems with synchronous message cycles (see Section 7.1) and ensure race-free execution with SRP [9].

8 Demonstration and performance analysis

For the design and measurements in this section, we used a Cortex-M3 microcontroller on a *Blue Pill* development board [14] running at 8 MHz and with zero memory wait states. Figure 2 depicts the example system implementation utilizing our proposed framework and Figure 3 illustrates the toolchain of our CRC framework.

[8] However, this does not prevent a message payload to be overwritten before it has been dispatched. Therefore, further system wide timing analysis and potentially larger buffers are required.

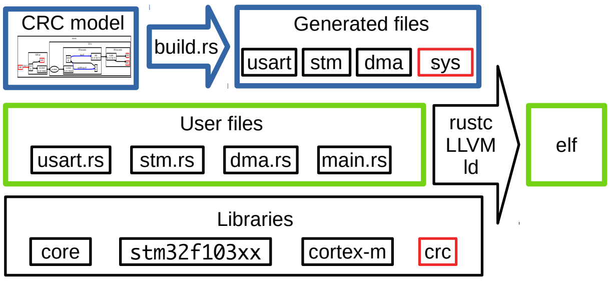


Figure 3: The `build.rs` build script is at heart of our toolchain. It analyzes the CRC/CRO model and generates the port binding proxies and system configuration, i.e., the statically allocated state and message memory. The user files that implement the CRO application logic contain no `unsafe` code. Among dependencies, the `crc` library contains the hardware related resource protection and scheduling primitives. `rustc` and `LLVM` compile all files and libraries for the `binutils-ld` linker to build a monolithic `elf` binary.

```
pub unsafe fn claim<R, F>(
    nvic_prio_bits: u8,
    ceiling: u8, f: F) -> R
where
    F: FnOnce() -> R,
{
    let max_priority = 1 << nvic_prio_bits;
    let old = basepri::read();
    let hw  = (max_priority - ceiling)
                    << (8 - nvic_prio_bits);
    basepri_max::write(hw); // sys ceiling
    let r = f();
    basepri::write(old);
    r
}
```

Listing 13: Resource protection with `claim`.

8.1 Characterization of overhead

In order to characterize the overhead, we performed a set of clock cycle accurate measurements with a 100% repeatability between runs. For all measurements, we compiled the code in `-release` mode.

The `claim` interface, as depicted in Listing 13, has an overhead of 4 clock cycles (call to return). We also observed this overhead when invoking object methods because the system applies the same `claim` mechanism to prevent data races on the object state. Our implementation of `claim` closely follows [3] and enforces compiler barriers around the critical section.

Synchronous messages are plain function calls, which allow to inline the code. In many cases, `rustc` opts to inline and eliminate the overhead of a function call. It also enables further optimization because it gives the compiler more local information about the behavior of the program.

Enqueuing an asynchronous message takes 20 clock cycles plus the time required to copy the message payload from the stack into a statically allocated buffer.

Dispatching asynchronous messages has a per message overhead of 26 clock cycles plus the time required to copy the message payload from a statically allocated buffer back into the stack.

The interrupt latency (11 clock cycles) plus the proxy overhead claiming the target object and entering the user code (3 clock cycles) determines the external event latency. It amounts on an 8 MHz MCU to $1.75us$.

The model offers a plethora of methods for response time analysis, taking into consideration preemption and blocking [7] as well as offsets [8]. Scheduling and resource protection overhead is $O(1)$, i.e., free of run-time dependencies. Hence, further scheduling analysis can utilize the characterizations as direct input.

8.1.1 Example system

We designed our example system with reactivity in mind. The environment and the application at hand set the timing constraints, which we specified in cycles as depicted in Figure 2.

The USART operates at $115.2kbps$, which is roughly $87us$ or 696 cycles to serve an arriving byte. For simplicity, we assume a single buffer.

The LED array consists of 24 daisy chained WS2812B-LEDs. In order to update each LED with a unique RGB value, the DMA peripheral sends a non-return-to-zero bit stream and latches the output on the end of the frame by holding the data line low for at least $50us$. The DMA operates at $400kHz$, which results in a transfer time of $1.5ms$. This is on the safe side at half of the maximum specified operation rate.

The design ensures that blocking will not be an issue, because the state of the STM is completely decoupled from the state of the DMA[9]. Alternatively, we could use an asynchronous message between the USART and the STM in the LED component to achieve the same effect of decoupling. When we see our LED application as a freestanding and re-usable component, there is no restriction on how to implement it, both synchronous and asynchronous calls work equally well.

Looking at the STM CRO, we set the interarrival time of `transition` events in the system to $10ms$, i.e., a frequency of $100Hz$. The number of preemptions during a $10ms$ period is roughly 115. I.e., the `transition` suffers $115 * C(receive)$ in the worst case. We measured a worst case execution time of 299 cycles for *receive*, amounting to a total of $4.3ms$.

The response time for a task is $r = C + P + B$, where C is the execution time, P is the preemption time (interference), and B is the blocking time. For the `transition`, we derive $r = 0.098ms + 4.3ms + 0$, which is a worst case estimation well under the required $10ms$ or 80000 cycles[10]. For this presentation, we conclude the response times of `receive` and `on_command` to be clearly within their timing requirements and skip the precise analysis.

We measured a CPU utilization of 13.25% at the maximum animation speed (100 frames per second).

[9] The $1.5ms$ transfer period blocks the DMA, but there is only $87us$ in between two USART events. Hence, a synchronous (blocking) approach is not sufficient.

[10] Computing the actual busy period of `transition` and taking the USART parsing logic into account allows to derive a less pessimistic estimation. Not all character inputs yield the worst case behavior.

```
text    data     bss     dec     hex   filename
3974     196     620    4790    12b6   crc-test
```

Listing 14: `arm-none-eabi-size`

8.2 Memory usage

The system compiled in `-release` mode shows a $4kB$ Flash memory footprint and less than $1kB$ of RAM usage. Listing 14 displays the actual sizes.

The DMA buffer requires 601 bytes to store the non-return-to-zero bit encoding including a postamble of 25 zeros to latch the data to the WS2812B LED array. In the STM CRO we store the RGB values of each individual LED ($24 * 3 = 72$ bytes) and send it with an async message buffer. In total, this amounts to 745 bytes. The remaining allocated RAM memory of 67 bytes holds additional CRO states (USART, STM, DMA) and message structure overhead.

We conclude the abstraction to be memory efficient and zero-cost in comparison to a handwritten implementation.

9 Conclusions and future work

In this paper, we present a Rust based component model for concurrent programming along with a framework for analysis and code generation that produces efficient, memory safe, race- and deadlock-free executables for single-core Stack Resource Policy (SRP) based scheduling. As the main contribution, we show that the CRC model allows a secure by construction design of embedded software, covering authentication for operations as well as abstractions for safe and secure data containers. For the underlying CRC framework, we discuss soundness in regard to the Rust memory model and SRP invariants.

Other contributions include key design decisions for the ecosystem under development, a feasibility demonstration on an ARM Cortex-M3 target, and the characterization of run-time overhead for resource protection and scheduling primitives.

For the prototype, we manually carried out the timing analysis and timer queue generation. Current and future work includes the analysis of arbitrary timing offsets to determine safe (yet tight) bounds for the number of outstanding asynchronous messages and the synthesis of queuing and timer primitives.

Based on recent advances of the RustBelt formal model [15], we project a formalization and mechanized proof of correctness.

References

[1] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black (2002), *Reactive objects*, in Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002, pp. 155–158.

[2] The Timber Language, webpage. http://www.timber-lang.org, last accessed 2017-09-16.

[3] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren (2013), *Real-time for the masses, step 1: Programming api and static priority srp kernel primitives*, in Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on, pp. 110–113, IEEE.

[4] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis (2017), *The case for writing a kernel in rust*, in Proceedings of the 8th Asia-Pacific Workshop on Systems, APSys '17, (New York, NY, USA), pp. 1:1–1:7, ACM.

[5] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis (2013), *Gpu programming in rust: Implementing high-level abstractions in a systems-level language*, in 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, pp. 315–324.

[6] The Rust Programming Language book, webpage. https://doc.rust-lang.org/book/, last accessed 2018-02-05.

[7] T. P. Baker (1991), *Stack-based scheduling for realtime processes*, Real-Time Syst., vol. 3, pp. 67–99.

[8] J. Mäki-turja and M. Nolin (2004), *Tighter response-times for tasks with offsets*, in Proc. of the 10 th International conference on Real-Time Computing Systems and Applications (RTCSA'04).

[9] P. Lindgren, M. Lindner, E. Fresk, D. Pereira, and L. Pinho (2014), *RTFM-core: Language and Implementation*. Embedded Systems Week, New Delhi, India.

[10] svd2rust, webpage. https://github.com/japaric/svd2rust, last accessed 2017-09-16.

[11] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady (2004), *Security in embedded systems: Design challenges*, ACM Trans. Embed. Comput. Syst., vol. 3, pp. 461–491.

[12] D. Kahn (1976), *The Codebreaker: The Story of Secret Writing*, Macmillam.

[13] S. Meyers (2005), *Effective C++: 55 specific ways to improve your programs and designs*, Pearson Education.

[14] Blue Pill compact STM32F103 board, webpage. https://wiki.stm32duino.com/index.php?title=Blue_Pill, last accessed 2018-09-07.

[15] RustBelt: Logical Foundations for the Future of Safe Systems Programming, webpage. http://plv.mpi-sws.org/rustbelt, last accessed 2017-09-16.

Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems (Part 1) [1]

Alan Burns

University of York, UK; email: alan.burns@york.ac.uk

Brian Dobbing

Altran Praxis, UK [+]

Tullio Vardanega

University of Padua, Italy; email: tullio.vardanega@unipd.it

1 Introduction

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable failure of the whole system. The choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

Traditional methods used for the design and development of complex applications, which concentrate primarily on functionality, are increasingly inadequate for hard real-time systems. This is because non-functional requirements such as dependability (e.g. safety and reliability), timeliness, memory usage and dynamic change management are left until too late in the development cycle.

The traditional approach to formal verification and certification of critical real-time systems has been to dispense entirely with separate processes, each with their own independent thread of control, and to use a *cyclic executive* that calls a series of procedures in a fully deterministic manner. Such a system becomes easy to analyse, but is difficult to design for systems of more than moderate complexity, inflexible to change, and not well suited to applications where sporadic activity may occur and where error recovery is important. Moreover, it can lead to poor software engineering if small procedures have to be artificially constructed to fit the cyclic schedule.

The use of Ada has proven to be of great value within high integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. Such subsets have been defined for Ada 83, but these have excluded tasking on the grounds of its non-determinism and inefficiency. Advances in the area of schedulability analysis currently allow hard deadlines to be checked, even in the presence of a run-time system that enforces preemptive task scheduling based on multiple priorities. This valuable research work has been mapped onto a number of new Ada constructs and rules that have been incorporated into the Real-Time Annex of the Ada language standard [RM D]. This has opened the way for these tasking constructs to be used in high integrity subsets whilst retaining the core elements of predictability and reliability.

The Ravenscar Profile is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements.

It is important to note that the Ravenscar Profile is silent on the non-tasking (i.e. sequential) aspects of the language. For example it does not dictate how exceptions should, or should not, be used. For any particular application, it is likely that constraints on the sequential part of the language

[1] **Editor note:** This paper includes chapters 1 to 5 of the report of the University of York, UK (University of York Technical Report YCS-2017-348, June 2017), which updates the original "Guide for the use of the Ada Ravenscar Profile in high integrity systems", published in 2003, considering the changes in the definition of Ada (Ada 2012). Chapters 1 to 5 present the Ravenscar profile, the rationale for the decisions taken and examples of usage. Chapters 6, which discusses the verification approaches appropriate to Ravenscar programs, and Chapter 7, which provides and extensive example, will be published in the next issue of the Journal. This updated Guide will also be published as an official ISO Technical Report (TR), with some changes. The paper in the Ada User Journal follows the York version, noting, where applicable, the changes in the ISO TR.

[+] now retired

will be required. These may be due to other forms of *static analysis* to be applied to the code, or to enable *worst-case execution time* information to be derived for the sequential code. The reader is referred to the ISO Technical Report, Guide for the Use of Ada Programming Language in High Integrity Systems [GA] for a detailed discussion on all aspects of static analysis of sequential Ada.

The Ravenscar Profile has been designed such that the restricted form of tasking that it defines can be used even for software that needs to be verified to the very highest integrity levels. The Ravenscar Profile has already been included in the ISO technical report [GA] referenced above. The aim of this guide is to give a complete description of the motivations behind the Ravenscar Profile, to show how conformant programs can be analysed and to give examples of usage.

Structure of the Guide

The report is organized as follows. The motivation for the development of the Ravenscar Profile is given in the next chapter. Chapter 3 includes the definition of the profile as specified by the Ada Standard; the definition is included here for convenience, but this report is not the definitive statement of the profile. In Chapter 4, the rationale for each aspect of the profile is described. Examples of usage are then provided in Chapter 5. The need for verification is an important design goal for the Ravenscar Profile: Chapter 6 reviews the verification approach appropriate to Ravenscar programs. Finally, in Chapter 7 an extended example is given. Definitions and references are included at the end of the report.

Readership

This report is aimed at a broad audience, including application programmers, implementers of run-time systems, those responsible for defining company/project guidelines, and academics. Familiarity with the Ada language is assumed.

Conventions

This report uses the *italics* face to flag the first occurrence of terms that have a defining entry in Chapter 8. For all Ada-related terms, the report follows the language reference manual [RM] style: it uses the Arial font where there is a reference to defined syntax entities (e.g. delay_relative_statement). For all other names (e.g. Ada.Calendar) it uses normal text font, as do language keywords in the text except that they are in **bold** face.

2 Motivation for the Ravenscar Profile

Before describing the Ravenscar Profile in detail, in this chapter we explain some of the reasoning behind its features. These primarily come from the need to be able to verify concurrent real-time programs, and to have these programs implemented reliably and efficiently.

In this chapter, we look mainly at scheduling theory, as this is the main driver for the definition of the restrictions of the Ravenscar Profile. In addition, there is a section that summarizes other program verification techniques that can be used with the profile.

2.1 Scheduling Theory

Recent research in scheduling theory has found that accurate analysis of real-time behaviour is possible given a careful choice of scheduling/dispatching method together with suitable restrictions on the interactions allowed between tasks. An example of a scheduling method is *preemptive fixed priority scheduling*. Example analysis schemes are *Rate Monotonic Analysis* (*RMA*) [1] and *Response Time Analysis* (*RTA*) [2].

Priority-based preemptive scheduling is normally used with a *Priority Ceiling Protocol* (*PCP*) to avoid unbounded priority inversion and deadlocks. It provides a model suitable for the analysis of concurrent real-time systems. The approach supports *cyclic* and *sporadic* activities, the idea of *hard*, *soft*, *firm*, and *non-critical* components, and controlled inter-process communication and synchronization. It is also scalable to programs for distributed systems.

Tool support exists for RMA and RTA, and for the static simulation of concurrent real-time programs. The primary aim of analysing the real-time behaviour of a system is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints. Whether the timing constraints are appropriate for meeting the requirements of the application is not an issue for scheduling analysis. Such verification requires a more formal model of the program and the application of techniques such as model checking – see Section 2.4.

2.1.1 Tasks Characteristics

The various tasks in an application will each have timing constraints. For *critical tasks*, these are normally defined in terms of *deadlines*. The deadline is the maximum time within which a task must complete its operation in response to an event.

Each task is classified into one of the following four basic levels of criticality according to the importance of meeting its deadline:

- Hard
  A *hard deadline task* is one that must meet its deadline. The failure of such a task to meet its deadline may result in an unacceptable failure at the system level.
- Firm
  A *firm deadline task* is one that must meet its deadline under "average" or "normal" conditions. An occasional missed deadline can be tolerated without causing system failure (but may result in degraded system performance). There is no value, and thus there is a system-level degradation of service, in completing a firm task after its deadline.
- Soft
  A *soft deadline task* is also one that must meet its deadline under "average" or "normal" conditions. An

occasional missed deadline can be tolerated without causing system failure (but may result in degraded system performance). There is value in completing a soft task even if it has missed its deadline.

- Non-critical
  A *non-critical task* has no strict deadline. Such a task is typically a background task that performs activities such as system logging. Failure of a non-critical task does not endanger the performance of the system.

2.1.2 Scheduling Model

At any moment in time, some tasks may be *ready* to run (meaning that they are able to execute instructions if processor time is made available). Others are *suspended* (meaning that they cannot execute until some event occurs) or *blocked* (meaning that they await access to a shared resource that is currently exclusively owned by another task). Suspended tasks may become ready synchronously (as a result of an action taken by a currently running task) or asynchronously (as a result of an external event, such as an interrupt or timeout, that is not directly stimulated by the current task).

With priority-based preemptive scheduling on a mono-processor, a priority is assigned to each task and the scheduler ensures that the highest priority ready task is always executing. If a task with a priority higher than the currently running task becomes ready, the scheduler performs a *context switch*, as soon as it can, to enable the higher-priority task to resume execution. The term "preemptive" indicates that this can occur because of an asynchronous event (i.e. one that is not caused by the running task).

Tasks will normally be required to interact as a result of contention for shared resources, exchange of data, and the need to synchronize their activities. Uncontrolled use of such interactions can lead to a number of problems:

- Unbounded *Priority Inversion* / Blocking
  where a high-priority task is *blocked* awaiting a resource in use by a low-priority task; as a result, ready tasks of intermediate priority may hold up the high priority task for an unbounded amount of time since they will run in preference to the low priority task that has locked the resource.

- *Deadlock*
  where a group of tasks (possibly the whole system) block each other permanently due to circularities in the ownership of and the contention for shared resources.

- *Livelock*
  where several tasks (possibly the whole system) remain ready to run, and do indeed execute, but fail to make progress due to circular data dependencies between the tasks that can never be broken.

- Missed *Deadline*
  where a task fails to complete its response before its deadline has expired due to factors such as system overload, excessive preemption, excessive blocking, deadlocks, livelocks or CPU overrun.

The restricted scheduling model that is defined by the Ravenscar Profile is designed to minimize the upper bound on blocking time, to prevent deadlocks, and (via tool support) to verify that there is sufficient processing power available to ensure that all critical tasks meet their deadlines.

In this model, tasks do not interact directly, but instead interact via shared resources known as *protected objects* [2]. Each protected object typically provides either a resource access control function (including a repository for the private data to manage and implement the resource), or a synchronization function, or a combination of both.

A protected object that is used for resource access control requires a mutual exclusion facility, commonly known as a *monitor* or *critical region*, where at most one task at a time can have access to the object. During the period that a task has access to the object, it must not perform any operation that could result in it becoming suspended. Ada directly supports protected objects and disallows internal suspension within these objects.

A protected object that is used for synchronization provides a signalling facility, whereby tasks can signal and/or wait on events. In the Ravenscar Profile definition, the use of protected objects for synchronization by the critical tasks is constrained so that at most one task can wait on each protected object. A simplified version of wait/signal is also provided in the Ravenscar Profile via the Ada Real-Time Annex functionality known as *suspension objects* [RM D.10]. These can be used in preference to the protected object approach for simple resumption of a suspended task, whereas the protected object approach should be used when more complex resumption semantics are required, for example including deterministic (*race-condition*-free) exchange of data between signaller and waiter tasks.

The Ravenscar Profile definition assures absence of deadlocks by requiring use of an appropriate locking policy. This policy requires a *ceiling priority* to be assigned to each protected object that is no lower than the highest priority of all its calling tasks, and results in the raising of the priority of the task that is using the protected object to this ceiling priority value. In addition to absence of deadlocks, this policy also allows an almost optimal time bound on the worst case blocking time to be computed for use within the schedulability analysis, thereby eliminating the unbounded priority inversion problem. This time bound is calculated as the maximum time that the object is in use by lower-priority tasks. Therefore, the smaller the worst-case time bound for this blocking period, the greater the likelihood that the task set will be schedulable.

[2] Editor Note: the ISO technical report adds the use of the atomic aspect to support object sharing. However, it also notes the need for static assurance of safe use of atomic objects and the use of protected objects as the preferable abstraction for shared date, as they are inherently safe.

The use of priority-based preemptive dispatching defines a mechanism for scheduling. The scheduling policy is defined by the mapping of tasks to priority values. Many different schemes exist for different temporal characteristics of the tasks and other factors such as criticality. What most of these schemes require is an adequate range of distinct priority values. Ada and the Ravenscar Profile ensure this.

2.2 Mapping Ada to the Scheduling Model

The analysis of an Ada application that makes unrestricted use of Ada run-time features including tasking rendezvous, select statements and abort is not currently feasible. In addition, the non-deterministic and potentially unbounded behaviour of many tasking and other run-time calls may make it impossible to provide the upper bounds on execution time that are required for schedulability analysis and simulation. Thus, Ada coding style rules and subset restrictions must be followed to ensure that all code within critical tasks is statically time-bounded, and that the execution of the tasks can be defined in terms of response times, deadlines, cycle times, and blocking times due to contention for shared resources.

The application must be decomposed into a number of separate tasks, each with a single thread of control, with all interaction between these tasks identified. Each task has a single primary invocation event. The tasks are categorized as *time-triggered* (meaning that they execute in response to a time event), or *event-triggered* (meaning that they execute in response to a stimulus or event external to the task). If a time-triggered task receives a regular invocation time event with a statically-assigned rate, the task is termed *periodic* or *cyclic*.

Protected objects must be introduced to provide mutually-exclusive access to shared resources (e.g. for concurrent access to writable global data) and to implement task synchronization (e.g. via some event signalling mechanism). This decomposition is normally the result of applying a design methodology suitable to describe real-time systems.

In order to be suitable for schedulability analysis, the task set to be analysed must be static in composition and have all its dependencies between tasks via protected objects. Tasks nested inside other Ada structures incur unwanted visibility dependencies and termination dependencies. Therefore, this model only permits tasks to be created at the *library level*, at system initialization time.

Hence, in the Ravenscar Profile, all tasks in the program are created at the library level.

Another consequence of requiring a static task set for schedulability analysis purposes is that the Ravenscar Profile must prohibit the dynamic creation of tasks and protected objects via *allocators*. This implies that the memory requirements for the execution of the task set (e.g. the task stacks) are resolved prior to, or during, elaboration of the program. In addition, the Ravenscar Profile prohibits the implementation from implicitly acquiring dynamic memory from the standard storage pool [RM 13.11(17)]. The data structures that are required by the run-time system should either be declared globally, so that the memory requirements can be determined at link time, or in such a way as to cause the storage to be allocated on the stack (of the *environment task*) during elaboration of the run-time system.

The Ravenscar Profile places no restrictions on the declaration of large or dynamic-sized Ada objects in the application other than prohibiting the implementation from implicitly using the standard storage pool to acquire the storage for these objects. It is acceptable for the memory for such objects to be allocated on the task stack.

2.3 Non-Preemptive Scheduling and Ravenscar

The definition of the Ravenscar Profile requires preemptive scheduling of tasks. However, a similar profile could be defined that specified non-preemptive execution. Much of the material and guidelines contained in this report would also apply to the non-preemptive case. Non-preemptive implementation for a mono-processor is in between the cyclic executive approach and the preemptive tasking approach with regard to ease of timing analysis, flexibility with regard to change, and responsiveness to asynchronous events. In common with the cyclic executive approach, there is no contention for shared resources, and there is no need to analyse the impact from asynchronous events. There is still, however, the need to break up long code sequences using voluntary suspension points (e.g. a delay_until_statement with a wakeup time argument that denotes a time in the past) to obtain reasonable responsiveness to asynchronous events.

2.4 Other Program Verification Techniques

In addition to the provision of support for schedulability analysis, the rationale behind the Ravenscar Profile definition is also to support other static program verification techniques, and to simplify the formal certification process. These other techniques are discussed briefly in this section.

2.4.1 Static Analysis

Static analysis is recognized as a valuable mechanism for verifying software. For example, it is mandated for safety critical applications that are certified to the UK Defence Standard 00-55 [DS]. Industrial experience shows that the use of static analysis during development eliminates classes of errors that can be hard to find during testing. Moreover, these errors can be eliminated by the developer before the code has been compiled or entered into the configuration management system, saving the cost of repeated code review and testing which results from faults that are discovered during *testing*.

Static analysis as a technology has a fundamental advantage over dynamic testing. If a program property is shown to hold using static analysis, then the property is guaranteed for all scenarios. Testing, on the other hand, may demonstrate the presence of an error, but the correct execution of a test only indicates that the program behaves correctly for the specific set of inputs provided by the test, and within the specific context that the test harness sets up. For all but the simplest systems, exhaustive testing of all possible combinations of input values and program contexts is infeasible. Typically, test cases are devised to represent broad classes of inputs, so that tests can be created that use a representative value from each possible input class. However, complex program state contexts are usually only creatable during integration and system testing, when it may be very difficult to simulate all possible operational states. Further, the impact of correcting errors that are found only at this stage of the lifecycle is generally large in comparison to errors found during development.

There are many methods of static analysis. By using combinations of these methods, a variety of properties can be guaranteed for a program. The following list of forms of analysis is drawn from a study of a variety of standards that is presented in the ISO Technical Report [GA]. Section 6.2 discusses how these analyses may be applied in the context of a concurrent Ravenscar Profile program.

Control Flow

Control flow analysis ensures that code is well structured, and does not contain any syntactically or semantically unreachable code.

Data Flow

Data flow analysis ensures that there is no executable path through the program that would result in access to a variable that does not have a defined value. Data flow analysis is only feasible on code that has valid control flow properties.

Information Flow

Information flow analysis is concerned with the dependencies between inputs and outputs within the code. It checks the specified dependencies against the implemented dependencies to ensure consistency. To be effective, information flow analysis needs to be performed with knowledge of the system requirements. It can be a powerful tool for demonstrating properties such as non-interference between critical and non-critical data.

Symbolic Execution

Symbolic execution generates a model of the function of the software in terms of parallel assignments of expressions to outputs for each possible path through the code. This can be used to verify the code without the need for a formal specification.

Formal Code Verification

Formal code verification is the process of proving the code is correct against a formal specification of its requirements. Each operation is specified in terms of the pre-conditions that need to be satisfied for the operation to be callable, and the post-conditions that hold following a successful call to the operation. The verification process demonstrates that, given the pre-conditions, execution of the operation always gives rise to the post-conditions. The level of proof depends on the information provided in the formal specification. This can vary depending on the aspects of the code that need to be verified; this can vary from the proof of a single invariant right up to full functional behaviour.

Proof of absence of run-time errors is a special form of formal code verification. This does not require the provision of a formal specification of the program. Instead, formal code verification techniques are used to demonstrate that, at every point in the code where a run-time error may occur, the pre-conditions on execution of that code and the current set of data values in the expression guarantee that the run-time error cannot occur. This is a very valuable property to be able to demonstrate, especially in systems where the occurrence of an unexpected run-time exception is generally unrecoverable, and the overhead of dynamic defensive mechanisms for preventing all such faults is unacceptable.

2.4.2 Formal Analysis

The formal analysis of concurrent programs has been a fruitful research topic for a number of years. Current standard techniques allow many important properties of programs to be statically checked.

Concurrent programs, whilst more expressive than their sequential counterparts, have a number of distinct error conditions that must be addressed during program development. The most common of these is deadlock, where all processes are blocked on a synchronization primitive with no processes left to undertake the necessary unblocking actions. In general, a concurrent program should possess two important properties:

1. *Safety* - the system of tasks should not get into an unsafe (undesirable) state (for example; deadlock, livelock).
2. *Liveness* - all desirable states of the task must be reached eventually (that is, useful progress should always be made).

In a real-time concurrent system, ‘liveness’ becomes ‘bounded liveness’ as desirable states must be reached by known deadlines.

Ada, like all other engineering languages, does not have its semantics defined in a formal mathematical way. Hence, it is necessary to link a model of the program with the program itself. This link cannot be formal but can be precise. The use of standard patterns for Ada tasks helps this linkage. The formal model could be derived from the code or, more likely in an engineering process, the model is derived from requirements, and the code is obtained via a series of refinements from the model.

There are two general forms for these models and two methods of extracting properties (behaviours) from these descriptions. First, an algebraic form could be used in one of the concurrency languages that does have formally defined semantics; examples being *CSP* (Communicating Sequential Processes) and *CCS* (Calculus of Communicating Systems). The other, more common, approach is to view the program as a collection of state-transition systems.

Verification comes either from a proof theoretic approach or via model checking. An algebraic description can be proved to be deadlock-free, for example, by the use of a theorem prover. Alternatively, a state-transition description (or an algebraic one) can be exercised by an exhaustive search of the set of states the program can enter. This 'checking of the model' can deduce that all safe states, and no unsafe states, can be reached.

The disadvantage of model checking is that an explosion of states can make it impossible to terminate the search. However, there have been considerable (and continuing) advances in the tools for model checking, and now sizeable systems can be verified in a respectably small number of hours of processing time. Theorem proving does not have this problem but it is a more skilled activity and theorem proving tools are not simple to use (i.e. the verification process is not automatic). A proof theoretic approach also has the advantage that it can show that a property is true 'for any number of tasks'; whereas model checking cannot generalize in this way – it will show it is true for six client tasks, say, but for seven the check must be made again. Combinations of proof and model checking are possible and are the subject of current research.

For real-time systems, it is possible to add time to the concurrency model and to then validate temporal aspects of program. Timed versions of formalisms such as CSP [CSP] exist and state-transition systems with clocks allow timing requirements to be expressed and subsequently verified by model checking. A common formalism for this type of state-transition system is called timed automata. Again, tool support for model checking sets of timed automata is well advanced. One of the very useful features of model checking tools is that they all produce a well-defined counter example for any failed check.

2.4.3 Formal Certification

In order to achieve formal certification of a software architecture and of its Ada implementation, it is necessary to provide verification evidence of safety and reliability of the Ada run-time system as well as for the application-specific components. The run-time system that is needed to implement the dynamic semantics of the full Ada concurrency model is complex, and the number of states that may be represented by its dynamic data structures is large. As a result, it is very challenging for a commercial Ada vendor to produce certification evidence to the highest integrity levels for an entire Ada run-time system.

The Ravenscar Profile definition greatly reduces the size and complexity of the required run-time system, to simplify the process of providing evidence of its safety and reliability. Ada concurrency features that have major impact on the run-time system semantics, such as abort, asynchronous transfer of control, multiple entry queues each with a list of waiting tasks, requeue statements, task hierarchy and dependency, and *finalization* actions of local protected objects, are eliminated. As a result, it is possible to create not only a small and highly efficient run-time system implementation, but also one that is amenable to the forms of verification applicable to sequential code as described in [GA], which may then be used as evidence to support the formal certification of an entire software system to the highest integrity levels.

3 The Ravenscar Profile Definition

3.1 Development History

The 8th International Real-Time Ada Workshop (IRTAW) was held in April 1997 at the small Yorkshire village of Ravenscar. Two position papers [3][4] led to an extended discussion on tasking profiles. By the end of the workshop, the Ravenscar Profile had been defined [5] in a form that is almost identical to its current specification.

At the 9th IRTAW [6] (March 1999) the Ravenscar Profile was again discussed at length. The definition was reaffirmed and clarified. The most significant change was the incorporation of Suspension Objects. An Ada Letters paper [5] became the de facto defining statement of the Ravenscar Profile.

By the 10th IRTAW [7] (September 2000) many of the position papers were on aspects of the Ravenscar Profile and its use and implementation. No major changes were made, although an attempt to standardize on the Restriction identifiers was undertaken. Time was spent on a non-preemptive version of the profile. Following the 10th IRTAW, the participants decided to forward the Ravenscar Profile to the ARG – the ISO body in charge of the maintenance of the Ada language – so that its definition could move from a de facto to a real standard. The HRG – the ISO body in charge of the high integrity aspects of the Ada language – was also tasked with producing a Rationale for the Ravenscar Profile, which resulted in the production of this guide.

At the 11th IRTAW [8] (April 2002), the formal definition of the profile as formulated by the ARG was agreed. It was confirmed that the Ravenscar Profile requires task dispatching policy FIFO_Within_Priorities and locking policy Ceiling_Locking.

Since 2002, the Ravenscar Profile has been a formal part of the definition of Ada. Each time the language is upgraded, the profile is revisited to make sure that it continues to have the right set of restrictions. The series of IRTAW workshops continues to review the Ravenscar Profile's definition. This last took place at the 18th IRTAW, in April 2016.

3.2 Definition

The definition of the Ravenscar Profile is now included in the Ada Standard. The definition is reported here for information only. The latest version of Ada defining the Ravenscar Profile is Ada 2012; ARG agreed changes for the next version of Ada are incorporated into the definition given here.

An application requests the use of the Ravenscar Profile by means of the configuration pragma Profile with the Ravenscar identifier:

```
   pragma Profile(Ravenscar);
```

There are, in general, two distinct ways of defining the details of a profile: either by defining what is in it, or by declaring those parts of Ada that are not. The 'official' definition defines the restrictions that are needed to reduce the full tasking model to Ravenscar. However, this gives a rather negative definition. Therefore, we shall first introduce the profile by focusing on the features it does contain.

3.3 Ravenscar Features

Following from the discussion on verification in the previous chapter, we are able to define an adequate set of tasking features. The Ravenscar Profile allows programs to contain:

- Task types and objects, defined at the library level.
- Protected types and objects, defined at the library level, with a maximum of one entry per object and with a maximum of one task queued at any time on that entry. The entry barrier must be a single Boolean variable (or a Boolean literal).
- Atomic and Volatile aspects.
- delay_until_statements.
- Ceiling_Locking policy and FIFO_Within_Priorities dispatching policy.
- The E'Count attribute for protected entries except within entry barriers.
- The Ada.Task_Identification package plus task attributes T'Identity and E'Caller.
- Synchronous task control.
- Task type and protected type discriminants.
- The Ada.Real_Time package.
- Protected procedures as statically bound interrupt handlers.
- Static allocation of task to cores on a multicore (or multiprocessor) platform so that each core hosts a separate set of tasks, to which the Ravenscar Profile's scheduling and locking policies apply locally.

Together, these form a coherent set of features that define an adequate language for expressing the programming needs of statically defined real-time systems.

3.3 Summary of Implications of pragma Profile(Ravenscar)

The following restrictions apply to the alternative mode of operation defined by the Ravenscar Profile. Some restrictions require language features to be omitted, others can be achieved by simply requiring that certain defined (standard) library packages are not incorporated into the program that is conforming to the Ravenscar Profile (i.e. there is no semantic dependency on the specified package).

The Ravenscar Profile is defined as follows [RM D.13]:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions(
         No_Abort_Statements,
         No_Dynamic_Attachment,
         No_Dynamic_CPU_Assignment,
         No_Dynamic_Priorities,
         No_Implicit_Heap_Allocations,
         No_Local_Protected_Objects,
         No_Local_Timing_Events,
         No_Protected_Type_Allocators,
         No_Relative_Delay,
         No_Requeue_Statements,
         No_Select_Statements,
         No_Specific_Termination_Handlers,
         No_Task_Allocators,
         No_Task_Hierarchy,
         No_Task_Termination,
         Simple_Barriers,
         Max_Entry_Queue_Length => 1,
         Max_Protected_Entries => 1,
         Max_Task_Entries => 0,
         No_Dependence =>
            Ada.Asynchronous_Task_Control,
         No_Dependence => Ada.Calendar,
         No_Dependence =>
            Ada.Execution_Time.Group_Budgets,
         No_Dependence => Ada.Execution_Time.Timers,
         No_Dependence => Ada.Synchronous_Barriers,
         No_Dependence => Ada.Task_Attributes,
         No_Dependence =>
            System.Multiprocessors.Dispatching_Domains);
```

4 Rationale

This chapter provides a description of each restriction, a detailed rationale for the imposition of each restriction and some general discussion about how to work within the restrictions while still retaining flexibility in the design and coding processes.

4.1 Ravenscar Profile Restrictions

4.1.1 Static Existence Model

The restrictions listed below ensure that the set of tasks and interrupts to be analysed is fixed and has static properties (in particular, base priority) after program elaboration. If a variable task set were to exist, then it would be impractical to perform static timing analysis of the program because of the dynamic nature of the requirements for CPU time and the meeting of deadlines.

No_Task_Hierarchy

> [RM D.7] *No task depends on a master other than the library-level master.*

The restriction No_Task_Hierarchy prevents the declaration of tasks local to procedures or to other tasks. Thus, tasks may only be created at the library level, i.e. within the declarative part of library level package specifications and bodies, including child packages and package subunits.

No_Task_Allocators

[RM D.7] *There are no allocators for task types or types containing task subcomponents.*

The restriction No_Task_Allocators prevents the dynamic creation of tasks via the execution of Ada allocators [RM 4.8].

No_Task_Termination

[RM D.7] *All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler set for the partition it should be called when the first task attempts to terminate.*

The restriction attempts to mitigate the hazard that may be caused by tasks terminating silently. Real-time tasks normally have an infinite loop as their last outermost statement.

No_Specific_Termination_Handlers

[RM D.7] *There is no use of a name denoting the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.*

The restriction No_Specific_Termination_Handlers ensures that the only termination handler defined for the program is a fall-back handler [RM C.7.3].

No_Abort_Statements

[RM D.7] *There are no abort_statements, and there is no use of a name denoting Task_Identification.Abort_Task.*

The restriction No_Abort_Statements ensures that tasks cannot be aborted. The removal of abort statements (and select then abort) significantly reduces the size and complexity of the run-time system. It also reduces non-determinacy.

No_Dynamic_Attachment

[RM D.7] *There is no use of a name denoting any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).*

The restriction No_Dynamic_Attachment excludes use of the operations in predefined package Ada.Interrupts, which contains primitives to attach and detach handlers dynamically during program execution. In conjunction with restriction No_Local_Protected_Objects (see below) this implies that interrupt handlers can only be attached statically using Attach_Handler applying to protected procedures within library-level protected objects. Note the types and names defined in Ada.Interrupts can be used.

No_Dynamic_Priorities

[RM D.7] *There are no semantic dependencies on the package Ada.Dynamic_Priorities, and no occurrences of the attribute Priority.*

The restriction No_Dynamic_Priorities disallows the use of the predefined package Ada.Dynamic_Priorities, thereby ensuring that the priority assigned at task creation is unchanged during task execution, except when the task is executing a protected operation, during which time it inherits the ceiling priority. Protected objects also have unchanging ceiling priorities (as the Priority attribute [RM 4.1.4] cannot be used).

No_Local_Timing_Events

[RM D.7] *Timing events are declared only at library level.*

The restriction No_Local_Timing_Events prevents the declaration of timing events local to procedures or tasks. Thus, Timing_Events may only be created at the library level.

4.1.2 Static Synchronization and Communication Model

These restrictions are a natural consequence of the static execution model, since a locally declared protected object is meaningless for mutual exclusion and task synchronization purposes if it can only be accessed by one task. Furthermore, a static set of protected objects is required for schedulability analysis.

No_Local_Protected_Objects

[RM D.7] *Protected objects are declared only at library-level.*

The restriction No_Local_Protected_Objects prevents the declaration of protected objects local to subprograms, tasks, or other protected objects.

No_Protected_Type_Allocators

[RM D.7] *There are no allocators for protected types or types containing protected type subcomponents.*

The restriction No_Protected_Type_Allocators prevents the dynamic creation of protected objects via Ada allocators [RM 4.8].

No_Select_Statements

[RM D.7] *There are no select_statements.*

Max_Task_Entries => N

[RM D.7] *Specifies the maximum number of entries per task.*

For the Ravenscar Profile, the value of Max_Task_Entries is zero.

The restrictions Max_Task_Entries => 0 and No_Select_Statements prohibit the use of Ada rendezvous for task synchronization and communication. This ensures that these operations are achieved using only the two supported task synchronization primitives: protected object entries and suspension objects, both of which exhibit the time-deterministic execution properties needed for static timing analysis.

4.1.3 Deterministic Memory Usage

The Ravenscar Profile contains two restrictions that are designed to prevent implicit dynamic memory allocation by the implementation. The Ravenscar Profile does not prevent the use of the standard storage pool or a user-defined storage pool via explicit allocators. However, if there were no application-level visibility or control over how the storage in the standard storage pool was managed, the use of this pool would not be recommended.

No_Implicit_Heap_Allocations

> [RM D.7] *There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.*
>
> The restriction No_Implicit_Heap_Allocations prevents the implementation from allocating memory from the standard storage pool other than as part of the execution of an Ada allocator.

No dependence on Ada.Task_Attributes

> [RM D.13] *There are no semantic dependencies on the package Ada.Task_Attributes.*
>
> The restriction No_Task_Attributes_Package prevents use of the predefined package Ada.Task_Attributes [RM C.7.2], which is used to dynamically create attributes of each task in the application. Attribute creation may cause implicit dynamic allocation of memory. Although an implementation is allowed to statically reserve space for such attributes and then to impose a restriction on usage, it is felt that support of this feature is not compatible with the static nature of Ravenscar programs.

4.1.4 Deterministic Execution Model

The following restrictions ensure deterministic execution:

Max_Protected_Entries => N

> [RM D.7] *Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.*
>
> For the Ravenscar Profile, the value of Max_Protected_Entries is 1.

Max_Entry_Queue_Length => N

> [RM D.7] *Defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error exception at the point of the call.*
>
> For the Ravenscar Profile, the value of Max_Entry_Queue_Length is 1, and a call can only be queued on a protected entry, since Max_Task_Entries is 0.
>
> The restrictions Max_Protected_Entries => 1 and Max_Entry_Queue_Length => 1 ensure that at most one task can be suspended waiting on a closed entry barrier for each protected object which is used as a task synchronization primitive. This avoids the possibility of queues of task calls forming on an entry, with the associated non-determinism of the length of the waiting time in the queue. It also avoids two or more barriers becoming open simultaneously as the result of a protected action, with the associated non-determinism of selecting which entry should be serviced first. The restriction also enables a tight time bound on the *epilogue* code to be determined.
>
> The Max_Entry_Queue_Length restriction may only be checkable at run time, in which case violation would result in the raising of the Program_Error exception at the point of the entry call. This is consistent with the Ada rule that states that Program_Error exception is raised upon calling Suspend_Until_True if another task is waiting on that suspension object (when the Detect_Blocking pragma is enabled as it is in the Ravenscar Profile) [RM D.10]. An application could further restrict a Ravenscar program so that only one task is able to call one specific entry. A static check could then be provided, but this goes beyond what the Ravenscar Profile defines.
>
> When the restriction Max_Entry_Queue_Length => 1 is in force, Queuing_Policy ([RM D.4]) has no effect, since there are no queues.

Simple_Barriers

> [RM D.7] *The Boolean expression in an entry barrier shall be either a static expression or a name that statically denotes a component of the enclosing protected object.*
>
> The restriction Simple_Barriers, coupled with Max_Protected_Entries => 1, ensures a deterministic execution time and absence of side effects for the evaluation of entry barriers at the epilogue of protected actions within a protected object that is used for task synchronization. There is also scope for additional optimization by the implementation since the barrier value is either static or can be read directly from one of the protected object components, without needing to be computed separately. If the application requires composite entry barrier expressions, this can be achieved by declaring an additional Boolean in the protected data and assigning the composite expression to the Boolean whenever its evaluation result may change. The Boolean variable must be declared within the protected object (or type).

No_Requeue_Statements

> [RM D.7] *There are no requeue_statements.*
>
> The restriction No_Requeue_Statements ensures deterministic task release from protected entry barriers used for task synchronization. The requeue_statement in Ada causes the current caller of a protected entry to be

requeued to a different entry dynamically, thereby making it difficult to perform static analysis of task release.

No dependence on Ada.Asynchronous_Task_Control

[RM D.13] *There are no semantic dependencies on the package Ada.Asynchronous_Task_Control.*

The restriction No_Asynchronous_Control excludes the use of asynchronous suspension of execution. This ensures that task execution is temporally deterministic. See also the comments made on No_Abort_Statements.

No_Relative_Delay

[RM D.7] *There are no delay_relative_statements, and there is no use of a name that denotes the Timing_Events.Set_Handler subprogram that has a Time_Span parameter.*

The restriction No_Relative_Delay prohibits use of the delay_relative_statement based on type Duration. This statement exhibits non-determinism with respect to the absolute time at which the delay expires in the case when the delaying task is preempted after calculating the required relative delay, but before actual suspension occurs. In contrast, the delay_until_statement is deterministic and should be used for accurate release of time-triggered tasks.

No dependency on Ada.Calendar

[RM D.13] *There are no semantic dependencies on the package Ada.Calendar.*

The restriction No_Calendar ensures that all timing is performed using the high precision afforded by the time type in package Ada.Real_Time [RM D.8], or by an implementation-defined time type. The Ada.Real_Time time type has a precision of the same order of magnitude as the real-time clock device on the underlying processor board. In contrast, the time type in package Calendar generally has much coarser precision than the real-time clock, due to the need to support a 200-year range, and so its use could result in less accuracy in task release times. In addition, only the clock available from Ada.Real_Time is required to be monotonic.

4.1.5 Simple Run-time Behaviour

To reduce the overheads required to support the full Ada model, some features are removed from the Ravenscar Profile: in particular, time-triggered tasks.

No dependency on Ada.Execution_Time.Group_Budgets

[RM D.13] *There are no semantic dependencies on the package Ada.Execution_Time.Group_Budgets.*

A Ravenscar runtime can monitor the execution time of tasks, but it does not support the sharing of a CPU budget within a group of tasks. Neither does it require a handler to be executed if a task executes beyond a defined level of execution time (hence the next restriction). This simplifies the runtime but makes it harder to construct programs that can recover from timing errors.

No dependency on Ada.Execution_Time.Timers

[RM D.13] *There are no semantic dependencies on the package Ada.Execution_Time.Timers.*

4.1.6 Parallel Semantics

More recent definitions of the Ada language have included features that provide more control over the execution of multi-tasking programs on parallel hardware. Such hardware includes multiprocessors (with various memory configurations), multi-core processor and various forms of heterogeneous architectures. The definition of the Ravenscar Profile has been extended to deal with these forms of truly parallel (rather than just concurrent) execution. The basic approach chosen for the Ravenscar Profile has been to support the static allocation of tasks to processors.

No_Dynamic_CPU_assignment

[RM D.13] *All of the tasks in the partition will execute on the same CPU unless the programmer explicitly uses aspect CPU to specify the CPU assignments for tasks.*

This results in tasks being statically assigned to processors.

No dependency on Ada.Synchronous_Barriers

[RM D.13] *There are no semantic dependencies on the package Ada.Synchronous_Barriers.*

Synchronous barriers [RM D.10.1] are used on some forms of parallel hardware. As they can be programmed by the user in a Ravenscar application, the use of the predefined package is not explicitly supported by the Ravenscar Profile.

No dependency on
System.Multiprocessors.Dispatching_Domains

[RM D.13] *There are no semantic dependencies on the package System.Multiprocessors.Dispatching_Domains*

Dispatching domains allow more structured approaches to parallel execution to be supported. Currently, this leads to programs that are deemed to be beyond what can be easily analysed; they are therefore not included in the Ravenscar Profile

4.1.7 Implicit Restrictions

The set of restriction identifiers for Ada does not represent an orthogonal set of restrictions with the result that some restrictions are implied by others. For example, No_Select_Statements implies Max_Select_Alternatives must be zero.

4.2 Ravenscar Profile Dynamic Semantics

4.2.1 Task Dispatching Policy

The task dispatching policy that is required by **pragma** Profile(Ravenscar) is FIFO_Within_Priorities [RM D.2].

4.2.2 Locking Policy

The locking policy that is required by **pragma** Profile(Ravenscar) is Ceiling_Locking [RM D.3]. This policy provides one of the lowest worst case blocking times for contention for shared resources, and so maximizes the schedulability of the task set when preemptive scheduling is used.

4.2.3 Queuing Policy

The queuing policy is not meaningful for **pragma** Profile(Ravenscar) since no entry queues can form. Thus queuing policy identifiers FIFO_Queuing and Priority_Queuing have no effect.

4.2.4 Additional Run-Time Errors Defined by the Ravenscar Profile

The Ada language standard defines a number of concurrency-related run-time checks that may lead to the raising of an exception. The Ravenscar Profile restrictions greatly reduce the quantity of these checks, and thus the number of exception cases that can occur. The two concurrency-related run-time checks that apply to Ravenscar programs are:

- detection of priority ceiling violation as defined by Ceiling_Locking policy;
- detection of violation of not more than one task waiting concurrently on a suspension object (via the Suspend_Until_True operation).

The Ravenscar Profile introduces some additional concurrency-related checks that are potentially detectable only at execution time:

- the maximum number of calls that are queued concurrently on an entry must not exceed one. Program_Error exception is raised if the error occurs (**pragma** Restrictions(Max_Entry_Queue_Length => 1));
- all tasks are non-terminating (**pragma** Restrictions(No_Task_Termination)).

A conforming implementation must document the effect of a task that attempts to terminate. Possible effects may include:

- allowing the task to terminate silently;
- holding the task in a permanent pre-terminated state;
- executing a task termination handler.

Whatever action is taken by the implementation, the application cannot assume that full task termination actions (including finalization) have been executed.

4.2.5 Potentially-Blocking Operations in Protected Actions

The Ravenscar Profile requires detection of the following bounded error in the Ada standard, with the consequential raising of Program_Error exception:

- execution of a potentially-blocking operation during a protected action (**pragma** Detect_Blocking).

The Ravenscar Profile definition does however significantly reduce the list of potentially-blocking operations that may occur during a protected action. In particular, the following potentially-blocking operations are eliminated by the Ravenscar Profile definition:

- a select_statement
- an accept_statement
- a task entry call
- a delay_relative_statement
- an abort_statement
- task creation or activation
- an external requeue_statement with the same target object as that of the protected action.

The Ravenscar Profile definition does not require detection, at compile time, of other potentially blocking operations defined by the language standard [RM 9.5.1 (16)]. In this case, it is allowed for the detection to occur at the point of execution of the potentially blocking operation within the called subprogram body.

The rationale for requiring detection of potentially-blocking operations in protected actions is to allow a highly efficient and temporally deterministic implementation of Ceiling_Locking policy on a mono-processor. In effect, the ceiling priority alone is sufficient to provide the required mutual exclusion without the need to use locks such as *mutexes* once it is guaranteed that the task cannot suspend co-operatively whilst inside the protected operation. This form of locking is also non-queuing on a mono-processor, with the associated benefit of removing the need to compute the worst-case duration that a task call may wait in the queue.

4.2.6 Exceptions and the No_Exceptions Restriction

The general concern within high integrity systems of the occurrence of unhandled exceptions is not addressed directly by the Ravenscar Profile since exceptions relate to the sequential, rather than the concurrent, part of the language. Consequently, whereas an unhandled exception will cause a sequential program to terminate, and hence offer an immediate opportunity for some program level control to invoke recovery actions, an unhandled exception during the execution phase of a concurrent program may not be detected. In particular, an unhandled exception can cause any of the following effects:

- silent abandonment of the execution of an interrupt handler;
- silent termination of a task;
- premature exit from a protected action.

The Ravenscar Profile statically avoids the possibility that an exception can be raised by an entry barrier via the restriction Simple_Barriers. In addition, the Ravenscar Profile imposes the restriction No_Task_Termination that requires the implementation to document the effect of a task attempting to terminate. Nevertheless, this is inadequate for most high integrity applications that require static demonstration of absence of exceptions due to run-time check failure. Some techniques are presented in Section 6.2 to address the topic of proof of absence of the concurrency-related run-time errors that may occur in a Ravenscar Profile program, using static analysis.

The Ada standard includes the identifier No_Exceptions as a valid argument for the Restrictions pragma. It should be noted that the inclusion of this pragma does not provide a static guarantee of exception freedom – it merely guarantees that the application code does not contain any explicit raise_statement, nor code generation for language-defined checks, nor any exception handlers. However, it is possible for an exception to be raised automatically by the underlying hardware, or by built-in code in the run-time system. There is a documentation requirement on the implementation to define such cases [RM H.4 (25)].

In addition, the language standard defines execution of a program to become *erroneous* if a language-defined check is suppressed via the No_Exceptions restriction and the conditions arise that would have caused the check to fail [RM H.4 (26)]. This is consistent with the suppression of checks using **pragma** Suppress [RM 11.5 (26)]. Since erroneous execution results in the behaviour of a program becoming undefined, the recommendation for high integrity systems is that the No_Exceptions restriction should only be used in conjunction with verification and analysis techniques (see Chapter 1) that can statically prove that no exceptions due to run-time check failure can occur. In this case, the No_Exceptions restriction is providing the additional safeguard that exception raising via explicit raise_statements will be prohibited at compile time.

4.2.7 Access to Shared Variables

The Ravenscar Profile requires all synchronization and communication between tasks and interrupt handlers to use data that are guaranteed to have mutually-exclusive access. This prevents any erroneous execution that might arise if concurrent access (that includes a write operation) to the same unprotected shared variable is permitted. Such access control is provided in Ada using one of the following constructs:

- a protected object;
- a suspension object;
- an *atomic* object (to which the Atomic aspect applies).

This access control model applies to the operational phase of the application, after program initialization via elaboration of library-level packages is complete. For each class of object above, it is possible to ensure that its initialization is completed as part of program elaboration.

There is an issue however, in that the semantics of Ada define task activation and interrupt handler attachment to occur during library-level elaboration code for objects that are declared within library-level packages. Consequently, it is the case that tasks will execute their declarative part and may proceed into their sequence_of_statements, and that interrupt handlers may execute, prior to the elaboration code for program initialization being completed. This scenario could give rise to the following undesirable effects:

- a task body or interrupt handler may suffer an access-before-elaboration exception;
- a task body or interrupt handler may access uninitialized data;
- a task body or interrupt handler may access unprotected data concurrently that it shares only with the thread of control that is performing the data initialization.

It is possible to program each task such that it suspends itself at the start of its sequence of statements, but this is not possible for interrupt handlers (although an application may be able to inhibit interrupts if the device allows). Furthermore, the code executed as part of task activation (prior to the suspension point) may suffer the effects listed above. In order to address this issue, the Partition_Elaboration_Policy is defined in the Ada standard (see below).

4.2.8 Elaboration Control

The new **pragma** Partition_Elaboration_Policy [RM H.6] is not part of the Ravenscar Profile, but it is closely related to it. If given the argument Sequential, this defines an alternative elaboration behaviour in which all tasks declared at the library level proceed to their activation only *after* the environment task has completed all its elaborations and the main program is leaving its declarative_part. It is only at that point that interrupt handlers are attached (so that no interrupt can be delivered earlier), and all tasks eventually start their concurrent execution. This **pragma** complements those that are defined by the Ravenscar Profile and helps achieve the goal that controlled access to global shared variables is met during program initialization.

5 Examples of use

This chapter illustrates some simple patterns of use of the Ravenscar Profile.

The Ravenscar Profile can be used with a variety of coding styles. However, if the user is required to perform program analysis, for example to check the schedulability of the tasks, then certain coding styles are recommended. Indeed, a small number of templates can cater for a large class of application needs. In the first eight sections of this chapter, we give examples that illustrate the straightforward use of the Ravenscar Profile. After that, in Sections 5.9 to 5.12, we show how the Ravenscar Profile can deal with requirements that would appear to lie outside of its scope.

With the 2012 version of the language specification, aspects should be used in place of most pragmas. Accordingly, we have replaced all occurrences of the obsolete pragmas with the corresponding aspect.

5.1 Cyclic Task

The task body for a cyclic (or periodic) task typically has, as its last statement, an outermost infinite loop containing one or more delay_until_statements. The basic form of a cyclic task has just a single delay statement either at the start or at the end of the statements within the loop. The Ravenscar Profile supports only one time type for use as the argument – Ada.Real_Time.Time, although a user-defined time type could be used.

Task termination is considered to be an error condition in Ravenscar-compliant code since there is no dynamic task creation (and hence the thread of control would be permanently lost). Hence, the loop that is presented in the template below is infinite.

A cyclic task will not usually contain any other form of voluntary-suspension statement in the infinite loop, since this would undermine the schedulability analysis.

The Ravenscar Profile supports the use of discriminants for task types and protected types. One use of a discriminant is to set differing priorities for task objects or protected objects that are of the same type by using it as the argument of the Priority aspect.

Discriminants can also be used to indicate the period of a cyclic task or other task parameters, including the assigned priority.

Example 1, Cyclic Template

```
task type Cyclic(Pri : System.Priority;
                 Cycle_Time : Positive)
  with Priority => Pri;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
      Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations as needed
begin
  -- Initialization code
  Next_Period := Ada.Real_Time.Clock + Period;
  loop
    delay until Next_Period;
    -- Wait one whole period before executing
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;

-- Now we declare two task objects of this type
C1 : Cyclic(20,200);
C2 : Cyclic(15,100);
```

Cyclic tasks normally exchange data through protected operations. In this coding style, there are no protected entries since the only activation event is on `delay until`. Conformance with the Ravenscar Profile requires that all shared data be placed in protected objects to avoid corruption. [3]

5.2 Co-ordinated release of Cyclic Tasks

The simple example illustrated above has a number of cyclic tasks that each read the clock and then suspend for time 'Period'. It can however by useful for all such tasks to co-ordinate their start times so that they share a common epoch. This can help to enforce precedence relations across tasks. To achieve this a protected object is used, which reads the clock on creation and then makes this clock value available to all cyclic tasks.

Example 2, Protected Object Implementing an Epoch

```
protected Epoch
  with Priority => System.Priority'Last is
  function Start_Time return Ada.Real_Time.Time;
private
  Start : Ada.Real_Time.Time := Ada.Real_Time.Clock;
end Epoch;

protected body Epoch is
  function Start_Time return Ada.Real_Time.Time is
  begin
    return Start;
  end Start_Time;
end Epoch;
```

A protected object is not strictly needed to this end, since a shared variable appropriately initialized will suffice. A more robust scheme and one that only reads the epoch time once a task actually needs it is as follows.

Example 3, Caller Initialized Epoch

```
protected Epoch
  with Priority => System.Priority'Last is
  procedure Get_Start_Time(
                  T : out Ada.Real_Time.Time);
private
  Start : Ada.Real_Time.Time;
  First : Boolean := True;
end Epoch;

protected body Epoch is
  procedure Get_Start_Time(
                  T : out Ada.Real_Time.Time) is
  begin
    if First then
     First := False;
     Start := Ada.Real_Time.Clock;
    end if;
    T := Start;
  end Get_Start_Time;
end Epoch;
```

This leads to the following further example.

Example 4, Cyclic Task Using Epoch

```
task type Cyclic(Pri : System.Priority;
                 Cycle_Time : Positive)
  with Priority => Pri;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
           Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations as needed
```

[3] Editor Note: the ISO technical report adds the possible use of atomic objects, statically proven free of race conditions.

```
begin
  -- Initialization code
  Epoch.Get_Start_Time(Next_Period);
  Next_Period := Next_Period + Period;
  loop
    delay until Next_Period;
    -- Wait until next period after epoch
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;
```

5.3 Cyclic Tasks with Precedence Relations

The use of priorities and a shared epoch can be used to enforce precedence between tasks with the same period, if the application can be restricted so that the tasks do not block during execution. An alternative scheme is to use an offset in time. Here, scheduling analysis is used to ensure that each task has completed before the next is released.

Example 5, Cyclic Tasks with Offsets

```
task type Cyclic(Pri : System.Priority;
                 Cycle_Time, Offset : Natural)
  with Priority => Pri;

task body Cyclic is
  Next_Period : Ada.Real_Time.Time;
  Period : constant Ada.Real_Time.Time_Span :=
           Ada.Real_Time.Microseconds(Cycle_Time);
  -- Other declarations
begin
  -- Initialization code
  Next_Period := Epoch.Start_Time +
                 Ada.Real_Time.Microseconds(Offset);
  loop
    delay until Next_Period;
    -- Wait until next period after offset
    -- Non-suspending periodic response code
    -- May include calls to protected procedures
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;

First : Cyclic(20,200,0);  -- Required to complete with
                           -- deadline 70
Second : Cyclic(20,200,70);
```

5.4 Event-Triggered Tasks

The task body for an event-triggered task that conforms to the Ravenscar Profile typically has, as its last statement, an outermost infinite loop whose first statement is either a call to a protected entry or a call to Ada.Synchronous_Task_Control.Suspend_Until_True using a Suspension Object. The suspension object is used when no other effect is required in the signalling operation; for example, no data is to be transferred from signaller to waiter. In contrast, the protected entry is used for more elaborate event signalling, when additional operations must accompany the resumption of the event-triggered task.

An event-triggered task will not usually contain any other form of voluntary-suspension statement in the infinite loop.

Example 6, An Event-Triggered Task

```
-- A suspension object, SO, is declared in a visible library
-- unit and is set to True in another (releasing) task

task type Sporadic(Pri : System.Priority)
  with Priority => Pri;

task body Sporadic is
  -- Declarations
begin
  -- Initialization code
  loop
     Ada.Synchronous_Task_Control.
                           Suspend_Until_True(SO);
    -- Non-suspending sporadic response code
  end loop;
end Sporadic;

Sp : Sporadic(17);
```

5.5 Shared Resource Control using Protected Objects

A protected object used to ensure mutually exclusive access to a shared resource, such as global data, typically contains only protected subprograms as operations, i.e. no protected entries. Protected entries are used only for task synchronization purposes where data exchange is involved. A protected procedure should be used when the internal state of the protected data must be altered, and a protected function should be used for information retrieval from the protected data, when the data remains unchanged.

The Ada Reference Manual states that the use of any form of voluntary-suspension statement during the execution of a protected operation is a bounded error [RM 9.5.1 (8)]. The Ravenscar Profile requires, via **pragma** Detect_Blocking, an implementation to detect this error (and hence to raise the Program_Error exception), other than in the case when suspension is due to execution outside of the Ada environment, for example within an underlying operating system call or within imported code that is written in another language.

It is essential to choose the correct value for the ceiling priority of the protected object. By default, the value is System.Priority'Last, unless the protected object contains interrupt handlers (see below). The chosen value must be at least as high as the highest priority task that calls one of the protected operations. If this is not the case, the Ada Reference Manual requires the Program_Error exception to be raised when a task with a priority higher than the ceiling priority makes a call to one of the protected operations. However, if the ceiling value is higher than necessary, there may be an increase in the blocking time that high priority tasks will suffer, and consequently a decrease in the overall schedulability of the system. Tool support may be available to determine the optimal ceiling value when the calling sequences can be statically analysed.

Example 7, Use of Protected Object for Mutual Exclusion

```
protected Shared_Data
  with Priority => 10  -- All callers must have priority no
                       -- greater than 10
is
  function Get return Data;  -- For some global type, Data
  procedure Put(D : in Data);
private
  Current : Data; -- Shared data declaration
end Shared_Data;

protected body Shared_Data is
  function Get return Data is
  begin
    return Current;
  end Get;
  procedure Put(D : in Data) is
  begin
    Current := D;
  end Put;
end Shared_Data;
```

5.6 Task Synchronization Primitives

Task synchronization, in the form of a wait/signal event model, can be achieved in the Ravenscar Profile using either a protected entry or a suspension object, as described above for event-triggered tasks.

The suspension object is the optimized form for a simple suspend/resume operation. The package Ada.Synchronous_Task_Control [RM D.10] is used to declare a suspension object, and the primitives Suspend_Until_True and Set_True are used for the suspend and resume operations respectively.

The use of protected objects with entries for task synchronization is restricted by the Ravenscar Profile. The protected object can have at most one entry declaration; the entry barrier must be a simple value that is either a Boolean literal or a Boolean variable that is part of the protected state; and at most one task is allowed to wait on the protected entry at any time (see Section 4.1.4). These restrictions provide the necessary determinism in knowing which waiting task is serviced first when entry_barriers become true, since there can be at most one such task call enqueued at it. This model is very similar to the suspension object approach except that:

- Data can be transferred from signaller to waiter atomically (i.e. without risk of a race condition) by use of parameters to the protected operations and additional protected data.
- Additional code can be executed atomically as part of signalling by use of the bodies of the protected operations.

Example 8, Event-Triggered Tasks Suspending on a Protected Entry

```
protected type Event(Ceiling : System.Priority)
  with Priority => Ceiling --  Ceiling priority defined for each
                              -- object
is
  entry Wait(D : out Data);
  procedure Signal(D : in Data);
private
  Current : Data;  -- Event data declaration
  Signalled : Boolean := False;
end Event;

protected body Event is
  entry Wait(D : out Data) when Signalled is
  begin
    D := Current;
    Signalled := False;
  end Wait;
  procedure Signal(D : in Data) is
  begin
    Current := D;
    Signalled := True;
  end Signal;
end Event;

Event_Object : Event(15);

task Event_Handler
  with Priority => 14; --  I.e. this must be not greater than 15

task body Event_Handler is
  -- Declarations, including D of type Data
begin
  -- Initialization code
  loop
   Event_Object.Wait(D);
   -- Non-suspending event handling code
  end loop;
end Event_Handler;
```

5.7 Minimum Separation between Event-Triggered Tasks

To ensure the timely execution of all tasks in a system it may be necessary to enforce a separation between sporadic tasks so that they cannot execute more frequently than some agreed value. This is easily achieved with a delay_until_statement. Doing so however introduces a second activation event into the code of the task's outer loop. In general, this can make the task more difficult to analyse. In Example 9 below however, it actually facilitates the analysis by ensuring a minimum separation between task activations. This happens because the two activation events are in effect subsequent.

Example 9, Event-Triggered Task with Minimum Separation

```
task Event_Handler
  with Priority(14);

task body Event_Handler is
  -- Declarations, including D of type Data
  Minimum_Separation : constant
     Ada.Real_Time.Time_Span := -- some appropriate
                                  -- value
  Next : Ada.Real_Time.Time;
begin
  -- Initialization code
  loop
    Event_Object.Wait(D);
    Next := Ada.Real_Time.Clock + Minimum_Separation;
    -- Non-suspending event handling code
    delay until Next;  -- this ensures minimum temporal
                       -- separation
  end loop;
end Event_Handler;
```

5.8 Interrupt Handlers

The code of an interrupt handler will often be used to initiate a response in an event-triggered task. This is because the code in the handler itself executes at the hardware interrupt level, and typically the major part of the processing of the response to the interrupt is moved into an event response task, which executes at a software priority level with interrupts fully enabled.

In Example 8 above, if signalling is to be achieved via an interrupt, then the procedure Signal should be defined as parameterless, and be identified as an interrupt handler by the aspect Attach_Handler. This aspect includes an argument of type Ada.Interrupts.Interrupt_ID that identifies the interrupt to which the handler applies.

The ceiling priority of a protected object that contains an interrupt handler must be in the range of System.Interrupt_Priority.

Example 10, Interrupt Handling via a Protected Entry

```
protected Interrupt_Event
  with Interrupt_Priority => System.Interrupt_Priority'Last
is
  entry Wait(D : out Data);
  procedure Signal
    -- Must be parameterless
    with Attach_Handler => Some_Interrupt_Id;
  -- Wait and Signal will execute with full interrupt lockout
private
  Current : Data;  -- Event data declaration
  Signalled : Boolean := False;
end Interrupt_Event;

protected body Interrupt_Event is
  -- Similar to the code in Example 8
  -- except that the setting of Current is
  -- obtained via a register during
  -- the execution of Signal rather than as an in parameter
```

5.9 Catering for Entries with Multiple Callers

In this and the following three sections we illustrate how to cater for situations that appear to need more functionality than provided by the Ravenscar Profile. In doing this we are not attempting to say that Ravenscar applications will be able to deal with all situations that full Ada covers. The tasking features of Ada represent a powerful set of abstractions for programming concurrent and real-time systems. To gain predictability and efficiency, the Ravenscar Profile has had to drop many of these features, and it is not appropriate to reintroduce them via a combination of programming tricks and conventions. However, situations may arise when a requirement in just part of a program seems outside of the Ravenscar Profile's definition. These can often be catered for by straightforward techniques that benefit from the other restrictions of the Ravenscar Profile.

Here we focus on the requirement for two (or more) tasks to call the same entry of some protected object. As an illustration, consider a situation in which a series of tasks create work items, while others consume them. If more than 10 (say) outstanding items ever accumulate then the two separate event-triggered tasks must be released. An atomicity requirement is that the two tasks are only released if both are available and only when new work items are created.

A non Ravenscar Example

```
protected Controller is
  entry Overload;  -- called by two tasks
  procedure Create;
  procedure Consume;
private
  Work_Items : Integer := 0;
  Released : Boolean := False;
end Controller;

protected body Controller is
  entry Overload when Released is
  begin
  if Overload'Count = 0 then -- barrier is closed when both
                                -- tasks have left
    Released := False;
  end if;
  end Overload;
  procedure Create is
  begin
    Work_Items := Work_Items + 1;
    Released := (Work_Items > 10 and
                 Overload'Count = 2);
    -- barrier is opened when more than 10 items
    -- and both tasks are waiting
  end Create;
  procedure Consume is
  begin
    Work_Items := Work_Items – 1;
  end Consume;
end Controller;
```

To conform with the Ravenscar Profile restrictions, two Controller protected objects are needed, one for each task. To get the required atomicity the second Controller must be called from the first.

Example 11, Using Multiple Protected Objects to Mimic an Entry Queue

```
protected First_Controller is
  entry Overload;  -- called by one task
  procedure Check_Called(OK : out Boolean);
private
  Released : Boolean := False;
end First_Controller;

protected body First_Controller is
  entry Overload when Released is
  begin
    Released := False; -- barrier set to False once task has
                       -- been released
  end Overload;
  procedure Check_Called(OK : out Boolean) is
  begin
    Released := (Overload'Count = 1);
    OK := Released; -- returns True if task waiting
  end Check_Called;
end First_Controller;

protected Second_Controller is
  entry Overload;  -- called by the other task
  procedure Create;
  procedure Consume;
private
  Work_Items : Integer := 0;
  Released : Boolean := False;
end Second_Controller;

protected body Second_Controller is
  entry Overload when Released is
  begin
    Released := False; -- barrier set to False once task has
                       -- been released
  end Overload;
  procedure Create is
  begin
    Work_Items := Work_Items + 1;
    if Work_Items > 10 and Overload'Count = 1 then
      First_Controller.Check_Called(Released);
    end if;   -- if Released is true then the first task
              -- has been released
              -- and the second one must also be released
  end Create;
  procedure Consume is
  begin
    Work_Items := Work_Items – 1;
  end Consume;
end Second_Controller;
```

Note that, in the Ravenscar Profile, once a task calls an entry, it cannot cancel the call; hence the above algorithm is safe. In the full language, task calls can be cancelled and therefore the above approach would not be guaranteed to work.

5.10 Catering for Protected Objects with more than one Entry

To illustrate the way a two-entry protected object can be transformed, consider the standard buffer with one task calling the buffer to extract an item and another task calling it to place items in the buffer. Usually both of these calls must be made via entries in a protected object as the extract call must block if the buffer is empty, and the place call must block if the buffer is full. To comply with the Ravenscar Profile restriction of only one entry in any protected object, a protected object is used for mutual exclusion only and two suspension objects are introduced for the necessary conditional synchronization.

Example 12, A Bounded Buffer Example In Ravenscar

```
package Buffer is
  procedure Place_Item(Item : Some_Type);
  procedure Extract_Item(Item : out Some_Type);
end Buffer;

package body Buffer is
  protected Buff is
    procedure Place(Item    : in Some_Type;
                    Success : out Boolean);
    procedure Extract(Item    : out Some_Type;
                      Success : out Boolean);
  private
    Buffer_Full : Boolean := False;
    Buffer_Empty : Boolean := True;
    -- other declarations
  end Buff;

  Non_Full, Non_Empty :
      Ada.Synchronous_Task_Control.Suspension_Object;

  procedure Place_Item(Item : Some_Type) is
    OK : Boolean;
  begin
    Buff.Place(Item, OK);
    if not OK then
      Ada.Synchronous_Task_Control.
                         Suspend_Until_True(Non_Full);
      -- note this is a task activation event
      Buff.Place(Item, OK); -- OK must be true
    end if;
    Ada.Synchronous_Task_Control.Set_True(Non_Empty);
  end Place_Item;

  procedure Extract_Item(Item : out Some_Type) is
    OK : Boolean;
  begin
    Buff.Extract(Item, OK);
    if not OK then
      Ada.Synchronous_Task_Control.
                      Suspend_Until_True(Non_Empty);
      -- note this is a task activation event
      Buff.Extract(Item, OK); -- OK must be true
    end if;
    Ada.Synchronous_Task_Control.Set_True(Non_Full);
  end Extract_Item;

  protected body Buff is
    procedure Place(Item    : in Some_Type;
                    Success : out Boolean) is
    begin
      Success := not Buffer_Full;
      if not Buffer_Full then
        -- put Item into Buffer
        Buffer_Empty := False;
        -- set Buffer_Full if appropriate
      end if;
    end Place;
    procedure Extract(Item   : out Some_Type;
                      Success: out Boolean) ) is
    begin
      Success := not Buffer_Empty;
```

```
        if not Buffer_Empty then
          -- extract Item from Buffer
          Buffer_Full := False;
          -- set Buffer_Empty if appropriate
        end if;
      end Extract;
    end Buff;
end Buffer;
```

5.11 Programming Timeouts

There may be situations where a call to a protected object's entry should be retracted after a period of time if the event that should release it has not occurred. In full Ada, this would be:

```
    select
      PO.Call;
      Timeout := False;
    or
      delay until Some_Time;
      Timeout := True;
    end select;
```

Identical functionality can be achieved in Ravenscar by the use of an extra task that is event-triggered and a protected object that is used to pass the timeout value to this task. This is illustrated below; note the expansion in code needed to accommodate this effect. The full language clearly has significant superior expressive power in this, and other, areas.

Example 13, Programming Timeouts in Ravenscar

```
protected PO is
  entry Call(Timeout : out Boolean);
  procedure Used_To_Release_Call;
  procedure Too_Late;
private
  Timed_Out : Boolean := False;
  Release : Boolean := False;
end PO;

protected body PO is
  procedure Too_Late is
  begin
    if Call'Count = 1 then
      Timed_Out := True;
      Release := True;
    end if;
  end Too_Late;
  procedure Used_To_Release_Call is
  begin
    Timed_Out := False;
    Release := True;
  end Used_To_Release_Call;
  entry Call(Timeout : out Boolean) when Release is
  begin
    Timeout := Timed_Out;
    Release := False;
    -- further non-suspending code if necessary
  end Call;
end PO;

protected Timer_Control is
  entry Wait(Wait_Time : out Ada.Real_Time.Time);
  procedure Set_Time(Wait_Time : Ada.Real_Time.Time);
private
  Timeout : Ada.Real_Time.Time;
  Released : Boolean := False;
end Timer_Control;

protected body Timer_Control is
  entry Wait(Wait_Time : out Ada.Real_Time.Time)
       when Released is
  begin
    Wait_Time := Timeout;
    Released := False;
  end Wait;
  procedure Set_Time(
              Wait_Time : Ada.Real_Time.Time) is
  begin
    Timeout := Wait_Time;
    Released := True;
  end Set_Time;
end Timer_Control;

task Timer; -- note this task has more than one
            -- activation event

task body Timer is
  T : Ada.Real_Time.Time;
begin
  loop
    Timer_Control.Wait(T);
    delay until T;
    PO.Too_Late;
  end loop;
end Timer;

-- application calls the following
Timer_Control.Set_Time(Some_Time);
PO.Call(Timeout);
```

5.12 Further Expansions to the Expressive Power of the Ravenscar Profile

If static timing analysis is not of interest to the application program and a more general model of tasks and interrupts is required, this can still be achieved with reasonable expressive power within the subset definition. However, as noted earlier, the Ravenscar Profile is not a substitute for the full language when that level of expressive power is needed.

- Dynamic creation and termination of tasks can be simulated by declaring a pool of event-triggered tasks at program start-up, each containing an infinite loop which has a suspending operation as its first statement, such that its execution can be invoked dynamically by one of the task synchronization primitives. Thus, by changing the settings of suspension objects and entry barriers, it is possible for certain tasks to have their execution disabled whilst others have execution enabled.
- Dynamic exchange of interrupt handlers, often required for applications performing *mode change*, can be simulated by embodying all the different handling code for a particular interrupt in one interrupt handler protected procedure, with each of the different actions being coded as case alternatives in a case statement, dependent on a mode selector. By changing the value of the mode selector, the same handler procedure can

perform different response actions at various times during program execution.

- Dynamic task priority change is also generally associated with mode change. This can be simulated by use of a separate event response task for each mode of operation (and assigning a different priority to each task as required), such that the execution of each task that belongs to a dormant mode is suspended until signalled when its mode becomes active.
- A similar effect to requeue can be achieved by completing the protected entry body and returning a status result to the caller, which can then emit a subsequent protected entry call to the intended destination of the requeue statement. If each protected entry is called only by a single task, then this alternative technique does not introduce any race conditions.

Similarly, if static timing analysis is not of interest, the classic non-timed rendezvous operations can still be achieved within the subset definition by use of suspension objects for synchronization and protected object entries for communication.

No conditional form of suspension is supported by the Ravenscar Profile. This can be simulated if a suspension object is used by polling the state of the suspension object (via the Current_State function in package Ada.Synchronous_Task_Control), or if a protected entry is used by polling the value of the protected data which controls the synchronization (i.e. the barrier Boolean).

References

[AI 249] Ravenscar Profile for high integrity systems, ARG, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ AIs/ AI-00265.TXT

[AI 265] Partition elaboration policy for high integrity systems, ARG (2002), http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00265.TXT

[AI 305] New pragma and additional restriction identifiers for real-time systems, ARG (2002), http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00305.TXT

[CSP] Hoare, C. A. R. (2004), *Communicating Sequential Processes*, Prentice Hall International. ISBN 0-13-153271-5.

[DO] DO-178C Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc. (2011).

[DS] U.K. Ministry of Defence (1997), 00-55 *Requirements of Safety Related Software in Defence Equipment.*

[GA] Guide for the use of Ada Programming Language in High Integrity Systems (2000), ISO/IEC TR 15942.

[RM] International Standard ANSI/ISO/IEC-8652:2012 (2015), *Ada 2012 Reference Manual*, Technical Corrigendum 1.

Bibliography

[1] C. Lui and J. Layland (1973), *Scheduling algorithms for multiprogramming in a hard real-time environment*, JACM, 20 (1), 46 - 61.

[2] M. Joseph and P. Pandya (1986), *Finding response times in a real-time system*, BCS Computer Journal, 29 (5), 390 - 395.

[3] A. Burns, and A. J. Wellings (1997), *Restricted Tasking Models*, Ada Letters, XVII (5), 27 - 32.

[4] B. Dobbing and M. Richard-Foy (1997), *T-SMART - Task Safe, Minimal Ada Realtime Toolset*, Ada Letters, XVII (5), 45 - 50, 1997.

[5] A. Burns (1999), *The Ravenscar Profile*, Ada letters, XIX (4), 49 - 52.

[6] Session Summary (1999), *The Ravenscar Profile and Implementation Issues*, Ada Letters, XIX (2), 12 - 14.

[7] Session Summary (2001), *Status and Future of the Ravenscar Profile*, Ada Letters, XXI (1), 5 - 8.

[8] Session Summary, *Ravenscar Profile*, Proceedings of the 11th International Real-Time Ada Worshop, Ada Letters.

[9] A. Burns and A. J. Wellings (2016), *Analysable Real-Time Systems: Programmed in Ada*, Amazon Books.

[10] J.W.S. Liu (2000), *Real-Time Systems*, Prentice Hall.

[11] A. Burns and A. J. Wellings (1994), *HRT-HOOD: A design method for hard real-time Ada*, Real-Time Systems, 6 (1), 73 - 114.

National Ada Organizations

Ada-Belgium

attn. Dirk Craeynest
c/o KU Leuven
Dept. of Computer Science
Celestijnenlaan 200-A
B-3001 Leuven (Heverlee)
Belgium
Email: Dirk.Craeynest@cs.kuleuven.be
URL: www.cs.kuleuven.be/~dirk/ada-belgium

Ada in Denmark

attn. Jørgen Bundgaard
Email: Info@Ada-DK.org
URL: Ada-DK.org

Ada-Deutschland

Dr. Hubert B. Keller
Karlsruher Institut für Technologie (KIT)
Institut für Angewandte Informatik (IAI)
Campus Nord, Gebäude 445, Raum 243
Postfach 3640
76021 Karlsruhe
Germany
Email: Hubert.Keller@kit.edu
URL: ada-deutschland.de

Ada-France

attn: J-P Rosen
115, avenue du Maine
75014 Paris
France
URL: www.ada-france.org

Ada-Spain

attn. Sergio Sáez
DISCA-ETSINF-Edificio 1G
Universitat Politècnica de València
Camino de Vera s/n
E46022 Valencia
Spain
Phone: +34-963-877-007, Ext. 75741
Email: ssaez@disca.upv.es
URL: www.adaspain.org

Ada-Switzerland

c/o Ahlan Marriott
Altweg 5
8450 Andelfingen
Switzerland
Phone: +41 52 624 2939
e-mail: president@ada-switzerland.ch
URL: www.ada-switzerland.ch