ADA USER JOURNAL

Volume 40
Number 2
June 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Unless the authors state explicitly otherwise, submission of an article is taken to imply that it represents original, unpublished work, not under consideration for publication elsewhere.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

This issue is being concluded shortly after our annual flagship conference, the 24th Ada-Europe International Conference on Reliable Software Technologies, which took place 11-14 June 2019, in Warsaw, Poland. The organizers must be congratulated for a very successful conference, with a very rich program, both technical and social.

This edition of the conference featured several important innovations, of which I would like to particularly note the lower registration fee for all participants, even more reduced for paper and presentation authors, and the new journal-based, open-access publication model for the peer-reviewed papers. This new model will be continued in the next year; the reader can find in this issue the call for contributions for the 25th Ada-Europe International Conference on Reliable Software Technologies, which will take place in Santander, Spain, in the week of 8-12 of June 2020.

During the conference week there was also another change, which directly relates to the Ada User Journal: the Journal has a new Editor-in-Chief, António Casimiro, from the University of Lisbon, Portugal. I am glad that António was willing to take the job, and I know that the Journal will be in the best of hands. Thank you very much António, and my best wishes for this new endeavour.

I had the pleasure to serve as Editor-in-Chief of the Ada User Journal for 12 years – 12 volumes totalling 48 issues. It has been a very pleasing task, challenging at times, but always worthwhile. And it would not have been possible, without the continuous and strong support of many people in the Journal editorial team. I would like to express a heartfelt thanks to Patricia López, Jorge Real, Dirk Craeynest, Santiago Urueña, Marco Panunzio, Kristoffer Nyborg Gregertsen and Alejandro R. Mosteo. Also, and special, to Jacob Sparre Andersen, no longer with us. Finally, and of course, the contributors and readers. The journal would not exist if not for them.

As for the contents of this issue, we start the publication of the proceedings of the industrial track of the 2019 Ada-Europe Conference. The first paper, by Maurizio Martignano, from Spazio IT, Italy, discusses the evolution of static analysers for C programs. The second paper, by a group of authors from fortiss GmbH, Germany, presents AdaHorn, a model checker to verify Ada programs using a Horn constraints solver.

The issue also concludes the publication of the Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems, 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, which has been updated to consider Ada 2012. The first part of the guide, published in the March issue of the Journal, included the definition, rationale and examples of use of the Ravenscar profile. This second, and final, part includes the verification approach appropriate to analyse Ravenscar programs, and an extended example of its use.

I am sure the readers will enjoy reading this issue of the journal, as I have enjoyed.

Signing off …

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

Tools to get you there. Safely.
Ada and The GNAT Pro High-Integrity Family
www.adacore.com
AdaCore
The GNAT Pro Company

Quarterly News Digest

Alejandro R. Mosteo

Centro Universitario de la Defensa de Zaragoza, 50090, Zaragoza, Spain; Instituto de Investigación en Ingeniería de Aragón, Mariano Esquillor s/n, 50018, Zaragoza, Spain; email: amosteo@unizar.es

Contents

Ada-related Events

[To give an idea about the many Ada-related events organised by local groups, some information is included here. If you are organising such an event feel free to inform us as soon as possible. If you attended one please consider writing a small report for the Ada User Journal.]

FOSDEM 2019 post hoc summary

From: dirk@orka.cs.kuleuven.be (Dirk Craeynest)
Date: Mon, 25 Feb 2019 07:04:32 -0000
Subject: FOSDEM 2019 Ada Developer Room - presentations & videos online
Newsgroups: comp.lang.ada, fr.comp.lang.ada, comp.lang.misc

---

*** Presentations, videos, pictures available online ***

9th Ada Developer Room at FOSDEM 2019

Saturday 2 February 2019

Université Libre de Bruxelles (ULB), Solbosch Campus, Room AW1.125

Avenue Franklin D. Roosevelt Laan 50, B-1050 Brussels, Belgium

Organized in cooperation with Ada-Europe

www.cs.kuleuven.be/~dirk/ada-belgium/events/19/190202-fosdem.html

fosdem.org/2019/schedule/track/ada

---

All presentations and video recordings as well as some pictures from the 9th Ada Developer Room, held at FOSDEM 2019 in Brussels recently, are available via the Ada-Belgium and FOSDEM web sites now.

- "Welcome to the Ada DevRoom" by Dirk Craeynest - Ada-Belgium
- "An Introduction to Ada for Beginning and Experienced Programmers" by Jean-Pierre Rosen - Adalog
- "Sequential Programming in Ada: Lessons Learned" by Joakim Strandberg - Mequinox
- "Autonomous Train Control Systems: a First Approach" by Julia Teissl - FH Campus Wien
- "Controlling the Execution of Parallel Algorithms in Ada" by Jan Verschelde - University of Illinois at Chicago
- "Persistence with Ada Database Objects" by Stephane Carrez - Twinlife
- "Shrink your Data to (almost) Nothing with Trained Compression" by Gautier de Montmollin - Ada-Switzerland
- "GSH: an Ada POSIX Shell to Speed Up GNU Builds on Windows" by Nicolas Roche - AdaCore
- "What is Safety-Critical Software, and How Can Ada and SPARK Help?" by Jean-Pierre Rosen - Adalog
- "Secure Web Applications with AWA" by Stephane Carrez - Twinlife
- "Distributed Computing with Ada and CORBA using PolyORB" by Frédéric Praca - Ada-France
- "Cappulada: Smooth Ada Bindings for C++" by Johannes Kliemann - Componolit
- "AZip Archive Manager: a full-Ada Open-Source Portable Application" by Gautier de Montmollin - Ada-Switzerland
- "Proof of Pointer Programs with Ownership in SPARK" by Yannick Moy - AdaCore
- "Alternative Languages for Safe and Secure RISC-V Programming" by Fabien Chouteau - AdaCore, in RISC-V DevRoom on Sat 2 Feb
- "RecordFlux: Facilitating Verification of Communication Protocols" by Tobias Reiher - Componolit, in Security DevRoom on Sun 3 Feb

Presentation abstracts, speaker bios, pointers to relevant information, copies of slides, links to corresponding pages and video recordings, are available via the Ada-Belgium and FOSDEM sites at the URLs above.

Some pictures are posted as well. If you have more pictures or other material you would like to share, or know someone who does, then please contact me.

Finally, thanks once more to all presenters and helpers for their work and collaboration, thanks to all the FOSDEM organizers and volunteers, thanks to the many participants for their interest, and thanks to everyone for another nice experience!

Dirk Craeynest, FOSDEM Ada DevRoom coordinator

Dirk.Craeynest@cs.kuleuven.be (for Ada-Belgium/Ada-Europe/SIGAda/WG9)

#AdaFOSDEM #AdaProgramming #AdaBelgium #AdaEurope

DeCPS workshop in Warsaw

From: dirk@orka.cs.kuleuven.be. (Dirk Craeynest)
Date: Wed, 3 Apr 2019 22:19:24 -0000
Subject: DeCPS 2019 - Dependable and Cyber-Physical Systems Engineering
Newsgroups: comp.lang.ada, fr.comp.lang.ada, comp.lang.misc

Call for Papers

DeCPS 2019 - Workshop on Challenges and new Approaches for Dependable and Cyber-Physical Systems Engineering

14 June 2019, Warsaw, Poland

Co-located with the Ada-Europe 24th International Conference on Reliable Software Technologies

Conference web site: http://www.ada-europe.org/conference2019

--- Scope ---

In recent years, the Internet of Things (IoT) has experienced an extraordinary development with a broad impact on society; however, there is still a gap between the physical world and the cyber one. Cyber Physical Systems (CPS) constitute a new class of engineered systems, integrating software control and autonomous decision making with signals from an uncertain and dynamic environment. Internet transformed the way people interact and deal with information. CPS technology transformed the way people interact with engineered systems. For this type of systems, it is necessary not only ensuring the safety of physical devices but also other factors such as information about customers, suppliers, and organizational strategies need to be secured. In the context of cyber systems, the Artificial Intelligence (AI) technologies can contribute to manage a huge amount of heterogeneous data that come from different sources without human intervention. To deliver certification, standards for machine safety are highly recommended as they give confidence to the regulatory. The generic standard for safety-related hardware and software might be applicable, however, due to increasing autonomy of robots there is still a potential for evolution of such regulations or standards. The proper combination of AI, CPS and IoT is therefore fundamental.

CPS are considered a disruptive technology which will transform the traditional manufacturing into Industry 4.0 solutions, and are used in a very wide spectrum of applications: smart mobility, autonomous driving, digital healthcare, smart grids and buildings, mobile co-operating autonomous robotic systems, digital consumer products and services. "In conclusion, the emerging Digital (R)-evolution relies heavily on Embedded Intelligent Systems technologies in domains where it is paramount that Europe takes leadership role" (Laila Gide, "The pathway to digital transformation: an opportunity for Europe", ARTEMIS Magazine 20 May 2016).

This workshop aims to provide a platform to industrial practitioners, researchers and engineers in academia to exchange of their ideas, research results, experiences in the field of dependable and cyber physical systems engineering, both a theoretical and practical perspective. To foster visibility and interaction, participation in the workshop will be also open to conference participants (at no extra cost).

--- Topics of interest ---

The topics of interest includes, but are not limited to:

* 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

The workshop will also include contributions from relevant projects in the domain, such as Future Factories in the Cloud (FiC), Productive 4.0, AMASS, ENABLE-S3, SafeCOP, SCOTT, etc.

--- Paper submission ---

Submission of regular papers (4 pages, AUJ style) at the following page: https://easychair.org/conferences/?conf=decps2019

The post-workshop proceedings will be published in the Ada User Journal

(http://www.ada-europe.org/auj/guide/).

--- Important dates ---

* Submission deadline: 30 April 2019

* Notification to authors: 17 May 2019

* Workshop: 14 June 2019

* After-workshop final version: 15 September 2019

* Publication in Ada User Journal: December 2019

--- Track Chairs ---

* Faiz Ul Muram, Mälardalen Univ., Sweden

--- Steering Committee ---

* Daniela Cancila, CEA LIST, France

* Martin Torngren, KTH Royal Institute of Technology, Sweden

* Alessandra Bagnato, SOFTEAM, France

* Cristina De Luca, Infineon Technologies Austria AG Austria

* Silvia Mazzini, INTECS Italy

* Laurent Rioux, Thales, France

* Barbara Gallina, Mälardalen Univ., Sweden

* Luis Miguel Pinho, Polytechnic Institute of Porto, Portugal

Dirk.Craeynest@cs.kuleuven.be, Ada-Europe 2019 Publicity Chair

Ada-Belgium Spring 2019 Event

From: dirk@orka.cs.kuleuven.be. (Dirk Craeynest)
Date: Sun, 5 May 2019 19:35:53 -0000
Subject: Ada-Belgium Spring 2019 Event, Sun 12 May 2019
Newsgroups: comp.lang.ada, fr.comp.lang.ada, be.comp.programming

----

Ada-Belgium Spring 2019 Event

Sunday, May 12, 2019, 12:00-19:00

Wavre area, south of Brussels, Belgium

including at 15:00

2019 Ada-Belgium General Assembly

and at 16:00

Ada Round-Table Discussion

http://www.cs.kuleuven.be/~dirk/ada-belgium/events/local.html

----

*** Announcement

The next Ada-Belgium event will take place on Sunday, May 12, 2019 in the Wavre area, south of Brussels.

For the 12th year in a row, Ada-Belgium organizes their "Spring Event", which starts at noon, runs until 7pm, and includes an informal lunch, the 26th General Assembly of the organization, and a round-table discussion on Ada-related topics the participants would like to bring up.

*** Schedule

* 12:00 welcome and getting started (please be there!)

* 12:15 informal lunch

* 15:00 Ada-Belgium General Assembly

* 16:00 Ada round-table + informal discussions

* 19:00 end

*** Participation

Everyone interested (members and non-members alike) is welcome at any or all parts of this event.

For practical reasons registration is required. If you would like to attend, please send an email before Thursday, May 9, 21:00, to Dirk Craeynest <Dirk.Craeynest@cs.kuleuven.be> with the subject "Ada-Belgium Spring 2019 Event", so you can get precise directions to the place of the meeting. Even if you already responded to the preliminary announcement, please reconfirm your participation ASAP.

If you are a member but have not renewed your affiliation yet, please do so by paying the appropriate fee before the General Assembly (you have also received a printed request via normal mail). If you are interested to join Ada-Belgium, please register by filling out the 2019 membership application form [1] and by paying the appropriate fee before the General Assembly. After payment you will receive a receipt from our treasurer and you are considered a member of the organization for the year 2019 with all member benefits [2]. Early enrollment ensures you receive the full Ada-Belgium membership benefits (including the Ada-Europe indirect membership benefits package).

As mentioned at earlier occasions, we have a limited stock of documentation sets and Ada related CD-ROMs that were distributed at previous events, as well as some back issues of the Ada User Journal [3]. These will be available on a first-come first-serve basis at the General Assembly for current and new members. (Please indicate in the above-mentioned registration e-mail that you're interested, so we can bring enough copies.)

[1] http://www.cs.kuleuven.be/~dirk/ada-belgium/forms/member-form19.html

[2] http://www.cs.kuleuven.be/~dirk/ada-belgium/member-benefit.html
[3] http://www.ada-europe.org/auj/home/

*** Informal lunch

The organization will provide food and beverage to all Ada-Belgium members. Non-members who want to participate at the lunch are also welcome: they can choose to join the organization or pay the sum of 15 Euros per person to the Treasurer of the organization.

*** General Assembly

All Ada-Belgium members have a vote at the General Assembly, can add items to the agenda, and can be a candidate for a position on the Board [4]. See the separate official convocation [5] for all details.

[4] http://www.cs.kuleuven.be/~dirk/ada-belgium/board/

[5] http://www.cs.kuleuven.be/~dirk/ada-belgium/events/19/190512-abga-conv.html

*** Ada Round-Table Discussion

As in recent years, we plan to keep the technical part of the Spring event informal as well. We will have a round-table discussion on Ada-related topics the participants would like to bring up. We invite everyone to briefly mention how they are using Ada in their work or non-work environment, and/or what kind of Ada-related activities they would like to embark on. We hope this might spark some concrete ideas for new activities and collaborations.

*** Directions

To permit this more interactive and social format, the event takes place at private premises in the Wavre area, south of Brussels. As instructed above, please inform us by e-mail if you would like to attend, and we'll provide you precise directions to the place of the meeting. Obviously, the number of participants we can accommodate is not unlimited, so don't delay...

Looking forward to meet many of you!

Dirk Craeynest, President Ada-Belgium

Dirk.Craeynest@cs.kuleuven.be

Acknowledgements

We would like to thank our sponsors for their continued support of our activities: AdaCore, and KU Leuven (University of Leuven).

If you would also like to support Ada-Belgium, find out about the extra Ada-Belgium sponsorship benefits:

http://www.cs.kuleuven.be/~dirk/ada-belgium/member-benefit.html#sponsor

----

Ada-Europe 2019

From: dirk@orka.cs.kuleuven.be. (Dirk Craeynest)
Date: Thu, 9 May 2019 05:47:27 -0000
Subject: 24th Int. Conf. Reliable Software Technologies, Ada-Europe 2019
Newsgroups: comp.lang.ada, fr.comp.lang. ada, comp.lang.misc

Call for Participation

*** PROGRAM SUMMARY ***

24th International Conference on Reliable Software Technologies - Ada-Europe 2019

11-14 June 2019, Warsaw, Poland

http://www.ada-europe.org/conference2019

Organized by EDC and Ada-Europe, in cooperation with ACM SIGAda, SIGBED, SIGPLAN and the Ada Resource Association (ARA)

*** Online registration open ***

*** Early registration discount until May 20 ***

*** Extensive info available on conference web site ***

*** Highly recommended to book your hotel ASAP ***

The 24th International Conference on Reliable Software Technologies - Ada-Europe 2019 visits Poland, for the first time, and is hosted in Warsaw from the 11th to the 14th of June. The conference is the latest in a series of annual international conferences started in the early 80's, under the auspices of Ada-Europe, the international organization that promotes knowledge and use of Ada and Reliable Software in general, into academic education and research, and industrial practice.

The Ada-Europe series of conferences has over the years become a leading international forum for providers, practitioners and researchers in reliable software technologies. These events highlight the increased relevance of Ada in general and in safety- and security-critical systems in particular, and provide a unique opportunity for interaction and collaboration between academics and industrial practitioners.

Extensive information is on the conference web site, such as an overview of the program, the list of accepted papers and industrial presentations, and descriptions of workshops, tutorials, keynote presentations, and social events. Also check the conference site for registration, accommodation and travel information. The 12-page Advance Program brochure is available there as well.

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

- lower registration fee for conference, unified for all participants;
- further reduced fee for all authors;
- lower registration fee for all tutorials;
- journal-based open-access publication model for peer-reviewed papers;
- an educational tutorial offered especially for those new to Ada;
- more compact program with two core days (Wed & Thu); tutorials on Tuesday, then exhibition opening mid-afternoon, followed by welcome aperitif for all participants;
- full-day DeCPS workshop on Friday (complementary with registration).

Quick overview

- Tue 11: tutorials, opening exhibition + AE GA, welcome reception
- Wed 12 & Thu 13: core program
- Fri 14: workshop

Proceedings

- peer-reviewed papers in open-access journal
- industrial presentation and tutorial abstracts in Ada User Journal

Conference & Program Chair

- Tullio Vardanega, University of Padua, Italy tullio.vardanega at unipd.it

Keynote speakers

- Tucker Taft, AdaCore, USA, "A 2020 View of Ada"
- other keynote to be confirmed (see conference web site)

Workshop (full day)

- 6th International Workshop on "Challenges and new Approaches for Dependable and Cyber-Physical Systems Engineering" (DeCPS 2019)

Tutorials (full day)

- "Controlling I/O Devices with Ada, using the Remote I/O Protocol" Philip Munts, Sweden
- "An Introduction to Ada" Jean-Pierre Rosen, Adalog, France

Papers and Presentations

- sessions on Assurance Issues in Critical Systems, Tooling Aid for Verification, Best Practices for Critical Applications, Uses of Ada in Challenging Environments, Verification Challenges, Real-Time Systems
- 9 refereed technical papers
- 8 industrial presentations and experience reports
- a speaker's corner on "Experience from 40 years of teaching Ada"

Vendor exhibition and networking area

- area features exhibitor booths, project posters, reserved vendor tables, and general networking options
- 4 companies already committed: AdaCore, PTC Developer Tools, Rapita Systems, Vector; some exhibition slots still available
- vendor presentation sessions in core program

Social events

- each day: coffee breaks in the exhibition space and sit-down lunches offer ample time for interaction and networking
- Tuesday afternoon: opening of exhibition & Ada-Europe General Assembly, Welcome Aperitif on terrace overlooking Warsaw Airport
- Wednesday evening: transportation to restaurant in town where Chopin was born, banquet with Polish cuisine, drinks, and live piano music
- Best Paper and Best Presentation awards will be handed out

Registration

- online registration is open at <https://registration.ada-europe.org/index.html>
- early registration discount until Monday May 20, 2019
- special low fee for authors
- discount for Ada-Europe, ACM SIGAda, SIGBED and SIGPLAN members
- extra discount for students
- registration includes coffee breaks and lunches
- full conference registration includes all social events
- tutorial fees substantially reduced
- payment possible by credit card or bank transfer
- see registration page for all details

Promotion

- recommended Twitter hashtags: #AdaEurope and/or #AdaEurope2019
- 12-page Advance Program brochure online at http://www.ada-europe.org/conference2019/AE-2019%20AP.pdf
- support Ada-Europe 2019 with promotional poster at http://www.ada-europe.org/conference2019/picts/AE2019_poster.pdf

Please make sure you book accommodation as soon as possible.

For more info and latest updates see the conference web site at http://www.ada-europe.org/conference2019.

We look forward to seeing you in Warsaw in June 2019!

Our apologies if you receive multiple copies of this announcement. Please circulate widely.

Dirk Craeynest, Ada-Europe'2019 Publicity Chair

Dirk.Craeynest@cs.kuleuven.be

*** 24th Intl. Conf. on Reliable Software Technologies - Ada-Europe'2019 June 11-14, 2019 * Warsaw, Poland * www.ada-europe.org/conference2019

Ada-related Resources

Ada on Social Media

From: Alejandro R. Mosteo <amosteo@unizar.es>
Date: Thu May 23 2019
Subject: Ada on Social Media

On March 12, 2019, Maxim Reznik created an English-language Telegram chat group, called "Ada", with description "Ada Programming Language and related technologies". It can be joined at https://t.me/ada_lang

On other front, the Google+ Ada Community seems to no longer exist.

Ada groups on various social media:

- LinkedIn: 2_813 (+101) members	[1]
- Reddit: 2_243 (+343) members	[2]
- StackOverflow: 1_183 (+183) watchers	[3]
- Freenode: 87 (-17) users	[4]
- Gitter: 42 (-15) people	[5]
- Telegram: 47 (new!) users	[6]
- Twitter: 6 (-2) tweeters	[7]

[1] https://www.linkedin.com/groups/114211/

[2] http://www.reddit.com/r/ada/

[3] http://stackoverflow.com/questions/tagged/ada

[4] #Ada on irc.freenode.net

[5] https://gitter.im/ada-lang

[6] https://t.me/ada_lang

[7] https://twitter.com/search?src=typd&q=%23AdaProgramming%20since%3A2019-02-23%20until%3A2019-05-23

Repositories of Open Source Software

From: Alejandro R. Mosteo <amosteo@unizar.es>
Date: Thu May 23 2019
Subject: Repositories of Open Source software

GitHub: 603 (+90) developers	[1]
Rosetta Code: 664 (+ 9) examples	[2]
36 (+3) developers	[3]
Sourceforge: 270 (+5) projects	[4]
Open Hub: 209 (+3) projects	[5]
Bitbucket: 87 (+5) repositories	[6]
Codelabs: 46 (+1) repositories	[7]
AdaForge: 8 repositories	[8]

[1] https://github.com/search?q=language%3AAda&type=Users

[2] http://rosettacode.org/wiki/Category:Ada

[3] http://rosettacode.org/wiki/Category:Ada_User

[4] https://sourceforge.net/directory/language:ada/

[5] https://www.openhub.net/tags?names=ada

[6] https://bitbucket.org/repo/all?name=ada&language=ada

[7] http://git.codelabs.ch/

[8] http://forge.ada-ru.org/adaforge

Language popularity rankings

From: Alejandro R. Mosteo <amosteo@unizar.es>
Date: Thu May 23 2019
Subject: Ada in language popularity rankings

- TIOBE Index: 36 (0.326%)	[1]
- IEEE Spectrum (general): 46	[2]
- IEEE Spectrum (embedded): 13	[2]

[1] https://www.tiobe.com/tiobe-index/

[2] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

Ada-related Tools

Debugging Ada programs

From: "Randy Brukardt" <randy@rrsoftware.com>
Date: Tue, 2 Apr 2019 17:07:24 -0500
Subject: Re: Intervention needed?
Newsgroups: comp.lang.ada

Does anyone spend much time in a debugger when writing Ada? Almost all of the time I do it is to track down compiler bugs (hopefully something that the average Ada user doesn't do often). With the default exception information, there is little need to debug anything the majority of the time.

Certainly, moving detection to compile-time is even better. But I don't see that changing the mostly non-existent use of debuggers much.

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Date: Wed, 3 Apr 2019 09:29:20 +0200

> Does anyone spend much time in a debugger when writing Ada?

Well if there were a working one. GDB does not count.

I am using tracing, but there are few cases where debugger could be easier to use. In the debugger you could inspect the states of variables and of other tasks. And you don't need to modify the code. It is quite often that I have to add, in addition to "standard" tracing, some more extensive tracing which I remove later.

[...]

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Date: Wed, 3 Apr 2019 19:15:20 +0200

On 2019-04-03 18:16, Simon Wright wrote:

> "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de> writes:

> [...]

>> Other debuggers work, GDB does not. If you have an Ada project of a moderate size GDB stops working.

>

> How big is "moderate"?

In none of my projects GDB works. I never tried to figure out if that is related to the number of compilation units or number of library projects involved.

When you click Debug->Initialize->your-main-program in GPS and debugger does not start you know you reached the point.

From: Maciej Sobczak <see.my.homepage@gmail.com>
Date: Wed, 3 Apr 2019 22:44:03 -0700

> In none of my projects GDB works.

>

> P.S. It never worked reliable in GPS and I bet it never will.

This is very troubling. I understand the sentiment here that Ada is so good in error prevention that debuggers are not needed at all, but what I find in projects I'm related with is that debuggers are not used for debugging anyway.

The major use for debuggers that I see is in integration testing, where test procedures expect particular values in particular variables (or even exact memory locations) in particular circumstances. The test is successful if such expectations are confirmed. Even for a presumably 100% correct program such a test has to be done if foreseen by project plans.

So, we have another paradox: Ada is so good in error prevention that the community does not care about having a proper debugger, and then the lack of working debugger prevents people from choosing Ada for projects that have rigorous integration testing culture. Part of the paradox is that such projects happen to be safety-critical, where Ada is supposed to be the preferred solution. And then they use C, where debuggers work like a charm.

Again: debuggers are not only for debugging and you better get them working right (by, well... debugging them?).

From: Maciej Sobczak <see.my.homepage@gmail.com>
Date: Thu, 4 Apr 2019 22:45:17 -0700

> As gdb can be scripted, the tests that Maciej describes can probably be automated,

Yes.

> albeit with considerable effort,

Not really. I would say there is no need for this effort to be higher than with any other form of test automation. Note that as with anything else in software, recurring problems can be mitigated by additional code. That is, if testing this way is difficult, then the difficulty is similar for the whole class of similar tests and as such that difficulty can be refactored away to additional utility (library/framework/etc.) with simpler (higher-level) interface.

> especially if the scripts should be robust to evolution of the SW under test (changing the line numbers of the required breakpoints, etc.)

This is a wider problem of traceability. You have to solve this problem anyway for the coverage analysis, for example. And the solution, whatever you happen to use (like tool-readable labels in source comments), will help with debugging, too.

In any case, yes, some projects need the debugger to test individual memory locations. The lack of proper tools is a technology risk.

> However, I don't think that gdb or other current debuggers are ideal tools for automated checking of internal states.

They are not. But a non-ideal working debugger is still better than a not working one.

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Wed, 3 Apr 2019 20:23:36 +0300

On 19-04-03 01:07 , Randy Brukardt wrote:

> Does anyone spend much time in a debugger when writing Ada?

I don't. I can't remember when I last used gdb or any other debugger, and in my ~30 years of Ada use I estimate that I have used a debugger on perhaps ten occasions. I have slightly more often used "monitor" programs to examine and alter memory and register contents when analysing problems in embedded programs, and those monitor programs can perhaps be considered crude debuggers. However, these cases involved the effects and meanings of HW control registers rather than ordinary program variables.

A propos, the name "debugger" is IMO one of the unfortunate historical misnomers in the programming domain. It is a misnomer because a "debugger" like gdb should certainly not be our main tool for removing bugs from programs. Diving into the debugger as the first step of analysing a program failure is akin to starting a new project by diving into coding and skipping the design phase. Moreover, the activity of removing a bug from program, which should be the meaning of the term "debugging", should certainly not consist just of a gdb/debugger session.

[...]

From: Bill Findlay <findlaybill@blueyonder.co.uk>
Date: Wed, 03 Apr 2019 18:48:42 +0100

On 3 Apr 2019, Niklas Holsti wrote:

> On 19-04-03 01:07 , Randy Brukardt wrote:

>> Does anyone spend much time in a debugger when writing Ada?

> I don't. I can't remember when I last used gdb or any other debugger, and in my ~30 years of Ada use I estimate that I have used a debugger on perhaps ten occasions.

I can trump that.

I have *never* used a "debugger" in much the same time with Ada.

~30 years ago I raced an experienced programmer who was looking for an error in his code with the DEC Ada debugger, while I inspected his compilation listing. I won.

VisualAda 1.2.1

From: Alex Gamper <alby.gamper@gmail.com>
Date: Fri, 17 May 2019 20:26:55 -0700
Subject: ANN: VisualAda (Ada Integration for Visual Studio 2017 & 2019) release 1.2.1

Newsgroups: comp.lang.ada

Dear Ada Community

VisualAda version 1.2.1 has been released.

Fixes include the following:

- UWP DLL is linking with both GCC and MS.
- UWP XAML application project template now correctly add project dependencies .
- Install / Uninstall Ada menu items.
- Fix determining path to gdb.exe.

- Minimum supported version of Visual Studio is now 2017 Update 6 (15.0.27413).

Please feel free to download the free plugin from the following URL: https://marketplace.visualstudio.com/items?itemName=AlexGamper.VisualAda

Gnu Emacs Ada mode

From: Stephen Leake <stephen_leake@stephe-leake.org>
Date: Sat, 23 Mar 2019 10:25:34 -0700
Subject: Gnu Emacs *Ada mode 6.1.0 released.*
Newsgroups: comp.lang.ada

Gnu Emacs Ada mode 6.1.0 is now available in GNU ELPA. This is a medium feature release; partial file parsing is now supported when using the process parser, and error correction is improved. This means the time spent parsing is independent of the file size, so it is fast enough even on the largest files.

The process parser requires a manual compile step, after the normal list-packages installation:

```
cd ~/.emacs.d/elpa/ada-mode-6.1.0
./build.sh
```

This requires AdaCore gnatcoll packages which you may not have installed; see ada-mode.info Installation for help in installing them.

AdaSubst

From: "J-P. Rosen" <rosen@adalog.fr>
Date: Fri, 19 Apr 2019 08:47:50 +0200
Subject: [Ann] Adasubst 1.6r5 released
Newsgroups: comp.lang.ada

Adalog is pleased to announce the release of a new version of AdaSubst.

This releases adds a new function: Instantiate. It replaces all generic instantiations with equivalent, explicit code. This is useful if your coding standard disallows generics on the ground that it is "hidden code", or if you use a validation or testing tool that does not handle generics properly.

Adasubst can be downloaded from http://www.adalog.fr/en/components.html#adasubst

And of course, it's free software.

Enjoy!

Win32 and WinRT Bindings

From: alby.gamper@gmail.com
Date: Sat, 27 Apr 2019 21:05:21 -0700
Subject: Ann: Win32 and WinRT bindings update
Newsgroups: comp.lang.ada

Dear Ada Community

The Win32 and WinRT bindings have both been updated to the latest Microsoft SDK version (10.0.18362). This version corresponds to the 19H1 release of Windows 10.

Packages/Source can be found at

https://github.com/Alex-Gamper/Ada-Win32

https://github.com/Alex-Gamper/Ada-WinRT

Alex

Simple Components

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Date: Tue, 14 May 2019 19:05:57 +0200
Subject: ANN: Simple Components v4.40
Newsgroups: comp.lang.ada

The software 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 various protocols implementations.

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

Changes to the previous version:

- The package OpenSSL was added to provide bindings to OpenSSL;
- The package GNAT.Sockets.Server.OpenSSL was added to support secure servers based on OpenSSL;
- Multiple procedures were added to the package GNAT.Sockets.Connection_State_Machine.ELV_MAX_Cube_Client to support devices topology management and time management;
- Race condition in Object.Release fixed. The profile of the primitive operation Object.Decrement_Count has been modified.

GtkAda Contributions

From: "Dmitry A. Kazakov" <mailbox@dmitry-kazakov.de>
Date: Tue, 14 May 2019 19:08:07 +0200
Subject: ANN: GtkAda Contributions v3.24

The software extends GtkAda 3.14.15, an Ada bindings to GTK+. It deals with the following issues:

- Tasking support;
- Custom models for tree view widget;
- Custom cell renderers for tree view widget;
- Multi-columned derived model;
- Extension derived model (to add columns to an existing model);
- Abstract caching model for directory-like data;
- Tree view and list view widgets for navigational browsing of abstract caching models;
- File system navigation widgets with wildcard filtering;
- Resource styles;
- Capturing resources of a widget;
- Embeddable images;
- Some missing subprograms and bug fixes;
- Measurement unit selection widget and dialogs;
- Improved hue-luminance-saturation color model;
- Simplified image buttons and buttons customizable by style properties;
- Controlled Ada types for GTK+ strong and weak references;
- Simplified means to create lists of strings;
- Spawning processes synchronously and asynchronously with pipes;
- Capturing asynchronous process standard I/O by Ada tasks and by text buffers;
- Source view widget support;
- SVG images support.

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

Changes to the previous version:

- The package GLib.Time_Zone was added

GCC 9.1.0 for MacOS

From: Simon Wright <simon@pushface.org>
Date: Wed, 08 May 2019 20:00:57 +0100
Subject: ANN: GCC 9.1.0 for MacOS
Newsgroups: comp.lang.ada

GCC 9.1.0 for Mac OS X El Capitan (10.11) is available at https://sourceforge.net/projects/gnuada/files/GNAT_GCC%20Mac%20OS%20X/9.1.0/

Also runs on macOS Mojave (10.14) and (untested) on Sierra (10.12) and High Sierra (10.13).

```
********************************
* DO NOT USE ON EARLIER
VERSIONS OF OS X *
*******************************
```

The native/ directory contains the 9.1.0 x86_64-apple-darwin15 compiler, together with tools from GNAT CE 2018 and various Github and other repositories.

The arm-eabi/ directory contains the 9.1.0 arm-eabi Darwin-hosted cross compiler.GNAT Community 2019

From: Alejandro R. Mosteo <amosteo@unizar.es>
Date: Thu May 30 13:58:51 CEST 2019
Subject: GNAT Community 2019 released

As seen in several social media sources, the new Community edition of GNAT as arrived and is available for download at:

https://www.adacore.com/download

From the release announce at [1] by Nicolas Setton:

We are pleased to announce that GNAT Community 2019 has been released! See https://www.adacore.com/download.

This release is supported on the same platforms as last year:

- Windows, Linux, and Mac 64-bit native
- RISC-V hosted on Linux
- ARM 32 bits hosted on 64-bit Linux, Mac, and Windows

GNAT Community now includes a number of fixes and enhancements, most notably:

- The SPARK language now has support for pointers, a fantastic milestone for the language! See https://blog.adacore.com/using-pointers-in-spark for more information about this new feature.
- The installer for Windows and Linux now contains pre-built binary distributions of Libadalang, a very powerful language tooling library for Ada and SPARK.

Check out the README for some additional platform-specific notes.

We hope you enjoy using SPARK and Ada!

[1] https://blog.adacore.com/gnat-community-2019-is-here

Componolit Ada Runtime 1.0.0

From: u/marc-kd
Date: Tue May 28 2019 14:41:16 GMT+0200 (CEST)
Subject: Componolit Ada Runtime 1.0.0
Newsgroups: reddit:/r/ada/ [1]

https://github.com/Componolit/ada-runtime/releases/tag/v1.0.0

[News Editor - From the above link:]

Generic Ada Runtime - A downsized Ada runtime which can be adapted to different platforms.

The Componolit Ada Runtime 1.0.0 builds upon GCC 8.3 and is compatible with GNAT Community 2019. It provides the following runtime features:

- Interfaces (C)
- Secondary stack
- Exception raising
- 64bit arithmetics
- Unchecked conversion

The following features are DEPRECATED and will be removed in future releases:

- GNAT IO

Parts of the runtime are proven to have no runtime errors:

- Secondary stack allocator
- String handling

Supported platforms:

- Genode
- Linux

[1] https://www.reddit.com/r/ada/comments/btzk99/componolit_ada_runtime_100/

Ada Inside

Boeing 737 MAX Software

From: Paul Rubin <no.email@nospam.invalid>
Date: Fri, 05 Apr 2019 14:16:20 -0700
Subject: Boeing 737 and 737 MAX software
Newsgroups: comp.lang.ada

Does anyone know anything about this? It has been under some criticism lately.

I have heard that the 777 software was almost entirely in Ada. It also sounds as if Boeing's software operation may have slipped in recent years, not good news for the 737 MAX.

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sat, 6 Apr 2019 21:45:24 +0300

[...]

As I've read more about these accidents than I usually do, I will boldly (and perhaps foolhardily) describe how I have understood it. All info is from public sources, I have no insider info. I am not a pilot, and moreover I write from recollection of my reading and have no references to give, so reader beware.

> On 19-04-06 20:30 , Dennis Lee Bieber wrote:
>
> Unless things have changed severely -- GE Aviation (formerly Smith's Aerospace, formerly Lear Siegler) produces the 737 FMS software (and also the processor boxes).
>
> However, I have the impression (from TV news) the software is functioning /as designed/.

All info I have seen agrees with that.

> Some reports have indicated that Boeing designed the hardware (and corresponding software requirements) such that only one sensor is used for the MCAS subsystem

There are two angle-of-attack (AoA) sensors, one on each side of the nose. They feed two redundant computers, each able to run MCAS. Normally only one MCAS instance is running and it uses only its "own" AoA sensor.

The original design of MCAS gave it rather little control authority, which is probably why this single-sensor approach was accepted.

> -- and a fault in that sensor results in MCAS attempting to prevent a (non) stall by pushing the nose down.

Yes, but MCAS does not apply a temporary nose-down command -- as if pushing the stick forward -- it changes the pitch trim, the overall angle of the horizontal stabilizer, giving the plane a permanent tendency to dive. This trim change can be overridden by the pilots, but only if they notice that it has happened.

In the original MCAS design, one activation of MCAS changed the pitch trim by a small amount, at most 0.6 degrees IIRC, and this limit was reported in the MCAS design documentation to the authorities. During testing, Boeing found that it was not enough, and they increased it quite a lot, to over 2 degrees IIRC. One source I read claimed that this change was _not_ updated in the documentation shown to the authorities.

Moreover, by design MCAS would repeat this trim change, with a certain minimum interval, as long as the AoA sensor reading remained too large and indicated a risk of stall. This iteration should converge and stop if the sensor is working, but if the sensor fails and is stuck at a high AoA (the false value reported in the second accident was around 60 degrees, IIRC) then MCAS will incrementally and cumulatively keep increasing the pitch trim and the diving tendency. If the pilots do not understand what is happening, they will find it ever harder to counteract the "dive" trim with stick inputs.

> Some hints in the news that Boeing is changing the requirements (well, in truth, the news only says Boeing is changing the software) to have MCAS cross-reference with other flight parameter data -- and making an optional bit of hardware (additional sensors) standard.

AIUI the modified MCAS will read both AoA sensors and will disable itself if they disagree, and the disagreement will also be reported by a display. This display is the new piece of HW which used to be an option. There are no new sensors, AIUI.

I believe Boeing are also changing the minimum interval between MCAS activations -- perhaps even allowing only one activation -- so as to prevent a cumulatively increasing "dive" trim.

In summary, it seems to me that the criticality of MCAS, and thus the need for redundant sensors, was not realized for two reasons: (1) in its initial design, MCAS command authority was small, and (2) the possibility of multiple repeated commands (due to a stuck sensor) and the resulting large cumulative command (large change of pitch trim) was not considered.

A kind of "criticality creep".

From: Dennis Lee Bieber <wlfraed@ix.netcom.com>
Date: Fri, 12 Apr 2019 18:15:07 -0400

On Fri, 12 Apr 2019 00:46:31 -0700 , tranngocduong@gmail.com declaimed the following:

> I know nothing about the software. But I don't think it is written in Ada. If it was, programmers must have chosen a wrong subtype.

It's Ada... (In the past, I was doing maintenance on the FMS "BootROM" code -- which, while not the actual run-time flight software, is responsible for doing CRC checks of the software and databases, reading new software from dataloaders, and loading which application is to run based upon external settings. The FMS software links with the same base "OS".

[...]

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Thu, 18 Apr 2019 21:20:19 +0300

On 19-04-18 19:21 , tranngocduong@gmail.com wrote:

[...]

> To my limited knowledge, AoA is a critical parameter that is used by many flight control algorithms, not just the MCAS. The real issue is thus the failure to detect an unreliable sensor. If the failure was a "feature, not bug", the entire flight software (and its certificate) would be questioned.

I've read rumours that even if the U.S. FAA lets the fixed 737 MAX fly again, other air safety authorities (Europe, for example) might not be satisfied, for that very reason -- suspicion that the flight software process was at fault.

From more recent descriptions of the two crashes, it seems that the problem also involves complex interaction between MCAS, the enabling or disabling of the elevation trim motors, restarts of the control computer, and the fact that manual correction of the elevation trim becomes impossibly hard when the MCAS-commanded large "dive" trim applies large aerodynamic forces to the trim mechanism. Thus the problem was not only in the software process, but also in the controllability of the aircraft under anomalies -- a chain of failures, as typical for accidents.

> [...] Would Boeing as a company risk its very existence by committing such a big mistake? I don't think so.

The suspicion involves Boeing sliding down two slippery slopes, as I understand it:

1) For MCAS in particular, its control authority was greatly increased from its first design to the flying version, but this was not propagated into a new consideration of its criticality.

2) For the process in general, an increasing complacency ("we know how to do it") and increasing delegation of checks from the FAA to Boeing (and other airplane builders), combined with specific driving forces for 737 MAX (urgency + desire to avoid pilot retraining).

I am reminded of the Space Shuttle O-rings... and perhaps also of the scandals with automotive SW hiding emissions, leading to multi-billion losses for the guilty European companies...

From: Paul Rubin <no.email@nospam.invalid>
Date: Thu, 18 Apr 2019 13:20:29 -0700

Niklas Holsti <niklas.holsti@tidorum.invalid> writes:

> On the issue of Ada subtypes, it seems to me that if the SW specification, design and coding considers sensor faults (as it of course should), the normal approach for such critical SW

One of the criticisms of the decisions leading to the MCAS software is that the software is certified only at DO-178B level C, defined as software whose consequences are (https://en.wikipedia.org/wiki/DO-178B):

Major – Failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries) or significantly increases crew workload (safety related)

This is instead of level A (catastrophic, the whole plane can be lost), or level B (hazardous, people can be injured). The rationale was that at worst MCAS going wrong would change the nose pitch by a few degrees and then the pilot could fix it. They didn't consider the possibility of it activating over and over again, tilting a few more degrees each time.

Since the software was treated as level C, its development and certification process was less rigorous than what it would have gotten at a more critical level.

Certifying and developing this system at level C instead of level A was itself obviously some kind of process failure. I believe finding out how that happened is one of the investigation's objectives.

Ada and other Languages

Pointer Ownership, Containers and Cursors in Ada, Rust, SPARK

From: Randy Brukardt <randy@rrsoftware.com>
Date: Tue, 12 Mar 2019 16:53:01 -0500
Subject: Re: Intervention needed?
Newsgroups: comp.lang.ada

[...]

My understanding is that the SPARK people are far into designing ownership contracts for Spark.

It's also possible that Ada 2020 will have a form of pointer ownership. (Unfortunately, we didn't make any conclusions on that during yesterday's meeting, so it's still in limbo, and we're getting very close to the finish line.) The current problem is that in Ada 2020 as it stands, it's not possible to write a containers implementation in pure Ada. You'd have to have some implementation hack to turn off some of the Legality Rules. Tucker has designed a solution, based on an ownership mechanism, but as it is new and barely vetted, it's unclear what we will do with it ultimately. Note that this solution will not provide the perfect safety that you would get with SPARK or Rust, but it would form the foundation of the SPARK solution and it would clearly catch a lot of issues with using pointers to implement ADTs. (And there is little other reason to use pointers, IMHO.)

From: Randy Brukardt <randy@rrsoftware.com>
Date: Mon, 18 Mar 2019 18:36:15 -0500

[...]

But a better question is whether the Rust borrow checker allows building proper ADTs for most data structures. Most of Tucker's proposals didn't have a safe way to build typical data structures like a doubly-linked list or the parent pointer of a tree structure. Leaving out these backward pointers means adding a substantial performance degradation for (possibly) common operations like node deletion. Depending on what you're doing, that could be a non-starter. I haven't had a chance to actually look at Rust's actual rules; Ada is hard enough and as we're in the home stretch for Ada 2020, I literally don't have time for much else. (Probably shouldn't be answering this message...)

Tucker's latest proposal does address the back pointer problem. So at least that can be done with checks.

[...]

From: Randy Brukardt <randy@rrsoftware.com>
Date: Tue, 19 Mar 2019 18:01:19 -0500

[...]

Some background here: The basic idea behind pointer ownership is to prevent various issues by enforcing an invariant -- that each allocated object is stored in exactly one pointer object. This is enforced with a variety of runtime and compile-time rules.

Now, it's clear that one can't even walk a data structure that way, so the idea of "borrowing" a pointer for a limited time was invented. Such borrowing has to be done in carefully controlled ways in order to keep it being safe -- for instance, no one can read or write the original pointer while it is borrowed.

Multiple long-lived pointers that point at a single object are simply not allowed. In part, that's done by making assignment either illegal or a move (where the source is nulled when the pointer is assigned).

For something like a cursor, that means that Rust-pointers couldn't be used to create the object. The entire point of a cursor is that it is a long-lived handle to a specific element in a larger data structure. One can't null out part of the data structure to create the handle, and if the assignment is banned completely, you could never create a valid cursor object in the first place.

There are similar issues with back pointers in a data structure, as you might guess.

From: Randy Brukardt <randy@rrsoftware.com>
Date: Tue, 19 Mar 2019 18:13:45 -0500

[...]

A cursor is a handle accessing an element of a larger container, nothing more or less than that. The primary usage is to connect data structures made up of multiple containers. For instance, consider a compiler symbol table. There is a tree structure that represents each of the declarations and their scopes, and a map structure that represents a mapping of names to nodes of the tree. The contents of that map is going to be tree cursors, each representing a declaration with a particular name.

[...]

The value of cursors is that they can be implemented by a range of abstractions with a range of checking, from array indices (as in the bounded containers and the vector) to pointers with a variety of schemes from no dangling checking to the bulletproof controlled cursor scheme.

[...]

From: Jere <jhb.chat@gmail.com>
Date: Fri, 22 Mar 2019 03:54:03 -0700

[...]

Not really. In Rust they don't even use cursors. They go straight to iterators. On top of that, the combination of being able to specify the lifetime of all of your variables (if you need to) and the ownership rules gives them 100% safety from dangling references when they create and use those iterators.

[...]

From: Randy Brukardt <randy@rrsoftware.com>
Date: Sat, 23 Mar 2019 02:53:32 -0500

If you don't store any cursors and just use iterators in Ada, you have the same level of safety: the tampering checks prevent any problems with iterators. (Well, unless you turn them off, of course, but if you remove the seatbelts, you can't much complain that they didn't protect you.)

I'd be interested to find out how Rust deals with the need to designate individual elements of a container (which is the primary reason that Ada exposes cursors).

From: Jere <jhb.chat@gmail.com>
Date: Sat, 23 Mar 2019 06:59:35 -0700

Keep in mind the rust paradigm is very different than Ada's. When you obtain an iterator of a container, you borrow ownership of the container. At that point it is impossible to tamper with the container because it is a compile time error to modify the container when something else has ownership of it. It's a compile time version of Ada's tampering checks using an ownership model. If you need to remove items as you iterate, there is a consuming version of the iterator that allows for that. It handles all the logic of keeping the iterator correct as you remove items. If you need to remove only specific items, it provides functions for that as well (but you would not iterate while using them do to the ownership/borrowing rules).

Additionally, Rust allows you to specify the lifetime of the iterator, the lifetime of the reference to the item, and how they relate to the lifetime of the container so that the compiler can guarantee that nothing dangles (it's a relative specification..container has lifetime A and everything else has either A or a lifetime relative to A).

If you were instead referring to indexing the container, for things like vector, Rust looks to see if the container implements the Index trait, and, if so, allows for the user to use the index operation on the container. It's very similar to making a cursor and setting the variable_indexing aspect and constant_indexing aspect, except rust doesn't expose the underlying equivalent cursor type. So you either work with Indexes or Iterators depending on your need.

From: Jere <jhb.chat@gmail.com>
Date: Tue, 19 Mar 2019 18:20:54 -0700

[...]

Rust has a couple levels of ownership and borrowing. It employs both spacial and temporal ownership rules. The 90% solution uses spacial rules (the ones you are most likely familiar with). For managing data across threads or doing complex data types, Rust also provides temporal ownership/borrowing. Think of the same distinction Ada has for named access types vs anonymous access types. You can catch a lot more at compile time with named access types, but anonymous access types can potentially have more runtime checking. In Rust, the standard reference scheme you are probably familiar with employ spacial ownership. If you need temporal ownership, then for single threaded you use things like the RC<> generic (reference counted), and for multi-threaded you use things like the ARC<> generic (atomic reference counted). There are other things of similar nature. These employ temporal rules, which can potentially require runtime checks, though the presence of the spacial ownership rules can help optimize out a lot of the run time checks. I kind of mentioned this above, but the spacial and temporal rules aren't mutually exclusive. They can work together when needed.

[...]

From: Olivier Henley <olivier.henley@gmail.com>
Date: Wed, 13 Mar 2019 06:44:05 -0700

On Wednesday, March 13, 2019 at 5:10:31 AM UTC-4, Maciej Sobczak wrote:

> So, seriously - what's wrong with pointers in Rust?

From that excerpt [1] by Oliver Scherer (Rust compiler contributor), it looks like the ownership aspect that comes with them is a real improvement:

"The two (obviously not a good amount of datapoints) large scale refactorings in Ada software that I've been part of have resulted in horrible hacks where people just spammed protected and pragma everywhere to get stuff working and bug free. The protected injections are because it's nearly impossible to figure out which things are accessed by multiple tasks without SPARK and you end up with undefined behaviour if you accidentally have a shared access to an unprotected memory location. The pragmas were reconfiguring things like stack size or disabling compiler warnings without actually thinking about what these changes meant.

Refactorings in Rust on the other hand are (compile-time) guaranteed to be free of race conditions, no matter how crazy you move stuff around or create new parallelism. Additionally the ownership concept lead to many libraries typestate encoding their API which makes misusing them a near impossibility (at compile-time) while Ada mainly catches those misuses at runtime via exceptions."

[...]

[1] "Why Rust was the best thing that could have happened to Ada". https://www.reddit.com/r/ada/comments/7wzrqi/why_rust_was_the_best_thing_that_could_have/

From: "Randy Brukardt" <randy@rrsoftware.com>
Date: Thu, 14 Mar 2019 17:41:12 -0500

Obviously, if your existing code isn't documented properly as to what needs to be task-safe, then refactoring it isn't going to work very well. Refactoring bad code is just going to give you bad code. :-) And almost all code in any language is bad code, because at some point people turned to "just make it work" mode, and stopped doing the things necessary for the code to be understandable. Using Ada helps, but surely doesn't eliminate this point.

In any case, Ada 2020 is very much about addressing this point. The new Nonblocking and Global contracts make is possible to declare tasking and memory side-effects, and the "conflict check policies" allow using that to prevent data races. (Note that there is a difference between a "data race", and "race conditions"; there are plenty of race conditions that aren't data races, and no programming language can statically prevent the latter, since they're caused of a sequence of operations. Well, other than not having any task interactions in the first place. :-)

In addition, conflict checks are enabled by default on the new parallel constructs, so you have to work at causing problems. (The parallel constructs are safer anyway, since they do not allow blocking, so there aren't any rendezvous and entry calls to worry about.) And they can be enabled on tasks as well (not done by default for the obvious reason of compatibility - but also for capability, tasks should mainly be used in Ada 2020 when one needs rendezvous and other constructs that can't be checked at compile-time).

The issue with this is that a dereference of an access value is almost always going to cause a conflict and thus be illegal. And the contracts for the containers are designed so that they can be used in parallel operations (presuming the actual parameters to the instance allow that). This means that no access types can be used to implement the containers, which is nonsense for the unbounded and indefinite containers. The ownership stuff is a proposal to limit that in the case of building ADTs, including the containers.

From: Olivier Henley <olivier.henley@gmail.com>
Date: Wed, 13 Mar 2019 06:23:59 -0700

Thanks to those who brought 'material' to the discussion.

a. The Rust thread is now closed and we did not slide into a flame war. Very good.

b. We definitely enlightened a whole bunch. You have no idea how many Rustaceans do not even know Ada exists. After all, awareness and politics are important. Very good.

c. From Randy's post, I find it exciting to see that this 'episode' is of actuality regarding Ada202X. Very good.

Thx

Ada Practice

Interviews to Ada Practitioners

From: Alejandro R. Mosteo <amosteo@unizar.es>
Date: Fri, 24 May 2019
Subject: Interviews to Ada enthusiasts

Tomek Wałkuski <tomek.walkuski@gmail.com>, co-founder at 98elements [5], is running a series of interviews [1] to people from the Ada community. Here is a list with a few extracted words from each interview since the last AUJ Issue:

- Fabien Chouteau interview [2]: "My name is Fabien Chouteau, I am embedded software engineer at AdaCore, hobbyist in electronics, instrument making and woodworking. [...] A couple years ago I started the Ada Drivers Library project, at first it was just a way to have fun with an ARM Cortex-M micro-controller board and see how Ada can be used on such hardware. It became a one stop shop for getting started in embedded Ada programming and sparked many other projects [...]"
- Edward Fish interview [3]: "I was introduced to Ada in one single class, Programming Languages, which did a high-level introduction/survey of various languages and instantly felt at-home. It did raise the question as to why a lot of the features aren't common in more languages [...]"
- Stéphane Carrez interview [4]: "Later I created another computer board based on 68HC11 and to use it I also did a complete port of the GNU compiler, the GNU binutils and the GNU debugger. My work was integrated in the FSF sources in 2000. The GNAT Ada compiler was working! I was able to run a small Ada program that fit in less than 256 bytes!"

[1] https://tomekw.com/tag/interview/

[2] https://tomekw.com/ada-programmers-fabien-chouteau/

[3] https://tomekw.com/ada-programmers-edward-fish/

[4] https://tomekw.com/ada-programmers-stephane-carrez/

[5] https://98elements.com/

Integer type with gaps

From: mario.blunk.gplus@gmail.com
Date: Fri, 29 Mar 2019 09:10:40 -0700
Subject: type definition for an integer with discrete range
Newsgroups: comp.lang.ada

Hello,

I'm looking for a way to define a type that runs from let say -100 to +100 with gaps of 5 width. Important is to make sure that a value like 7 cannot be assigned to the type.

something like:

```
type number is new integer range
      -100 .. 100;

-- or

subtype number is integer range
      -100 .. 100;

-- with this special thing or something like
-- that:

for number'small use 5;   -- cannot applied
-- here. works with fixed point types only
```

Thanks !

From: Simon Wright <simon@pushface.org>
Date: Fri, 29 Mar 2019 21:24:53 +0000

[...]

What about this?

```
pragma Assertion_Policy (Check);
with Ada.Text_Io; use Ada.Text_Io;
procedure Type_Integer is
   subtype Number is Integer range
         -100 .. 100
   with Dynamic_Predicate => Number
         mod 5 = 0;
   V : Number;
begin
   V := 0;
   Put_Line ("0'image is " & V'Image);
   V := -50;
   Put_Line ("-50'image is " & V'Image);
   V := 42;
   Put_Line ("42'image is " & V'Image);
end Type_Integer;
```

Executing gives

```
$ ./type_integer
0'image is  0
-50'image is -50
raised SYSTEM.ASSERTIONS.
ASSERT_FAILURE : Dynamic_Predicate
failed at type_integer.adb:12
```

Gauss Error Function in Ada

From: leov@gammawizard.com
Date: Mon, 1 Apr 2019 09:28:58 -0700
Subject: Erfc() function in ADA
Newsgroups: comp.lang.ada

Greetings, I have been looking into reimplementing a collection of numerical heavy code from R/C++ into ADA and so far things seem doable. My only question is about the support for the error function and in particular the complementary error function erfc(). I assume this is library dependent so I would appreciate any information if erfc() is part of the ADA standard library or perhaps provided by GNAT in some form?

From: gautier_niouzes@hotmail.com
Date: Mon, 1 Apr 2019 10:01:07 -0700

You can get easily the error function from the Phi function which is available in the following library: http://mathpaqs.sourceforge.net/

From: gautier_niouzes@hotmail.com
Date: Tue, 2 Apr 2019 03:39:57 -0700

A few random remarks...

1) For further references: there is now in Mathpaqs (rev. 153+) a separate Erf_function package. Since Phi_function.Phi uses Erf(x) anyway, it's better to have access to Erf directly.

2) About the Numerical Recipies: be careful, some versions support only 7-8 digits (single precision), so numerical errors cumulate very quickly.

3) Some good stuff can be found in the Alglib and Cephes libraries, in C, Fortran or Pascal

4) Simple special functions (with one parameter) could well be in an official Ada.Numerics.Generic_Special_Functions (low maintenance effort for compiler vendors)

5) Don't forget to check: https://www.adaic.org/ada-resources/tools-libraries/

6) Perhaps the Alire system has some math packages?

Porting GNAT bare-board runtime to a new target

From: Daniel Way <p.waydan@gmail.com>
Date: Sun, 7 Apr 2019 19:13:07 -0700
Subject: Understanding GNAT Bare Board Run-time for Cortex-M
Newsgroups: comp.lang.ada

I'm trying to port the bare-board GNAT run-time to a Coretex-M0+ (NXP KV11Z7) processor. I'm new to concurrency and have been reading through the run-times for the STM32 targets to understand how the tasks and protected objects are implemented, however, there seems to be a web of dependencies between the different packages and wrappers of wrappers of wrappers for types and subprograms.

* Is there any tool available to scan through the source code and generate a graphical call graph to help visualize the different dependencies?

* Has anyone on the forum successfully ported a bare-board run-time? What was your experience and do you have any tips?

* Is porting the run-time just a matter of updating the linker, a few packages, and a GPR script, or is there some fundamental implementation changes to consider?

Thank you,

Daniel

From: Simon Wright <simon@pushface.org>
Date: Mon, 08 Apr 2019 08:36:59 +0100

Daniel Way <p.waydan@gmail.com> writes:

> I'm trying to port the bare-board GNAT run-time to a Coretex-M0+ (NXPKV11Z7) processor. I'm new to concurrency and have been reading through the run-times for the STM32 targets to understand how the tasks and protected objects are implemented, however, there seems to be a web of dependencies between the different packages and wrappers of wrappers of wrappers for types and subprograms.

Yes.

> * Is there any tool available to scan through the source code and generate a graphical call graph to help visualize the different dependencies?

Pass.

> * Has anyone on the forum successfully ported a bare-board run-time? What was your experience and do you have any tips?

AdaCore have published a guide for porting their runtime[0].

GNAT CE 2018 includes a ravenscar-sfp-microbit runtime.

My Cortex GNAT RTS[1] is based on FreeRTOS[2] and includes an RTS for the nRF51 as found in the BBC micro:bit. That's a cortex-m0, but as far as I can see [3] the differences from the m0+ are minimal.

The main issue I had was with the clock; the nRF51 doesn't have a system tick, instead I had to use RTC1 (I think AdaCore used RTC0).

> * Is porting the run-time just a matter of updating the linker, a few packages, and a GPR script, or is there some fundamental implementation changes to consider?

That would be it (also the runtime.xml file) but the problem is identifying _which_ packages to change! I wouldn't expect many from the microbit RTS, it's likely to be clock setup and interrupt naming. It would help if you had an SVD to generate the board peripheral dependencies.

[0] https://github.com/AdaCore/bb-runtimes/tree/community-2018/doc/porting_runtime_for_cortex_m

[1] https://github.com/simonjwright/cortex-gnat-rts

[2] https://www.freertos.org

[3] https://community.cypress.com/docs/DOC-10652

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Mon, 8 Apr 2019 10:46:56 +0300

On 19-04-08 05:13 , Daniel Way wrote:

[...]

> * Is there any tool available to scan through the source code and generate a graphical call graph to help visualize the different dependencies?

I know of no free tool that generates graphical call-graphs. I've used the non-graphical call tree information from GPS.

> * Has anyone on the forum successfully ported a bare-board run-time? What was your experience and do you have any tips?

In my last project, I ported the small-footprint Ravenscar run-time for the SPARC architecture from the generic off-the-shelf AdaCore version to a specific SPARC LEON2 processor embedded in an SoC for processing satellite navigation signals, the AGGA-4 SoC.

My advice is to first understand the differences between the original target processor and the new target processor, especially in these areas:

- Basic processor architecture, and especially if there is some difference in the instruction set or in the sets of registers that must be saved and restored in a task switch. In my case there was no difference, so I did not have to modify the task-switch code nor the Task Control Block structure. For porting across various models of the same processor architecture, perhaps the most likely difference is in the presence or absence of a floating-point unit and dedicated floating-point registers.

- The HW timers. In my case the RTS used two HW timers, and there were some differences: the bit-width was different (32 instead of 24) and the HW addresses and interrupt numbers were different. The corresponding parts of the RTS had to be adapted, but in my case the changes were small, and the logic of the code did not change.

- Interrupts and traps. Differences may have to be implemented in the assembly-language code that initially handles interrupts and traps. In my case, the architecture was the same (the structure of the trap table and most of the HW error traps) but the set of external interrupt traps was different, because of the particular I/O devices available on the new target. This difference (if any) becomes visible to application programs through Ada.Interrupts.
- "Console" I/O, usually some form of UART accessible via GNAT.Text_IO. In my case, the UARTs in the new target were quite different from the standard LEON2 UARTs, so I had to reimplement the low-level I/O operations (Put character, Put string, etc.).
- Memory layout. Where in the address space is the ROM (or flash), where is the RAM, where are the I/O control registers? Any differences in the layout must be implemented in the linker command script, which in my case was a file called leon.ld. The Ada RTS code probably does not have to change for this reason, and did not change in my case.

Once all that is sorted out, you will probably have to modify the start-up assembly-language code, which in my case was in the file crt0.S. This deals with HW initialization (clearing registers, stopping any I/O that might be running, disabling interrupts, etc.) and SW initialization, which means to set up the stack for the environment task and then enter the body of that task.

> * Is porting the run-time just a matter of updating the linker, a few packages, and a GPR script, or is there some fundamental implementation changes to consider?

If you are porting from one implementation of the same architecture to another (in your case ARM Cortex M<n> with the Thumb-1/2 instruction sets, if I understand right), IMO it is unlikely that any fundamental changes are required. However, if there are differences in the instruction set (with M0+ omitting some instructions available larger members and perhaps used in the original RTS) be sure to use the correct target options for the compiler so as to avoid generating code that will not run on the M0+. If there is a major difference in instruction sets (say, porting from Thumb-2 to Thumb-1) you will have to review and perhaps modify all the assembly-language RTS parts, and all assembly-language code insertions in the Ada RTS code, and all the code in crt0.S.

HTH. I think others on this group have more experience with ARM Cortex run-time systems and can probably offer better advice.

Heart of Darkness

From: "J-P. Rosen" <rosen@adalog.fr>
Date: Sat, 20 Apr 2019 17:58:48 +0200
Subject: Re: Anonymous Access and Accessibility Levels
Newsgroups: comp.lang.ada

Le 20/04/2019 à 17:29, Jere a écrit :

> I was trying to get a bit better at understanding how accessibility levels work with respect to anonymous access types. I have GNAT to test out things, but I think I am running into various bugs, so I am not seeing the exceptions or compilation errors I would expect. It could also be that I misunderstand the rules (They are difficult somewhat).

In my tutorial about memory management, I explain that there are 34 special cases in 3.10.2 (AKA "heart of darkness"). Enter at your own risk.

From: "Randy Brukardt" <randy@rrsoftware.com>
Date: Wed, 24 Apr 2019 18:27:52 -0500

[...]

I suspect that accessibility implemented by compilers is essentially whatever the ACATS tests require. I know that I've never spent time on it in Janus/Ada beyond that -- it simply isn't worth self-inflicted pain. Thus, my advice is that accessibility works like one would expect in basic cases, and do not go beyond basic cases unless you like pain.

From: "Randy Brukardt" <randy@rrsoftware.com>
Date: Mon, 22 Apr 2019 17:11:19 -0500

[...] there are special rules for allocators, for objects created as return objects, and many other special cases. [...]

As always, I suggest the following rules:

(1) Do not use anonymous access types unless you absolutely need one of the special capabilities that can only be done with them.

(2) Under no circumstances, do anything that cannot be checked statically. (So no one should use dynamic accessibility checks of anonymous access parameters or SAOAATs [Editor's note: Stand-Alone Object of an Anonymous Access Type]).

(3) Think three times before depending upon access parameter dispatching and anonymous access-to-subprograms.

(A) If you find that you really need these things, complain to the ARG that you should be able to but cannot do these things with named access types. (This limitation is idiotic, as it requires repeating long declarations at every usage.) [I need help getting this fixed!!]

(4) Keep access types out of visible specifications (since they make memory management much harder, and locks in clients to suboptimal memory management).

From: "Randy Brukardt" <randy@rrsoftware.com>
Date: Wed, 24 Apr 2019 18:21:57 -0500

[...]

In any case, I don't believe that dynamic accessibility checking buys anything at all. Indeed, 98% of my code has to resort to 'Unchecked_Access in order to be compilable at all. I generally wrap the uses in a controlled type that cleans up the accesses as needed (that's how Claw works, for instance). The dynamic checks are mainly a hazard to be avoided rather than anything helpful. (Unlike a static check, it's hard to prove that a dynamic check can't fail, so it remains as a hazard for a future call added in maintenance.)

Licensing woes

[Often, when Ada compilers are discussed, the licensing model of GNAT arises in conversation. What follows discusses the limitations imposed by the pure GPLv3 license of the GNAT runtime in Community editions —News Editor.]

From: Maciej Sobczak <see.my.homepage@gmail.com>
Date: Mon, 27 May 2019 23:43:06 -0700
Subject: Re: Needed - Ada 2012 Compiler.
Newsgroups: comp.lang.ada

On Tuesday, May 28, 2019 at 1:25:03 AM UTC+2, Optikos wrote:

> Hence why Alex was correctly indicating that GPL Community Edition forestalls most practical forms of commercial business activity

I have an impression that nowadays "most forms of commercial business activity" involve setting up an account for accessing whatever on-line service.

This is why most apps today are free, anyway. In this context, GPL license on the app is not a problem at all.

No, I do not applaud the GPL licensing. I only state that the landscape of "commercial business activity" has significantly changed from what it was say two decades ago.

I also think that you are overestimating the willingness of customers to engage in further business activity of reproducing and re-selling what they have bought from you. This concept is being demonized since ever, but I don't think it has any bigger significance than a "traditional" counterfeiting.

No, I don't applaud GPL as a licensing scheme. I just don't consider it to be a showstopper.

> by entirely prohibiting AdaCore-esque dual licensing

Wrong. You can write your program (or a library) and sell it in the form of source code with whatever license you wish and allow your customer to compile it using whatever compiler they have. The compiler that you have used to verify (!) your product has no impact on the licensing of your source code. Thus, dual- or closed- licensing is still possible. Feel free to complete this scheme with any kind of NDA or other forms of legal agreements with your customers.From: Maciej Sobczak <see.my.homepage@gmail.com>

Date: Tue, 28 May 2019 22:54:03 -0700

[...] let's go back a little to better understand the workflow.

1. You write some code. It can be a standalone app or a library.

2. You can put whatever license you wish on your source code.

3. You can deliver it (the source code!) to your users with that license.

Finished.

OK, so you think it might be a good idea to verify this code a little bit before selling it to your customers - you know, test it or at least check whether it compiles at all. So you add an additional points to the scheme above:

1a. You compile your code with whatever compiler you have.

1b. You run your tests or perform whatever other verification activities to make sure that your product has an expected quality level.

These two points have no impact on points 2. and 3. above.

I will agree that this scheme is not satisfactory for the case of applications distributed via App Stores, or for users who don't want to be involved in technical activities like compiling something on their own - this is understandable, and in such cases a turn-key product needs to be delivered. But it is a very satisfactory scheme for the case of libraries, which become included in this kind of workflow on the user side anyway.

Ada in Jest

Lightening the mood in serious discussion

From: Jeffrey R. Carter <spam.jrcarter.not@spam.not.acm.org>
Date: Tue, 12 Mar 2019 16:41:28 -0500
Subject: Re: Intervention needed?
Newsgroups: comp.lang.ada

I have no desire to register to post on a [Rust] forum full of people who like to use pointers. It's bad enough being on one full of people who like to use anonymous access types.

From: J-P. Rosen <rosen@adalog.fr>
Date: Fri, 22 Mar 2019 15:09:36 +0100

Le 22/03/2019 à 12:10, Lucretia a écrit : .

>> He [a Rust forum poster] also told me that Ada compilers aren't allowed to do certain kinds of optimizations that for example c, c++ (and Rust and other PL via LLVM) are doing.

>

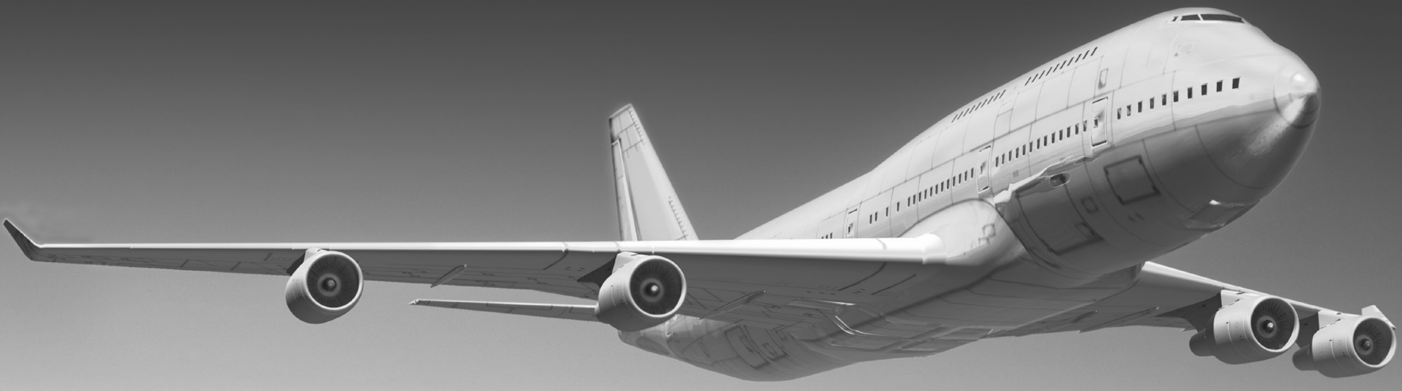
> How true is this?

In Ada, the principle is that the compiler has an obligation of result, i.e. that the "external effect" (see 1.1.3(9)) of the compiled program must be the same as the effect defined by the canonical execution.

Basically, this means that the compiler can do any optimization provided the result is correct. Going farther than that would mean allowing the compiler to generate incorrect programs... Maybe that's what C/C++ compilers are doing ;-)

Revolutionize your software verification

Efficiency, Automation, Reliability



RAPITA Systems
A DANLAW Company

Unit testing · System testing · Coverage analysis · Timing analysis
V&V services · Multicore timing services · DO-178C training
Ada · C · C++

www.rapitasystems.com

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

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.
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.
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.

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.

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. Deadline for early registration: July 7, 2019.

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 14th **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.

☺ 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 11-13 12th **International Conference on the Quality of Information and Communications Technology** (QUATIC'2019). Ciudad Real, Spain. Topics include: all quality aspects in ICT systems engineering and management; quality in ICT process, product and applications domains; practical studies; etc. Tracks on quality aspects in model-driven engineering, DevOps development, process improvement and assessment, verification and validation, evidence-based software engineering, etc.

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.

September 19-20 13th ACM/IEEE **International Symposium on Empirical Software Engineering and Measurement** (ESEM'2019). Porto de Galinhas, Brazil. Deadline for submissions: 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 is 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. 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: July 13, 2019 (ACM SIGBED Student Research Competition abstracts).

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 17-18 9th **Workshop on Model-Based Design of Cyber Physical Systems** (CyPhy'2019). New York City, NY, USA. In conjunction with ESWEEK 2019. Deadline for submissions: August 12, 2019 (abstracts), August 16, 2019 (papers).

☺ 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: 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 20-25 **Onward!** 2019. Topics include: everything to do with programming and software, including processes, methods, languages, communities, and applications; different ways of thinking about, approaching, and reporting on programming language and software engineering research.
>
> 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: 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.

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.

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.

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: July 15, 2019 (workshop papers), August 16, 2019 (student volunteers).

November 25-29 22nd **Brazilian Symposium on Formal Methods** (SBMF'2019). São Paulo, Brazil. Topics include: techniques and methodologies (such as model-driven engineering, development methodologies with formal foundations, software evolution based on formal methods, ...); specification and modeling languages (such as well-founded specification and design languages, formal aspects of popular languages, logic and semantics for programming and specification languages, code generation, formal methods of programming paradigms (such as objects, aspects, and component), formal methods for real-time, hybrid, and safety-critical systems, ...); theoretical foundations (such as type systems, models of concurrency, security, ...); verification and validation (such as abstraction, modularization and refinement techniques, correctness by construction, model checking, static analysis, formal techniques for software testing, software certification, ...); experience reports regarding teaching formal methods; applications (such as experience reports on the use of formal methods, industrial case studies, tool support). Deadline for submissions: July 16, 2019 (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: July 15, 2019 (workshop paper abstracts), July 22, 2019 (workshop papers), 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.

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

2020

January 14-17 12th **Software Quality Days** (SWQD'2020). Vienna, Austria. Topics include: improvement of software development methods, processes, and artifacts; testing and quality assurance of software and software-intensive systems; domain-specific quality issues such as embedded, medical, automotive systems; novel trends in software quality; etc.

Apr 25 - May 01 23rd **European Joint Conferences on Theory and Practice of Software** (ETAPS'2020). Dublin, Ireland. Events include: ESOP (European Symposium on Programming), FASE (Fundamental Approaches to Software Engineering), FoSSaCS (Foundations of Software Science and Computation Structures), TACAS (Tools and Algorithms for the Construction and Analysis of Systems). Deadline for submissions: August 30, 2019 (nominations EAPLS PhD Award).

♦ June 08-12 25th **Ada-Europe International Conference on Reliable Software Technologies** (AEiC 2020 aka Ada-Europe 2020). Santander, Spain. Sponsored by Ada-Europe. Deadline for submissions: January 7, 2020 (journal-track papers, industrial presentation outlines, tutorial and workshop proposals), 31 March 2020 (Work-in-Progress papers).

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

ptc® apexada | ptc® objectada®

Complete Ada Solutions for Complex Mission-Critical Systems

- Fast, efficient code generation
- Native or embedded systems deployment
- Support for leading real-time operating systems or bare systems
- Full Ada tasking or deterministic real-time execution

Learn more by visiting: **ptc.com/developer-tools**

ptc

25th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2020)

8-12 June 2020, Santander, Spain

Conference Chair

Michael González Harbour
University of Cantabria, Spain
mgh@unican.es

Program Chair

Mario Aldea Rivas
University of Cantabria, Spain
aldeam@unican.es

WiP Chair

Kristoffer Nyborg Gregertsen
SINTEF Digital, Norway
kristoffer.gregertsen@sintef.no

Educational Tutorial & Workshop Chair

Jorge Garrido Balaguer
Technical University of Madrid, Spain
jorge.garrido@upm.es

Exhibition & Sponsorship Chair

Ahlan Marriott
White Elephant GmbH, Switzerland
software@white-elephant.ch

Publicity Chair

Dirk Craeynest
Ada-Belgium & KU Leuven, Belgium
dirk.craeynest@cs.kuleuven.be

General Information

The 25th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2020 aka Ada-Europe 2020) will take place in Santander, Spain. The conference schedule includes a technical program, vendor exhibition and parallel tutorials and workshops.

The 2020 edition of the conference continues the major revamp in the registration fees introduced in 2019, redesigned to extend participation from industry and academia, and to reward contributors, especially but not solely, students and post-doc researchers.

Schedule

7 January 2020	Submission of journal-track papers, industrial presentation outlines, and tutorial and workshop proposals
20 March 2020	Notification of acceptance for journal-track papers and industrial presentations
31 March 2020	Submission of Work-in-Progress (WiP) papers
30 April 2020	Notification of acceptance for WiP papers

Topics

The conference is a leading international forum for providers, practitioners and researchers in reliable software technologies. The conference presentations will illustrate current work in the theory and practice of the design, development and maintenance of long-lived, high-quality software systems for a challenging variety of application domains. The program will allow ample time for keynotes, Q&A sessions and discussions, and social events. Participants include practitioners and researchers from industry, academia and government organizations active in the promotion and development of reliable software technologies.

The topics of interest for the conference include but are not limited to:

- Design and Implementation of Real-Time and Embedded Systems,
- Design and Implementation of Mixed-Criticality Systems,
- Theory and Practice of High-Integrity Systems,
- Software Architectures for Reliable Systems,
- Methods and Techniques for Quality Software Development and Maintenance,
- Ada Language and Technologies,
- Mainstream and Emerging Applications with Reliability Requirements,
- Achieving and Assuring Safety in Machine Learning Systems,
- Experience Reports on Reliable System Development,
- Experiences with Ada.

Refer to the conference website for the full list of topics.

Call for Journal-Track Papers

The **journal-track papers** submitted to the conference are full-length papers that must describe mature research work on the conference topics. They must be original and shall undergo anonymous peer review. The corresponding authors shall submit their work by 7 January 2020.

Accepted journal-track papers will get a presentation slot within a technical session of the conference and they will be published in an **open-access journal**, with no additional costs to authors.

Call for WiP-Track Papers

The **Work-in-Progress papers (WiP-track)** are short (4-page) papers describing evolving and early-stage ideas and new research directions. They must be original and shall undergo anonymous peer review. The corresponding authors shall submit their work by 31 March 2020, strictly in PDF and following the Ada User Journal style (*http://www.ada-europe.org/auj/*).

Authors of accepted WiP-track papers will get a presentation slot within a regular technical session of the conference and will also be requested to present a poster. The papers will be published in the Ada User Journal as part of the proceedings of the Conference.

The conference is listed in the principal citation databases, including DBLP, Scopus, Web of Science, and Google Scholar. The Ada User Journal is indexed by Scopus and by EBSCOhost in the Academic Search Ultimate database.

Call for Industrial Presentations

The conference seeks **industrial presentations** that deliver insightful information value but may not sustain the strictness of the review process required for regular papers. The authors of industrial presentations shall submit their proposals, of at least 1 page in length, by 7 January 2020, strictly in PDF and following the Ada User Journal style (*http://www.ada-europe.org/auj/*).

The Industrial Committee will review the submissions anonymously and make recommendations for acceptance. The authors of accepted contributions shall be requested to submit a short (one or two pages) abstract, for inclusion in the conference booklet, and be invited to deliver a talk at a regular technical session of the conference. These authors will also be required to expand their contributions into articles for publication in the Ada User Journal, as part of the proceedings of the Industrial Program of the Conference.

Awards

Ada-Europe will offer an honorary award for the best presentation.

Call for Educational Tutorials

The conference is seeking tutorials in the form of educational seminars including hands-on or practical demonstrations. Proposed tutorials can be from any part of the reliable software domain, they may be purely academic or from an industrial base making use of tools used in current software development environments. We are also interested in contemporary software topics, such as IoT and artificial intelligence and their application to reliability and safety.

Tutorial proposals shall include a title, an abstract, a description of the topic, an outline of the presentation, the proposed duration (half day or full day), and the intended level of the tutorial (introductory, intermediate, or advanced). All proposals should be submitted by e-mail to the Educational Tutorial Chair.

The authors of accepted full-day tutorials will receive a complimentary conference registration. For half-day tutorials, this benefit is halved. The Ada User Journal will offer space for the publication of summaries of the accepted tutorials.

Call for Workshops

Workshops on themes that fall within the conference scope may be proposed. Proposals may be submitted for half- or full-day events, to be scheduled at either end of the conference days. Workshop proposals should be submitted to the Workshop Chair. The workshop organizer shall also commit to producing the proceedings of the event, for publication in the Ada User Journal.

Call for Exhibitors

The commercial exhibition will span the core days of the main conference. Vendors and providers of software products and services should contact the Exhibition Chair for information and for allowing suitable planning of the exhibition space and time.

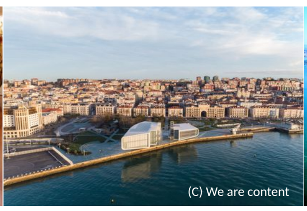
Special Registration Fees

Authors of accepted contributions and all students will enjoy reduced registration fees.

Venue

Santander is a nice tourist city in the north of Spain, with a well-connected airport and at a 100 km drive from Bilbao airport.

The conference venue and hotel is the Bahia Hotel in the city center and beside Santander bay.



A “New” C Static Analyzer: the Compiler

Maurizio Martignano

Spazio IT – Soluzioni Informatiche s.a.s., San Giorgio Bigarello, Italy; Maurizio.Martignano@spazioit.com

Abstract

While in the past in the C/C++ world compilers and static analyzers took two separate paths and were two separate lines of tools, nowadays they are coming back together, especially the Clang compiler and its Clang/LLVM based static analyzers. The paper will show why and how this “reunion” is beneficial, especially when analyzing large codebases. In particular the paper first will present these relatively new analyzers, then it will show how these tools are currently integrated in code quality platforms – e.g. SonarQube; finally, the paper will describe the author’s recent results in terms of improving the analyzers - code quality platforms integration and facilitating the adoption and execution of static analysis in software projects.

Keywords: C/C++, Static Analysis, Clang, LLVM, libclang, libadalang, SonarQube

1 (a historical) Introduction

The size of software systems in spacecrafts has increased dramatically over time: e.g. in 1981 NASA Space Shuttle Primary Flight Software was about 400 K lines of code (LOC) while in 2012 Curiosity had 2.5 MLOC; in 2008 ESA Automatic Transfer Vehicle (ATV) had about 1 MLOC. A similar trend has occurred in avionics: e.g.: in 1974 an F16A plane had 135 KLOC while in 2012 an F35 had about 10 MLOC. An even more dramatic increase has occurred in the automotive sector where a nowadays car software ranges from 10 to 150 MLOC. This increase in the software size, and consequently its importance and criticality, calls for new and more efficient methodologies and tools able to properly support Independent Software Verification and Validation (ISVV) activities, so that they are feasible and economic even when applied to very large codebases.

In the C language, from the very beginning, around the end of the seventies, a decision was taken to somehow separate the compiler from the static analyzer (“Lint”) where the compiler gives the programmer as much freedom (and responsibility) as possible and the “Lint” analyzer looks for programming and stylistic errors, bugs, questionable constructs and so on… This initial separation has caused over time the independent evolution of compilers and static analyzers as well as the communities of people working on them. While the compiler communities concentrated on the actual software trends, i.e. its continuous increase in size (by definition, a build system must be able to build the system), the static analyzer communities have concentrated on developing deeper and more detailed analysis techniques without really considering their actual applicability to large codebases. Nowadays, especially thanks to Clang (C language family front-end) and LLVM (compiler infrastructure), two open source software systems mostly developed by Apple, Microsoft, Google, ARM, Sony, Intel and Advanced Micro Device, the compiler and the static analyzer are getting back together, in the attempt of combining the compiler efficiency with analytical power of the static analyzer. In Clang and LLVM [1] systems all main functionalities are available as libraries: e.g. the lexer is a library (clangLex”) used by the both the compiler (Clang) and its static analyzers (Clang-Check and Clang-Tidy).

Section two of this paper will present various types of static analyzers, differentiated based on the technique they use, i.e. pattern matching, data flow analysis, abstract interpretation and model checking. The paper will show the actual applicability of these techniques to large codebases.

Section three will present the Clang/LLVM based static analyzers, will show their main characteristics and how they perform on large codebases. The section will present also “libclang”, a C programming interface (API) to the compilation system, accessible via C and Python. The section will end presenting Clang “JSON Compilation Database Format Specification” [2], which is a data “format for specifying how to replay single compilations independently of the build system”. This format is becoming a “de-facto” standard among tools vendors and simplifies the task of properly configuring static analyzers.

Section four will present the current status of the art of the integration of the Clang/LLVM static analyzers with SonarQube [3], an open source code quality platform. Section five will present the author’s current on code analysis, the “SAFe Toolset”, a toolset based on open source tools, able to facilitate and simplify the execution of static analysis on large codebases.

2 Static Analyzers

It is relatively common to categorize static analyzers based on the techniques they use to find issues, that is:

- pattern matching – the analyzer looks for particular constructs in the code at both lexical and syntactical levels – e.g. PC-Lint [4], Cppcheck [5];
- abstract interpretation – the analyzer verifies the code by “executing” it on some kind of abstract machine that approximates the original system – e.g.

Polyspace [6] and Frama-C Eva [7] and Clang-Check [8];

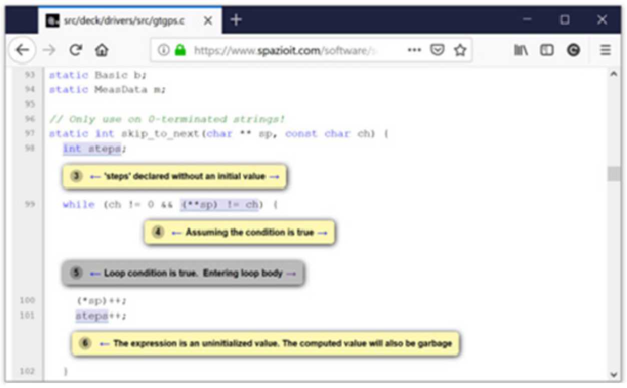
- data-flow analysis – the analyzer keeps track of a set of values and how they might change during (virtual) execution – e.g. Cppcheck and Clang-Check;
- Hoare logic – the analyzer uses a set of logical rules for reasoning rigorously about the code – e.g. Frama-C WP on C code extended with ACSL [9] (Ansi C Specification Language);
- (bounded) model checking – the analyzer works on a finite state machine representation of the code under analysis – e.g. CBMC [10].

Techniques like pattern matching and data-flow analysis are able to process relatively large codebases and this is because their analyses are rather “shallow”; on the contrary all the others (i.e. abstract interpretation, Hoare logic and model checking) do not scale well with the size of the code under analysis. Analyzers like CBMC and Frama-C attempt to perform very “deep” analyses; they use GNU GCC for parsing while they implement their own semantic analyzers - and this is where they expose some problems: in no way the commercial or open source community developing / supporting a given analyzer tool has the same momentum and energy of the community behind a compiler (be it GCC or Clang). The C and C++ programming languages are very much alive and in continuous evolution; so, for a tool developer/community it is quite difficult keeping up with the pace of this evolution (and the continuously increasing demand in performances and efficiency). One possible way of coping with this “lack of scalability issue” is to use the “shallow” analyzers to first identify critical areas in the codebase and then apply the “deep” analyzers only to these critical and smaller areas.

Another problem is that some of the “deep” analyzers, in order to properly function, require the code to be extended with additional information (e.g. Frama-C WP and C code enriched with ACSL annotation). In the majority of the cases it is very unlikely that a project has enough resources to annotate the source code with the required extra information.

3 Clang/LLVM based Static Analyzers, “libclang”, and JSON Compilation Database

In the Clang/LLVM website it is possible to read “The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. (…) The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many CPUs. (…) Clang is an LLVM native C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles.” In fewer words Clang/LLVM is a compilation toolchain where absolutely everything is built in a modular fashion as collection of libraries. In this toolchain the two static analyzers are Clang-Check (a.k.a. Clang-SA) and Clang-Tidy. Clang-Check relies on a set


src/deck/drivers/src/gtgps.c
https://www.spazioit.com/software/

```
static Basic b;
static MeasData m;

// Only use on 0-terminated strings!
static int skip_to_next(char ** sp, const char ch) {
  int steps;
    3 ← 'steps' declared without an initial value →
  while (ch != 0 && (**sp) != ch) {
                4 ← Assuming the condition is true →
    5 ← Loop condition is true.  Entering loop body →
    (*sp)++;
    steps++;
     6 ← The expression is an uninitialized value. The computed value will also be garbage
  }
```

Figure 1 - Clang-Check Graphic Output

of Clang modules to perform things like lexical analysis, parsing, semantic analysis, AST manipulation and the like. Clang-Tidy relies on the very same Clang modules plus some additional modules of Clang-Check itself (this is why Clang-Tidy can be considered a sort of superset of Clang-Check). So, for instance, the lexer and parser of the static analyzers are the very same lexer and parser of the compiler: when the compiler evolves to keep up with the changes and improvements in the language, the very same evolution occurs also in the static analyzers (i.e. no disconnect between the compiler and the static analyzer worlds). Clang-Check implements path-sensitive, inter-procedural analysis based on symbolic execution and data flow analysis techniques. Even if the analyzer attempts to perform a sort of abstract interpretation it is still very efficient because it does not perform this interpretation on the entire codebase but only on “suspected” areas, identified via fast techniques like pattern matching; the performed checks are listed in document [11]. Clang-Tidy adds up on top of Clang-Check and the list of its checks is available in document [12].

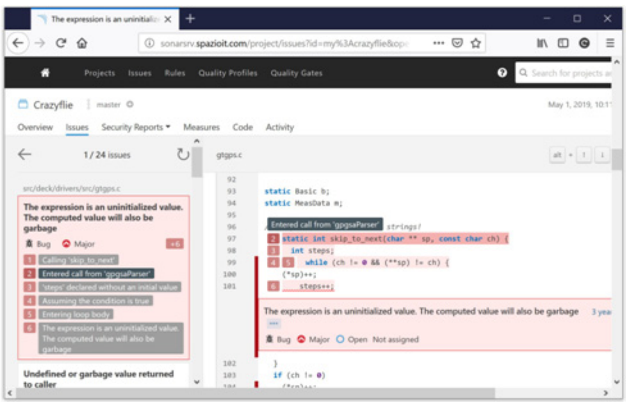
What really makes Clang-Check stand apart from other static analyzers is its ability to show graphically from within the code the causes of a potential issue/bug, as shown here below.

“libclang” is nothing but a simple C API (with Python bindings) exposing Clang functionalities (i.e. modules) to external applications; thanks to “libclang” also these third-party applications can use the very same modules/libraries of Clang (for instance they could parse a C program as efficiently as Clang does). This is similar to what was available in Ada with the ASIS (Ada Semantic Interface Specification) library [13]; also, with this library it is possible to build Ada tools, different from the compilers. The ASIS library implemented an “unforgiving / deep parsing” algorithm, that is while the analysis was pretty detailed, the code processed by the library had to be syntactically correct. C static analyzer of the pasts, e.g. “Lint”, were using “forgiving / shallow parsing” algorithms, able to process also code syntactically incorrect but limiting their exploration capabilities to analyses not too much detailed. “libclang”, on the contrary implements a “forgiving / deep parsing” algorithm (containing also portions of the semantic analysis), that can access all the detailed information available to the compiler itself but can also process incomplete, syntactically incorrect codebases and is very performant. “libclang” advantages are obvious, up to the point that a sort of a new version of ASIS library, called “libadalang” [14], adopts the same “forgiving / deep parsing” approach. The difference between Ada ASIS (or “libadalang”) and “libclang” is that, once again, Ada ASIS and “libadalang” are software products/tools separated from the actual Ada Compilers (e.g. the GNAT compiler); and this is why, for example, ASIS does not yet support Ada 2012.

When performing code analysis static analyzers are used to examine the source code; the more static analyzers the better, the more issues are found. The problem is that, at least up to now, each static analyzer uses its own format for the configuration files controlling their behavior (e.g. which source files to analyze, how to analyze them, and so on…). In the attempt of trying to solve this problem, the Clang/LLVM community has developed a standard format specification, called “JSON Compilation Database Format Specification”, defining “a format for specifying how to replay single compilations independently of the build system”. This “de facto” standard is rather young, immature: several build tools (e.g. cmake [15], compiledb [16], bear [17], ninja [18],) support it but in (slightly) different ways; the same applies to static analyzers - e.g. among Clang-Check, Clang-Tidy, Cppcheck and SonarQube C/C++ Community Plugin [19] only the first two (obviously) properly support it.

4 Integration with SonarQube

Static analyzers produce their results in the form of tabular data (e.g. a *.csv or a *.xml file) where per each found issue, the provided information is the issue type, where it was found (file name, line number) and some additional explanation. It is very difficult to assess the found issues in this format. An already mentioned exception is Clang-Check, which can produce graphic html pages with the flow of events causing a given issue to occur. SonarQube, an open source quality platform, is a web application able to gather the analyses results produced by the various analyzers and display them from



The expression is an uninitialized value. The computed value will also be garbage

Bug Major +6

1 Calling 'skip_to_next'
2 Entered call from 'gpgsaParser'
3 'steps' declared without an initial value
4 Assuming the condition is true
5 Entering loop body
6 The expression is an uninitialized value. The computed value will also be garbage

Undefined or garbage value returned to caller

```

static Basic b;
static MeasData m;

Entered call from 'gpgsaParser' strings/
static int skip_to_next(char ** sp, const char ch) {
  int steps;
    while (ch != 0 && (**sp) != ch) {
  (*sp)++;
    steps++;
```

The expression is an uninitialized value. The computed value will also be garbage 3 yea

Bug Major Open Not assigned

```
  }
  if (ch != 0)
```

Figure 2 - Clang-Check Sensor Output displayed in SonarOube

within the source code itself. SonarQube is language agnostic and the interface between SonarQube and a particular language is provided by “Plugins”. Inside the “Plugin” per each supported static analyzer there is a “Sensor” that actually converts the analyzer results into a format processable by SonarQube. For the C and C++ languages there are two Plugins, one commercial developed by SonarSource [20] and one open source [19]. The open source plugin supports Clang-Check and Clang-Tidy (as well as PC-Lint, Cppcheck and some other analyzers) and the author has always been modifying, improving it according to the various project needs; in particular the author has been working on the Clang-Check and PC-Lint “Sensors”. It has to be noted that the Clang-Check in the community plugin has gone through a dramatic improvement towards the end of April 2019 (version 1.3.0-SNAPSHOT); in particular now it is able to show multilocation issues as well as if not better than the static analyzer itself, as shown here below.

5 The SAFe Toolset

The SAFe (Static Analysis Framework) Toolset is an Ubuntu Virtual Machine containing various open source tools that can be used to perform Software Verification and Validation. The actual contents of this virtual machine (as per April 2019) are described here [21]; in this context is enough to mention that the machine includes Clang/LLVM compilation system, with its static analyzers – Clang-Check and Clang-Tidy, Sonar-Qube and the SAFacilitator (Static Analysis Facilitator).

The SAFacilitator is an application based on the Compilation Database Format Specification developed by the Clang/LLVM foundation and its major functionality is the simplification of the production of the static analyzers configuration files.

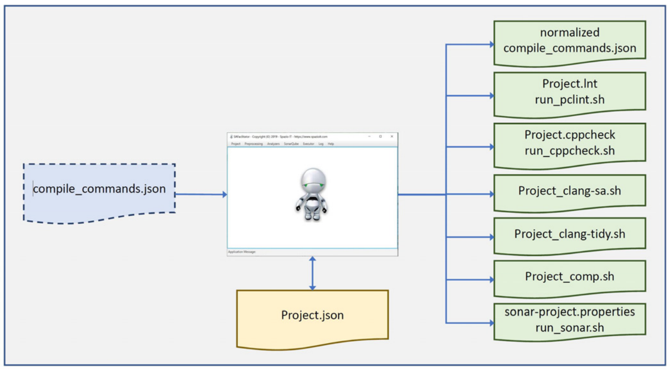

compile_commands.json
Project.json
normalized compile_commands.json
Project.lnt run_pclint.sh
Project.cppcheck run_cppcheck.sh
Project_clang-sa.sh
Project_clang-tidy.sh
Project_comp.sh
sonar-project.properties run_sonar.sh

Figure 3 – The SAFacilitator (Static Analysis Facilitator)

The SAFacilitator, for a given codebase, allows the creation and editing of a single project file containing all the information required to properly drive and control the execution of the selected static analyzers on that codebase. This information may initially derive from an externally provided compilation database file. Once the information contained in the project file is correct, it is used to automatically generate all the configuration files

and execution scripts required by the static analyzers (and SonarQube). The development of the "SAFe Toolset" has been funded by the European Space Agency Contract number RFP/3-15558/18/NL/FE/as.

6 Conclusions and Future Work

This paper has stressed some few simple but important points:

1. The increase in the size of software codebases demands for faster and more efficient static analyzers.
2. When the compiler and the static analyzer(s) are separate, are not built form the same modules and libraries, there is no guarantee that they will follow at the same pace the natural evolution of the programming language.
3. When performing code analysis, the more static analyzers are used, the better. But then configuring all these tools (in input) and assessing all produced results (in output) may become quite complex and difficult. And this is where tools like the SAFacilitator (in input) and like SonarQube (in output) come to the rescue.

References

[1] http://clang.llvm.org/

[2] https://clang.llvm.org/docs/JSONCompilationDatabase.html

[3] https://www.sonarqube.org/

[4] https://www.gimpel.com/

[5] http://cppcheck.net/

[6] https://www.mathworks.com/products/polyspace.html

[7] https://frama-c.com/value.html

[8] https://clang-analyzer.llvm.org/

[9] http://frama-c.com/wp.html

[10] http://www.cprover.org/cbmc/

[11] https://clang-analyzer.llvm.org/available_checks.html

[12] https://clang.llvm.org/extra/clang-tidy/checks/list.html

[13] http://gnat-asis.sourceforge.net/

[14] https://github.com/AdaCore/libadalang

[15] https://cmake.org/

[16] https://github.com/nickdiego/compiledb

[17] https://github.com/rizsotto/Bear

[18] https://ninja-build.org/

[19] https://github.com/SonarOpenCommunity/sonar-cxx

[20] https://www.sonarsource.com/

[21] https://www.spazioit.com/pages_en/sol_inf_en/code_quality_en/safe-toolset-en/

Verification of Ada Programs with AdaHorn

Tewodros A. Beyene, Christian Herrera, Vivek Nigam
fortiss GmbH, Forschungsinstitut des Freistaats Bayern für softwareintensive Systeme und Services
Guerickestr. 25, 80805 München, Germany; email: {beyene, herrera, nigam}@fortiss.org

Abstract

We propose AdaHorn*, a model checker for verification of* Ada *programs with respect to correctness properties given as assertions.* AdaHorn *translates an Ada program together with its assertion into a set of Constrained Horn Clauses, and feeds it to a Horn constraints solver. We evaluate the performance of* AdaHorn *on a set of Ada programs inspired by C programs from the software verification competition (SV-COMP). Our experimental results show that* AdaHorn *outputs correct results in more cases than GNATProve, which is a widely used Ada verification framework.*

Keywords: Ada Verification, Model Checking, Horn Constraints Solving.

1 Introduction

Ada [1] is widely used by systems developers in the avionics, space, military and railways domains due to its features like strong typing, explicit concurrency, support for design-by-contract, non-determinism, etc., that enable developers to build robust and dependable safety critical systems. Prominent Ada analysis tools that support the development of dependable systems in Ada include *GNATProve* [2] and *Polyspace* [3]. These tools perform static analysis on Ada programs for detecting runtime errors, such as *array out-of-bounds*, *arithmetic overflow* and *division by zero*. It is known that static analysis tools often yield *false positives*, i.e. wrongly concluding that errors occur in a program, and sometimes even *false negatives*, i.e. wrongly concluding that a program does not have any error.

In this work, we aim to advance the support available for the analysis and verification of Ada programs by proposing AdaHorn, a Horn constraints-based model checker for verifying Ada programs with respect to correctness properties written as assertions. Similar to the *SeaHorn* [4] and *JayHorn* [5] frameworks, which respectively verify *C* and *Java* programs, AdaHorn translates Ada programs into a set of Constrained Horn Clauses (CHCs) [6, 7, 8] which are solved by well-known constraint solvers such as Eldarica [9] and Z3 [10]. In general, a CHC correspond to a clause with at most one positive occurrence of an uninterpreted predicate. One can also think of a CHC as a fragment of first-order formulas modulo background theories, where its constraints are formulated using a given background theory [6].

In this work, AdaHorn supports a small but non-trivial subset of Ada data types, namely integer, floating-point and boolean data types as well as arrays of these types, and Ada program constructs, which include *procedures*, *functions*, *for/while loops*, *if-then-else statements*, *case statements*, *procedure/-function calls* and *assertions*.

The contribution of this work is twofold: (1) AdaHorn, which is the first Horn constraints-based model checker for Ada programs. (2) A Horn constraints generator for Ada programs, that takes Ada programs as input and produces a set of CHCs. This makes various Horn constraints-based program analysis, verification and synthesis techniques available to Ada programs.

2 Preliminiaries

In this section, we introduce the set of Constrained Horn Clauses (CHC) that AdaHorn uses as an intermediate language to encode an Ada verification problem. We also discuss the class of Ada programs that can be handled by the current implementation of AdaHorn.

2.1 Constrained Horn Clauses

In general, a Constrained Horn Clause correspond to a clause that has at most one positive occurrence of an uninterpreted predicate. One can also think of a Constrained Horn Clause as a fragment of first-order formulas modulo background theories, where its constraints are formulated using a given background theory [6]. We use the symbol $\mathcal{A}$ to denote a background theory. In this paper, we let $\mathcal{A}$ be quantifier-free linear arithmetic.

Definition 1. CHC is defined by the following grammar:

$$\begin{aligned}
\Pi &::= HC \wedge \Pi \mid \top \\
HC &::= \forall vars : body \rightarrow head \\
pred &::= upred \mid \Phi \\
head &::= pred \\
body &::= \top \mid pred \mid body \wedge body \\
vars &::= \text{the set of all variables in a given clause} \\
upred &::= \text{an uninterpreted predicate applied to terms} \\
\Phi &::= \text{a formula whose terms and predicates are interpreted over } \mathcal{A}
\end{aligned}$$

We use Π to denote a set (conjunction) of CHC, while HC refers to a single Constrained Horn Clause.

2.2 Classes of Target Ada Programs

The current implementation of AdaHorn does not support all language features and constructs of Ada. However, all basic constructs of Ada that can be used to write programs of medium complexity are supported. These include (1) integer, floating-point and boolean data types, and self-defined ranges over these types, (2) arrays, (3) assertions, (4) while and for loops, (5) procedures and functions (together with their corresponding calls), and (6) if-then-else statements.

The complete class of supported constructs is given in Figure 1, which presents the grammar for the set of supported Ada programs. In that grammar, *range* stands for a finite range of the underlying data type. An example integer range can be $0..4$. *Id* corresponds to a typical identifier that can be used as function names, variable reference, etc. The symbol * denotes the Kleene closure of the underlying grammar item.

We aim to develop AdaHorn further by incrementally adding support not only for more constructs but additional Ada language features. One candidate language construct to add is *protected object* [1], which is useful for mutual exclusion problems. Similarly, a candidate language feature to consider in the future is the *Ravenscar profile* [11], which is useful for real-time and high-integrity applications.

```
⟨program⟩           ::= ⟨with_use⟩* ⟨pkg⟩ | ⟨with_use⟩* ⟨pkg_body⟩ |
                        ⟨with_use⟩* ⟨proc_fun_body⟩
⟨with_use⟩          ::= 'with' ⟨id⟩';' 'use' ⟨id⟩';'
⟨pkg⟩               ::= 'package' ⟨id⟩ 'is' ⟨decl⟩* ⟨proc_fun_decl⟩*
                        'end' ⟨id⟩';'
⟨pkg_body⟩          ::= 'package body' ⟨id⟩ ⟨proc_fun_body⟩*
                        'end' ⟨id⟩';'
⟨proc_fun_body⟩     ::= ⟨proc_body⟩ | ⟨fun_body⟩
⟨proc_body⟩         ::= 'procedure' ⟨id⟩ ['('⟨formal_part⟩')'] 'is'
                        ⟨decl⟩* ⟨proc_fun_body⟩* 'begin' ⟨stmt⟩*
                        'end' ⟨id⟩';'
⟨fun_body⟩          ::= 'function' ⟨id⟩ ['('⟨formal_part⟩')']
                        'return' ⟨type⟩ 'is' ⟨decl⟩*
                        ⟨proc_fun_body⟩* 'begin' ⟨stmt⟩* 'return'
                        ⟨stmt_rtn⟩';' 'end' ⟨id⟩';'
⟨formal_part⟩       ::= ⟨formal_param_spec⟩
                        (';' ⟨formal_param_spec⟩)*
⟨formal_param_spec⟩ ::= ⟨var⟩ (',' ⟨var⟩)* ':' ⟨mode⟩
                        ⟨type⟩ [':=' ⟨expr⟩]
⟨decl⟩              ::= ⟨var⟩ ':' 'array' '(' range ')' 'of' ⟨type⟩
                        [':=' '(' initial values for array ')'] ';' |
                        ⟨var⟩ ':' ⟨type⟩ [':=' ⟨expr⟩]';'
⟨var⟩               ::= ⟨id⟩
⟨proc_fun_decl⟩     ::= 'procedure' ⟨id⟩ '('⟨formal_part⟩')'';'
                        | 'function' ⟨id⟩ '('⟨formal_part⟩')'
                        'return' ⟨type⟩';'
⟨mode⟩              ::= 'in' | 'in out' | 'out'
⟨type⟩              ::= 'Integer' | 'Float' | 'Boolean'
⟨expr⟩              ::= arithmetic logical expression
⟨stmt⟩              ::= ⟨var⟩ ':=' ⟨expr⟩ | ⟨stmt1⟩ ';' ⟨stmt2⟩
                        | 'pragma assert' '('⟨expr⟩')' ';' |
                        'if' ⟨expr⟩ 'then' ⟨stmt1⟩ 'else' ⟨stmt2⟩
                        'end if' ';' | ⟨proc_fun_call⟩ ';' | 'while'
                        ⟨expr⟩ 'loop' ⟨stmt⟩ 'end loop' ';' | 'for'
                        ⟨id⟩ 'in' ⟨type⟩ 'range' range 'loop'
                        ⟨stmt⟩ 'end loop'';' | 'case' ⟨expr⟩ 'is'
                        ⟨case_option⟩ (⟨case_option⟩*) 'end case'
                        ';'
⟨proc_fun_call⟩     ::= procedure_name | procedure_name
                        '('⟨actual_part⟩')' | function_name |
                        function_name '('⟨actual_part⟩')'
⟨actual_part⟩       ::= ⟨actual_param_spec⟩
                        (',' ⟨actual_param_spec⟩)*
⟨actual_param_spec⟩ ::= ⟨expr⟩ | ⟨var⟩ | function_name | function_name
                        '('⟨actual_part⟩')'
⟨stmt_rtn⟩          ::= ⟨var⟩ ':=' ⟨expr⟩ | ⟨expr⟩
⟨case_option⟩       ::= 'when' ⟨discrete_choice⟩ '=>' ⟨stmt⟩ ';'
⟨discrete_choice⟩   ::= ⟨expr⟩ | range | 'others'
```

Figure 1: Grammar for supported Ada programs.

3 Architecture of AdaHorn

Inspired by similar approaches for verifying *C* and *Java* programs [4, 5], the architecture of AdaHorn consists of three layers. The architecture is shown in Figure 2.

- *Front-end*: This layer makes use of the *gnat2xml* utility in the *GNAT Compiler Tool* [12] to obtain an XML serialisation of an abstract syntax tree of the input Ada program.
- *Middle-end (glue code)*: In this layer, AdaHorn takes the abstract syntax tree generated by the *gnat2xml* utility as input, and translates them into a set of CHCs. AdaHorn performs a top-down, recursive descent through the syntax tree of the given Ada program. It introduces auxiliary predicates and generates a set CHCs over these predicates.

  The interested reader finds in [6] a reference for translating program constructs into CHCs. Moreover, this step needs to also make special considerations for constructs that are not directly supported by the Horn constraints solvers used in this work. For example, as arrays may result in Horn clauses with nonlinear structure or higher orders, further processing needs to be done to translate resulting constraints into array-free Horn constraints [13].

  As this constraints generating middle-end layer, also called the glue code, is the main contribution of this work, it is explained in more detail in Section 4.
- *Back-end*: This layer takes the generated CHCs as input and pass them to a CHC solver to get a result, which is the final result of AdaHorn's model checking procedure. Any CHC solver can be employed in principle. However, we have used the Eldarica and Z3 solvers in this work. Possible results of the solving step are: SAT(the CHCs are satisfiable), UNSAT(the CHCs are unsatisfiable), and UNKNOWN(the solver is not able to output any conclusive result).

CHCs have enjoyed recent success as languages of intermediate representation, and a promising set of frameworks for verification tasks ranging from temporal verification to synthesis and game solving have recently been proposed [14, 15, 16]. The Horn constraints generator alone in AdaHorn can also be useful in making all these Horn constraints-based verification and synthesis technologies available to Ada programmers and verification engineers. AdaHorn is implemented in Java.

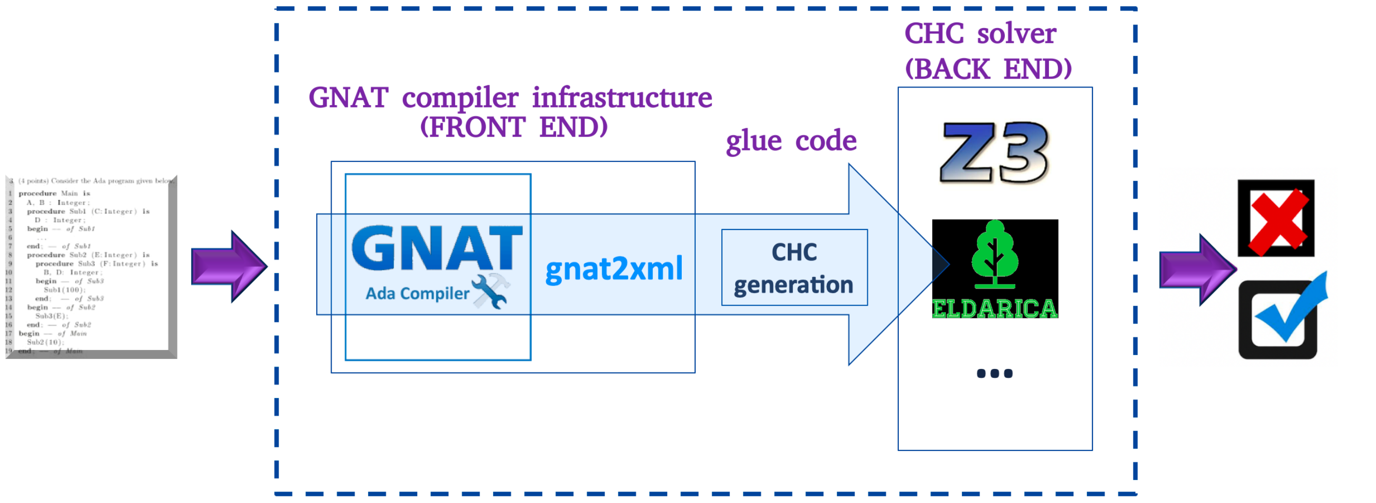


Figure 2: Architecture of AdaHorn

4 Constraints Generation

In this section, we describe AdaHorn's constraint generation process and illustrate how supported Ada constructs are encoded as CHCs by AdaHorn.

There are two considerations in the Ada to CHC translation:

1. The first one is encoding program states, which are valuations of program variables at certain critical locations of the program. These include loop entries and exits, procedure call and return locations, function call and return locations, etc.
2. The second one is encoding state transitions that occur during the execution of the program by translating involved Ada constructs into their corresponding CHC.

In the rest of this section, we illustrate the constraint generation process using two simple Ada programs that contain non-trivial constructs from the grammar in Figure 1. Both of our example Ada programs are structured into 3 files; a main Ada project file, a specification file and an implementation file. The constraint generation procedure needs to take into consideration all three files to generate a collective set of CHCs. We provide the corresponding CHCs encoding for each example, together with discussion on how the translation is done for supported Ada language constructs. AdaHorn generates CHC in the *SMT-Lib Standard* [17].

Example 1. Our first example Ada program, which is shown in Figure 3, simply increments an input integer value by 10 and returns the result. As mentioned above, this program is structured into a project file (*Figure 3a*), an implementation file (*Figure 3b*), and a specification file (*Figure 3c*).

The program initialises its two integer variables, *res* and *x*, to 0 (*line* 3 *in Figure 3a*), and calls the function *sum* over *x* (*line* 5 *in Figure 3a*). Note that the specification file for the function *sum* is given in Figure 3c, and its corresponding implementation file is given in Figure 3b. Finally, the value

```
1    with prog1;
2    procedure gmain is
3       res, x: Integer := 0;
4    begin
5       res := prog1.sum(x);
6    end gmain;
```

(a) project file

```
1    package prog1 is
2       function sum(j: in out Integer)
3       return Integer;
4    end prog1;
```

(b) specification file

```
1    package body prog1;
2       function sum (j: in out Integer)
3       return Integer is
4       begin
5          return j+10;
6       end sum;
7    end prog1;
```

(c) implementation file

Figure 3: A simple example program.

returned by the function is stored in the variable *res* (*line* 5 *in Figure 3a*).

The set of CHCs generated by AdaHorn for this program is given in Figure 4. Each state of the Ada program is encoded as an uninterpreted predicate over the program variables. For example, the predicate *gmain_call_init*, which is defined over the empty set of variables, encodes the state from which the program starts execution. The start of the program execution is encoded as the clause $(assert\ (=>\ true\ gmain\_call\_init))$ with just *true* in the body (*line 8*). Once the program starts running, the first statement it executes is initialisation of its variables *res* and *x* to *0* (*line 9*). The call to the function *sum* (line 5 in Figure 3a) is encoded in *line 10*. The clause in *line 12* of Figure 4 encodes the addition operation in *return j+10* (line 5 in Figure 3b) of the Ada program. The return statement itself is encoded In *line 12*, where that result of the sum operation is stored in the auxiliary variable $\_RetV$. The value of auxiliary variable $\_RetV$ is assigned back to the variable *res* of the caller *gmain* function

```
1   ( declare −fun gmain_s0 (Int  Int )  Bool)
2   ( declare −fun gmain_s1 (Int  Int )  Bool)
3   ( declare −const  gmain_call_init  Bool)
4   ( declare −const gmain_call_end Bool)
5   ( declare −fun Prog1_Sum_s0 (Int) Bool)
6   ( declare −fun Prog1_Sum_call_init  ( Int )  Bool)
7   ( declare −fun Prog1_Sum_call_ret ( Int )  Bool)
8   ( assert  (=> true   gmain_call_init ))
9   ( assert  ( forall  (( res  Int )  (x  Int )) (=> (and (=  res  0)
          (= x  0))  gmain_call_init )  (gmain_s0 res  x))))
10  ( assert  ( forall  (( res  Int )  (x  Int )) (=> (gmain_s0 res  x)
          (Prog1_Sum_call_init  x))))
11  ( assert  ( forall  (( j  Int )) (=> (Prog1_Sum_call_init  j )  (
          Prog1_Sum_s0 j))))
12  ( assert  ( forall  (( j  Int )  (_RetV Int )) (=> (and (= _RetV
          (+ j  10))  (Prog1_Sum_s0 j))  (Prog1_Sum_call_ret
          _RetV))))
13  ( assert  ( forall  (( res  Int )  (x  Int )  ( res ’  Int )  (_RetVV
          Int ))(=> (and  (gmain_s0 res  x)  (Prog1_Sum_call_ret
          _RetVV) (= res ’  _RetVV)) (gmain_s1 res ’  x))))
14  ( assert  ( forall  (( res  Int )  (x  Int )) (=> (gmain_s1 res  x)
          gmain_call_end)))
```

Figure 4: Generated CHCs for the program in Figure 3

```
1   with prog2;
2   procedure gmain is
3   begin
4       prog2. initArray ;
5   end gmain;
```

(a) project file

```
1   package prog2 is
2       procedure initArray ;
3   end prog2;
```

(b) specification file

```
1   package body prog2;
2       procedure initArray
3           arr  :  array  (1..10)  of  Integer ;
4           temp:  Integer ;
5       begin
6           arr (1):=  0;
7           temp:= arr (1);
8           arr (3):=  temp;
9           pragma Assert ( arr (1)  =  arr (3));
10      end  initArray ;
11  end prog2;
```

(c) implementation file

Figure 5: A program with array read and write operations.

as encoded in *line 13*. The constraint in *line 14* denotes the end of the call to procedure *gmain*.

Example 2: In this example, we show how a set of CHC is generated for a program with arrays. The program is given in Figure 5. Like the previous example, here also we have three files. Our main focus, however, will be on the implementation file (*Figure 5c*) as it contains all the important variables and statements of the program from users' perspective. The program is defined over an array variable *arr*, which defines an array of length 10, and an integer variable *temp*. During execution, the program calls the *initArray* procedure which is specified and implemented in the package *prog2*. As shown in Figure 5c, the procedure does three assignments: the assignments *arr(1):= 0;* and *arr(3):= temp;* (on lines 6 and 8) involve array write operations, whereas the assignment *temp:=arr(1)* (on line 7) involves array read operation. Finally, as a correctness property the procedure ensures the array has equal values on indices 1 and 3 with the assert statement *pragma Assert (arr(1) = arr(3))*.

This example is particularly interesting as verification of many properties over arrays often require inferring universally quantified invariants over arrays. However, no general algorithms exist for checking if such universally quantified array invariants hold, let alone inferring them. In this paper, we have applied a system of rules for transforming atomic array read and write statements into a system of non-linear Horn Clauses over scalar variables only [13]. We find this approach efficient compared to other approaches for handling arrays as it does not introduce a new abstract domain or a new interpolation procedure for arrays. Instead, it generates an abstraction as a scalar problem, that can be fed to any solver that can handle non-linear Horn Clauses.

The set of CHCs for this example is given in *Figure 6*. The declarations for the boolean variables and predicates used in the generated CHCs are given in *lines 1 - 9*. The first important clause to consider is at *line 13* that defines the predicate *Prog2_InitArray_s0*. This predicate represents the state of the program before the input array variable arr is updated. The auxiliary variable *arrIND* denotes an index of *arr*. Note that this index takes only the values of the indices of the array accessed in the corresponding Ada program. As the aim of this example is to illustrate the handling of arrays during CHCs generation, our focus will be the array read and write operations in the example Ada program.

Let us now discuss the crux of the implemented array handling method. A predicate *pred(arr, temp)*, such as the one encoding the states for our example program, is assumed to internally have a table structure with a set of entries $(ind_1, val_1, temp)$, $(ind_2, val_2, temp)$, $\ldots(ind_n, val_n, temp)\}$, where $arr[ind_i] = val_i$ for each $1 \leq i \leq n$. However, we do not want to flatten the array eagerly. Rather, what we want to do is to keep the array as abstract as possible and refer to concrete values only when the need arises. With this in mind, let us try to see how we handle array read and write operations one by one:

- write operation: Assume we want to encode $arr[5] := 10$. Here we will replace val_5 by 10 in $(ind_5, val_5, temp)$, and for the rest of the entries, i.e., $(ind_j, val_j, temp)$ for each $j \neq 5$, val_j stays the same. Let us assume predicates $pred_0$ and $pred_1$ represent states of the program before and after the write operation $arr[5] := 10$. Our encoding of the write operation consists of two CHCs: (1) $pred_0(ind, val, temp) \wedge ind = 5 \rightarrow pred_1(ind, 10, temp)$ (2) $pred_0(ind, val, temp) \wedge ind \neq 5 \rightarrow pred_1(ind, val, temp)$. In our example, there are the two write operations on lines 6 and 8 in Figure 5. Their corresponding CHCs can be found on lines 14-15 and 18-19 in Figure 6, respectively.
- read operation: Assume we want to encode $temp := arr[5]$. Here we will have to replace $temp$ by val_5

```
1    ( declare −const gmain_s0 Bool)
2    ( declare −const gmain_s1 Bool)
3    ( declare −const  gmain_call_init  Bool)
4    ( declare −const gmain_call_end Bool)
5    ( declare −fun Prog2_InitArray_s0  (  Int  Int )  Bool)
6    ( declare −fun Prog2_InitArray_s1  (  Int  Int )  Bool)
7    ( declare −fun Prog2_InitArray_s2  (  Int  Int )  Bool)
8    ( declare −const  Prog2_InitArray_call_init  Bool)
9    ( declare −const Prog2_InitArray_call_end  Bool)
10   ( assert  (= true  gmain_call_init ))
11   ( assert  (=  gmain_call_init  gmain_s0))
12   ( assert  (= gmain_s0  Prog2_InitArray_call_init  ))
13   ( assert  ( forall  ((arrIND Int) ( arr  Int) (temp Int))(=>
          Prog2_InitArray_call_init  (Prog2_InitArray_s0
         arrIND arr  temp))))
14   ( assert  ( forall  ((arrIND Int) ( arr  Int) ( arr ’  Int) (
         temp Int))(=> (and (Prog2_InitArray_s0 arrIND arr
         temp)(= arrIND 1)(= arr ’  0))(Prog2_InitArray_s1
         arrIND arr ’  temp))))
15   ( assert  ( forall  ((arrIND Int) ( arr  Int) ( arr ’  Int) (
         temp Int))(=> (and (Prog2_InitArray_s0 arrIND arr
         temp)(not (= arrIND 1)))  (Prog2_InitArray_s1
         arrIND arr  temp))))
16   ( assert  ( forall  ((arrIND Int) ( arr  Int) (temp Int) (
         temp’ Int)) (=> (and (Prog2_InitArray_s1  arrIND
          arr  temp)(= arrIND 1)(= temp’ arr ))(
         Prog2_InitArray_s3  arrIND arr  temp’))))
17   ( assert  ( forall  ((arrIND1 Int) (arrIND2 Int) ( arr2  Int)
         ( arr1  Int) (temp Int) (temp’ Int))(=> (and (
         Prog2_InitArray_s1  arrIND1 arr1 temp)(
         Prog2_InitArray_s1  arrIND2 arr2 temp)(= arrIND 1)
         (not (= arrIND1 arrIND2))(= temp’ arr1 ))(
         Prog2_InitArray_s3  arrIND2 arr2 temp’))))
18   ( assert  ( forall  ((arrIND Int) ( arr  Int) (temp Int) ( arr ’
          Int)) (=> (and (Prog2_InitArray_s3  arrIND arr
         temp)(= arrIND 3)(= arr ’  temp))(Prog2_InitArray_s4
          arrIND arr ’  temp))))
19   ( assert  ( forall  ((arrIND Int) ( arr  Int) ( arr ’  Int) (
         temp Int)) (=> (and (Prog2_InitArray_s3  arrIND arr
          temp)(not (= arrIND 3)))  (Prog2_InitArray_s4
         arrIND arr  temp))))
20   ( assert  ( forall  ((arrIND1 Int) (arrIND2 Int) ( arr1  Int)
          ( arr2  Int) (temp Int)) (=> (and (
         Prog2_InitArray_s4  arrIND1 arr1  temp) (
         Prog2_InitArray_s4  arrIND2 arr2  temp) (= arrIND1
         1)(= arrIND2 3))(= arr1  arr2))))
21   ( assert  ( forall  ((arrIND Int) ( arr  Int) (temp Int)) (=>
         (Prog2_InitArray_s4  arrIND arr  temp)
         Prog2_InitArray_call_end )))
22   ( assert  (=  Prog2_InitArray_call_end  gmain_s1))
23   ( assert  (= gmain_s1 gmain_call_end))
```

Figure 6: Corresponding set of CHC for the Ada program in Figure 5.

in $(ind_i, val_i, temp)$, for all i. Note that since $temp$ is not an array variable, its value is independent of an index. Let's assume predicates $pred_0$ and $pred_1$ represent states of the program before and after the read operation. Our encoding of the read operation consists of two CHCs: (1) $pred_0(ind, val, temp) \wedge ind = 5 \rightarrow pred_1(ind, val, val)$: this constraint ensures that if the entry has the index involved in the read, we simply copy val to $temp$. (2) $pred_0(ind_1, val_1, temp) \wedge pred_0(ind_2, val_2, temp) \wedge ind1 \neq 5 \wedge ind2 = 5 \rightarrow pred_1(ind_1, val_1, val_2)$: this constraint ensures that if the entry does not have the index involved in the read, it looks for another entry with the index involved in the read operation, and then it copies $val2$ from the other entry to $temp$. Note that this constraint is non-linear as there are two instances of $pred_0$ in the body of the CHC. This is where the role of non-linear Horn constraints encoding comes into play. In our example, there is a read operation on lines 7 in Figure 5, and its corresponding CHCs can be found on lines 16-17 in Figure 6.

5 Experiments

We evaluate AdaHorn on a set of Ada benchmarks that consists of four categories of programs: *arrays*, *floats*, *loops*, and *simple* [1]. The first three categories are inspired by C programs from the software verification competition SV-COMP 2017 [18]. For the C programs that can exclusively be translated to the subset of Ada handled in this work, we have manually created equivalent Ada programs. Programs in the *simple* category are written by the authors of this paper.

The arrays and floats categories contain programs whose assertion involves array and float variables, respectively. In the loops category, the program assertions are placed within while and for loop constructs. Programs in the simple category were constructed with integer variables and without any complex Ada construct like loop or logic statements. While assertions in the first three categories do not capture specific classes of properties, programs in the simple category contain assertion that captures runtime properties such division by zero, integer and floating-point over(under)-flow and array out of bounds.

The verification task is to prove if an assertion placed in a given program is valid or not. A timeout is reached if the verification task can not complete in 1000 seconds (indicated by **TO**). Results for each tool are classified into one of the following four classes by comparing it with the expected result: (1) **True Positive (TP)** - tool correctly indicates an assertion is not valid, (2) **True Negative (TN)** - tool correctly indicates an assertion is valid, (3) **False Positive (FP)** - tool wrongly indicates an assertion is not valid, and (4) **False Negative (FN)** - tool wrongly indicates an assertion is valid. In addition, AdaHorn may not be able to solve its generated constraints leaving the final result unknown (indicated by **UN**).

Coming to the results, GNATProve takes less than 3 seconds for verifying each benchmark, whereas AdaHorn timed out for one example and took less than 60 seconds to verify the remaining examples. GNATProve concludes false positives in 48 occasions (out of 68 benchmarks) and 2 false negatives! After reporting those false negatives to developers of GNATProve, we were told that GNATProve generally does not output correct results if intermediate checks have failed, which is the case for the programs related to those false negatives. This is mainly due to GNATprove's assumption that any prior check must be correct in order to prove the next ones. While AdaHorn does not result in any false negatives, it results in 4 false positives. The reason behind the false positives was the difference in precisions for floating-point numbers between the Ada compiler and the Horn constraint solvers used by

[1] https://bitbucket.org/umaya/adabenchmarksfromsvcomp17

benchmark category	number of programs	GNATProve					AdaHorn					
		TP	TN	FP	FN	TO	TP	TN	FP	FN	TO	UN
arrays	20	1	0	19	0	0	8	10	1	0	1	0
floats	20	1	4	13	2	0	5	8	3	0	0	4
loops	20	4	2	14	0	0	7	13	0	0	0	0
simple	8	0	5	2	0	0	2	6	0	0	0	0

Table 1: Comparison of results between GNATProve and AdaHorn

AdaHorn (Eldarica and Z3). AdaHorn, however, is not able to verify 4 benchmarks due to failure of the Horn constraints solvers to conclude whether input CHCs are satisfiable or unsatisfiable.

Finally, we would like to point out that AdaHorn has been created as a subproject in the context of a cooperation with one of our partners from the aviation industry. As a next step in our cooperation we plan to evaluate AdaHorn on our partner's industrial Ada code.

References

[1] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg (2013), *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, vol. 8339 of LNCS, Springer.

[2] J. G. P. Barnes (2003), *High Integrity Software - The SPARK Approach to Safety and Security*, Addison-Wesley.

[3] Polyspace. `https://www.mathworks.com/products/polyspace.html`.

[4] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas (2015), *The SeaHorn Verification Framework*, vol. 9206 of LNCS, pp. 343–361, Springer.

[5] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf (2016), *JayHorn: A Framework for Verifying Java Programs*, vol. 9779 of LNCS, pp. 352–358, Springer.

[6] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko (2015), *Horn Clause Solvers for Program Verification*, vol. 9300 of LNCS, pp. 24–51, Springer.

[7] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey (2015), *Horn Clauses as an Intermediate Representation for Program Analysis and Transformation*, TPLP, vol. 15, no. 4-5, pp. 526–542.

[8] N. Bjørner, K. L. McMillan, and A. Rybalchenko (2012), *Program Verification as Satisfiability Modulo Theories*, in SMT 2012, vol. 20 of EPiC Series in Computing, pp. 3–11, EasyChair.

[9] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer (2012), *A Verification Toolkit for Numerical Transition Systems - Tool Paper*, vol. 7436 of LNCS, pp. 247–251, Springer.

[10] L. M. de Moura and N. Bjørner (2008), *Z3: An Efficient SMT Solver*, vol. 4963 of LNCS, pp. 337–340, Springer.

[11] A. Burns, B. Dobbing, and G. Romanski (1998), *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*, vol. 1411 of LNCS, pp. 263–275, Springer.

[12] Project Hi-Lite / GNATprove (2014).

[13] D. Monniaux and L. Gonnord (2015), *An Encoding of Array Verification Problems into Array-Free Horn Clauses*, CoRR, vol. abs/1509.09092.

[14] T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko (2014), *A constraint-based approach to solving games on infinite graphs*, SIGPLAN Not., vol. 49, pp. 221–233.

[15] T. A. Beyene, C. Popeea, and A. Rybalchenko (2016), *Efficient CTL Verification via Horn Constraints Solving*, vol. 219 of EPTCS, pp. 1–14, 2016.

[16] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko (22015), *Horn Clause Solvers for Program Verification*, pp. 24–51, Cham: Springer International Publishing.

[17] C. Barrett, P. Fontaine, and C. Tinelli (2017), *The SMT-LIB Standard: Version 2.6*, tech. rep., Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[18] D. Beyer (2017), *Software Verification with Validation of Results - (Report on SV-COMP 2017)*, in TACAS 2017, vol. 10206 of LNCS, pp. 331–349.

Automate Your Ada Unit Testing

With VectorCAST/Ada

VectorCAST/Ada is an integrated software test solution that significantly reduces the time, effort, and cost associated with testing Ada software components necessary for validating safety- and mission-critical embedded systems.

> Complete test-harness construction for unit and integration testing
> Test execution from GUI or scripts
> Code coverage analysis
> Regression Testing
> Code complexity calculation
> Automatic test creation based on decision paths
> User-defined tests for requirements-based testing
> Test execution trace and playback to assist in debugging
> Integrations with best of breed requirements traceability tools

More information: **www.vector.com/vectorcast**

Vector Austria GmbH | www.vector.com

Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems (Part 2) [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

6 Verification of Ravenscar Programs

Chapter 1 described the motivation for the Ravenscar Profile in terms of the need to verify the temporal behaviour of concurrent real-time programs. In this chapter, we provide an introduction to the forms of verification that can applied to Ravenscar applications, to deliver dependable systems.

The approach to verification in the presence of Ada tasking is similar in many ways to that traditionally used for cyclic executives. Each thread of control is independently verified for conformance with its precise/formal specification, for example by performing requirements-based testing or by use of static analysis tools on its sequential behaviour. Then, the program as a whole is verified against all its timing constraints. This latter stage differs from the cyclic executive approach in the presence of priority-based preemptive task scheduling in that it can be automated by the use of, for example, a Response Time Analysis (RTA) tool to verify that a given task set meets its deadlines. The tool-based approach greatly simplifies the process of verification of timing constraints during development, and of re-verification after the system has undergone modification during maintenance.

The effects of arbitrary dynamic preemption can be statically analysed by considering all accesses to the global state of the program as being volatile, e.g. two successive reads to the same global state variable may deliver different values (as for reads of values delivered by an external device).

The approach to verification in the presence of Ada tasking is similar in many ways to that traditionally used for cyclic executives. Each thread of control is independently verified for conformance with its precise/formal specification, for example by performing requirements-based testing or by use of static analysis tools on its sequential behaviour. Then, the program as a whole is verified against all its timing constraints. This latter stage differs from the cyclic executive approach in the presence of priority-based preemptive task scheduling in that it can be automated by the use of, for example, a Response Time Analysis (RTA) tool to verify that a given task set meets its deadlines. The tool-based approach greatly simplifies the process of verification of timing constraints during development, and of re-verification after the system has undergone modification during maintenance.

The effects of arbitrary dynamic preemption can be statically analysed by considering all accesses to the global state of the program as being volatile, e.g. two successive reads to the same global state variable may deliver different values (as for reads of values delivered by an external device).

The core set of Ravenscar Profile run-time system packages can be developed to the most stringent software development standards so that these packages are suitable for inclusion in an application that requires certification against an applicable standard such as RTCA DO-178B [DO].

In this chapter, we look at four levels of verification:

- Static analysis of sequential code
- Static analysis of concurrent code
- Scheduling analysis
- Formal analysis

6.1 Static Analysis of Sequential Code

As discussed in the introduction, the Ravenscar Profile is silent about those features of the sequential language that

[1] **Editor note:** This paper includes chapters 6 to 7 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 (published in the Ada User Journal, Volume 40, Number 1, March 2019) present the Ravenscar profile, the rationale for the decisions taken and examples of usage. Chapter 6 discusses the verification approaches appropriate to Ravenscar programs, and Chapter 7 provides and extensive example of the profile use. 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

should be used with the profile (apart from requiring no implicit use of the heap). Similarly, it is not appropriate here to discuss the forms of static analysis that should be used to verify the functional behaviour of each task. The reader is referred to the ISO Technical Report Guide for the use of Ada in high integrity applications [GA].

6.2 Static Analysis of Concurrent Code

The two main goals of applying static analysis techniques to Ravenscar programs are:

- to obtain the same level of proof and data / information flow analysis for concurrent programs as is currently achievable for a sequential program;
- to obtain proof of absence of the concurrency-related run-time errors, to supplement the proof of absence of run-time errors that is currently achievable for sequential code.

The concurrency-related run-time errors that apply to Ravenscar programs are described in Sections 4.2.4 and 4.2.5.

In addition, it is highly desirable if the implementation-defined effect of task termination in the presence of the No_Task_Termination restriction can be eliminated.

The remainder of this section addresses various techniques for producing static analysis evidence to meet the above goals. These verification processes are made possible by the following assertions about the behaviour of a valid Ravenscar program:

- Each task and interrupt handler execution is deferred until after program elaboration is complete.
- Tasks do not terminate.
- All task communication is via protected shared variables (predominantly using protected objects). [2]
- All protected shared variables are initialized during library-level elaboration code.

6.2.1 Program-wide Information Flow Analysis

Current technology supports data flow analysis, information flow analysis, and proof based on pre- and post-conditions and invariants, for sequential code only. The goal is to extend this to Ravenscar programs that include tasks, protected objects and interrupt handlers.

The data dependency information that is currently used to analyse sequential programs can be applied to each task and each interrupt handler in the concurrent program as an independent entity. Thus the existing tools and techniques can verify each thread of control in isolation, including its use of privately accessed global data. This then leaves only the issue of the verification of the interactions between the threads of control as represented by the set of protected shared variables.

[2] Editor Note: the ISO technical report updates the bullet to also consider the use of atomic objects, which are statically proven free of race conditions.

The protected shared variables are required to be initialized by the library-level elaboration code in order to ensure that uninitialized shared data is not used. If initialization were instead performed during the operation phase, a race condition could be introduced. For a suspension object, initialization is defined by the Ada standard to occur at the point of declaration. For a protected object or an atomic object, all fields should be initialized either as part of object elaboration, or using library-level package elaboration code. In conjunction with the use of **pragma** Partition_Elaboration_Policy(Sequential) this ensures that no thread of control can access any shared state that has not been fully initialized.

After the initialization phase is complete, the protected shared variables can be modelled for data and information flow analysis purposes if we assume that their data is volatile. Since the data can be updated at any time due to the effects of preemption and interrupt occurrence, any specific task's view of a protected shared variable must assume that the value may change at any time. For example, two successive reads by a task of a protected shared variable may deliver different results and similarly, the value read by a task following a write by the same task cannot be assumed to be the written value. This volatility is the same abstraction as that used to model access to external program data, such as that which has an address clause or is an imported variable (via the Import aspect). Thus, assuming that the static analysis technique supports access to volatile external data, concurrent access to protected data can be modelled in the same way. As a result, each thread of control can now be described both in terms of its sequential data and information flow, and in terms of its interactions with volatile protected shared variables.

Having obtained the analysis of each thread of control that includes its interactions with the protected state, it is then possible to combine the analyses to form the overall data and information flow for the program as a whole, across the task and interrupt handler boundaries. This allows the designer to make assertions about how the entire program should behave in terms of the effect that it has on its external inputs (including interrupts) to produce its external outputs. These assertions can then be verified by the analysis to the same degree of confidence as is currently achievable in a sequential program.

This form of static analysis does not address the timing or ordering properties of the program. Later sections in this chapter address these topics by describing the use of RTA and other forms of formal analysis, such as model checking, which can prove statically the timing properties of the program.

6.2.2 Absence of Run-time Errors

Existing static analysis techniques can be used to prove absence of run-time errors due to language-defined exceptions within sequential code. The corresponding guidance on the sequential code constructs that may be used to achieve this goal is contained in the Technical Report [GA]. These techniques can be independently applied to each individual thread of control (task, main program or interrupt handler) of a Ravenscar program.

In order to extend these existing techniques to a full Ravenscar program, it is necessary to address the various forms of run-time check failure that relate directly to the concurrency features. These can be broken down into the following groups:

- Errors during program elaboration, such as access-before-elaboration or use of uninitialized data.
- Errors after program elaboration is complete, during the normal operation phase of the application, in particular the exceptions that are cited in Sections 4.2.4 and 4.2.5.
- Erroneous behaviour during normal operation, in particular concurrent access to unprotected shared variables (see Section 4.2.7).
- Implementation-defined behaviour as a result of violation of the No_Task_Termination restriction.

The following sub-sections discuss various techniques that can be applied to verify statically that these forms of error cannot occur.

Elaboration Errors

Within a sequential program, detection of access before elaboration errors is generally straightforward during program development due to the repeatable nature of the elaboration order, and the raising of Program_Error exception at the point of failure, causing the program to terminate. Having obtained a correct elaboration order during development, this ordering is usually predictable except when a switch to a different compiler vendor, or an upgrade to a new product version from the same vendor that uses a different algorithm for any units that have implementation-defined ordering, is performed. This implicit order variation can be prevented by explicit use of elaboration order pragmas, once a correct order has been established.

Within a concurrent program however, access to global data that is not yet initialized by the elaboration code may occur as a result of race conditions that vary between development mode and deployment mode, due to factors such as the use of hardware of differing performance or memory access times, inclusion or exclusion of checking code, differences in interpretation of priority, scheduling variations etc. These race conditions are more likely to be present because of the Ada rule that a library-level task shall be activated by its master package prior to the execution of that master's body elaboration code, and also prior to the execution of the elaboration code of later library units in the overall program elaboration order. Another contributing factor to the race condition is that having completed its activation, the Ada task proceeds into its normal execution code, and so must be programmed to immediately suspend to prevent this code from executing whilst program elaboration is still incomplete. Similar concerns apply to the execution of interrupt handlers after attachment - an interrupt may trigger execution of a handler prior to completion of program elaboration, and in this case, the handler cannot be programmed to suspend, of course. Such an error may actually occur silently - the task or interrupt handler may read an uninitialized value of a shared variable and not cause any exception to be raised, even in the presence of **pragma** Normalize_Scalars.

There are several solutions that can mitigate this hazard statically. The most obvious one is to ensure that all shared variables of a Ravenscar program are initialized at the point of declaration. However this is inappropriate in the case when elaboration code in the body is needed to set a correct initial value. Logically, it is highly desirable if we can assert that the dynamic semantics of the program are unaffected whether global shared data is initialized at the point of declaration, or by library package body elaboration code, assuming a correct elaboration order for the sequential elaboration code has been enforced using elaboration control pragmas.

In order to achieve the static guarantee that all library units have been elaborated prior to the activation of any task and prior to the invocation of any interrupt handler, the Partition_Elaboration_Policy pragma was added to the Ada standard. If this pragma is used with argument Sequential, then all task activation and interrupt handler attachment is deferred until after all program elaboration code is complete, i.e. just prior to the call of the main subprogram (see also Section 4.2.7).

Execution Errors Causing Exceptions

Sections 4.2.4 and 4.2.5 identify the concurrency-related run-time checks that are required of a conformant implementation of the Ravenscar Profile. In the following sub-sections, we examine techniques for static elimination of these error conditions.

Max_Entry_Queue_Length and Suspension Object Check

The static detection of absence of entry queue length violation may be achieved by applying further constraints on the application code, namely that at most one task object can call each protected entry. This also implies that the task objects, protected objects and protected entries are statically identified. Static identification of an object excludes its name being determined dynamically such as via a function result, a dynamic array index, the dereferencing of an access value etc. A less restrictive scheme that shows that there is no program state in which more than one task may be calling the same protected object would require more extensive analysis, such as the use of model checking (see Section 6.4). The same approach can be applied to the static detection of absence of more than one task waiting on each suspension object at any time.

Priority Ceiling Violation Check

The static detection of absence of priority ceiling violation can be achieved assuming the following further constraints:

- all task objects and protected objects have a static priority (this may be supplied via a static expression of a type discriminant for example);
- the protected object call chain (including nested protected object calls) that is made by each task object and each interrupt handler is statically determinable, by requiring static identification of the target protected object in all cases.

Potentially Blocking Operations in a Protected Action

The static detection of absence of execution of a potentially blocking operation within a protected action is feasible given the additional constraint on the use of indirect subprogram calls, which then allows the call trees to be statically determined. The presence of any of the following constructs in any protected or subprogram body in the call tree that is rooted in a protected operation body would then be statically disallowed:

- a protected entry_call_statement;
- a delay_statement;
- a call to Ada.Synchronous_Task_Control.Suspend_ Until_True;
- a call to any other language-defined subprogram that is defined to be potentially blocking [RM 9.5.1 (8-16)].

In addition, the determination of the call trees would enable static detection of an external subprogram call with the same target protected object as that of the protected action, assuming the restriction that the target protected object is always statically identified.

A slightly less restrictive scheme may be possible that uses formal verification methods such as model checking (see Section 6.4) to determine if a program state exists such that a protected action would cause execution of a potentially blocking operation (which may be within conditionally-executed code, although this style is not recommended).

It may also be possible to support detection of potentially blocking operations in the presence of indirect procedure calls if a pre-condition that specifies a non-blocking property is asserted prior to each indirect call, and that property is shown to be satisfied statically by all possible procedures that can be invoked by that call. Similarly, the check for circularity in the protected object call chain may be possible even in the case of non-statically identified protected objects, by imposing a pre-condition that none of the potentially called protected objects invoke operations of any protected objects that are higher in the call chain.

Task Termination

The Ravenscar Profile defines a static task set and prohibits dynamic task creation. The intent is that all tasks are created during program start-up, but in any mode of operation, some of them may be dormant, waiting on a synchronization event. A task that is no longer required to be executed would wait on its event indefinitely. In this model, task termination is considered to be an error case and hence the restriction No_Task_Termination is required by the Ravenscar Profile. The effect of violation of the No_Task_Termination restriction is implementation-defined.

Task termination within the restrictions of the Ravenscar Profile can occur only as a result of normal exit from the task body, or as a result of an unhandled exception.

- The case of avoidance of normal exit can be statically analysed if a coding restriction is placed on the task body code - the final statement must either be an infinite loop or else be a compound statement (such as a conditional or case statement) that can only cause an infinite loop to be executed.
- The case of showing absence of exceptions by static analysis has already been covered in Section 4.2.6 and in the sub-sections above.

The combination of these two techniques can be used to ensure statically that task termination cannot occur, and hence also that no implementation-defined behaviour that results from task termination can be invoked.

Use of Unprotected Shared Variables [3]

The intent of the Ravenscar Profile is that tasks and interrupt handlers should not make concurrent use of an unprotected shared variable - all interactions involving tasks or interrupt handlers are recommended to be via protected and atomic objects, where an atomic object is either a suspension object or one that has the Atomic aspect applied to it or its type. The avoidance of unprotected shared variables is generally a requirement of high integrity systems, although detection of this erroneous case is not mandated by the Ravenscar Profile definition.

The static detection of absence of unprotected shared variables can be achieved assuming the restriction that the use of all global variables of unprotected type by each task object and by each interrupt handler is statically identifiable. All global objects that are either of a protected type or an atomic type may be safely shared, and so no static identification is required for these. Static verification can then ensure that no unprotected global variable is accessed by more than one thread of control.

Note that if a task object or interrupt handler shares global data only with program elaboration code, i.e. the elaboration code initializes global data that is subsequently privately used by a single task or interrupt handler, then this data does not need to be protected if the Partition_Elaboration_Policy pragma is used with the argument Sequential, since this pragma ensures that the

[3] Editor note: the ISO technical report updates this section noting that appropriate use of atomic must be guaranteed, statically proven free of race conditions.

elaboration is complete prior to any task execution or interrupt attachment (and hence there can be no sharing violation).

6.3 Scheduling Analysis

The use of scheduling theory was mentioned in Chapter 1. Here we provide more details on the procedure to be followed. The aim is to introduce the form this analysis takes as it is not appropriate within this report to give a full tutorial on this material; such material can be found in text books (for example [9] and [10]). Ravenscar facilitates the use of these techniques as it supports priority-based dispatching and ceiling locking on protected objects. To apply these techniques, however, further constraints on application code must be made. All tasks must have a single invocation event and allow other parameters to be analysed or measured – see below.

In this section, priority assignment is considered first, then two forms of analysis are introduced: Rate Monotonic Analysis and Response Time Analysis.

6.3.1 Priority Assignment

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 depending on the temporal characteristics of the task and other factors such as criticality. For hard deadline tasks it is usually assumed that the following three parameters are known:

T – Period; time interval between consecutive arrivals of the task

D – Deadline; required latest completion time for the task (relative to its arrival)

C – Computation time; worst-case execution time needed for the task to complete one activation.

For periodic tasks, T is the time interval between releases. For sporadic tasks, T is the minimum inter-arrival time for the event that releases the task. The three parameters (T, D, C) are always given in the same time units. So (30ms, 20ms, 2.73ms) defines a task that (at maximum) is released every 30ms; must complete within 20ms; and that has a maximum computation time of 2.73ms. These latter values are obtained either by measurement or by some form of static timing analysis (or a combination of the two).

If all tasks are hard and criticality itself is not taken into account (because we require all tasks to always meet their deadline) then there is an optimal algorithm for assigning priority if D <= T for all tasks. By optimal we mean that the algorithm is as good as any other fixed priority scheme. The optimal algorithm is called Deadline Monotonic and simply assigns priority based on deadline – the shorter the deadline the higher the priority. In the special case when D = T for all tasks this scheme is known as Rate Monotonic.

An important property of fixed priority dispatching is that the lower priority tasks are the most vulnerable to missing a deadline if there is a run-time problem such as a task executing for more than its assumed maximum C. Because of this property the systems designer may wish to place the highly critical tasks at higher priorities than the Deadline Monotonic scheme would advise. This may reduce schedulability but is perfectly valid and is amenable to Response Time Analysis (see below).

Another reason to raise a task priority is to reduce *jitter* on input and/or output actions. Higher priority tasks have a more regular execution pattern and hence important events such as reading a sensor or writing to an actuator will occur with less variation from one period to the next. Scheduling analysis will only ensure that a task completes somewhere between its release and its deadline. One way of reducing jitter is thus to reduce the deadline of the tasks that perform jitter-sensitive I/O. If this is done, then the Deadline Monotonic priority assignment scheme will automatically allocate a higher priority.

Most scheduling schemes assume that each task is assigned a unique priority. Any Ada runtime for Ravenscar will support at least 32 priorities (and may indeed support many more). Although maximum schedulability does require distinct priorities for the tasks, it is unusual for an application to be so close to being unschedulable that it requires these unique priorities. Response Time Analysis can again deal with shared priority values. It should also be noted that that some real-time kernels can exploit the knowledge that tasks share priority to reduce the memory requirement. This is achieved by noting that two (or more) tasks that share a priority level never execute at the same time and hence can 'share' a task stack.

Once a priority map has been agreed for the set of tasks within the application the priorities for the protected objects can be assigned systematically.

6.3.2 Rate Monotonic Utilization-based Analysis

For a constrained set of temporal characteristics there exists a very simple schedulability test that quickly verifies if all deadlines will always be met. The constraints are that D=T for all tasks, and that priorities are assigned using the Rate Monotonic scheme. In practice this means that all tasks are hard and periodic. Each task must finish before its next release and there is no additional requirement to control jitter. If we assume, initially, that the program does not contain protected objects (i.e. all tasks execute independently) then the schedulability test is simply a matter of checking the utilization of the task set. For each task, the fraction of a complete processor it needs is given by C/T. If this is summed across all tasks this gives the total utilization of the application. Clearly, this value must not be more than 1.0 or the system is never going to be schedulable. The actual upper bound (which is less than 1.0) is given by the following formula, which is a function of n, the number of tasks in the system.

$$\sum_{i:=1}^{n} \left(\frac{C_i}{T_i}\right) \leq n(2^{1/n} - 1)$$

As n gets arbitrarily large, this expression converges on a single value. This is the famous 'Rate Monotonic' result, which says that a utilization of less than 0.69 will always furnish a schedulable system.

Once protected objects (POs) are introduced, blocking can occur. Here a task when released can be prevented from executing by the currently executing 'low' priority task running with a 'high' ceiling value while in a PO. For each task, the maximum blocking time, B, can be calculated. This is the maximum time a lower priority task can be executing with a priority equal or higher than the task currently under consideration. As noted in Chapter 1, the use of *Immediate Priority Ceiling Protocol* (*IPCP)* on POs does reduce blocking to its minimum value. The utilization test is now augmented with the result that each task must be examined in turn; so for task *j* [4]:

$$\sum_{i:=1}^{j}\left(\frac{C_i}{T_i}\right)+\frac{B_j}{T_j}\le j(2^{1/j}-1)$$

The blocking term for the lowest-priority task is 0, as it cannot suffer blocking by definition.

The simplicity of the utilization-based test makes it a very attractive one to use. Yet, it only applies to the constrained set of task characteristics. Moreover, it is a necessary but not sufficient test. If the application passes the test, all timing constraints will be met. If it fails the test, instead, the system may still be schedulable. A better test is needed in these circumstances. The following is one such example.

6.3.3 Response Time Analysis

Response time analysis is a general technique. It will deal with any priority assignment scheme and any relationship between D and T, (although its simple form requires D<=T). Moreover, it is a necessary and sufficient scheme for most situations. Like the utilization-based method it is easily incorporated into tools – many of which already exist.

The form of the analysis is quite straightforward. Firstly, the worst-case (longest) completion time for each task is calculated. This is known as the task response time, R. Secondly, these R values are compared, trivially, with the deadlines to make sure that R is less than D for all tasks. The response time equation is as follows (the *hp* function delivers the set of task with priority higher than task i):

$$R_i = C_i + B_i + \sum_{j\in hp(i)}\left\lceil\frac{R_i}{T_j}\right\rceil C_j$$

As ceiling functions are used, the unit for time is chosen so that all parameters are represented as integers.

The equation is solved by forming a recurrence relation:

[4] Editor note: this equation has been corrected in relation to the York Technical Report, and is as provided in the ISO technical report.

$$\omega_i^{k+1} := C_i + B_i + \sum_{j\in hp(i)}\left\lceil\frac{\omega_i^k}{T_j}\right\rceil C_j$$

The initial value of the iteration variable is the task's computation time. Iteration continues until either the same value is obtained on two successive iterations (in which case the response time has been calculated) or the value rises higher than the task's deadline (in which case the task is not schedulable).

The above description represents the 'textbook' version of the analysis. The engineering version requires extra terms to capture the *overhead* of actual implementation. Firstly, overheads such as context switches can be assigned to the task that caused them (by incorporating them into the C parameter). Next, the kernel overheads associated with manipulating the delay queue, handling clock interrupts and the releasing of tasks must be factored in. The specific form this takes will depend on the structure of the kernel – but the kernel must provide the data needed to model this overhead. This is a documentation requirement specified in the Real-Time Annex which is discussed further in the following section. For an example on how to include this term in the analysis see the textbooks [9] and [10]. Finally, the overheads incurred by the application's interrupts must be accounted for. We must know a bound on the arrival of such interrupts, and the execution time of each attached handler must be known. Putting these values together allows a set of interrupt overhead terms to be included in the Response Time Analysis.

The appropriate use of the Ravenscar Profile and the scheduling results outlined in the previous three sections provide a sound engineering basis for constructing high integrity real-time systems. The theory is mature and tool support is available.

6.2.4 Documentation Requirement on Run-time Overhead Parameters

There are a number of places in the Reference Manual where documentation requirements and metrics are required of an implementation. Those of most relevance to Ravenscar are:

- C.3 concerning the interrupt model
- D.2.3 (11 - 13) concerning maximum duration of priority inversion
- D.8 (33 - 45) concerning clock accuracy
- D.9 (8, 11, 13) concerning the precision of **delay until**
- D12 (5) concerning interrupt blocking
- D.12 (6-7) concerning overhead involved with the use of protected objects

Unfortunately, this is not a comprehensive list of the data needed to fully model the overheads caused by the run-time system. Typically also needed are:

- Cost of context switches between tasks
- Cost of handling delay queue operations

Both of these factors may, contingent on the implementation of queues with the run-time system, depend on the number of tasks in the application's program. Nevertheless, if timing analysis is to be used on a Ravenscar program, it is necessary to have one of the following:

- Evidence of all necessary parameters
- A means by which the programmer can measure these parameters
- Formulae by which these parameters can be calculated.

6.4 Formal Analysis of Ravenscar Programs

The Ravenscar profile supports only a simple concurrency model with the error conditions being relatively easy to avoid. For example, the use of shared resources (via projected objects with ceiling priorities) cannot lead to deadlock. Nevertheless, to gain a very high level of assurance it may be necessary to formally analyse a Ravenscar program. As outlined in Section 2.4, such analysis takes the form of either mechanized proof (via a theorem prover) or model checking.

There is already experience of using model checking to validate Ravenscar programs. It is possible to add worst-case and best-case execution times for state transitions and to then check that deadlines are never missed. Alternatively, model checking can be used to validate the top-level description of the timing constraints – leaving scheduling analysis to check deadline satisfaction once execution times from the implementation are known. Typical of the verification that can be achieved with this approach is to check some end-to-end deadline through a number of tasks assuming each task itself meets its timing requirements. Each task is represented by an automaton and each protected object by a shared variable (there are no problems with mutual exclusion in these formal models).

As with Ada itself, there can never be a formal map between a Ravenscar program and its model. However, the use of standard paradigms and libraries of associated (reusable) models allows a high integrity process to be defined.

This demonstrates that formal approach can be applied effectively to Ravenscar programs, but this does not imply that all high integrity Ravenscar programs need this level of verification. For many systems, static analysis of each task will be sufficient to generate the appropriate level of confidence.

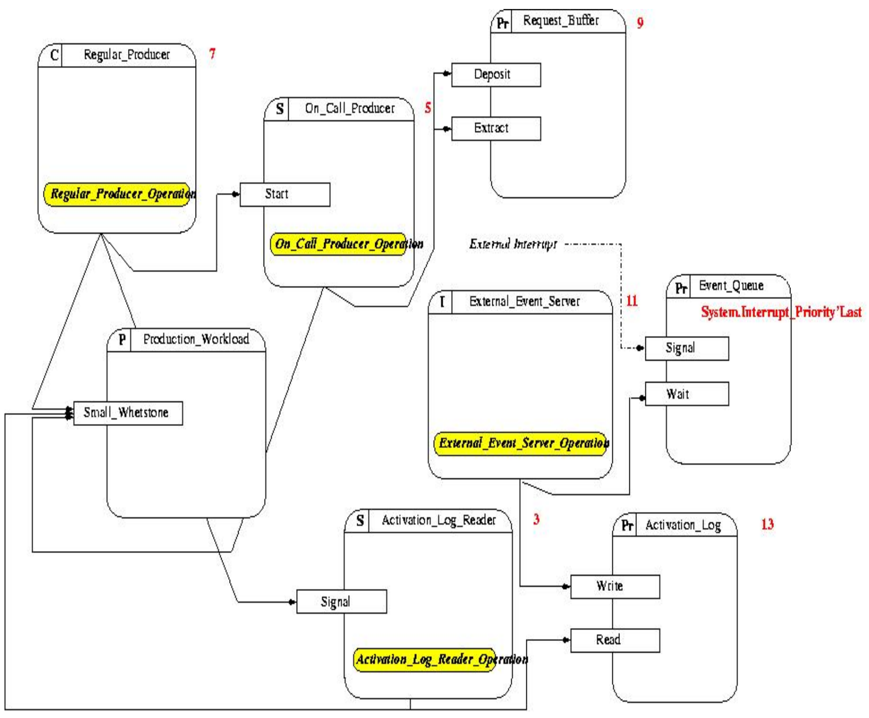
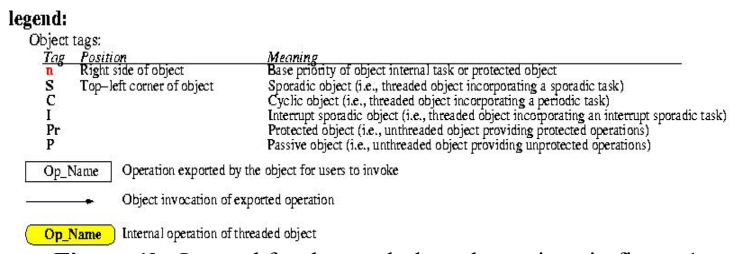


Figure 1. Schematic architecture of the example Ravenscar application.


legend:

Object tags:

Tag	*Position*	*Meaning*
n	Right side of object	Base priority of object internal task or protected object
S	Top–left corner of object	Sporadic object (i.e., threaded object incorporating a sporadic task)
C		Cyclic object (i.e., threaded object incorporating a periodic task)
I		Interrupt sporadic object (i.e., threaded object incorporating an interrupt sporadic task)
Pr		Protected object (i.e., unthreaded object providing protected operations)
P		Passive object (i.e., unthreaded object providing unprotected operations)

Op_Name Operation exported by the object for users to invoke

Object invocation of exported operation

Op_Name Internal operation of threaded object

Figure 1b: Legend for the symbols and notations in figure 1

7 Extended Example

The example presented in this chapter is designed to illustrate the expressive power of the Ravenscar Profile and the associated coding paradigms (discussed in Chapter 5), which aim to facilitate off-line scheduling analysis in the form outlined in Chapter 6.

The extended application example uses all of the concurrency components permitted by the Ravenscar Profile. The structure of the example models, on a reduced and simplified scale, the operation of real-world embedded real-time systems. The presentation of the example also outlines the information required for, and obtained from, the execution of deadline monotonic priority assignment and off-line scheduling analysis.

7.1 A Ravenscar Application Example

The example system includes a periodic process that handles orders for a variable amount of workload. Whenever the request level exceeds a certain threshold, the periodic process farms the excess load out to a supporting sporadic process. While such orders are executed, the system may receive interrupt requests from an external source. Each interrupt treatment records an entry in an activation log. When specific conditions hold, the periodic process releases a further sporadic process to perform a check on the interrupt activation entries recorded in the intervening period. The policy of work delegation adopted by the system allows the periodic process to ensure the constant discharge of a guaranteed level of workload. The correct implementation of this policy also requires assigning the periodic process a higher priority than those assigned to the sporadic processes, so that guaranteed work can be performed in preference to subsidiary activities.

Figure 1, overleaf, shows an HRT-HOOD [11] like representation of the system, while the legend, in figure 1b, recalls the meaning of the symbols and notations used in the diagram.

In HRT-HOOD terms, the system comprises:

- 4 active (i.e. threaded) objects respectively called: Regular_Producer, On_Call_Producer, Activation_Log_Reader, External_Event_Server;
- 1 passive (i.e. unthreaded) object called Production_Workload;
- 3 protected objects respectively called: Request_Buffer, Event_Queue, Activation_Log

The operation of the system proceeds as follows:

- Regular_Producer, which figure 1 tags as **C**yclic, embeds a fixed-rate periodic task that carries out a given amount of workload. The example represents the execution of this workload by the invocation of the well-known Small_Whetstone procedure exported by the shared passive object Production_Workload.
- When Regular_Producer determines that the required amount of workload exceeds its ceiling capacity, it delegates the excess workload out to On_Call_Producer. On_Call_Producer, which figure 1 tags as sporadic, embeds a sporadic task whose activation is specifically invoked to take over the excess workload of Regular_Producer.
- The sporadic activation and the associated workload transfer occur by means of a typical Ravenscar data-oriented synchronization: Regular_Producer invokes the Start operation exported by On_Call_Producer with a parameter characterising the service request. The Start operation enqueues the request in a private queue embedded within the protected object Request_Buffer. We need to protect the buffer because we allow new service requests to come in while the sporadic task is busy executing old ones. This follows from the decision to assign Regular_Producer a higher base priority than that of On_Call_Producer, which we opted for to ensure the discharge of a guaranteed level of workload in preference to the execution of subsidiary activities.
- A successful enqueueing releases the On_Call_Producer sporadic task, which indefinitely waits on an empty queue. The sporadic task fetches the request parameter from the top of the queue and performs the requested amount of workload in the same way as Regular_Producer. An invocation of Start fails when the queue held within Request_Buffer is full; for example, as a result of a (transient) rate of requests faster than service execution. Static analysis of the relationship between the maximum frequency of activation requests and the longest service time incurred by the sporadic task of On_Call_Producer should be used to prevent failure events of this kind.
- While the system carries out the required level of workload (whether regular or excess), an external device may occasionally raise an interrupt to signal its call for attention. In keeping with the Ravenscar programming model, the example application maps the arrival of the external interrupt to the invocation of a protected procedure. Object Event_Queue exports the procedure in question, which we call Signal.
- The service associated with the raising of the interrupt is carried out by the sporadic task embedded in External_Event_Server, which is tagged interrupt-activated sporadic. To simplify the coding of the example, and in keeping with the programming model that minimizes the amount of activity performed at interrupt priority, we have limited the extent of this interrupt service to the storing of an activation record in a protected buffer. The recording occurs by invocation of procedure Write exported by protected object Activation_Log. The use of a protected buffer to hold the activation record offers the natural mechanism to preserve data integrity in the face of independent read and write activities.
- In order for the system to monitor the arrival of service requests from the external device, when certain conditions hold, the periodic process embedded in

Regular_Producer requests the task embedded in the Sporadic object Activation_Log_Reader to examine the latest activation record stored by the interrupt service carried out by External_Event_Server. Activation_Log_Reader does this by invoking the Read procedure of Activation_Log. This style of work partitioning between Regular_Producer and Activation_Log_Reader uses the Ravenscar concurrency mechanisms to allocate activities with differing degrees of importance to distinct tasks. This approach aids system modelling. It also favours the specialization of tasks, which is a way of using the Ravensvar Profile definition to facilitate static analysis of the system.

- The activation request issued by Regular_Producer for this purpose uses the other form of synchronization permitted by the Ravenscar Profile: the data-less synchronization supported by suspension objects. Procedure Signal exported by Activation_Log_Reader performs this synchronization on a suspension object internally held by the object. As HRT-HOOD provides no specific object representation for suspension objects, we have used the convention that procedures by the name Signal exported by sporadic objects be understood as implemented by invocation of a private suspension object embedded within the object. Conversely, procedures by the name Start exported by sporadic objects are implemented by invocation of the Deposit procedure exported by an associated protected object. (Note that Signal is also the name of the protected procedure attached to an interrupt, which dispatches the activation event to interrupt-activated sporadic objects.)

7.2 Code

The Ravenscar Profile model does not inherently require the application to use any particular coding style for the execution of cyclic and sporadic tasks, protected objects, and interrupt handlers. However, if the application is required to pass schedulability analysis, certain task templates (patterns or stereotypes) and corresponding coding styles are useful in defining the activities that are to be analysed. These task templates are described in Chapter 5 and are used to code the example application outlined above.

Note that, in order to emphasise the stereotype nature of the task templates in the example, we have relegated all the parametric components of the application code into support packages named with "_Parameters" trailer added to the name of the corresponding base package. (The code of these support packages is provided in the closing section of this example.)

The Ravenscar-compliant HRT-HOOD coding convention has individual terminal objects in the system implemented as distinct library-level packages that carry the name of the corresponding object. An HRT-HOOD terminal object is one that cannot be further decomposed and therefore contains at most one type of primitive Ravenscar concurrency component. As each package, associated with a terminal object, by definition contains a single task or protected object, the corresponding entity carries the name of the enclosing package (and thus of the corresponding object).

Cyclic Task

The example uses one cyclic task, named Regular_Producer, the code of which is shown below. The non-suspending operation of Regular_Producer and its supporting definitions are defined in the Regular_Producer_Parameters package shown at the end of this section.

Regular_Producer

```
with Regular_Producer_Parameters;
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
package Regular_Producer is
  task Regular_Producer
     with Priority =>
     Regular_Producer_Parameters.
                              Regular_Producer_Priority;
       -- Assigned by deadline monotonic analysis
end Regular_Producer;

with Ada.Real_Time;
with Activation_Manager;
package body Regular_Producer is
   Period : constant Ada.Real_Time.Time_Span :=
        Ada.Real Time.Milliseconds
                      (Regular_Producer_Parameters.
                                    Regular_Producer_Period);
   task body Regular_Producer is
      use Ada.Real_Time;
      -- For periodic suspension
      Next_Time : Ada.Real_Time.Time;
   begin
      -- For tasks to achieve simultaneous activation
      Activation_Manager.Activation_Cyclic(Next Time);
      loop
         Next_Time := Next_Time + Period;
         -- Non-suspending operation code
         Regular_Producer_Parameters.
                              Regular_Producer_Operation;
         -- Time-based activation event
         delay until Next_Time; -- Delay statement at
                                -- end of loop
      end loop;
   exception
      when Error : others =>
         --  Last rites: for example
         Ada.Text_IO.Put_Line
           ("Something has gone wrong here: " &
              Exception_Information (Error));
   end Regular_Producer;
end Regular_Producer;
```

Event-response (Sporadic) Tasks

The example application includes three sporadic tasks, one per type of sporadic activation permitted by the profile: the activation of On_Call_Producer uses a protected object with a suspending entry; the activation of Activation_Log_Reader uses a suspension object; and the activation of External_Event_Server uses a protected object with a suspending entry attached to an interrupt. We first look at the code of the respective sporadic tasks and then turn our attention to the corresponding synchronization objects.

The non-suspending operation of On_Call_Producer and its supporting definitions are defined in the On_Call_Producer_Parameters package shown at the end of this section.

On_Call_Producer

```
with On_Call_Producer_Parameters;
package On_Call_Producer is
  -- Non-suspending operation with queuing of data
  function Start(Activation_Parameter : Positive)
    return Boolean;
  task On_Call_Producer
    with Priority =>
      On_Call_Producer_Parameters.
                              On_Call_Producer_Priority;
      -- Assigned by deadline monotonic analysis
end On_Call_Producer;

with Request_Buffer;
with Activation_Manager;
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
package body On_Call_Producer is
  -- To hide the implementation of the event buffer
  function Start(Activation_Parameter : Positive)
    return Boolean is
  begin
    return Request_Buffer.Deposit(Activation_Parameter);
  end Start;
  task body On_Call_Producer is
    Current_Workload : Positive;
  begin
    -- For tasks to achieve simultaneous activation
    Activation_Manager.Activation_Sporadic;
    loop
      -- Suspending request for activation event with
      -- data exchange
      Current_Workload := Request_Buffer.Extract;
      -- Non-suspending operation code
      On_Call_Producer_Parameters.
         On_Call_Producer_Operation (Current_Workload);
    end loop;
  exception
    when Error : others =>
      --  last rites: for example
      Ada.Text_IO.Put_Line
        ("Something has gone wrong here: " &
          Exception_Information (Error));
  end On_Call_Producer;
end On_Call_Producer;
```

The non-suspending operation of Activation_Log_Reader and its supporting definitions are defined in the Activation_Log_Reader_Parameters package shown at the end of this section.

Activation_Log_Reader

```
with Activation_Log_Reader_Parameters;
package Activation_Log_Reader is
  -- non-suspending parameterless operation
  --+ with no queuing of activation requests
 procedure Signal;
  task Activation_Log_Reader
    with Priority =>
      Activation_Log_Reader_Parameters.
                       Activation_Log_Reader_Priority;
      --  assigned by deadline monotonic analysis
end Activation_Log_Reader;

with Ada.Synchronous_Task_Control;
with Activation_Manager;
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
package body Activation_Log_Reader is
  Local_Suspension_Object :
     Ada.Synchronous_Task_Control.Suspension_Object;
  procedure Signal is
  begin
    Ada.Synchronous_Task_Control.
                     Set_True(Local_Suspension_Object);
  end Signal;
  procedure Wait is
  begin
    Ada.Synchronous_Task_Control.
          Suspend_Until_True (Local_Suspension_Object);
  end Wait;
  task body Activation_Log_Reader is
  begin
    -- for tasks to achieve simultaneous activation
    Activation_Manager.Activation_Sporadic;
    loop
      -- suspending parameterless request of
      -- activation event
      Wait;
      -- non-suspending operation code
      Activation_Log_Reader_Parameters.
                     Activation_Log_Reader_Operation;
    end loop;
  exception
    when Error : others =>
      --  last rites: for example
      Ada.Text_IO.Put_Line
        ("Something has gone wrong here: " &
          Exception_Information (Error));
  end Activation_Log_Reader;
end Activation_Log_Reader;
```

The non-suspending operation of External_Event_Server and its supporting definitions are defined in the External_Event_Server_Parameters package shown at the end of this section.

External_Event_Server

```
with External_Event_Server_Parameters;
package External_Event_Server is
  task External_Event_Server
    with Priority =>
      External_Event_Server_Parameters.
                       External_Event_Server_Priority;
end External_Event_Server;

with Event_Queue;
with Activation_Manager;
with Ada.Text_IO;
with Ada.Exceptions; use Ada.Exceptions;
package body External_Event_Server is
  procedure Wait renames Event_Queue.Handler.Wait;
  task body External_Event_Server is
```

```
begin
   -- for tasks to achieve simultaneous activation
   Activation_Manager.Activation_Sporadic;
   loop
      -- suspending request for external activation event
      Wait;
      -- non-suspending operation code
      External_Event_Server_Parameters.
                                 Server_Operation;
   end loop;
  exception
   when Error : others =>
      --  last rites: for example
      Ada.Text_IO.Put_Line
        ("Something has gone wrong here: " &
          Exception_Information (Error));
  end External_Event_Server;
end External_Event_Server;
```

Shared Resource Control Protected Object

The example application uses one protected object, named Activation_Log, to control access to a shared resource. The auxiliary package Activation_Log_Parameters shown at the end of this section defines all the parameters that characterize the activity of Activation_Log.

Activation_Log

```
with Activation_Log_Parameters;
with Ada.Real_Time;
package Activation_Log is
  type Range_Counter is mod 100;
  protected Activation_Log
    with Priority =>
     Activation_Log_Parameters.Activation_Log_Priority
   -- Must be ceiling of users' priority
  is
   -- Records interrupt service activation:
   -- non-suspending operation
   procedure Write;
   -- Retrieves the last activation record:
   -- non-suspending operation
   procedure Read
       (Last_Activation  : out Range_Counter;
        Last_Active_Time : out Ada.Real_Time.Time);
  private
   Activation_Counter : Range_Counter := 0;
   Activation_Time : Ada.Real_Time.Time;
  end Activation_Log;
  procedure Write renames Activation_Log.Write;
  procedure Read
      (Last_Activation  : out Range_Counter;
       Last_Active_Time : out Ada.Real_Time.Time)
    renames Activation_Log.Read;
end Activation_Log;

package body Activation_Log is
  protected body Activation_Log is
   procedure Write is
   begin
      Activation_Counter := Activation_Counter + 1;
      Activation_Time := Ada.Real_Time.Clock;
   end Write;
   procedure Read(Last_Activation  : out Range_Counter;
              Last_Active_Time : out Ada.Real_Time.Time)
   is
    begin
      Last_Activation := Activation_Counter;
      Last_Active_Time := Activation_Time;
    end Read;
  end Activation_Log;
end Activation_Log;
```

Task Synchronization Primitives

The suspension object is the optimized form for a simple suspend/resume operation. The package Ada.Synchronous_Task_Control 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. We have seen an example of use of the former in the code of Activation_Log_Reader shown above, whereby the Activation_Log_Reader package exports a Signal procedure that invokes Set_True on the local suspension object on which the Activation_Log_Reader sporadic task suspends by invoking Suspend_Until_True within the call to its internal Wait operation.

As mentioned earlier, the activation of On_Call_Producer is controlled by the use of a protected object named Request_Buffer, which provides a suspending entry named Extract and a releasing procedure named Deposit.

The auxiliary package Request_Buffer_Parameters shown at the end of this section defines all the parameters that characterize the activity of Request_Buffer.

Request_Buffer

```
package Request_Buffer is
  function Deposit(Activation_Parameter : Positive)
     return Boolean;
  function Extract return Positive;
end Request_Buffer;

with Request_Buffer_Parameters;
package body Request_Buffer is
  type Request_Buffer_Index is
    mod Request_Buffer_Parameters.
                         Request_Buffer_Range;
  type Request_Buffer_T is
     array(Request_Buffer_Index) of Positive;
  protected Request_Buffer is
   with Priority =>
     Request_Buffer_Parameters.Request_Buffer_Priority
     --  Must be ceiling of users' priority
  is
    procedure Deposit
      (Activation_Parameter : Positive;
       Response           : out Boolean);
    entry Extract(Activation_Parameter : out Positive);
  private
    My_Request_Buffer : Request_Buffer_T;
    Insert_Index : Request_Buffer_Index :=
                         Request_Buffer_Index'First;
    Extract_Index : Request_Buffer_Index :=
                         Request_Buffer_Index'First;
    -- The Request_Buffer is initially empty
    Current_Size : Natural := 0;
    -- The guard is initially closed
    -- so that the first call to Extract will block
    Barrier : Boolean := False;
  end Request_Buffer;
  -- We encapsulate the call to protected procedure
  -- Deposit in a function
  -- that returns a Boolean value designating the
  -- success or failure of
  -- the operation. This coding style allows for
  -- a more elegant coding
  -- of the call
  function Deposit(Activation_Parameter : Positive)
    return Boolean is
    Response : Boolean;
  begin
    Request_Buffer.Deposit(
                     Activation_Parameter, Response);
    return Response;
  end Deposit;
  -- We encapsulate the call to protected entry Extract
  -- in a function
  -- that returns the Positive value designating the
  -- workload level passed
  -- by Regular_Producer on to On_Call_Producer.
  -- This coding style allows
  -- for a more elegant coding of the call
  function Extract return Positive is
    Activation_Parameter : Positive;
  begin
    Request_Buffer.Extract(Activation_Parameter);
    return Activation_Parameter;
  end Extract;

protected body Request_Buffer is
    entry Extract(Activation_Parameter : out Positive)
      when Barrier is
    begin
      Activation_Parameter :=
                     My_Request_Buffer(Extract_Index);
      Extract_Index := Extract_Index + 1;
      Current_Size := Current_Size - 1;
      -- We close the barrier when the buffer is empty
      -- this also prevents the counter from
      -- becoming negative
      Barrier := (Current_Size /= 0);
    end Extract;
    procedure Deposit
       (Activation_Parameter : Positive;
        Response             : out Boolean) is
    begin
      if Current_Size < Natural(Request_Buffer_Index'Last)
       then
        My_Request_Buffer(Insert_Index) :=
                             Activation_Parameter;
        Insert_Index := Insert_Index + 1;
        Current_Size := Current_Size + 1;
        Barrier := True;
        Response := True;
      else
        -- There is no room for insertion, hence the
        -- Deposit returns
        -- with a failure (we might have used as well
        -- an over-writing
        -- policy as long as the call returned)
        Response := False;
      end if;
    end Deposit;
  end Request_Buffer;
```

Interrupt Handler

The example system handles one external interrupt, which is serviced by the interrupt sporadic task External_Event_Server.

Event_Queue is the protected object that provides the Signal procedure attached to the interrupt and the Wait suspending entry invoked by External_Event_Server.

The auxiliary package Event_Queue_Parameters shown at the end of the section holds all the definitions required by Event_Queue.

Event_Queue

```
with External_Event_Server_Parameters;
package Event_Queue is
  protected Handler
   with Interrupt_Priority =>
     External_Event_Server_Parameters.
                         Event_Queue_Priority
     -- Must be in the range of System.Interrupt_Priority
  is
    procedure Signal
      with Attach_Handler => Some_Interrupt_Id;
    entry Wait;
  private
    -- The entry barrier must be simple
    -- (i.e. Boolean expression)
    Barrier : Boolean := False;
  end Handler;
end Event_Queue;

package body Event_Queue is
  protected body Handler is
    procedure Signal is
    begin
      Barrier := True;
    end Signal;
    entry Wait when Barrier is
    begin
      Barrier := False;
    end Wait;
  end Handler;
end Event_Queue;
```

7.3 Scheduling Analysis

In order to use the deadline monotonic algorithm to assign priorities to all tasks and protected objects in the above application example, we need to determine the respective real-time attributes. This is done in table 1.

As soon as we know the worst-case execution time of the non-suspending internal operations performed by the tasks of our example, we can use response time analysis to confirm the feasibility of the real-time attributes of the task set in table 1.

As we mentioned above and as figure 1 illustrates, the example application uses the Small_Whetstone algorithm to control the computational workload of Regular_Producer, On_Call_Producer and Activation_Log_Reader. The way this occurs is shown in the respective auxiliary packages.

Task name	***Task type***	***Period / Minimum inter-arrival time***	***Deadline***	***Execution time***	***Response time***	***Priority***
Regular_Producer	Cyclic	1000	500			7
On_Call_Producer	Sporadic	≥ 1,000	800			5
Activation_Log_Reader	Sporadic	≥ 1,000	1,000			3
External_Event_Server	Interrupt sporadic	5,000	100			11
Protected object name	***User tasks***				***Ceiling priority***	
Request_Buffer	Regular_Producer (Deposit), On_Call_Producer (Extract)				9	
Event_Queue	External interrupt (Signal), External_Event_Server (Wait)				System.Interrupt_ Priority'Last	
Activation_Log	External_Event_Server (Write), Activation_Log_Reader (Read)				13	

Table 1. Real-time attributes of tasks and protected objects in example application. All time values are in milliseconds.

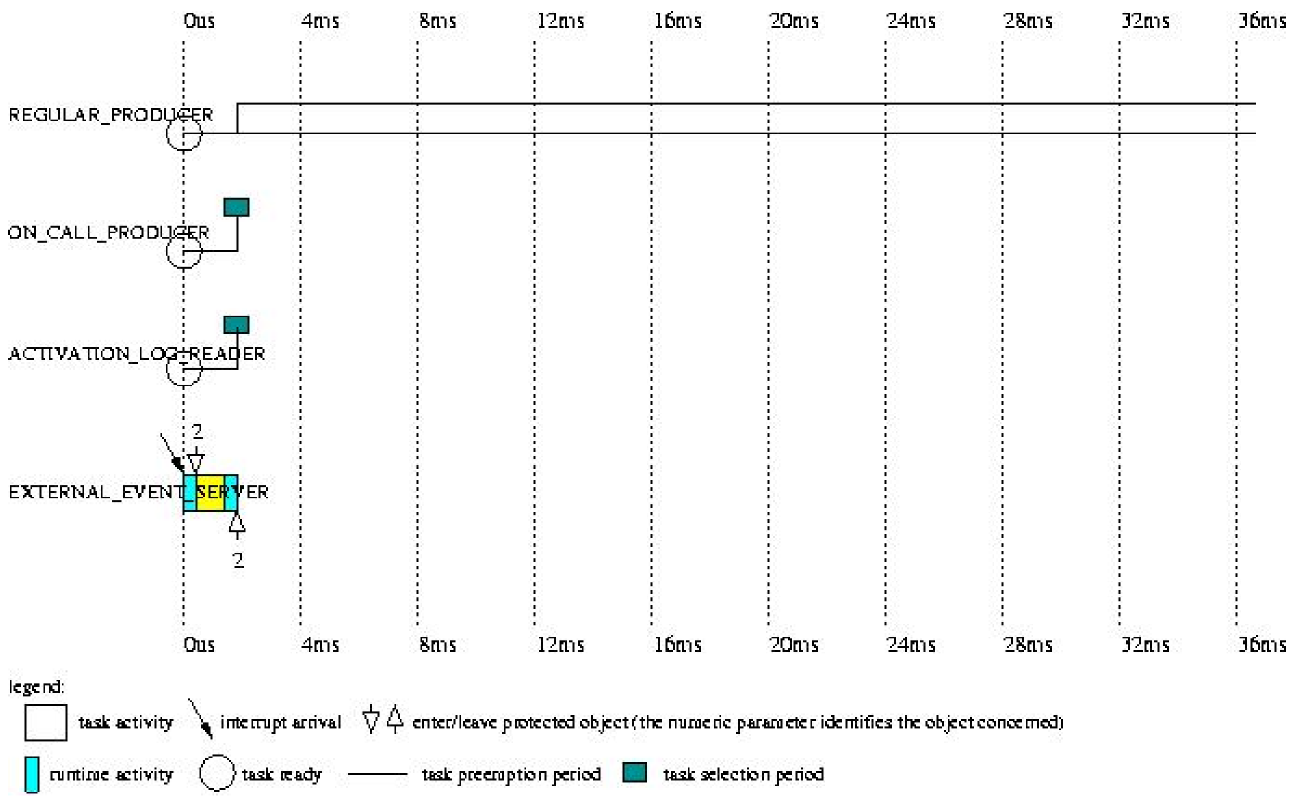


Figure 2. Schedule of task execution near the time of system activation.

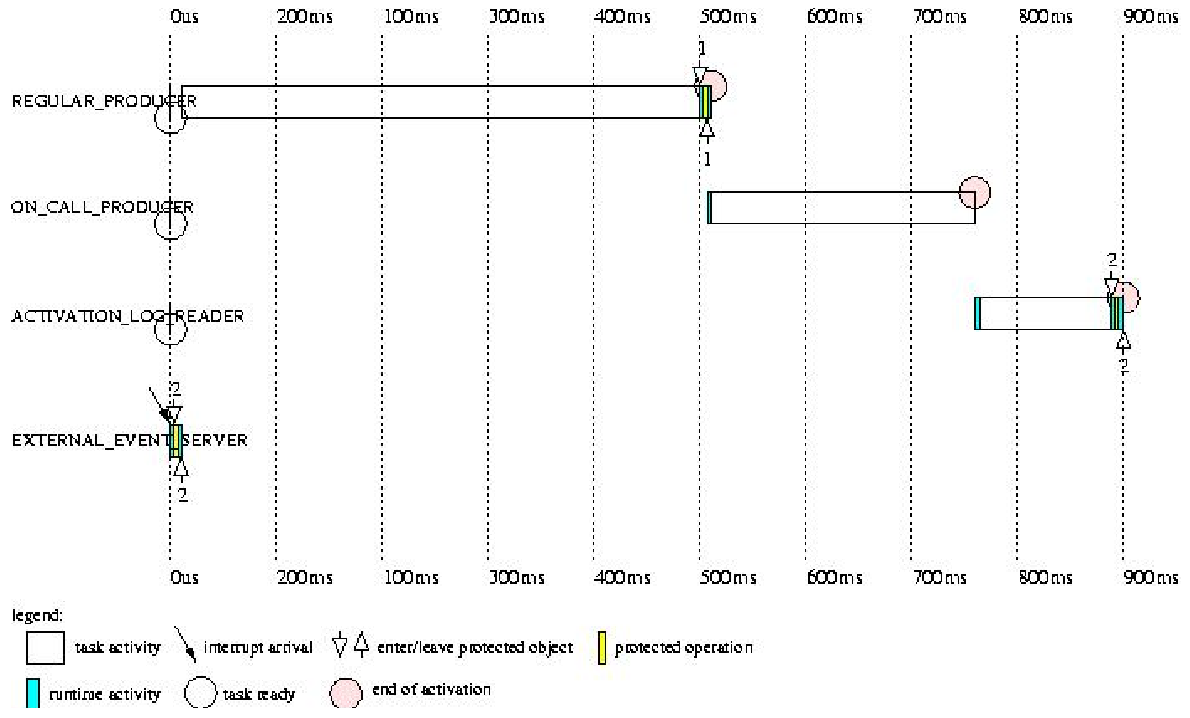


Figure 3. Schedule of execution for one complete activation of all tasks in the example application.

Knowing the processing power of the designated target processor and the runtime overheads associated to the execution of the Ravenscar tasking model (e.g. select and context switch time; insert and remove from delay queue; insert and remove from single-position entry queue) we may achieve precise estimates of the required execution time for all tasks and thus allow the use of response time analysis.

By way of example, for one particular assignment of computational workload to the tasks in the system and for the priority assignment shown in table 1, we obtain the schedule of execution shown in figure 2 for the region near the time of system activation (which assumes the arrival of the 1st external interrupt at notional time 0) and in figure 3 for one complete activation of all tasks in the task set.

7.4 Auxiliary Code

The auxiliary code includes the various operation parameter packages referred to in the earlier descriptions. It also includes the Activation_Manager, which provides a common time reference to all tasks in the application, and the specification of the Production_Workload package, which provides the synthetic workload that keeps the application tasks busy. The body of that package (which is the well-known Whetstone benchmark) is omitted.

Regular Producer operation parameters

```
with System;
package Regular_Producer_Parameters is
  Regular_Producer_Priority :
     constant System.Priority := 7;
  Regular_Producer_Period :
     constant Natural := 1_000; -- in milliseconds
  procedure Regular_Producer_Operation;
end Regular_Producer_Parameters;
```

```
with Auxiliary;
with On_Call_Producer;
with Production_Workload;
with Activation_Log_Reader;
with Ada.Text_IO;
package body Regular_Producer_Parameters is
  -- approximately 5,001,000 processor cycles of
  -- Whetstone load
  -- on an ERC32 (a radiation-hardened SPARC for
  -- space use) at 10 Hz
  Regular_Producer_Workload : constant Positive := 756;
  -- approximately 2,500,500 processor cycles
  On_Call_Producer_Workload : constant Positive := 278;
  -- the parameter used to query the condition
  -- for the activation of On_Call_Producer
  Activation_Condition :
     constant Auxiliary.Range_Counter := 2;
  procedure Regular_Producer_Operation is
  begin
    -- we execute the guaranteed level of workload
    Production_Workload.
          Small_Whetstone(Regular_Producer_Workload);
    -- then we check whether we need to farm excess
    -- load out to
    -- On_Call_Producer
    if Auxiliary.Due_Activation(Activation_Condition) then
      -- if yes, then we issue the activation request with
      -- a parameter
      -- that determines the workload request
      if not On_Call_Producer.
                   Start(On_Call_Producer_Workload) then
        -- we capture and report failed activation
        Ada.Text IO.Put_Line("Failed sporadic activation.");
      end if;
    end if;
    -- we check whether we need to release Activation_Log
    if Auxiliary.Check_Due then
      Activation_Log_Reader.Signal;
    end if;
    -- finally we report nominal completion of the
    -- current activation
    Ada.Text_IO.Put_Line("End of cyclic activation.");
  end Regular_Producer_Operation;
end Regular_Producer_Parameters;
```

On_Call_Producer operation parameters

```
with System;
package On_Call_Producer_Parameters is
  On_Call_Producer_Priority :
     constant System.Priority := 5;
  procedure On_Call_Producer_Operation(
                                              Load : Positive);
end On_Call_Producer_Parameters;

with Production_Workload;
with Ada.Text_IO;
package body On_Call_Producer_Parameters is
  procedure On_Call_Producer_Operation(Load : Positive)
is
  begin
    -- we execute the required amount of excess workload
    Production_Workload.Small_Whetstone(Load);
    -- then we report nominal completion of
    -- current activation
    Ada.Text_IO.Put_Line("End of sporadic activation.");
  end On_Call_Producer_Operation;
end On_Call_Producer_Parameters;
```

Activation_Log operation parameters

```
with System;
package Activation_Log_Parameters is
  Activation_Log_Priority : constant System.Priority := 3;
end Activation_Log_Parameters;
```

Activation_Log_Reader operation parameters

```
with System;
package Activation_Log_Reader_Parameters is
  Activation_Log_Reader_Priority :
      constant System.Priority := 3;
  procedure Activation_Log_Reader_Operation;
end Activation_Log_Reader_Parameters;

with Production_Workload;
with Activation_Log;
with Ada.Real_Time;
with Ada.Text_IO;
package body Activation_Log_Reader_Parameters is
  -- approximately 1,250,250 processor cycles of
  -- Whetstone load on an ERC32 (a radiation-hardened
 -- SPARC for space  use) at 10 Hz
  Load : constant Positive := 139;
  procedure Activation_Log_Reader_Operation is
    Interrupt_Arrival_Counter :
                    Activation_Log.Range_Counter := 0;
    Interrupt_Arrival_Time : Ada.Real_Time.Time;
  begin
    -- we perform some work
    Production_Workload.Small_Whetstone(Load);
    -- then we read into the Activation_Log buffer
    Activation_Log.Activation_Log.
                    Read(Interrupt_Arrival_Counter,
                         Interrupt_Arrival_Time);
    -- and finally we report nominal completion of
    -- current activation
    Ada.Text_IO.
      Put_Line("End of parameterless sporadic activation.");
  end Activation_Log_Reader_Operation;
end Activation_Log_Reader_Parameters;
```

External_Event_Server operation parameters

```
with Ada.Interrupts.Names;
with System;
package External_Event_Server_Parameters is
  -- a target-specific interrupt
  The_Interrupt : constant Ada.Interrupts.Interrupt ID :=
     Ada.Interrupts.Names.External_Interrupt_2;
  -- the interrupt priority should be at the appropriate level
  -- (we set it to 'Last because the example handles no
  --  other interrupts)
  Event_Queue_Priority :
     constant System.Interrupt_Priority :=
                          System.Interrupt_Priority'Last;
  -- the interrupt sporadic priority is determined by deadline
  -- monotonic analysis
  External_Event_Server_Priority :
     constant System.Priority := 11;
  procedure Server_Operation;
end External_Event_Server_Parameters;

with Activation_Log;
package body External_Event_Server_Parameters is
  procedure Server_Operation is
  begin
    -- we record an entry in the Activation_Log buffer
    Activation_Log.Write;
  end Server_Operation;
end External_Event_Server_Parameters;
```

Request_Buffer operation parameters

```
with System;
package Request_Buffer_Parameters is
  -- the request buffer priority must ceiling of its
  -- users' priorities
  Request_Buffer_Priority : constant System.Priority := 9;
  -- proper analysis will determine the appropriate size
  -- of the request
  -- buffer
  Request_Buffer_Range : constant Positive := 5;
end Request_Buffer_Parameters;
```

The Activation_Manager provides two facilities.

- A common epoch for all tasks in the system.
- A mechanism for all tasks to suspend until a common time, in order to achieve a co-ordinated release after elaboration. This achieves the effect of **pragma** Partition_Elaboration_Policy(Sequential);.

Activation_Manager internals

```
with Ada.Real_Time;
package Activation_Manager is
  use Ada.Real_Time;
  function Clock return Ada.Real_Time.Time
     renames Ada.Real_Time.Clock;
  -- global start time relative to which all periodic events
  -- in system will be scheduled
  System_Start_Time :
      Ada.Real_Time.Time : constant := Clock;
  -- relative offset of task activation after elaboration
  -- (milliseconds)
  Relative_Offset : constant Natural := 100;
  Task_Start_Time :
    constant Ada.Real_Time.Time_Span :=
            Ada.Real_Time.Milliseconds(Relative_Offset);
  -- absolute time for synchronization of task activation
  -- after elaboration
  Activation_Time : constant Ada.Real_Time.Time :=
    System_Start_Time + Task_Start_Time;
  procedure Activation_Sporadic;
  procedure Activation_Cyclic
      (Next_Time : out Ada.Real_Time.Time);
end Activation_Manager;

package body Activation_Manager is
  procedure Activation_Sporadic is
  begin
    delay until Activation_Time;
  end Activation_Sporadic;
  procedure Activation_Cyclic
      (Next_Time : out Ada.Real_Time.Time) is
  begin
    Next_Time := Activation_Time;
    delay until Activation_Time;
  end Activation_Cyclic;
end Activation_Manager;
```

Auxiliary definitions and services

```
package Auxiliary is
  type Range_Counter is mod 5;
  function Due_Activation(Param : Range_Counter)
    return Boolean;
  type Run_Counter is mod 1_000;
  Factor : constant Natural := 3;
  function Check_Due return Boolean;
end Auxiliary;

package body Auxiliary is
  Request_Counter : Range_Counter := 0;
  Run_Count : Run_Counter := 0;
  -- we establish an arbitrary criterion for the activation of
  -- On_Call_Producer
  function Due_Activation(Param : Range_Counter) return
Boolean is
  begin
    Request_Counter := Request_Counter + 1;
    -- we make an activation due according to the caller's
    -- input parameter
    return (Request_Counter = Param);
  end Due_Activation;
  -- we establish an arbitrary criterion for the activation of
  -- Activation_Log_Reader
  function Check_Due return Boolean is
    Divisor : Natural;
  begin
    Run_Count := Run_Count + 1;
    Divisor := Natural(Run_Count) / Factor;
    -- we force a check due according to an
    -- arbitrary criterion
    return ((Divisor*Factor) = Natural(Run_Count));
  end Check_Due;
end Auxiliary;
```

Production_Workload (specification only)

```
package Production_Workload is
 procedure Small_Whetstone(Kilo_Whets : Positive);
end Production_Workload;
```

8 Definitions, Acronyms, and Abbreviations

Allocator
An Ada construct used to create an object dynamically [RM 4.8].

Atomic
An operation performed by a task which is guaranteed to produce the same effect as if it were executing in total isolation and without interruption.

Blocked
The state of a task when its execution is prevented, while waiting for mutually-exclusive access to a shared resource which is currently held by a lower priority task.

Bounded error
An implementation- or language-defined error in the application program whose effect is predictable and documented.

Ceiling priority
The priority of a shared resource. The static default priority of all processes that use the resource must be less than or equal to the ceiling priority.

Context switch
The replacement of one task by another as the executing task on a processor.

Critical region
A sequence of statements that must appear to be executed indivisibly.

Critical task
A task whose deadline is significant and whose failure to meet its deadline could cause system failure.

CSP (Communicating Sequential Processes)
A notation for specifying and analyzing concurrent systems.

CSS (Calculus of Communicating Systems)
An algebra for specifying and reasoning about concurrent systems.

Cyclic executive
A scheduler that uses procedure calls to execute each periodic process in a predetermined sequence at a predetermined rate.

Cyclic task
A task whose execution is repeated based on a fixed period of time, also known as a periodic task.

Deadline
The maximum time allowed to a task to produce a response following its invocation.

Deadlock
A situation where a group of tasks (possibly the whole system) block each other permanently.

Dynamic testing
An analysis method that determines properties of the software by observing its execution (cf. static analysis).

Erroneous execution
A program state in which execution of the program becomes unpredictable as the result of an error. The errors that result in this state are defined in the language reference manual [RM 1.1.5 (9-10)].

Environment Task
The implicit outermost task which executes the program elaboration code and then calls the main subprogram (if any) [RM 10.2 (8)].

Epilogue
The code executed by the Ada run-time system to service the entry queues as defined in RM 9.5.3(13).

Event-Triggered Task
A task whose invocation is triggered either by an asynchronous action by another task, or by an external stimulus such as an interrupt.

Finalization
An Ada operation which occurs for controlled objects at the point of their destruction [RM 7.6.1].

Firm deadline task
A task whose failure to meet a deadline does not necessarily cause a failure of the application program. There is no value in completing a firm task after its deadline.

Hard deadline task
A task whose failure to meet a deadline may cause a failure of the application program.

IPCP (Immediate Priority Ceiling Protocol, also known as **Priority Ceiling Emulation)**
A technique to minimize the blocking time for contention for shared resources, protected by a protected object. This is provided by the locking policy Ceiling_Locking in Ada [RM D.3].

Jitter
The variation in time between the occurrence of a periodic event and a period of the same frequency.

Library level
The level at which an object which has global accessibility [RM 3.10.2 (22)].

Livelock
A situation where several tasks (possibly comprising the whole system) remain ready to run, and execute, but fail to make progress.

Liveness
The property that a set of tasks will reach all desirable states.

Mode change
A change in operating characteristics of a system that requires a co-ordinated change in the operation of several different processes in the system.

Monitor
A module containing one or more critical regions; all variables that must be accessed under mutual exclusion are hidden and all procedure calls are guaranteed to execute with mutual exclusion.

Mutex
A locking mechanism used to ensure mutually exclusive access to a shared resource.

Non-critical task
A task with no strict timing requirements.

Overhead
The execution time within the Ada run-time system which must be included in the schedulability analysis.

PBPS (Priority-Based Preemptive Scheduling)
This ensures that, if a high priority task becomes ready to run when a lower priority task is executing on the processor, the high priority task will replace the lower priority task immediately as the executing task.

PCP (Priority Ceiling Protocol)
A set of techniques that bound the blocking time for contention for shared resources. One such protocol, implemented in Ada, is IPCP.

Periodic task
A task whose execution is repeated based on a fixed period of time, also known as a cyclic task.

Preemptive fixed priority scheduling
A scheduling method in which each process has a static priority and the scheduler ensures that the currently selected process is the ready process with the highest priority.

Priority inversion
This occurs when a high-priority task is blocked waiting for a shared resource (including the CPU itself) currently in use by a low-priority task.

Protected object
An Ada construct which is used to provide mutually-exclusive access to shared resources and as a task synchronization primitive.

Race condition
A timing condition that causes processes to operate in an unpredictable sequence so that operation of the system may be incorrect.

Ready
The state of a task when it is no longer suspended. The task, however, will not execute whilst all the available processor resource can be used by higher priority ready tasks.

RMA (Rate Monotonic Analysis)
A mathematical method based on utilization which is used to prove that a set of tasks with static (and simple) characteristics will meet its deadlines in the presence of PBPS.

RTA (Response Time Analysis)
A mathematical method based on calculating latest completion time which is used to prove that a set of tasks with static characteristics will meet its deadlines in the presence of PBPS.

Safety
The property that a set of tasks cannot reach any undesirable state from any desirable state.

Soft deadline task
A task whose failure to meet a deadline does not necessarily cause a failure of the application program. There is value in completing a soft task even if it has missed its deadline.

Sporadic task
An event-triggered task with defined minimum inter-arrival time.

Static analysis
A group of analysis techniques that determine properties of the system from analysis of the program code (cf. dynamic testing).

Suspended
The state of a task when its execution is stopped due to execution of a language-defined construct that waits for a given time (e.g. a delay statement) or an event.

Suspending operation
An operation which causes the current task to be suspended until released by another task, a timer event or an interrupt handler.

Suspension object
An Ada construct [RM D.10] which is used for basic task synchronization, i.e. suspend and resume, which do not involve data transfer.

Time triggered task
A task whose invocation is triggered by the expiry of a delay set by that task.

WG9
The Ada Working Group, ISO/IEC JTC1/SC22/WG9. It is the group tasked with the interpretation and maintenance of the Ada Language Standard.

Worst-case execution time
A maximum bound on the time required to execute some sequential code.

TR Acknowledgements

This report is the result of input from a number of people. The authors acknowledge the contributions of Peter Amey, Rod Chapman, Stephen Michell, Juan Antonio de la Puente, Phil Thornley, David Tombs, Jeff Cousins, Ed Schonberg, Joyce Tokar, Jorge Real and the permission by Aonix Inc (now PTC) to use sections of their cross-development guide for the ObjectAda/Raven® product as the textual basis of the initial version of this report.

References

[AI 249] Ravenscar Profile for high integrity systems, ARG, http://www.ada-auth.org/cgi-bin/cwsweb.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] RTCA Inc. (2011), DO-178C Software Considerations in Airborne Systems and Equipment Certification.

[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. Liu 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*, Proc. of the 11th International Real-Time Ada Workshop, 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