The journal for the international Ada community

Ada User Journal

Volume 44
Number 1
March 2023

Produced by Ada-Europe

Ada-Europe General Secretary

Dene Brown
SysAda Limited
Signal Business Center
2 Innotec Drive
BT19 7PD Bangor
Northern Ireland, UK

Tel: +44 2891 520 560
Email: Secretary@Ada-Europe.org
URL: www.ada-europe.org

Information on Subscriptions and Advertisements

Ada User Journal (ISSN 1381-6551) is published in one volume of four issues. The Journal is provided free of charge to members of Ada-Europe. Library subscription details can be obtained direct from the Ada-Europe General Secretary (contact details above). Claims for missing issues will be honoured free of charge, if made within three months of the publication date for the issues. Mail order, subscription information and enquiries to the Ada-Europe General Secretary.

For details of advertisement rates please contact the Ada-Europe General Secretary (contact details above).

ADA USER JOURNAL

Volume 44
Number 1
March 2023

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

We are starting another year and another volume of the Ada User Journal, Volume 44. A long history that started in 1989, when the publication was named Ada-Europe News. I suggest the reader to have a look at the full history of the publication, by visiting the dedicated “History” page on the AUJ webpage. For this new year, we expect to have the opportunity to publish a whole lot of interesting technical contributions, coming from the Work-in-Progress and Industrial tracks of the Ada-Europe International Conference on Reliable Software Technologies (AEiC 2023), as well as from the co-located workshops DeCPS and ADEPT. This year the AEiC conference will take place in Lisbon, Portugal, from the 13th to the 16th of June, and the reader is invited to participate (a Call for Participation is provided in the Forthcoming Events section).

In the present issue, we include the second part of the HILT’22 - Supporting a Rigorous Approach to Software Development Workshop, and the complete proceedings of the 2022 ADEPT: AADL by its practitioners Workshop.

HILT 2022 was the seventh in a series of conferences and workshops focused on the use of High Integrity Language Technology. The workshop was held in October 2022, with the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE’2022. The program included two keynotes and nine papers, of which four were published in the previous AUJ issue and five are now provided. They address topics related to formal methods, assurance, and to use cases.

The first edition of the ADEPT: AADL by its practitioners Workshop took place in June 2022, with AEiC 2022. We publish the proceedings of ADEPT, which include the eight papers that were presented in the workshop. As expectable, all the papers focus on the Architecture Analysis and Design Language (AADL), either by describing tools and methods that are somehow related to the language, or by presenting examples and experiences in using the language for modelling concrete applications. Given the success of the first edition, the second edition of the ADEPT workshop is organized again this year, with AEiC 2023.

In addition to the rather long technical part of this March issue, the reader will find the News Digest and the Calendar and Events sections, as usual prepared by their editors, respectively Alejandro Mosteo and Dirk Craeynest.

Antonio Casimiro
Lisboa
March 2023
Email: AUJ_Editor@Ada-Europe.org

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

[Messages without subject/newsgroups are replies from the same thread. Messages may have been edited for minor proofreading fixes. Quotations are trimmed where deemed too broad. Sender’s signatures are omitted as a general rule. —arm]

Preface by the News Editor

Dear Reader,

We are fast approaching the main Ada-Europe yearly event, the AEiC conference. You can get the gist of this year's plans in this post: [1].

For the technically minded, several intricate aspects of the language are discussed in, e.g., [2] and [3]. It seems one never stops learning new details about Ada!

Finally, from the ‘Ada-not-the-language’ department (usually very quiet), several resources that promise lots of fun: An opera based on the wonderful work of Sydney Padua, the steampunk comic where Ada Lovelace and Charles Babbage team up to fight crime (!) is in the making [4]. Based on the same duo of scientists, you can already get your hands on an educational tabletop game [5]. Lastly, you may want to check an impressive Ada Lovelace cosplay sewn entirely from scratch [6].

[1] “AEiC 2023 - Ada-Europe Conference - Final Deadline Approaching”, in Ada-related Events.

[2] “Ada Array Contiguity”, in Ada Practice.

[3] “Assignment Access Type with Discriminants”, in Ada Practice.

[4] “Babbage & Lovelace - The Opera”, in Ada-related Events.

[5] “Table Game”, in Ada and Education.

[6] “Ada Lovelace Cosplay”, in Ada in Jest.

Sincerely,
Alejandro R. Mosteo.

Ada-related Events

Babbage & Lovelace - The Opera

From: Simon Wright <simon@pushface.org>
Subject: Babbage & Lovelace - The Opera
Date: Mon, 16 Jan 2023 18:30:56 +0000
Newsgroups: comp.lang.ada

https://guerillaopera.org/repertoire/thrilling-adventures

From: Dirk Craeynest <dirk@orka.cs.kuleuven.be>
Date: Thu, 19 Jan 2023 12:17:08 -0000

> https://guerillaopera.org/repertoire/thrilling-adventures

As I reacted on Twitter when I saw Sydney Padua @sydneypadua announcing this opera adaption of her graphic novel "The Thrilling Adventures of Lovelace and Babbage":

-------start-quote-------

The late Robert Dewar, of #AdaProgramming language fame, would have loved this. For some history, look for "The Maiden and the Mandate" in https://ada-europe.org/archive/auj/auj-41-1-withcovers.pdf and https://adacore.com/adacore25...

-------end-quote-------

Those were hilariously funny fully staged musical performances at several ACM SIGAda and Ada-Europe conferences, which I was lucky enough to attend. It would be great if AdaCore could put online one of the video recordings that were made at the time.

Dirk

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

* 27th Ada-Europe Int. Conf. Reliable Software Technologies (AEiC 2023)

* June 13-16, 2023, Lisbon, Portugal, www.ada-europe.org/conference2023

Ada Stand at FOSDEM 2023

[Past event for the record. —arm]

From: Dirk Craeynest <dirk@orka.cs.kuleuven.be>
Subject: Ada Stand at FOSDEM 2023 - Sat 4 & Sun 5 Feb (was: No Ada DevRoom in FOSDEM 2023, alternative DevRooms and Ada-Europe) support
Date: Thu, 2 Feb 2023 13:13:26 -0000
Newsgroups: comp.lang.ada, fr.comp.lang.ada, comp.lang.misc

Reminder: FOSDEM 2023 takes place this weekend, Sat 4 and Sun 5, in Brussels, Belgium. See www.fosdem.org.

Even though we didn't manage to get an Ada DevRoom this year […], the Ada FOSDEM team has an Ada stand in the "Education" group on level 2 of building K at the ULB site, with theme "It's time to learn Ada!"

Looking forward to meet many Adaists!

Dirk Craeynest, Ada FOSDEM team

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

AEiC 2023 - Ada-Europe Conference - Final Deadline Approaching

[For the record, as the deadline is past. —arm]

From: Dirk Craeynest <dirk@orka.cs.kuleuven.be>
Subject: AEiC 2023 - Ada-Europe conference - Final Deadline Approaching
Date: Thu, 16 Feb 2023 09:39:33 -0000
Newsgroups: comp.lang.ada, fr.comp.lang.ada, comp.lang.misc

FINAL Call for Contributions

27th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2023)

13-16 June 2023, Lisbon, Portugal

www.ada-europe.org/conference2023

*** FINAL submission DEADLINE 27 February 2023 ***

Organized by Ada-Europe in cooperation with ACM SIGAda (approval pending) and the Ada Resource Association (ARA)

#AEiC2023 #AdaEurope #AdaProgramming

General Information

The 27th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2023) will take place in Lisbon, Portugal. The conference schedule comprises a journal track, an industrial track, a work-in-progress track, a vendor exhibition, parallel tutorials, and satellite workshops.

* Journal-track submissions present research advances supported by solid theoretical foundation and thorough evaluation.

* Industrial-track submissions highlight the practitioners' side of a challenging case study or industrial project.

* Work-in-progress-track submissions illustrate a novel research idea that is still at an initial stage, between conception and first prototype.

* Tutorial submissions guide attenders through a hands-on familiarization with innovative developments or with useful features related to reliable software.

Schedule

[CLOSED] Extended submission deadline for journal-track papers

27 February 2023: Submission deadline for industrial-track and work-in-progress-track papers, tutorial & workshop proposals

20 March 2023: First round notification for journal-track papers, acceptance notification for other submission types

13-16 June 2023: Conference

Scope and 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:

- Formal and Model-Based Engineering of Critical Systems;
- Real-Time Systems;
- High-Integrity Systems and Reliability;
- Ada Language;
- Applications in a variety of domains.

More specific topics are described on the conference web page.

Call for Journal-track Submissions

Following a journal-first model, this edition of the conference again includes a journal track, which seeks original and high-quality papers that describe mature research work on the conference topics. Accepted journal-track papers will be published in the "Reliable Software Technologies (AEiC2023)" Special Issue of JSA -- the Journal of Systems Architecture (Scimago Q1 ranked, impact factor 5.936).

[Submission details removed. Call is closed now.]

Authors who have successfully passed the first round of review will be invited to present their work at the conference. Please note that the AEiC 2023 organization committee will waive the Open Access fees for the first four accepted papers, which do not already enjoy OA from personalized bilateral agreements with the Publisher. Subsequent papers will follow JSA regular publishing track.

Call for Industrial-track Submissions

The conference seeks industrial practitioner presentations that deliver insight on the challenges of developing reliable software. Especially welcome kinds of submissions are listed on the conference web site. Given their applied nature, such contributions will be subject to a dedicated practitioner-peer review process. Interested authors shall submit a one-to-two pages abstract, by 27 February 2023, via EasyChair at https://easychair.org/my/conference?conf=aeic2023, selecting the "Industrial Track". The format for submission is strictly in PDF, following the Ada User Journal style. Templates are available at http://www.ada-europe.org/auj/guide.

The abstract of the accepted contributions will be included in the conference booklet. The corresponding authors will get a presentation slot in the prime-time technical program of the conference and will also be invited to expand their contributions into full-fledged articles for publication in the Ada User Journal, which will form the proceedings of the industrial track of the Conference. Prospective authors may direct all enquiries regarding this track to its chairs Alexandre Skrzyniarz (alexandre.skrzyniarz at fr.thalesgroup.com) and Sara Royuela (sara.royuela at bsc.es).

Call for Work-in-Progress-track Submissions

The work-in-progress track seeks two kinds of submissions: (a) ongoing research and (b) early-stage ideas. Ongoing research submissions are 4-page papers describing research results that are not mature enough to be submitted to the journal track. Early-stage ideas are 1-page papers that pitch new research directions that fall within the scope of the conference. Both kinds of submissions must be original and shall undergo anonymous peer review. Submissions by recent MSc graduates and PhD students are especially sought. Authors shall submit their work by 27 February 2023, via EasyChair at https://easychair.org/my/conference?conf=aeic2023, selecting the "Work in Progress Track". The format for submission is strictly in PDF, following the Ada User Journal style. Templates are available at http://www.ada-europe.org/auj/guide.

The abstract of the accepted contributions will be included in the conference booklet. The corresponding authors will get a presentation slot in the prime-time technical program of the conference and will also be offered the opportunity to expand their contributions into 4-page articles for publication in the Ada User Journal, which will form the proceedings of the WiP track of the Conference. Prospective authors may direct all enquiries regarding this track to the corresponding chairs Bjorn Andersson (baandersson at sei.cmu.edu) and José Cecílio (jmcecilio at fc.ul.pt).

Awards

Ada-Europe will offer an honorary award for the best technical presentation, to be announced in the closing session of the conference.

Call for Tutorials

The conference seeks tutorials in the form of educational seminars on themes falling within the conference scope, with an academic or practitioner slant, including hands-on or practical elements. 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), the intended level of the contents (introductory, intermediate, or advanced), and a statement motivating attendance. Tutorial proposals shall be submitted by e-mail to Tutorial and Education Chair, Luís Miguel Pinho (lmp at isep.ipp.pt), with subject line: "[AEiC 2023: tutorial proposal]". Tutorial proposals shall be submitted by 27 February 2023. The authors of accepted full-day tutorials will receive a complimentary conference registration, halved for half-day tutorials. The Ada User Journal will offer space for the publication of summaries of the accepted tutorials.

Call for Workshops

The conference welcomes satellite workshops centred on themes that fall within the conference scope. Proposals may be submitted for half- or full-day events, to be scheduled at either end of the AEiC conference. Workshop organizers shall also commit to producing the proceedings of the event, for publication in the Ada User Journal. Workshop proposals shall be submitted by e-mail to the Workshop Chair, Frank Singhoff (singhoff at univ-brest.fr), with subject line: "[AEiC 2023: workshop proposal]". Workshop proposals shall be submitted at any time but no later than the 27 February 2023. Once submitted, each workshop proposal will be evaluated by the conference organizers as soon as possible.

Call for Exhibitors

The conference will include a vendor and technology exhibition. Interested providers should direct inquiries to the Exhibition & Sponsorship Chair, Ahlan Marriott (ahlan at Ada-Switzerland.ch).

Venue

The conference will take place at the Hotel Fénix Lisboa, near downtown Lisbon, Portugal. June is full of events in Lisbon, including the festivities in honour of St. António (June 13 is the town holiday), with music, grilled sardines, and popular parties in Alfama and Bairro Alto neighbourhoods. There's plenty to see and visit in Lisbon, so plan in advance!

Organizing Committee

- Conference Chair

António Casimiro,
University of Lisbon, Portugal
casim at ciencias.ulisboa.pt

- Journal-track Chair

Elena Troubitsyna,
KTH Royal Inst. of Technology, Sweden
elenatro at kth.se

- Industrial-track Chairs

Alexandre Skrzyniarz,
Thales, France
alexandre.skrzyniarz at fr.thalesgroup.com

Sara Royuela,
Barcelona Supercomputing Center, Spain
sara.royuela at bsc.es

- Work-In-Progress-track Chairs

Bjorn Andersson,
Carnegie Mellon University, USA
baandersson at sei.cmu.edu

José Cecílio,
University of Lisbon, Portugal
jmcecilio at fc.ul.pt

- Tutorial and Education Chair

Luis Miguel Pinho,
ISEP, Portugal
lmp at isep.ipp.pt

- Workshop Chair

Frank Singhoff,
University of Brest, France
singhoff at univ-brest.fr

- Exhibition & Sponsorship Chair

Ahlan Marriott,
White Elephant GmbH, Switzerland
ahlan at Ada-Switzerland.ch

- Publicity Chair

Dirk Craeynest,
Ada-Belgium & KU Leuven, Belgium
Dirk.Craeynest at cs.kuleuven.be

- Webmaster

Hai Nam Tran,
University of Brest, France
hai-nam.tran at univ-brest.fr

Previous Editions

Ada-Europe organizes annual international conferences since the early 80's. This is the 27th event in the Reliable Software Technologies series, previous ones being held at Montreux, Switzerland ('96), London, UK ('97), Uppsala, Sweden ('98), Santander, Spain ('99), Potsdam, Germany ('00), Leuven, Belgium ('01), Vienna, Austria ('02), Toulouse, France ('03), Palma de Mallorca, Spain ('04), York, UK ('05), Porto, Portugal ('06), Geneva, Switzerland ('07), Venice, Italy ('08), Brest, France ('09), Valencia, Spain ('10), Edinburgh, UK ('11), Stockholm, Sweden ('12), Berlin, Germany ('13), Paris, France ('14), Madrid, Spain ('15), Pisa, Italy ('16), Vienna, Austria ('17), Lisbon, Portugal ('18), Warsaw, Poland ('19), online from Santander, Spain ('21), and Ghent, Belgium ('22).

Information on previous editions of the conference can be found at

http://www.ada-europe.org/confs/ae.

Our apologies if you receive multiple copies of this announcement.

Please circulate widely.

Dirk Craeynest, AEiC 2023 Publicity Chair
Dirk.Craeynest@cs.kuleuven.be

* 27th Ada-Europe Int. Conf. Reliable Software Technologies (AEiC 2023)
* June 13-16, 2023, Lisbon, Portugal, www.ada-europe.org/conference2023

(V3.1)

Ada-Europe Conference - 6 March Extended Final Deadline

From: Dirk Craeynest <dirk@orka.cs.kuleuven.be>
Subject: Ada-Europe Conference - 6 March Extended Final Deadline
Date: Sat, 25 Feb 2023 14:33:00 -0000
Newsgroups: comp.lang.ada, fr.comp.lang.ada, comp.lang.misc

The recently posted reminder for the Ada-Europe 2023 Conference triggered several requests for extra time. To give all authors the same opportunity to further refine their submission, the organizers decided that the deadline for industrial- and work-in-progress-track abstracts, and for tutorial and workshop proposals will be extended by 1 week until Monday, 6 March 2023. 1+ week remains!

FINAL UPDATED Call for Contributions

27th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2023)

13-16 June 2023, Lisbon, Portugal

*** EXTENDED FINAL submission DEADLINE 6 March 2023 ***

Industrial- and Work-in-Progress-track: submit via
https://easychair.org/my/conference?conf=aeic2023
select "Industrial Track" or "Work in Progress Track"

Tutorials: submit to Tutorial and Education Chair,
Luís Miguel Pinho <lmp @ isep.ipp.pt>
subject "[AEiC 2023: tutorial proposal]"

Workshops: submit to Workshop Chair, Frank Singhoff
<singhoff @ univ-brest.fr>
subject "[AEiC 2023: workshop proposal]"

For more information please see the full Call for Papers at
www.ada-europe.org/conference2023

#AEiC2023 #AdaEurope #AdaProgramming

Our apologies if you receive multiple copies of this announcement.

Please circulate widely.

Dirk Craeynest, AEiC 2023 Publicity Chair
Dirk.Craeynest@cs.kuleuven.be

* 27th Ada-Europe Int. Conf. Reliable Software Technologies (AEiC 2023)*
* June 13-16, 2023, Lisbon, Portugal, www.ada-europe.org/conference2023

(V4.1)

Ada and Education

Table Game

From: Mockturtle <framefritti@gmail.com>
Subject: Table game
Date: Tue, 17 Jan 2023 05:56:31 -0800
Newsgroups: comp.lang.ada

Really?!?

https://www.amazon.com/Artana-AAX14001-Lovelace-Babbage/dp/B07WHMG5Y8

[The link is for a tabletop game with the blurb “Play as a pioneer of early computing, like Ada Lovelace or Charles Babbage, to build a program that solves problems for famous patrons like Charles Darwin, Mary Shelley, and more!”. It is priced at 19.98$ and has 4.5/5 stars rating with 50 reviews at the time of this writing. —arm]

Ada-related Resources

[Delta counts are from February 12th to April 5th. —arm]

Ada on Social Media

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Ada on Social Media
Date: 5 Apr 2023 17:36 CET[b]
To: Ada User Journal readership

Ada groups on various social media:

- Reddit: 8_349 (+58) members	[1]
- LinkedIn: 3_436 (+18) members	[2]
- Stack Overflow: 2_323 (+14) questions	[3]
- Telegram: 160 (+1) users	[4]
- Gitter: 219 (+68*) people	[5]
- Ada-lang.io: 107 (+6) users	[6]
- Libera.Chat: 74 (-8) concurrent users	[7]
- Twitter: 22 (-10) tweeters	[8]
44 (-5) unique tweets	[8]

* Gitter has migrated its messaging to the Matrix open standard. The [5] reference has been updated accordingly.

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

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

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

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

[5] https://app.gitter.im/#/room/#ada-lang_Lobby:gitter.im

[6] https://forum.ada-lang.io/u

[7] https://netsplit.de/channels/details.php?room=%23ada&net=Libera.Chat

[8] http://bit.ly/adalang-twitter

Repositories of Open Source Software

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Repositories of Open Source software
Date: 5 Apr 2023 17:45 CET[c]
To: Ada User Journal readership

Rosetta Code: 924 (+4) examples	[1]
40 (+1) developers	[2]
GitHub: 763* (=) developers	[3]
Alire: 337 (+13) crates	[4]
Sourceforge: 240 (=) projects	[5]
Open Hub: 214 (=) projects	[6]
Codelabs: 54 (=) repositories	[7]
Bitbucket: 31 (=) repositories	[8]

* This number is unreliable due to GitHub search limitations.

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

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

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

[4] https://alire.ada.dev/crates.html

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

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

[7] https://git.codelabs.ch/?a=project_index

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

Language Popularity Rankings

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Ada in language popularity rankings
Date: 5 Apr 2023 17:36 CET[d]
To: Ada User Journal readership

[Positive ranking changes mean to go up in the ranking. —arm]

- TIOBE Index: 28 (-5) 0.42% (-0.18%)	[1]
- PYPL Index: 19 (-2) 0.83% (-0.11%)	[2]
- IEEE Spectrum (general): 35 (=) Score: 1.16	[3]
- IEEE Spectrum (jobs): 33 (=) Score: 0.79	[3]
- IEEE Spectrum (trending): 32 (=) Score: 3.95	[3]

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

[2] http://pypl.github.io/PYPL.html

[3] https://spectrum.ieee.org/top-programming-languages/

Ada-related Tools

Embedded AVR Ada Setup - Linux Edition

From: Stéphane Rivière <stef@genesix.org>
Subject: ANN: Embedded AVR Ada Setup - Linux edition
Date: Thu, 12 Jan 2023 11:21:30 +0100
Newsgroups: comp.lang.ada

Hi all,

Embedded AVR Ada Setup - Linux edition.

Thanks to the work of Rolf Ebert (AVR-Ada and AVR-Ada to Alire conversion), Fabien Chouteau and AdaCore (GNAT-AVR, GNAT-AVR to Alire conversion, Alire promotion) and their friendly help, here is a tutorial to get the most pleasant environment to develop in Ada on 8-bit AVR targets under Linux.

Based on Alire and GNAT Studio 23 it allows real-time debugging in GNAT Studio as if you were in a native X86_64 environment.

This was an opportunity to get acquainted with Alire while keeping our usual GNAT Studio based environment, which integrates perfectly with Alire. Thanks to the author Alejandro R. Mosteo, who also wrote a very interesting presentation of Alire in AUJ Vol 39, Number 3, Sept 2018, P 189.

This work is part of a more general desire to empower the Ada community with respect to the defunct GNAT CE. We therefore adhere to this new policy of Adacore. Between this new direction, the arrival of Alire, the availability of many Crates, the first successes of the community in building GNAT Studio independently, the arrival of Rust which is good for the visibility of our favorite language, Ada is certainly entering a new era :)

https://github.com/sowebio/adam-doc (GNAT Studio & project example additional files)

https://github.com/sowebio/adam-doc/blob/master/Ada%20Development%20on%20AVR%20Microcontroller.pdf

Feedback and criticism are welcome.

Short Video on Getting Started with GtkAda in 2023

From: Stephen Merrony <merrony@gmail.com>
Subject: A Short Video on Getting Started with GtkAda in 2023
Date: Sat, 14 Jan 2023 01:01:13 -0800
Newsgroups: comp.lang.ada

I made a quick video showing how easy it is to get started writing a Gtk application in Ada these days...

https://youtu.be/IofrV5hsUvg

[Video running time is 11:03 minutes. —arm]

Gnu Emacs Ada Mode 8.0.4 Released

From: Stephen Leake stephen.leake84@gmail.com
Subject: Gnu Emacs Ada mode 8.0.4 released.
Date: Wed, 25 Jan 2023 05:27:57 -0800
Newsgroups: comp.lang.ada

Gnu Emacs Ada mode 8.0.4 is now available in GNU ELPA.

All Ada mode executables can now be built with Alire (https://alire.ada.dev/); this greatly simplifies that process.

gpr-query and gpr-mode are split out into separate GNU ELPA packages. You must install them separately (Emacs install-package doesn't support "recommended packages" like Debian does).

Ada mode can now be used with Eglot; this is controlled by new variables:

ada-diagnostics-backend - one of wisi, eglot, none

ada-face-backend - one of wisi, eglot, none

ada-indent-backend - one of wisi, eglot, none

ada-statement-backend - one of wisi, eglot, none

ada-xref-backend - one of GNAT, gpr_query, eglot, none

The diagnostic, face, indent, and statement backends default to wisi if the wisi parser is found in PATH, to eglot if the Ada LSP server is found, and none otherwise. The xref backend defaults to gpr_query if the gpr_query executable in PATH, to GNAT otherwise.

ada-diagnostics-backend controls the source of compilation error messages while editing.

ada-statement-backend controls statement motion; forward-sexp, wisi-goto-statement-end, etc. ada-xref-backend controls wisi-goto-spec/body and Emacs xref commands.

In addition, name completion is provided by eglot if any of the other backends are using eglot; eglot completion is always better than wisi.

The current AdaCore language server (version 23) supports face but not indent. The current version of eglot (1.10) does not support face. The Language Server Protocol does not support statement motion. So for now, eglot + ada_language_server only provides xref and completion.

The AdaCore language server ada_language_server is installed with GNATStudio (which ada-mode will find by default), or can be built with Alire. If you build it with Alire, either put it in PATH, or set gnat-lsp-server-exec.

I have not tested ada-mode with lsp-mode. You can set ada-*-backend to 'other to experiment with that, or tree-sitter, or some other backend. tree-sitter will be fully supported in the next ada-mode release.

The required Ada code requires a manual compile step, after the normal list-packages installation:

```
cd ~/.emacs.d/elpa/ada-mode-7.3beta*
```

```
./build.sh
```

```
./install.sh
```

If you have Alire installed, these scripts use it.

Ada and Other Languages

Carbon New Language

From: Gautier Write-Only Address <gautier_niouzes@hotmail.com>
Subject: Carbon
Date: Fri, 22 Jul 2022 14:13:08 -0700
Newsgroups: comp.lang.ada

[This thread is a bit dated as it was deemed less of a priority due to space constraints in past issues. —arm]

Next attempt to replace C/C++ without really replacing it: Carbon!

You will notice, as usual, a few aspects borrowed from Ada - and one point inspired by Ada 83 (which was relaxed in a later Ada version) :-)

https://mybroadband.co.za/news/software/453410-googles-carbon-programming-language-aims-to-replace-c.html

https://devclass.com/2022/07/20/google-brands-carbon-language-as-experimental-successor-to-c/

https://9to5google.com/2022/07/19/carbon-programming-language-google-cpp/

https://thenewstack.io/google-launches-carbon-an-experimental-replacement-for-c/

From: John Mccabe <john@nospam.mccabe.org.uk>
Date: Sat, 23 Jul 2022 09:09:57 -0000

I read that stuff yesterday and, yet again, shook my head in disbelief :-(

The bit where I laughed was where it was claimed that C++ is building technical debt because it's not changing quickly enough; C++ is currently a mess because it's changing too quickly! Half-baked, and half-implemented ideas are going into 'standards' in the full knowledge that they'll change again in the next one. Even g++ doesn't provide 100% support for C++17 (https://gcc.gnu.org/projects/cxx-status.html#cxx17)!

Carbon is likely to be even worse; every 'new' language that promises the earth, without being designed in a rigorous way, ends up with the same problems. Java - I started playing with that in the 90s and got frustrated that every update brought more and more depreciation warnings in. Python - 2.x -> 3.0 was a massive jump (and took years to gain traction) because the 'designers' just hadn't done a very good job to start with! Rust? Mmm

As for the 'reuse C++ syntax'; why the obsession with that? C++ syntax is really bad! (Semantics, in some cases, are another level - how many languages need a book like "C++ Gotchas"?!).

Aaaaarrrrgghhh!

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 23 Jul 2022 15:14:15 +0200

> Next attempt to replace C/C++ without really replacing it: Carbon!

We have just learned how dangerous carbon is for our climate. Yet these few privileged keep on pumping it up! (:-))

From: Stéphane Rivière <stef@genesix.org>
Date: Sat, 23 Jul 2022 15:49:05 +0200

> We have just learned how dangerous carbon is for our climate. Yet these few privileged keep on pumping it up! (:-))

Carbon language bad, green language good

From: Luke A. Guest <laguest@archeia.com>
Date: Sun, 24 Jul 2022 10:38:58 +0100

> Next attempt to replace C/C++ without really replacing it: Carbon!

Saw this last week and immediately thought they'd failed on one of their "design goals," i.e. to be "readable”.

> You will notice, as usual, a few aspects borrowed from Ada - and one point inspired by Ada 83 (which was relaxed in a later Ada version) :-)

What did they take from Ada?

From: John Mccabe <john@mccabe.org.uk>
Date: Tue, 26 Jul 2022 10:31:42 -0700

> What did they take from Ada?

Certainly not the approach to making life easier and less error-prone for developers.

I've got involved in a couple of discussions on their forum, and I'm inclined to think they just want C++ but taken out of the control of ISO/IEC WGs steering committees.

They're pretty much not considering changing any of the aspects of C++ that make it such a heap of junk (IMO, of course), including, but not limited to:

1. arrays
2. enums
3. (both of the above when used together :-))
4. symbols - overuse, duplication, inconsistency
5. implicit stuff
6. pretend strong typing
7. forcing developers to deal manually with numeric values that don't fit into an n-byte range, where n is a whole number

It really is shockingly soul-destroying watching all that. What's worse is that, from what I've seen over the years, the new languages that have been developed in a more 'relaxed' way than Ada (well, evolved, really, like Java, Python etc) and have become relatively successful have taken a good 10 years or so to get to that point, yet the discussions on the Carbon forum are all about how to appeal to _current_ developers who're used to C++; not _future_ developers who, ideally, would _never_ be used to C++!

From: Nasser M. Abbasi <nma@12000.org>
Date: Thu, 28 Jul 2022 18:48:49 -0500

Since Ada has solved these problems a long time ago, then why are people still reinventing the wheel? Why are they not just using Ada? Ada is free software.

Maybe there is something in Ada that prevents it from being widely adopted and used? [...]

From: John Mccabe <john@nospam.mccabe.org.uk>
Date: Fri, 29 Jul 2022 11:03:36 -0000

> why are people still reinventing the wheel?

Possibly for the same reason that I was so anti-Ada in my early years; it takes getting used to and people are lazy.

Looking at some of the languages that have come out in recent years, it's obvious that people can't be bothered to type much; "fn"/"def" (or, even, nothing!) instead of "function"/"procedure", "{"/"}" instead of "begin"/"end", "&&" instead of "and", "||" instead of "or" (!!!) etc.

From what I can see, some of the "moderators" on that Carbon group don't have much real professional software development experience, so I suspect they really have no clue about what they could achieve with Ada, and have little understanding of some of the constraints that embedded, especially bare-metal, systems impose on what you can and can't include in a program. I'm thinking here of things like heap-unfriendly container classes, such as (in Swift) arrays that are automatically expandable when you append a new item, rather than being fixed size etc.

There also seems to be a bit of an obsession with the time between "empty editor window" and "executable available", rather than "empty editor window" and "executable that actually does what you want"!

Also, as Devin says, compiler availability is an issue, from the point of view of actually _using_ Ada.

However, from the point of view of creating a new language, the fact that so many people clearly think it _has_ to be the C/C++ way is quite disturbing, especially since, as I think I mentioned, it's going to be a number of years until any new language really makes its mark, so new languages should be taking future developers into account, not just pandering to the laziness of existing ones!

At this point I think I should make it clear that, although I think Ada has some great features (and I regularly espouse them amongst my colleagues), I don't use Ada in the software I'm developing. I'd like to, but it would take me a lot of time to get back to a level in Ada where I'd be comfortable creating a relatively substantial codebase from scratch. The alternative would be to go and join a team that's already using Ada, but every Ada job I've seen come up locally is to support code that was written in Ada 95; I'd rather be looking at Ada 2005 -> if I was to make that jump.

From: Gautier Write-Only Address <gautier_niouzes@hotmail.com>
Date: Fri, 29 Jul 2022 11:59:21 -0700

[...] IMHO the only way to make Ada more popular is to create popular applications with it.

From: Dennis Knorr <dennis.knorr@gmx.net>
Date: Sat, 6 Aug 2022 16:18:12 +0200

> Maybe there is something in Ada that prevents it from being widely adopted and used?

An opinion from a bystander who wants to like Ada, this is only after I looked the resources and the community up a bit two years ago again. you do not have to agree, it's just my experience and sometimes gut feeling.

* Bad to no marketing
* sometimes elitism by members of the community/Ada fans
* no modern feeling toolchain (Even Lazarus+Pascal or Gambas has a more modern feeling toolchain, and that says a lot)
* not much free software built with it
* not much free software for the toolchain available
* not much libraries which are ready and easy to use as a beginner
* no modern/up2date books and articles (especially in other languages than in English) seem to be available.
* the free Ada Compiler seems slow and a while back it generated relatively big binaries and the result was not very fast.

Just a few concrete examples to back that up:

* Is there a web playground or repl shell trying or learning/trying Ada or some of its prominent modules?
* There's no modern book in German about modern Ada and its libraries
* There's no syntax highlighting package in vim for Ada
* No exercises like for example Ruby Koans
* It *Looks* like there are no libraries which make it easy use Ada for programming (think json/document formats, http/mail/mime protocols, algorithms or cryptography libraries)

I know there are libraries out there, but they are hard to find, not promoted/marketed and I saw developers (also in other languages, I admit that) talking like, if you do not understand it, you should go back to toy languages like python.

I also know that not all bullet points above are really true to the fullest, but most of them from the outside look like it and also have at least some grain of truth in there.

If someone would write a book in German, how to write Ada and use $cryptolibrary, $networklibrary and how to integrate it in one's favorite development software, this surely would be very interesting to many.

The ONLY thing where I see Ada Marketing in the free software world is FOSDEM. But it is in its own Room. Ada people would need to go out and say: hey, look we also can do good stuff, look, an https server with letsencrypt support with library in 30 lines.

To be honest, I am curious how the community here will react to it. I mean, I got the Book "Programming in Ada 2005" as a present and I liked it, but after reading the introduction (first 2-3 chapters I think) back then (was like over 15 years ago) I saw no libraries which I can use. and I was not that big a programmer to write them myself.

From: A.J. <ianozia@gmail.com>
Date: Sat, 6 Aug 2022 10:48:00 -0700

I agree with you on some of these points. Ada never seemed to be big on marketing, at least outside of specific niches, and from a learning & resources aspect, it took me reading Barnes' Programming in Ada 2012 cover-to-cover to properly grok the language. With that being said, things have been changing a lot in the last two years.

https://learn.adacore.com is a decent resource in that it gives you a little Ada interpreter with code snippets you can test out yourself right in the browser. It's not exactly a "web playground or repl shell" but it's pretty good and seems to support the standard library.

From a library and tooling standpoint, I would check out Alire. It takes a matter of minutes to get from not having any Ada compilers installed at all to compiling your own hello example and there's a lot of libraries already supported (https://alire.ada.dev/crates.html). To bring, for example, Gnatcoll_sqlite, into your project, you would simply just type "alr with gnatcoll_sqlite" while in that directory.

[...]

Then of course there's the awesome-ada repository that has some nice resources, albeit they seem to mostly be in English: https://github.com/ohenley/awesome-ada

From: G.B.
<bauhaus@notmyhomepage.invalid>
Date: Sun, 7 Aug 2022 11:08:43 +0200

> * There's no modern book in German about modern Ada and its libraries

What's the competition, considering C#, Swift, Java or C? I.e., an original work written by a German author, bought and studied by many? There used to be a number of books on Ada written in German when the market had developed ideas of a government mandate, the ideas producing corresponding opportunities.

> * There's no syntax highlighting package in vim for Ada

:syntax enable

(Does vim feature in a modern feeling tool chain, though?)

> * No exercises like for example Ruby Koans

> * It *Looks* like there are no libraries which make it easy use Ada for programming (think json/document formats, http/mail/mime protocols,

AWS, GNATColl, $ alr with json.

> algorithms or cryptography libraries

Just use one that you can trust. If you need it to be more Ada-ish, ChaCha20 cipher and Poly1305 digest have just been mentioned a few postings ago. If algorithms can address securing the entire computation...

There used to be the PAL, which is the Public Ada Library, easy to find. A bit dated, and reflecting the hype back then, I guess.

I gather that, currently, and in the past, Ada tools are also focusing on topics of embedded computers, a fairly large and attractive market. JSON or MIME, perhaps even interpreters are present, but I think not central to control stuff near sensors and actuators. How does one compute deterministic responses before a deadline using Node.js?

[...]

From: Dennis Knorr
<dennis.knorr@gmx.net>
Date: Mon, 8 Aug 2022 23:38:59 +0200

> What's the competition, considering C#, Swift, Java or C?

From the absolute amount in English, these languages or Python or Rust have more books. Hell, even Raku has more books.

Python, Kotlin(!), C# have more german books and also more current ones. I bet in five years from now there will be more German books about Carbon than about Ada, even if you include the old ones. [...]

>> * There's no syntax highlighting package in vim for ada

> :syntax enable

Okay, that I did not know.

> (Does vim feature in a modern feeling tool chain, though?)

Well, okay, Intellij is called more modern of course or VSCode, but you still can craft modern tooling onto vim and it works well.

[...]

From: Randy Brukardt
<randy@rrsoftware.com>
Date: Mon, 8 Aug 2022 23:12:44 -0500

> P.S. Nobody writes Ada books these days because they do not sell.

Do *any* programming books really sell? If so, why? :-)

There are plenty of free, on-line resources for pretty much any programming language. Why pay for something you can get free?

When someone starts talking about books, I think they're a troll. I can understand complaints about having trouble finding stuff (although Google should find AdaIC.org pretty easily, it's usually pretty high in Ada results, and most of the good stuff is linked from there), and lack of hype, and so on. But there's lots of good stuff if one looks (or asks here - if someone knows about here they're ready to use Ada).

AdaIC has an Ada-specific search engine which hopefully makes it easier to find Ada stuff than a general engine like Google.

From: Paul Rubin
<no.email@nospam.invalid>
Date: Mon, 08 Aug 2022 23:05:12 -0700

> When someone starts talking about books, I think they're a troll.

I don't know of any online Ada docs that I'd call helpful past the beginner level (Ada Distilled). Someone here recommended a book to me a year or two ago and I bought a copy. It looks good but has just been sitting around waiting for me to read it. I haven't done that because I haven't had any occasion to mess with Ada, and have too many other pending projects. One of these days.

From: John Mccabe
<john@nospam.mccabe.org.uk>
Date: Tue, 9 Aug 2022 07:22:13 -0000

>There are plenty of free, on-line resources for pretty much any programming language. Why pay for something you can get free?

FWIW, I may be 'old school', but I buy loads of programming books. That obviously doesn't qualify me to answer the question of whether "*any* programming books really sell", but the main reasons I like books are that they tend to be more constrained and cohesive than jumping around websites (at least, the decent ones are :-)). Also for those times when I want to flick back and forth between sections quickly, when I don't want to be staring at a screen and so on. One particular reason is that, unless I've actually got a block of free time to be experimenting with stuff, using a Web browser presents multiple, frustrating distractions, and it's often the case that an example of something you might want to do has no explanation about how it works (books, especially Ada As A Second Language, if I remember correctly, are generally fairly good at that bit), so that leads to more searching, more jumping about webpages and, nowadays, a helluva lot of stale and misleading information.

So, basically, that's why I pay for books.

From: John Perry <devotus@yahoo.com>
Date: Tue, 9 Aug 2022 18:19:47 -0700

> Do *any* programming books really sell? If so, why? :-)

Having recently left a university, I can attest that schoolbooks are still a thing, and that includes textbooks on computer programming. I recently inherited from a member of this forum a nice textbook on Data Structures in Ada, but it was based on Ada 95, and I'm not sure it's in print anymore. In fact, and alas, only three of the Ada-based textbooks I find "easily" on Amazon date from the early- to mid-90s, and of the three recent ones I find, only the Barnes book is of good quality.

I'd be delighted if someone would prove me wrong.

From: Paul Rubin <no.email@nospam.invalid>
Date: Tue, 09 Aug 2022 23:20:43 -0700

> of the three recent ones I find, only the Barnes book is of good quality. I'd be delighted if someone would prove me wrong.

Analysable Real-Time Systems: Programmed in Ada by Prof. Alan Burns is from 2016 and looks pretty good. It is the book I mentioned that I got on the recommendation of someone here. I've flipped through it but still haven't read it.

From: Randy Brukardt <randy@rrsoftware.com>
Date: Wed, 17 Aug 2022 20:02:53 -0500

The Ada 2012 book by Peter Chapin looks promising, although I don't think he's finished it (https://github.com/pchapin/tutorialada). There is a PDF version of it available on-line.

Otherwise, I recommend Ada Distilled (https://www.adaic.org/wp-content/uploads/2010/05/Ada-Distilled-24-January-2011-Ada-2005-Version.pdf) [Ada 2005], and the Craft of Object Oriented Programming (http://archive.adaic.com/docs/craft/craft.html) [Ada 95], depending on the programmer's level. These are all written in the textbook style, and are all available for free on the Internet. I don't think you miss too much learning with an Ada 95 book first (most of the newer stuff is fairly obvious, or needs a textbook of its own.

The Wikibook is also a good choice

(http://en.wikibooks.org/wiki/Ada_Programming), but you do have to be on-line to use it (the others are downloadable and thus usable locally).

I find the Barnes book to be too much of a good thing (sorry, John!). When John gave me a copy at a Paris ARG meeting, I put it on my lap to look through it (since my hotel room was tiny), my legs got numb after not very long. I know better than to put it on a body part again. :-) I'd recommend it as a reference for serious Ada programmers, since it tries to cover everything (the latest version has an Ada 2022 appendix).

Ada Practice

Working with Library Versions

[This thread splinter veered off from the announcement of ada-lang-io towards library management. Other topics have been pruned out. —arm]

From: Paul Jarrett <jarrett.paul.young@gmail.com>
Date: Tue, 11 Oct 2022 21:21:31 -0700
Date: Sun, 09 Oct 2022 09:13:11 -0700
Newsgroups: comp.lang.ada

> Adahome.com is sort of like that, but it is run by some company and hasn't been updated in forever.

https://ada-lang.io/ is designed to be updateable for a long time and open to community contributions by being completely open source. There're already multiple people who have permissions to merge changes to help ensure longevity.

ada-lang.io is indexed using Algolia, so the entire site (including the Ada 2022 draft RM) is searchable.

Someone else wrote a tool for searching through all code in Alire crates at https://search.synack.me/

> I am not sure if package manager is a good idea if it does not refer to the target system's packaging tools, e.g. DEB, RPM, MSI etc. The main problem with that stuff is usually architectural. Most of it is plain aggregation of source code, which is utterly wrong. The very idea to rebuild everything each time on each client is atrocious both with regard to wasting computing resources as well as testing, safety, consistency, interoperability inside the target.

Alire can do additional build steps and other things.

As an application developer, having the code available helps in auditing third-party software for security reasons, build it in a debug configuration for troubleshooting, and also provides the means to locally fix bugs or adapt the library if needed. Isolating libraries and including them with a package manager on a per project basis eases setup also by not making developers have to look up or use multiple installers.

I've seen inconsistencies in builds when developers who rely on the system libraries (installed by things like apt) join the project at different times -- the earliest developers might be on libfoo-1.2 whereas newer developers are on libfoo-1.4. You don't run into this problem if the repo points to the applicable dependencies and everyone builds everything locally. It also avoids other problems such as if your system's package manager doesn't have a particular library version, and the project builds that library from source. It's not perfect and there's other problems that you run into, but it often does help understanding what is being built in the project more clearly. Alire even takes this an entire step further by being able to install and manage the toolchain as well (gprbuild and GNAT).

Package managers also simplify having multiple projects using the same library, but different and possibly incompatible versions on the same system. You get a snapshot in time and a more consistent path to get a build working for new developers, or on a new system. There are limitations due to what systems open source library writers have available to test on, so you shouldn't just blanket trust code you pull in though, and you should be careful how you use it.

Overall, Alire makes the experience building and developing in Ada for me on Windows, Mac and Linux, considerably simpler and more efficient, by providing the same interface for use across all of these systems.

With the beautiful site styling done by onox, someone pointed to ada-lang.io should be able to download Alire, install a toolchain, make a project and build in less than 15 minutes or so (depending on download and install time). The work done by Fabien and Alejandro, and everyone else who has contributed to Alire to make this happen within the last couple years is absolutely incredible. Combined with Maxim's fantastic work on the Ada language plugin for Visual Studio Code, it's a great experience for first-time users of the language.

[...]

> Maybe a web forum would be a good idea, because many people nowadays see Usenet newgroups as an outdated thing. So the fact that the community mostly relies on comp.lang.ada may turn them off.

There's a dedicated forum now at https://forum.ada-lang.io/

From: Stephen Leake <stephen_leake@stephe-leake.org>
Date: Wed, 12 Oct 2022 17:06:26 -0700

>> I am not sure if package manager is a good idea if it does not refer to the target system's packaging tools, e.g. DEB, RPM, MSI etc.

Alire can define crates that import system libraries, using those tools. They are subject to the same version checks as other Alire crates.

>> The very idea to rebuild everything each time on each client is atrocious both with regard of wasting computing resources as well as testing, safety, consistency, interoperability inside the target.

Actually, it's better for consistency; that's why Alire does it.

I don't understand what you mean by "testing" here; how does compiling from source affect testing?

Same for "interoperability".

> I've seen inconsistencies in builds when developers who rely on the system libraries [...]

More precisely, an Alire crate can specify precisely which version of each dependency it requires/is compatible with.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Thu, 13 Oct 2022 08:58:16 +0200

> Actually, it's better for consistency; that's why Alire does it.

Consistency is easier to enforce on pre-built deployments, obviously. Moreover libraries usually provide integrated checks and/or have some target platform policy, e.g. naming and placement conventions.

> I don't understand what you mean by "testing" here; how does compiling from source affect testing?

Because one can run tests on pre-built packages impossible to run on the sources. For example, network/hardware protocols. In order to test a protocol implementation one needs complex mock setups the client simply does not have. Such tests may run for many hours etc.

> Same for "interoperability".

See above. You cannot run integration tests on the client, it is just silly.

>> [...] You don't run into this problem if the repo points to the applicable dependencies and everyone builds everything locally.

No difference whether deployment is in source or pre-built. Dependencies must be enforced regardless. However it is far easier to do with pre-built packages.

> More precisely, an Alire crate can specify precisely which version of each dependency it requires/is compatible with.

It seems so. Multiple versions at once are not supported. E.g. when you are working on two projects both dependent on different versions of another project:

B -> A.1

C -> A.2

Or even the same project, e.g. when doing some migration from one version to another.

From: Fabien Chouteau <fabien.chouteau@gmail.com>
Date: Fri, 14 Oct 2022 01:41:32 -0700

> Multiple versions at once are not supported. [...]

Yes of course, different crates can depend on different versions of the same crate.

> Or even the same project, e.g. when doing some migration from one version to another.

Not sure how you would do that? Link two different versions of the same library in an executable? That's not going to work.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Fri, 14 Oct 2022 12:05:00 +0200

> Yes of course, different crates can depend on different versions of the same crate.

It is about whether both A's can be installed and coexist on the same machine.

> Not sure how you would do that? Link two different versions of the same library in an executable? That's not going to work.

Same as above. You have B.1 -> A.1 and B.* -> A.2. You want to install both A.1 and A.2 and work on B.* while checking on B.1.

In the long gone time of common sense, a project code management system would use a virtual file system and map different parts of the project's graph onto a structure of folders arranged by versions. Today one would use something ugly like a virtual machine or incredibly ugly like a docker.

From: Stephen Leake <stephen_leake@stephe-leake.org>
Date: Fri, 14 Oct 2022 04:19:05 -0700

> It is about whether both A's can be installed and coexist on the same machine.

In Alire, "installed" means "checked out the source code into a local directory".

If A depends on a system library that is a shared object file, and those are different versions, then it depends on the OS; Debian can handle this nicely, Windows only via separate directories and search paths.

> Same as above. You have B.1 -> A.1 and B.* -> A.2. You want to install both A.1 and A.2 and work on B.* while checking on B.1.

And the solution is the same as well.

> [...] a project code management system would use a virtual file system and map different parts of the project's graph onto a structure of folders arranged by versions.

What prevents that now?

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Fri, 14 Oct 2022 15:05:06 +0200

> What prevents that now?

Nothing except that it is to be done manually. Why not download a source archive and bother with anything? It is Turing-complete, after all... (:-))

The advantage of a file system is that developing image will be automated and consistent. And you would not need to move any files physically. Alire is extremely slow because it must pull all files [and then compile them on top of that].

Furthermore, a virtual file system shares duplicates of the same version of the same file. When you work with naked Git you must have as many copies as you have projects. Same applies to virtual machines and dockers. It is a huge overhead for nothing.

Moreover, a virtual file system is instant so long you do not access a file for read or write. Which is the case for gprbuild, make and other tools which use timestamps and then never look into files.

With a virtual file system you can automatically check in all files on closing if it was open for write and never worry about command-line mess or plug-ins. Any tool will work out of the box.

From: G.B. <bauhaus@notmyhomepage.invalid>
Date: Sun, 16 Oct 2022 10:54:57 +0200

> Furthermore, a virtual file system shares duplicates of the same version of the same file. When you work with naked Git you must have as many copies as you have projects. Same applies to virtual machines and dockers. It is a huge overhead for nothing.

Inasmuch as versions are subject to business, software configuration management is just work that requires resources to get it done. Problem solved. (Well, not for the small shop on a budget, granted.)

To what extent can static linking make B.1 and B.2 exist on the same system?

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 16 Oct 2022 11:20:33 +0200

> Inasmuch as versions are subject to business, software configuration management is just work that requires resources to get it done.

Yes, human resources especially. It is a self-feeding system that exists in each organization. It creates problems in order to justify its continuous growth. Modern time tools excel at wasting and perfect outright meaninglessness.

> Problem solved. (Well, not for the small shop on a budget, granted.)

I cannot say that ClearCase, which did things more or less right 20 years ago, was for small business either. (:-)) AFAIK it is still available and GNAT Studio supports it. However, IBM (Rational, actually) fulfills its existential end goal of wasting personal and hardware resources by other, no less efficient, techniques... (:-))

> To what extent can static linking make B.1 and B.2 exist on the same system?

To a full extent! (:-))

Sorry, I do not understand your question...

Arrays with Discriminated Task Components

From: Adamagica <christ-usch.grein@t-online.de>
Subject: Arrays with discriminated task components
Date: Sat, 24 Dec 2022 03:44:27 -0800
Newsgroups: comp.lang.ada

I've got a task type with a discriminant:

```
type Index is range 1 .. N;
task type T (D: Index);
```

Now I want an array of these tasks, where each task knows its identity (the index) via the discriminant, an iterated_component_association:

```
Arr: array (Index) of T :=
  (for I in Index => ???);
```

How can I do this?

This works with access, but I find this extremely ugly:

```
Arr: array (Index) of access T :=
 (for I in Index => new T (I));
```

Alternatively, I could use the traditional method with a Start entry with the index as parameter:

```
task type T is
  entry Start (D: Index);
end T;
```

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sat, 24 Dec 2022 20:05:19 +0200

One way is to give the discriminant a default value that is a function call that returns a new identifier on each call:

```
Next_Index : Index := Index'First;
-- The value returned by the next call
-- of New_Index.
function New_Index return Index
-- Returns a unique Index value (up to N).
is
   Result : constant Index := Next_Index;
begin
   if Next_Index < Index'Last then
      Next_Index := Next_Index + 1;
   -- else report error?
   end if;
   return Result;
end New_Index;
task type T (D: Index := New_Index);
```

Then you can declare the array without any initial value:

```
Arr: array (Index) of T;
```

and the initialization of each task in the array makes its own call to New_Index and gets its own identifier value.

A bit sneaky but has the advantage that it extends automatically to two arrays of tasks, or one array and some separate single declarations of tasks, etc.

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sat, 24 Dec 2022 23:41:34 +0100

> One way is to give the discriminant a default value that is a function call that returns a new identifier on each call:

No, this does not guarantee that the task's discriminant is its index in the array, which is a requirement of the question.

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sun, 25 Dec 2022 18:16:56 +0200

This seems to work with GNAT, but I'm not entirely sure if it is legal (could there be a conflict between the default value of the task discriminant, which is the same for all tasks in the array, and the actual discriminants which are different for each task in the array?):

```
N : constant := 10;
type Index is range 1 .. N;
task type T (D: Index := Index'First);
-- A default value for D is needed to make
-- the type constrained, as
-- required by the Arr declaration below.
function New_T (I : in Index)
return T
is
begin
  return TI : T (D => I)
  do
    null;
  end return;
end New_T;
Arr: array (Index) of T := (for I in Index =>
   New_T(I));
```

Whether this is any less ugly than the heap allocation method is doubtful.

From: Adamagica <christ-usch.grein@t-online.de>
Date: Mon, 26 Dec 2022 08:39:23 -0800

Thanx, Niklas and Jeffrey. I just didn't think of the generator function.

Sockets, Streams, and Element_Arrays

From: Mark Gardner <magardner2017@gmail.com>
Subject: Sockets, Streams, and Element_Arrays: Much confusion
Date: Sat, 31 Dec 2022 14:11:55 +0200
Newsgroups: comp.lang.ada

Hello, I've been having a bit of difficulty doing some UDP socket programming in Ada. As outlined in my stackoverflow question here (https://stackoverflow.com/q/74953052/7105391), I'm trying to reply to messages I am getting over UDP.

GNAT.Sockets gives me a Stream_Element_Array, which I can't find any documentation on how to make use of other than "You should also be able to get a Stream, which you should use instead" (About ten years ago, on this very newsgroup, somebody said not to use streams with UDP, or at least not GNAT.Sockets.Stream).

Adasockets gives me a String, which I can work with, except it throws away the data recvfrom gives it, apparently making it impossible to reply to the querying address.

At this point, I'm half-tempted to make my own binding, but as I've never done that sort of thing before, I thought I'd ask the wisdom of the Usenet if there is a way to convert a Stream_Element_Array into the exotic types of Unsigned_16 and String.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 31 Dec 2022 14:11:11 +0100

> GNAT.Sockets gives me a Stream_Element_Array [...]

Stream_Element_Array is declared in Ada.Streams as

```
type Stream_Element_Array is
   array(Stream_Element_Offset range <>)
of aliased Stream_Element;
```

For communication purposes it is an array of octets. Your datagram is represented as a Stream_Element_Array or a slice of.

As for streams, yes, it does not make sense to use them for networking, unless you override all stream primitives. The reasons for that are

- non-portability of predefined primitives

- low efficiency for complex data types

- encoding inefficiency as well

You will need to handle some application protocol artifacts, checksums, counters, strange encodings, sequence numbers etc. It is easier to do this directly on the Stream_Element_Array elements.

And, well, do not use UDP, except for broadcasting. There is no reason to use it. For multicast consider delivery-safe protocols like PGM. For single cast use TCP/IP. (If you need low latency see the socket NO_DELAY option)

From: Mark Gardner <magardner2017@gmail.com>
Date: Sat, 31 Dec 2022 15:50:29 +0200

> For communication purposes it is an array of octets.

According to RM 13.13.1, "Stream_Element is mod implementation-defined" which to me says there is no guarantee that they will be octets, unless this is specified elsewhere?

> You will need to handle some application protocol artifacts, checksums, counters, strange encodings, sequence numbers etc. It is easier to do this directly on the Stream_Element_Array elements.

So, how would I do this directly on the elements? I mean, if it is an octet-array to a string, I expect an element-to-element copy, or type conversion to work, but what about integers? Do I need to do something like My_Int:=Unsigned_8(octet(1))+2**8* Unsigned_8(octet(2)); or whatever endianness demands? Or is this the time to learn how to use Unchecked_Conversion?

> And, well, do not use UDP, except for broadcasting.

Well, my use case just so happens to be broadcasting, and re-broadcasting data across a binary-tree-like p2p network.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 31 Dec 2022 15:16:05 +0100

> According to RM 13.13.1, "Stream_Element is mod implementation-defined"

GNAT.Sockets is GNAT-specific. All GNAT compilers have Stream_Element 8 bits. I can imagine some DSP implementation with Stream_Element of 32 bits. But realistically add

```
pragma Assert (Stream_Element'Size >= 8);
```

and be done with that.

[...]

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sat, 31 Dec 2022 16:18:50 +0100

> According to RM 13.13.1, "Stream_Element is mod implementation-defined"

The ARM has always tried to ensure that the language could be implemented on any kind of processor. Thus you have implementation-defined separate definitions of Storage_Element and Stream_Element, which need not be the same, and no guarantee that Interfaces contains declarations of Integer_8 or Unsigned_8.

But these days almost everything is byte oriented, so unless you need what you're writing to work on some unusual H/W, you can presume that both of these are bytes, and that Interfaces contains those declarations.

From: Simon Wright <simon@pushface.org>
Date: Sat, 31 Dec 2022 17:39:07 +0000

> About ten years ago, on this very newsgroup, somebody said not to use streams with UDP, or at least not GNAT.Sockets.Stream.

The reasoning behind the recommendation not to use streams with UDP was as follows (there's a faint possibility that it no longer applies!)

If the data type you want to send is e.g.

```
type Message is record
   Id  : Integer;
   Val : Boolean;
end record;
```

and you create a datagram socket and from that a stream, then use Message'Write to the stream, GNAT will transmit each component of Message separately in canonical order (the order they're written in the type declaration). This results in two datagrams being sent, one of 4 bytes and one of 1 byte.

If you take the same approach at the destination, Message'Read reads one datagram of 4 bytes, and one of 1 byte, and it all looks perfect from the outside. If the destination is expecting a 5 byte record, of course, things won't work so well.

The approach we adopted was to create a 'memory stream', which is a chunk of memory that you can treat as a stream (see for example ColdFrame.Memory_Streams at [1]). With Ada2022, you should be able to use Ada.Streams.Storage.Bounded [2].

Message'Write the record into the memory stream; transmit the written contents as one datagram.

To read, create a memory stream large enough for the message you expect; read a datagram into the memory stream; Message'Read (Stream => the_memory_stream, Item => a_message);

You can use gnatbind's switch -xdr to "Use the target-independent XDR protocol for stream oriented attributes instead of the default implementation which is based on direct binary representations and is therefore target-and endianness-dependent".

[1] https://github.com/simonjwright/coldframe/blob/master/lib/coldframe-memory_streams.ads

[2] http://www.ada-auth.org/standards/22rm/html/RM-13-13-1.html#p25

From: Mark Gardner <magardner2017@gmail.com>
Date: Sat, 31 Dec 2022 21:36:40 +0200

> The approach we adopted was to create a 'memory stream'

Wait, so if I know what shape my data is, and use a memory_stream (like the one in the Big Online Book of Linux Ada Programming chapter 11 [1]), I'm fine using Stream, in conjunction with Get_Address? That's wonderful. Not at all frustrated that I just wasted approximately three working days looking for a solution to a problem that didn't exist.

> Message'Write the record into the memory stream; transmit the written contents as one datagram.

I'm guessing with Memory_Stream'Write(Socket_Stream, Buffer);?

> To read, create a memory stream large enough for the message you expect

Does this second buffer need to be added? If the datagram arrives (UDP), shouldn't GNAT.Sockets.Stream() be able to handle it?

> You can use gnatbind's switch -xdr to "Use the target-independent XDR protocol for stream oriented attributes [...]

Oh fun, I didn't think of that aspect. Thanks! Would I have to pass it as a command line flag, or would there be some kind of pragma I could use?

Thanks for the help so far, and happy new year!

[1] http://www.pegasoft.ca/resources/boblap/11.html#11.12

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 31 Dec 2022 21:16:18 +0100

> I'm guessing with Memory_Stream'Write(Socket_Stream, Buffer);?

No, you create a memory stream object. Then you write your packet into it:

```
My_Message'Write
(My_Memory_Stream'Access);
```

Once written you use the accumulated stream contents to write it into the socket. An implementation of a memory-resident stream is very simple. E.g. see: http://www.dmitry-kazakov.de/ada/strings_edit.htm#Strings_Edit.Streams

My advice would be not to do this. It is wasting resources and complicated being indirect when 'Write and 'Read are compiler-generated. If you implement

'Write and 'Read yourself, then why not call these implementations directly. It just does not make sense to me. I always wonder why people always overdesign communication stuff.

Build messages directly in a Stream_Element_Array. Use system-independent ways to encode packet data. E.g. chained codes for integers. Mantissa + exponent for real numbers. If you have Booleans and enumerations it is a good idea to pack them into one or two octets to shorten the packets. All this is very straightforward and easy to implement.

You can also consider using some standard data representation format, e.g. ASN.1. An Ada ASN.1 implementation is here:
http://www.dmitry-kazakov.de/ada/components.htm#ASN.1

You describe your message in ASN.1 as an Ada tagged type derived from building blocks. Then you can encode and decode it directly from Stream_Element_Array. I would not recommend that either. ASN.1 is quite overblown.

Happy New Year!

From: philip...@gmail.com <philip.munts@gmail.com>
Date: Sat, 31 Dec 2022 14:32:17 -0800

> And, well, do not use UDP, except for broadcasting

I have to disagree here. UDP is perfectly fine for RPC-like (Remote Procedure Call) transactions on a local area network. And it is orders of magnitude easier to implement on microcontrollers than TCP. An Ada program using UDP to communicate with data collecting microcontrollers makes perfect sense in some contexts. I use it for my Remote I/O Protocol.

The only trick is that the server (or responder, as I like to call it) and client (or initiator) can't quite use the same code.

Here is my generic package for UDP with fixed length messages:

https://github.com/pmunts/libsimpleio/blob/master/ada/objects/messaging-fixed-gnat_udp.ads

https://github.com/pmunts/libsimpleio/blob/master/ada/objects/messaging-fixed-gnat_udp.adb

Getting between Stream_Element_Array and a byte array is a pain and I wound up just looping over arrays, copying one byte at a time. If somebody has a better idea, let me know.

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sat, 31 Dec 2022 23:49:33 +0100

> Getting between Stream_Element_Array and a byte array is a pain and I wound up just looping over arrays, copying one byte at a time. If somebody has a better idea, let me know.

You should be able to use Unchecked_Conversion for that.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 31 Dec 2022 23:55:07 +0100

> I have to disagree here. UDP is perfectly fine for RPC-like (Remote Procedure Call) transactions on a local area network.

RPC and other synchronous exchange policies should be avoided as much as possible.

Having said that, implementation of RPC on top of streams is incomparably easier than on top of UDP.

> And it is orders of magnitude easier to implement on microcontrollers than TCP.

Not at all. You need:

- Safe transmission and error correction on top of UDP;
- Buffering and sorting out incoming datagrams;
- Maintaining sequence numbers;
- Splitting messages that do not fit into a single datagram and reassembling them on the receiver side;
- Buffering on the sender side to service resend requests.

This is extremely difficult and a huge load for a microcontroller.

> Getting between Stream_Element_Array and a byte array is a pain and I wound up just looping over arrays, copying one byte at a time. If somebody has a better idea, let me know.

Use "in situ" conversion if you are concerned about copying. E.g.

```
pragma Import (Ada, Y);
for Y'Address use X'Address;
```

From: Simon Wright <simon@pushface.org>
Date: Sat, 31 Dec 2022 23:41:11 +0000

> My advice would be not to do this. [...]

It has to depend on the design criteria.

If you need something now, and it's not performance critical, and you have control over both ends of the channel, why not go for a low-brain-power solution?

On the other hand, when faced with e.g. SNTP, why not use Ada's facilities (e.g. [1]) to describe the network packet and use unchecked conversion to convert to/from the corresponding stream element array to be sent/received?

I'd have thought that building messages directly in a stream element array would be the least desirable way to do it.

[1] https://sourceforge.net/p/coldframe/adasntp/code/ci/default/tree/SNTP.impl/sntp_support.ads

Real_Arrays on Heap with Clean Syntax

From: Jim Paloander <dhmos.altiotis@gmail.com>
Subject: Real_Arrays on heap with overloaded operators and clean syntax
Date: Sun, 22 Jan 2023 13:34:18 -0800
Newsgroups: comp.lang.ada

Dear Ada lovers,

with stack allocation of Real_Vector (1 .. N) when N >= 100,000 I get STACK_OVERFLOW ERROR while trying to check how fast operator overloading is working for an expression

X := A + B + C + C + A + B, where A,B,C,X are all Real_Vector (1 .. N).

So my only option was to allocate on the heap using new. But then I lost the clean syntax

X := A + B + C + C + A + B

and I had to write instead:

X.all := A.all + B.all + C.all + C.all + A.all + B.all.

This is really ugly and annoying because when you are using Real_Arrays for implementing some linear algebra method who relies heavily on matrix vector products and vector updates, you do need to allocate on the heap (sizes are determined in runtime) and you do need a clean syntax. So, is there any way to simplify my life without using the .all or even without declaring A,B,C,X as access Real_Vector?

Thanks for your time!

From: Joakim Strandberg <joakimds@kth.se>
Date: Sun, 22 Jan 2023 13:56:27 -0800

Easiest solution is probably to declare a new task and specify the stack size using the Storage_Size aspect. Allocate as much stack space as you need to be able to do the calculations and do all the allocations on the declared task, not on the environment task. You will avoid the unnecessary heap allocations and have nice clean syntax.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 22 Jan 2023 23:13:14 +0100

You can define "+" on the access type, which should probably be an arena pointer for performance reasons:

```
Arena : Mark_And_Release_Pool;
type Real_Vector_Ptr is access
```

```
 Real_Vector;
for Real_Vector_Ptr'Storage_Pool use
Arena;
function "+" (Left, Right :
Real_Vector_Ptr)
  return Real_Vector_Ptr is
begin
  if Left'Length /= Right'Length then
    raise Constraint_Error;
  end if;
  return Result : Real_Vector_Ptr :=
  new Real_Vector (Left'Range) do
    for I in Result'Range loop
      Result (I) :=
        Left (I) + Right (I - Left'First +
        Right'First);
    end loop;
  end return;
end "+";
```

You can overload that with

```
function "+" (Left : Real_Vector_Ptr;
Right : Real_Vector)
  return Real_Vector_Ptr is
begin
  if Left'Length /= Right'Length then
    raise Constraint_Error;
  end if;
  return Result : Real_Vector_Ptr :=
  new Real_Vector (Left'Range) do
    for I in Result'Range loop
      Result (I) :=
        Left (I) + Right (I - Left'First +
        Right'First);
    end loop;
  end return;
end "+";
```

and with

```
function "+" (Left : Real_Vector;
 Right : Real_Vector_Ptr)
  return Real_Vector_Ptr;
```

Then you will be able to write:

```
X := A + B + C + C + A + B;
-- Use X
Free (X); -- Pop all arena garbage
```

But of course, the optimal way to work large linear algebra problems is by using in-place operations, e.g.

```
procedure Add (Left : in out Real_Vector;
Right : Real_Vector);
```

etc.

Regards,
Dmitry A. Kazakov

http://www.dmitry-kazakov.de

From: Jim Paloander
<dhmos.altiotis@gmail.com>
Date: Sun, 22 Jan 2023 14:49:09 -0800

> It is my impression that in the Ada community the preferred way of working is in general stack only. [...]

With great depression I realized that the preferred way is of stack only. This is very restrictive excluding all scientific modelling involving solvers for partial differential equations, linear algebra kernels, etc. It is insane. Completely insane. 3D simulations of physical phenomena may involve billions of grid-cells and at each grid-cell several unknowns are defined (velocity, pressure, temperature, energy, density, etc). That is why they are using Fortran or C++, but Ada has really cool stuff for so many things, why not vectors and matrices and heap allocation? Would you please give me an example, I googled and I cannot find a single example demonstrating how to use a task with the declaration of stack size. Why is there so little information online about so important things such as allocation?

From: Gautier Write-Only Address
<gautier_niouzes@hotmail.com>
Date: Sun, 22 Jan 2023 15:14:03 -0800

Note that Real_Arrays does not specify where things are allocated (heap or stack).

Only when you define "x : Real_Vector (1 .. n)", it is on stack. You can always write something like the snippet below.

Anyway, after a certain size, you may have to find compromises, like avoiding operators (they do too many allocations & deallocations in the background, even assuming elegant heap-allocated objects) and also give up plain matrices, against sparse matrices or band-stored matrices, typically for solving Partial Differential Equations.

```
with Ada.Numerics.Generic_Real_Arrays;
procedure Test_Large is
  type Float_15 is digits 15;
  package F15_R_A is new
    Ada.Numerics.Generic_Real_Arrays
    (Float_15);
  use F15_R_A;
  procedure Solve_it
   (x : in     Real_Vector;
    y : out   Real_Vector;
    A : in     Real_Matrix) is
  begin
   null;  --  Here, the big number-crunching
  end;
   n : constant := 10_000;
  type Vector_Access is access
  Real_Vector;
  type Matrix_Access is access Real_Matrix;
  x, y : Vector_Access := new Real_Vector
   (1 .. n);
  A    : Matrix_Access := new Real_Matrix
  (1 .. n, 1 .. n);

begin
  Solve_it (x.all, y.all, A.all);
  --  !! Deallocation here
end;
```

From: Leo Brewin
<leo.brewin@monash.edu>
Date: Mon, 23 Jan 2023 12:14:47 +1100

Here is a slight variation on the solution suggested by Gautier. It uses Ada's "rename" syntax so that you can avoid all the .all stuff. I use this construction extensively in my large scale scientific computations.

```
with Ada.Numerics.Generic_Real_Arrays;
with Ada.Unchecked_Deallocation;
procedure Test_Large is
  type Float_15 is digits 15;
  package F15_R_A is new
   Ada.Numerics.Generic_Real_Arrays
   (Float_15);
  use F15_R_A;
  procedure Solve_it
   (x : in     Real_Vector;
    y : out   Real_Vector;
    A : in     Real_Matrix) is
  begin
   null;  --  Here, the big number-crunching
  end;
  n : constant := 10_000;
  type Vector_Access is access
  Real_Vector;
  type Matrix_Access is access
  Real_Matrix;
  x_ptr, y_ptr : Vector_Access := new
  Real_Vector (1 .. n);
  A_ptr        : Matrix_Access := new
  Real_Matrix (1 .. n, 1 .. n);
  x : Real_Vector renames x_ptr.all;
  y : Real_Vector renames y_ptr.all;
  A : Real_Matrix renames A_ptr.all;
  procedure FreeVector is new
    Ada.Unchecked_Deallocation
    (Real_Vector,Vector_Access);
  procedure FreeMatrix is new
    Ada.Unchecked_Deallocation
   (Real_Matrix,Matrix_Access);
begin
  Solve_it (x, y, A);
  -- Deallocation here
  FreeVector (x_ptr);
  FreeVector (y_ptr);
  FreeMatrix (A_ptr);
end;
```

From: Jim Paloander
<dhmos.altiotis@gmail.com>
Date: Sun, 22 Jan 2023 22:01:58 -0800

Thank you very much, would a for Real_Vector_Access'Storage_Pool use localPool; save you from the need to free the vectors and matrix yourself?

On the other hand, is there any way of avoiding temporaries? Possibly a modern version of the Real_Array using expression generic syntax or something similar? Since you are using scientific computations extensively, you must be aware of Fortran. Have you compared Fortran's complex numbers with Ada's for inner products or similar computations to see who is faster? You see, I like a lot of things about Ada, but the syntax is really difficult to follow. Sometimes it gives me the impression that it is more difficult than really needed to be. For example there should be a way for Real_Arrays to allocate memory internally and not on the stack directly like containers. And for containers to provide an indexer Vector(i) and overloaded operators similarly to Real_Vectors. But the fact that they do not give me the impression that this Language, although being designed by the army for mission critical applications, never realized that modern mission critical need to simplify mathematical calculations providing an easy syntax. I am surprised that after so many years and so many updates to the Standard the designers of the Language did not realize that such mathematical syntax should be simplified to attract more people from scientific computing, who are tired with Fortran 10000 ways of declaring something a variable.

From: Egil H H <ehh.public@gmail.com>
Date: Sun, 22 Jan 2023 23:50:11 -0800

> wanted to find the video where Jean Pierre Rosen talks about how memory is handled in the Ada language from FOSDEM perhaps 2018-2019.

It was in 2016: https://archive.fosdem.org/2016/schedule/event/ada_memory/

From: J-P. Rosen <rosen@adalog.fr>
Date: Mon, 23 Jan 2023 09:51:55 +0100

>> Something came up and I had to send my previous reply/e-mail as is. I wanted to find the video where Jean Pierre Rosen talks about how memory is handled in the Ada language from FOSDEM perhaps 2018-2019. Unfortunately I have been unable to find it.

>>

> It was in 2016:

> https://archive.fosdem.org/2016/schedule/event/ada_memory/

Thanks Egil, you were faster than me...

I also have a full tutorial at several Ada-Europe conferences. No video, but I can send the slides to those interested.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 23 Jan 2023 09:28:46 +0100

> I was not sure whether or not it can be avoided with Implicit_Dereference,

> type Accessor (Data: not null access Element) is limited private with Implicit_Dereference => Data;

If you create a new wrapper type, anyway, then it is easier to define operations directly on that new type.

> Otherwise what you described for operator+ one has to do for every operator overloaded inside Real_Arrays package.

You should not use the standard library anyway. It is not intended for large problems, which require specific approaches and methods, like sparse matrices, concurrent processing and so on.

> The optimal way to work large linear algebra problem is what you describe because unfortunately Ada does not allow what Fortran does since 30 years ago or more.

I am not sure what you mean. It is quite possible to design a wrapper datatype allocating vectors/matrices in the pool. E.g. Ada's Unbounded_String is such a thing. Real_Arrays were not designed this way because see above.

> But in C++ you can reproduce the same functionality as Fortran using Expression Templates and Template Metaprogramming.

Nothing prevents you from wrapping Real_Array in a generic way:

```
generic
   with package Real_Arrays is new
Numerics.Generic_Real_Arrays (<>);
package Generic_Pool_Real_Arrays is
   ...
end Generic_Pool_Real_Arrays;
```

> Perhaps Ada should allow something like that. Because for maintainability reasons the best would be to write the mathematical expressions as close as possible to the mathematical formulas.

There is no problem with that as you can define operations on pointers.

From: G.B. <bauhaus@notmyhomepage.invalid>
Date: Mon, 23 Jan 2023 09:39:39 +0100

> Are you aware of any libraries similar to Real_Arrays, but who allocated memory internally using heap?

The most natural way to work with an array of FPT numbers is for the programmer to declare an array indexed by some index type. Done. If GNAT gets in the way there, it might be worth a note sent to its maintainers. Whenever a programmer is tasked with considering memory allocation, then depending on one's propensity towards working on memory allocation it is inconvenient and distracting. Math programs don't make you do this, I think.

Also, std::vector and its relatives shield the programmer from the absurdly clever, yet unreadable memory allocation that needs to be stuffed behind the scenes. More importantly, though, C++ introduced std::move semantics after a few decades of its existence, to address copying when using chains of +. It might be interesting to see Ada's in-situ construction of return values in comparison.

> Similarly to the Containers.Vector. But Vector has such an awful syntax. There should be something like an indexer [i] similarly to the C++ std::vector to make things simpler

.at() does some of what Ada does. Is

```
v.at(k) = 4;
```

less awful than

```
v(k) := 4;
```

?

Another thing: Mathematical notation has ellipsis, thus

```
A + B + ... + Y + Z;
```

Most general purpose languages don't have ellipsis for this kind of expression. However, even mathematical formulas use what programmers can usually achieve, too. The usual

```
\sum_k A_k.
```

No "+" at all, and an array of vectors, not single ones. Going further, some like to write

```
reduce("+", A);
```

In Ada, you could have a generic function for this, or use a function pointer.

The .all thing vanishes automatically whenever you want to refer to a particular component of the pointed-at object, as opposed to all of them. So, A.all(K) is the same as A(K). Likewise, .all can be dropped if want to invoke the pointed-at subprogram if it has parameters.

Broadcast / Iterate to All Connection Objects via Simple Components?

From: A.J. <ianozia@gmail.com>
Subject: Broadcast / iterate to all Connection objects via Simple Components?
Date: Tue, 7 Feb 2023 12:29:39 -0800
Newsgroups: comp.lang.ada

Hello everyone,

In an effort to better learn network programming in Ada, I've been working through the Protohacker Challenges (https://protohackers.com/), and the current challenge (number 3) is to create a chat server.

I am using a TCP Connections Server with Simple Components, specifically a Connection_State_Machine, but I've run into a problem. I'm trying to send a message received via "procedure Process_Packet (Client : in out Server_Connection)" to all connected Clients.

My (potentially incorrect) thought on how to accomplish this is to iterate through all of the clients currently connected, and use Send to send the message received to those clients. I've been struggling with how to actually do this though, since I couldn't use "function Get_Clients_Count (Listener : Connections_Server) return Natural" from within Process_Packets.

Another thought I had could be to just place every message received in a central queue, and then once all of the packets have been received, to then process that queue and send the results to every connected client.

I tried overriding "procedure On_Worker_Start (Listener : in out Connections_Server)", thinking that I could use it to read such a queue, but it never seemed to be called from within my program and I'm still unsure how to iterate through the Connection objects anyway.

Am I approaching this the right way, or am I missing something very obvious? I've read the test files that came with Simple Components, including the data server but couldn't see a way to get each client to interact with each other. If I didn't explain this well enough, please let me know, I'll be happy to clarify.

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Wed, 8 Feb 2023 10:55:11 +0100

For an example of this, see the Chattanooga demo that comes with Gnoga (https://sourceforge.net/projects/gnoga/). A screenshot and intermittently working (not right now) on-line version are available at https://sourceforge.net/p/gnoga/wiki/Gnoga-Gallery/.

From: Emmanuel Briot <briot.emmanuel@gmail.com>
Date: Sun, 12 Feb 2023 23:28:26 -0800

I am not sure how familiar you are with Network programming in general (not just as it would be done in Ada). Using a blocking Send could actually kill your performance. You mentioned you would be sending a message to one client after another. Imagine one of the clients has small socket buffers, and is busy doing something else at the moment so not reading your message immediately. If you are sending a large message, your server would only be able to send part of the message, then it would block until the client has read enough that there is space again in the socket buffers to send the rest of the message. That could take ... days. In the meantime, your server is not doing anything else, and no other client gets sent anything...

Instead, you need to use non-blocking sockets. When Send returns, it has sent whatever it could for the moment. You then need to monitor the socket (and all other similar ones) using something like select (which is limited to sockets < 1024, so pretty useless for an actual server in practice) poll (better version of select) or epoll (the best in my opinion). I have written a similar server that has 25000 concurrent clients, and serves them all with 10 worker tasks. That would never fly with blocking sockets.

A similar approach when receiving messages from clients, by the way. The message might have sent only part of its message, so you need to give up temporarily, and come back to it when poll tells you there is something new to read.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 13 Feb 2023 09:30:22 +0100

> Using a blocking Send could actually kill your performance. [...] A similar approach when receiving messages from clients, by the way.

Yes. All networking in Simple components is built on non-blocking sockets (socket select).

P.S. This poses difficulties for users, who see all communication turned upside down being driven by arbitrary socket events rather than by the protocol logic. This was a reason I argued for introducing co-routines with task interfaces in Ada.

From: Emmanuel Briot <briot.emmanuel@gmail.com>
Date: Mon, 13 Feb 2023 00:44:01 -0800

> sockets (socket select).

Have you taken a look at epoll(), on Linux it is so much more natural to use, and so much more efficient in practice. The example I mentioned above (a server with 25_000 concurrent connections) cannot work with select (which only accepts file descriptors up to 1024), and is slow with poll (since the result of the latter is the number of events, and we need to iterate over all registered sockets every time).

> This was a reason I argued for introducing co-routines with task interface in Ada.

In my own code, I basically provide an epoll-based generic framework. One of the formal parameters is a `Job_Type` with one primitive operation `Execute`. The latter is initially called when a new connection is established, and is expected to do as much non-blocking work as it can (Execute is run in one of the worker tasks). When it cannot make progress, it returns a tuple (file_descriptor, type_of_event_to_wait_for) to indicate when it needs to be called again in the future, for instance some data became available to read on the specified file_descriptor. The intent is that the `Job_Type` is implemented as a state machine internally.

Of course, a state machine is one of the two ways I know (along with a task) to implement the equivalent of a co-routine in Ada. So I 100% agree with you that co-routines would be very useful in simplifying user code, in particular in the scenario we are discussing here!

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 13 Feb 2023 11:55:07 +0100

> Have you taken a look at epoll(), on Linux ?

The implementation is on top of GNAT.Sockets, so no.

> It is so much more natural to use, and so much more efficient in practice.

Well, if there is Linux kernel level support why it is not used in socket select as it is in epoll? I would expect them do that at some point or drop epoll... (:-))

> [...] The intent is that the `Job_Type` is implemented as a state machine internally.

Yes, state machine is what I want to avoid. With complex layered protocols it imposes incredible difficulties requiring auxiliary stacks and buffers with errors almost intractable either by testing or by formal proofs.

> So I 100% agree with you that co-routines would be very useful in simplifying user code, in particular in the scenario we are discussing here!

I'd like to have special Ada "tasks" acting as co-routines on top of proper tasks yielding when the socket buffer is empty or full.

From: Emmanuel Briot <briot.emmanuel@gmail.com>
Date: Mon, 13 Feb 2023 03:07:04 -0800

> Well, if there is Linux kernel level support why it is not used in socket select as it is in epoll?

Because in practice the Linux developers don't get to modify such APIs, which are mandated by Posix, or Unix, or some RFC. So the API for select and poll will *never* change.

epoll is definitely the modern approach on Linux, until of course someone finds something even better. epoll is fully thread safe too, which is very nice when used from Ada. Using select() is totally outdated at this point, and means you can never handle more than 1000 simultaneous clients, and that only if you do not have other file descriptors open (database, files,...)

The person who developed GNAT.Sockets has left AdaCore a while ago, so "they" (which I assume is what your message was referring to) are actually unlikely to update that. They also have strong concerns about platform-agnostic support, and epoll is linux-specific at this point (likely also BSD). There exist multiple libraries out there that provide an API common to multiple platforms, and that use epoll on linux. Maybe that's what would make sense, but nowadays with Alire, I would expect someone to build a crate there rather than AdaCore modify GNAT.Sockets.

> Yes, state machine is what I want to avoid. With complex layered protocols it imposes incredible difficulties requiring auxiliary stacks and buffers

with errors almost intractable either by testing or by formal proofs.

Tell me about auxiliary stacks :- In practice, in my experience, you can have a single incoming buffer which is used by one state, and then another when the first state is no longer active,... so we do not need to have too many buffers, but that definitely is not trivial. Currently, I have a stack of iterators reading from a socket, buffering on top of that, then decompressing LZ4 data, then decoding our binary encoding to Ada values.

> I'd like to have special Ada "tasks" acting as co-routines on top of proper tasks yielding when the socket buffer is empty or full.

This is an approach we had discussed at AdaCore before I left. There are multiple drawbacks here: the limited stack size for tasks by default (2MB), the fact that entries cannot return indefinite types, the fact that currently those tasks are assigned to OS threads (so too many of them does impact resource usage),...

A colleague had found an external library that would provide several stacks and thus let people implement actual co-routines. We did not do much more work on that, but it was a nice proof of concept, and efficient. I think things are mostly blocked now, as the ARG has been discussing these topics for quite a few years now.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 13 Feb 2023 12:57:19 +0100

> [...] This is an approach we had discussed at AdaCore before I left. [...]

My idea is to have these as pseudo-tasks scheduled by the Ada run-time and not mapped onto any OS threads. A proper thread would pick up such a task and run it until it yields. The crucial point is to use the stack of the pseudo-task in place of the thread's stack or backing it up and cleaning the portion of the stack at the point of yielding, whatever.

> [...] the ARG has been discussing these topics for quite a few years now.

I have an impression that ARG's view on co-routines totally ignores the use case of communication stacks and other cases state machines show their ugly faces...

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Mon, 13 Feb 2023 15:22:19 +0200

[snip discussion of network programming details, retain discussion about co-routines]

So your co-routines would (1) have their own stack and (2) be independently schedulable, which implies (3) having their own execution context (register values, instruction pointer, etc.) How is that different from the Ada concept of a "task"? How could the ARG separate between a "task" and a "co-routine" in the Ada RM?

There exist Ada compilers and run-times where the tasking concept is implemented entirely in the run-time system, without involving the underlying OS (if there even is one). That approach was mostly abandoned in favour of mapping tasks to OS threads, which makes it easier to use potentially blocking OS services from tasks without blocking the entire Ada program.

So is your problem only that using OS threads is less "efficient" than switching and scheduling threads of control in the run-time system? If so, that seems to be a quality-of-implementation issue that could be solved in a compiler-specific way, and not an issue with the Ada language itself.

The point (from Emmanuel) that task entries cannot return indefinite types is certainly a language limitation, but seems to have little to do with the possible differences between tasks and co-routines, and could be addressed on its own if Ada users so desire.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 13 Feb 2023 16:10:15 +0100

> So your co-routines would (1) have their own stack and (2) be independently schedulable, which implies (3) having their own execution context (register values, instruction pointer, etc.)

Sure. You should be able to implement communication logic in a natural way:

1. Read n bytes [block until finished]

2. Do things

3. Write m bytes [block until finished]

4. Repeat

> How is that different from the Ada concept of a "task"?

It is no different, that the whole point of deploying high level abstraction: task instead of low level one: state machine.

> How could the ARG separate between a "task" and a "co-routine" in the Ada RM?

Syntax sugar does not bother me. I trust ARG to introduce a couple of reserved words in the most annoying way... (:-))

> So is your problem only that using OS threads is less "efficient" than switching and scheduling threads of control in the run-time system?

This too. However the main purpose is control inversion caused by callback architectures. A huge number of libraries are built on that pattern. This is OK for the library provider because it is the most natural and efficient way. For the user implementing his own logic, be it communication protocol, GUI etc. it is a huge architectural problem as it distorts the problem space logic. So the goal is to convert a callback/event driven architecture into plain control flow.

> If so, that seems to be a quality-of-implementation issue that could be solved in a compiler-specific way, and not an issue with the Ada language itself.

In Ada 83 there was no way to pass a procedure as a parameter. We used a

task instead... (:-))

But sure, a possibility to delegate a callback to an entry call without intermediates is certainly welcome.

[...]

From: J-P. Rosen <rosen@adalog.fr>
Date: Mon, 13 Feb 2023 16:43:31 +0100

> that task entries cannot return indefinite types is certainly a language limitation

That's what Holders are intended for... (changing indefinite types into a definite one)

From: Jeremy Grosser <jeremy@synack.me>
Date: Mon, 13 Feb 2023 08:40:05 -0800

> epoll is definitely the modern approach on Linux, until of course someone finds something even better.

For high performance networking, io_uring [1] is the new kid on the block, but the API involves a scary amount of pointer manipulation, so I'm not convinced that it's safe to use yet.

While epoll is thread safe, there are some subtleties. If you register a listening socket with epoll, then call epoll_wait from multiple threads, more than one thread may be woken up when the socket has a waiting incoming connection to be accepted. Only one thread will get a successful return from accept(), the others will return EAGAIN. This wastes cycles if your server handles lots of incoming connections. The recently added (kernel >=4.5) EPOLLEXCLUSIVE flag enables a mutex that ensures the event is only delivered to a single thread.

> They also have strong concerns about platform-agnostic support, and epoll is linux-specific at this point (likely also BSD). [...]

On BSD, the kqueue [2] API provides similar functionality to epoll. I believe kqueue is a better design, but you use what your platform supports.

libev [3] is the library I see used most commonly for cross-platform evented I/O. It will use the best available polling syscalls on whatever platform it's compiled for. Unfortunately, it's composed mostly of C preprocessor macros.

I've already written an epoll binding [5] that's in the Alire index. GNAT.Sockets provides the types and bindings for the portable syscalls.

For the Protohackers puzzles, I've written a small evented I/O server using those bindings [6]. Note that this server does not use events for the send() calls yet, which may block, though in practice it isn't an issue with the size of the payloads used in this application. I do plan to refactor this to buffer data to be sent when the Writable (EPOLLOUT) event is ready.

So far, I've found tasks and coroutines to be unnecessary for these servers, though coroutines would make it possible to implement Ada.Streams compatible Read and Write procedures, providing a cleaner interface that doesn't expose callbacks to the user.

[1] https://lwn.net/Articles/776703/

[2] https://people.freebsd.org/~jlemon/papers/kqueue.pdf

[3] https://linux.die.net/man/3/ev

[4] https://github.com/JeremyGrosser/epoll-ada

[5] https://github.com/JeremyGrosser/protohackers/blob/master/src/mini.adb

From: philip...@gmail.com <philip.munts@gmail.com>
Date: Mon, 13 Feb 2023 17:55:52 -0800

> In an effort to better learn network programming in Ada, I've been working through the Protohacker Challenges (https://protohackers.com/), and the current challenge (number 3) is to create a chat server.

I know it probably defeats the purpose of what you are trying to learn, but you are going to wind up just reinventing AMQP (broker based, meaning there is a intermediary computer running something like RabbitMQ to manage message queues) or ZeroMQ (brokerless), both implementations of so-called enterprise messaging protocols. Both seem to scale pretty well to thousands of clients.

It is pretty easy to do an Ada thin binding for the ZeroMQ C library libzmq.

From: A.J. <ianozia@gmail.com>
Date: Sat, 18 Feb 2023 17:27:02 -0800

Thank you for all of the responses and discussion, it pointed me in the right direction! The "chat server"[1] (if you could call it that) does work, and my friends and I were able to telnet into it and chat. One of my friends even tried throwing things at the server to break it, but it didn't crash!

Dmitry, maintaining a list of clients was the vital part I was missing. I played around with using synchronized queues and tasks, but ended up defaulting to an ordered map with a UID as the key and wrapped it in a protected type. I couldn't get Send() to send more data than Available_To_Send (after calling it, Available_To_Send ended up returning 0, and continued to do so despite wrapping Send() in a loop), but increasing the send buffer to 8kb per connection worked fine. I would simply loop through that ordered map each time I needed to send something to all of the clients.

I really like simple components, and it would be neat if the GNAT maintainers implement epoll in the backend for Linux systems, kqueue for BSD and MacOS. Any server I write will be for Linux though anyway. I'm also interested in trying to benchmark Simple Component's connections server (both pooled and standard) against epoll to see how it fares. Perhaps the clever tasking that the Connections Server utilizes can keep up with epoll despite what GNAT.Sockets utilizes!

Regarding coroutines vs tasks, I think at a high level it's hard to differentiate, but at a lower level, when I think of tasks vs what a coroutine would be, I think of Go, and their "goroutines."[2] Creating a task in Ada, at least on Linux, ends up creating a pthread, and you get all of the overhead that comes with threading (it's initialized in the kernel). coroutines are managed by the go runtime (I believe in user space) and have much less overhead to create or manage, since it's not creating a specific thread.

Ada 202x supports the "parallel" block[3] though I understand no runtime has utilized it yet-- would that end up being a coroutine or is it meant for something else?

[1] https://github.com/AJ-Ianozi/protohackers/tree/main/budget_chat/src

[2] https://www.geeksforgeeks.org/golang-goroutine-vs-thread/

[3] http://www.ada-auth.org/standards/22rm/html/RM-5-6-1.html

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sun, 19 Feb 2023 16:37:51 +0200

> Creating a task in Ada, at least on Linux, ends up creating a pthread

With the current GNAT compiler, yes. But not necessarily with all Ada compilers, even on Linux.

> coroutines are managed by the go runtime (I believe in user space) and have much less overhead to create or manage

Some Ada compilers may have run-times that implement Ada tasks within the run-time, with minimal or no kernel/OS interaction.

> Ada 202x supports the "parallel" block [...]-- would that end up being a coroutine or is it meant for something else?

As I understand it, the parallel execution constructs (parallel blocks and parallel loops) in Ada 2022 are meant to parallelize computations using multiple cores -- that is, real parallelism, not just concurrency.

The Ada2022 RM describes each parallel computation in such a parallel construct as its own thread of control, but all operating within the same task, and all meant to be /independent/ of each other. For example, a computation on a vector that divides the vector into non-overlapping chunks and allocates one core to each chunk.

Within a parallel construct (in any of the parallel threads) it is a bounded error to invoke an operation that is potentially blocking. So the independent computations are not expected to suspend themselves, thus they are not co-routines.

The parallelism in parallel blocks and parallel loops is a "fork-join" parallelism. In other words, when the block or loop is entered all the parallel threads are created, and all those threads are destroyed when the block or loop is exited.

So they are independent threads running "in" the same task, as Dmitry wants, but they are not scheduled by that task in any sense. The task "splits" into these separate threads, and only these, until the end of the parallel construct.

Moreover, there are rules and checks on data-flow between the independent computations, meant to exclude data races. So it is not intended that the parallel computations (within the same parallel construct) should form pipes or have other inter-computation data flows.

Ada Array Contiguity

From: Rod Kay <rodakay5@gmail.com>
Subject: Ada array contiguity.
Date: Mon, 20 Feb 2023 00:34:55 +1100
Newsgroups: comp.lang.ada

I've been told that Ada array elements are not guaranteed to be contiguous unless the 'Convention C' aspect is applied.

Is this correct?

From: J-P. Rosen <rosen@adalog.fr>
Date: Sun, 19 Feb 2023 15:28:23 +0100

The strength of Ada is that it protects you from all implementation details, thus allowing compilers to choose the most efficient implementation. Therefore, the answer is yes.

(BTW: try to find a definition of "contiguous". At byte level? At word level? What if the element does not fill a byte?)

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sun, 19 Feb 2023 16:59:42 +0200

> Therefore, the answer is yes.

I tried to find a rule on "contiguity" in the Ada 2022 RM, but failed. Can you point to one? Perhaps this rule is a consequence of C standard rules for arrays (pointer arithmetic), and the general idea that Ada should allow Convention C for a type only if that type is really compatible with the C compiler (in question).

For a constrained array type I would choose to specify the size of the component type, and the size of the array type to be the length of the array times the component size. That should (also) ensure that the elements are stored contiguously (if the Ada compiler accepts this size specification).

It seems (RM B.3(62.4/3)) that Ada compilers are not required to support Convention C for unconstrained array types. RM B.3 (Interfacing with C/C++) declares such types with the Pack aspect, but that may or may not (AIUI) give a contiguous representation.

> (BTW: try to find a definition of "contiguous". At byte level? At word level? What if the element does not fill a byte?)

Indeed. But it seems to me that Arr'Size = Arr'Length * Comp'Size is the meaning usually intended for programming purposes.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 19 Feb 2023 16:08:09 +0100

> it seems to me that Arr'Size = Arr'Length * Comp'Size is the meaning usually intended for programming purposes.

Rather: the bit offset of an element is a linear function of its position.

From: J-P. Rosen <rosen@adalog.fr>
Date: Sun, 19 Feb 2023 18:10:44 +0100

> it seems to me that Arr'Size = Arr'Length * Comp'Size is the meaning usually intended for programming purposes.

Certainly not if Comp'Size is not an integer number of bytes.

From: Niklas Holsti <niklas.holsti@tidorum.invalid>
Date: Sun, 19 Feb 2023 19:54:13 +0200

> Certainly not if Comp'Size is not an integer number of bytes.

I'm not so certain. By choosing various roundings-up of the component size, you can choose between "bit-contiguous", "byte-contiguous", etc.

For example, bit-contiguous with 2-bit components:

```
type Comp is (A, B, C, D) with Size => 2;
type Arr is array (1 .. 10) of Comp
  with Pack, Size => 10 * Comp'Size;
```

Nybble-contiguous with Comp'Size => 4, byte- (octet-) contiguous with Comp'Size => 8, etc.

(However, I haven't checked that eg. GNAT does the "right thing" with such Size clauses, just that it accepts them. It does require the Pack aspect for the array type when Comp'Size is not a multiple of 8.)

> Rather: the bit offset of an element is a linear function of its position.

That is ordering by index, but not contiguity: there may still be gaps between elements. However, I assume you meant that the slope of the linear function equals the component size, and then it includes contiguity.

The relationship of index order to memory-location order is certainly an aspect that should be considered when interfacing to C or HW.

Pet peeve: on more than one occasion I have been disappointed that Ada representation clauses do not let me specify the index-order of packed array elements in a word, relative to the bit-numbering order, and I have had to fall back to using several scalar-type record components, c1 .. c7 say, instead of one array-type component, c(1..7).

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 19 Feb 2023 20:05:28 +0100

> That is ordering by index, but not contiguity: there may still be gaps between elements. [...]

No gaps = packed = the most dense representation.

Contiguity is rather that the gaps are regular and can be considered a part of each element. E.g. a video buffer with strides is not contiguous.

> The relationship of index order to memory-location order is certainly an aspect that should be considered when interfacing to C or HW.

An definition of contiguous array equivalent to linearity is that the array body representation is isomorphic to slicing.

> Pet peeve [...]

This is as blasphemous as asking for n-D slices... (:-))

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sun, 19 Feb 2023 23:02:36 +0100

> I've been told that Ada array elements are not guaranteed to be contiguous unless the 'Convention C' aspect is applied.

The ARM says little about how the compiler represents objects in the absence of representation clauses. However, ARM 13.7(12) (http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-13-7-1.html#I5653) says, "Storage_Array represents a contiguous sequence of storage elements."

ARM 13.9(17/3) (http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-13-9.html#I5679) says that a compiler that supports Unchecked_Conversion should use a contiguous representation for certain constrained array subtypes.

Using convention Fortran should also ensure a contiguous representation, add can apply (unlike convention C) to multidimensional arrays.

From: J-P. Rosen <rosen@adalog.fr>
Date: Mon, 20 Feb 2023 08:12:41 +0100

> type Comp is (A, B, C, D) with Size => 2;

> type Arr is array (1 .. 10) of Comp

> with Pack, Size => 10 * Comp'Size;

> Nybble-contiguous with Comp'Size => 4, byte- (octet-) contiguous with Comp'Size => 8, etc.

Of course, if you add representation clauses, the compiler will obey them. But the OP's question was whether it was /guaranteed/ to have contiguous representation, and the answer is no - for good reasons.

From: Rod Kay <rodakay5@gmail.com>
Date: Thu, 2 Mar 2023 00:22:25 +1100

Thank you all for the replies.

To summarise then, contiguity is not guaranteed unless the array is of convention C, convention Fortran or representation clauses are applied.

Ada.Containers.Vectors Capacity

From: Rod Kay <rodakay5@gmail.com>
Subject: Is this a compiler bug?
Date: Sun, 19 Mar 2023 17:17:20 +1100
Newsgroups: comp.lang.ada

Came across this during a port of the Box2D physics engine.

It's a generic Stack package using 'ada.Containers.Vectors' to implement the stack.

One generic parameter is the 'initial_Capacity' of the stack, used in the 'to_Stack' construction function, via the Vectors 'reserve_Capacity' procedure.

In the 'to_Stack' function, the Capacity is reserved correctly but in the test program when the stack is created and assigned to a variable, the capacity is 0.

Here is the (very small) source code ...

https://gist.github.com/charlie5/7b4d863227a510f834c2bfd781dd50ba

The output I get with GCC 12.2.0 is ...

```
[rod@orth bug]$ ./stack_bug

to_Stack ~ Initial Capacity: 256

to_Stack ~ Before reserve:  0

to_Stack ~ After reserve:  256

stack_Bug ~ Actual Capacity: 0
```

Regards.

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sun, 19 Mar 2023 11:33:50 +0100

I think this is acceptable behavior. See ARM A.18.2 (147.19/3, 147.20/3, & 147.b/3) (http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-A-18-2.html). The first two sections define the behavior of procedure Assign, while the last states "Assign(A, B) and A := B behave identically".

Assign (A, B) only changes the capacity of A if A.Capacity < B.Length.

So if the compiler does not use build-in-place for the initialization of the variable, then the assignment of the function result should not change the capacity of the variable from its (apparent) default of zero (there is, of course, no requirement for the capacity of a default-initialized vector).

The discussion of capacities for vectors is only meaningful for a subset of possible implementations, so messing with capacities may have no meaningful effect at all.

For an unbounded stack based on a linked list (with no concept of capacity) you could use PragmARC.Data_Structures.Stacks.Unbounded.Unprotected (https://github.com/jrcarter/PragmARC/blob/Ada-12/pragmarc-data_structures-stacks-unbounded-unprotected.ads).

From: Rod Kay <rodakay5@gmail.com>
Date: Mon, 20 Mar 2023 13:24:40 +1100

Thank you, Jeffrey, for the detailed reply.

I'm now using a limited record with an extended return for 'build-in-place' initialisation and am getting the behavior I desired.

Why Don't All Initialising Assignments Use 'build-in-place'?

From: Rod Kay <rodakay5@gmail.com>
Subject: Why don't all initialising assignments use 'build-in-place' ?
Date: Tue, 21 Mar 2023 23:06:03 +1100
Newsgroups: comp.lang.ada

I'm sure there must be a good reason. All I can think of is that it may somehow break backwards compatibility wrt controlled types (a vague stab in the dark).

Any thoughts?

From: Randy Brukardt <randy@rrsoftware.com>
Date: Sat, 25 Mar 2023 03:39:14 -0500

(1) Didn't want to make work for implementers.

(2) You shouldn't be able to tell (since it is required for all cases involved finalization). Finalization is the only way to inject user-defined code into the initialization process.

(3) True build-in-place can be expensive and complex (especially for array types).

(4) Build-in-place requires functions compiled to support it (must pass in the place to initialize into). That might not be the case (especially if a foreign convention is involved). Also see (3) - an implementation might have a cheaper way to return some types that doesn't support build-in-place.

There's probably more, those are off the top of my head. If it is cheap, it would be silly for an implementation to do anything else. (Don't ask what Janus/Ada does. ;-) Otherwise, most people want the fastest possible code.

From: Rod Kay <rodakay5@gmail.com>
Date: Sun, 26 Mar 2023 16:10:33 +1100

Thanks, Randy. I somehow imagined that build-in-place would be faster :/.

So using 'extended return' *everywhere* would decrease performance, I guess.

From: Jeffrey R.Carter <spam.jrcarter.not@spam.acm.org.not>
Date: Sun, 26 Mar 2023 12:41:00 +0200

> So using 'extended return' *everywhere* would decrease performance, I guess.

You seem to think that using an extended return requires building in place. This is not required by the ARM.

"Built in place" is defined in ARM 7.6 (17.1/3-17.p/3) (http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-7-6.html#I4005). An initial value is required to be built in place when

1. The object (or any part of the object) being initialized is immutably limited
2. The object (or any part of the object) being initialized is controlled and the initialization expression is an aggregate

In all other cases, it is up to the compiler to decide whether or not to build in place.

This holds regardless of the the kind of return statement used if the initialization expression is a function call.

Thus the initialization of an immutably limited object is done in place even if the initialization expression is

* an aggregate

* a function call with a simple return statement

while the initialization of an integer object may be by copy even if the initialization expression is a function call with an extended return statement.

From: Rod Kay <rodakay5@gmail.com>
Date: Mon, 27 Mar 2023 15:44:33 +1100

> You seem to think that using an extended return requires building in place. This is not required by the ARM.

Yes, I did rather think that. Appreciate the correction.

Assignment Access Type with Discriminants

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Subject: Assignment access type with discriminants
Date: Wed, 22 Mar 2023 10:19:28 +0100
Newsgroups: comp.lang.ada

I stumbled on a curious fact.

The value of an object with a discriminant can be changed to a value with a different discriminant if the type's discriminants are defaulted.

Right?

Wrong! Not through an access type!

```
procedure Test is
   type F is (F1, F2, F3);
   type Foo (K : F := F1) is record
      case K is
         when F1 =>
            X1 : Integer;
         when F2 =>
            X2 : Float;
         when F3 =>
            X3 : String (1..2);
      end case;
   end record;
   type Foo_Ptr is access all Foo;
   X : aliased Foo;
   P : Foo_Ptr := X'Access;
begin
   X := (F2, 1.0);   -- OK
   P.all := (F1, 3); -- Error!
end Test;
```

Is this a compiler bug or intentional language design? Any language lawyers?

From: Björn Lundin <bnl@nowhere.com>
Date: Wed, 22 Mar 2023 10:31:58 +0100

> I stumbled on a curious fact. [...] Is this a compiler bug or intentional language design? Any language lawyers?

I get

```
Execution of ./test terminated by
unhandled exception

raised CONSTRAINT_ERROR :
test.adb:18 discriminant check failed

Call stack traceback locations:

0x402c33 0x402b27 0x7f335b5cfd8e
0x7f335b5cfe3e 0x402b63
0xfffffffffffffe

bnl@hp-t510:/usr2$ gnatls -v

GNATLS Pro 22.2 (20220605-103)
```

Linux 64bit - ubuntu 22.04

So it is (also) present on that platform at least

From: G.B.
<bauhaus@notmyhomepage.invalid>
Date: Wed, 22 Mar 2023 15:10:44 +0100

Some experiments point at the general access type.

```
   type Foo_Ptr is access Foo; -- sans `all`
   X : Foo;
   P : Foo_Ptr := new Foo;
   type Foo1 is new Foo_Ptr (K => F1);
begin
   X := (F2, 1.0);   -- OK
   P.all := (F1, 3); -- _no_ Error!
   Foo1 (P).all := (F1, 3);
end Test;
```

(Doesn't rejection for general access types seem reasonable if assignment would otherwise require adjusting the storage layout of a variable, including all access paths to components? Just guessing.)

From: Dmitry A. Kazakov
<mailbox@dmitry-kazakov.de>
Date: Thu, 23 Mar 2023 12:51:03 +0100

> Some experiments point at the general access type.

You get no error because you do not change the discriminant. Change your code to:

```
   P.all := (F2, 1.0); -- Error!
```

> (Doesn't rejection for general access types seem reasonable if assignment would otherwise require adjusting the storage layout of a variable, including all access paths to components?

I guess that an implementation must allocate memory for any value unless you constraint the discriminants in a subtype. But I am not a language lawyer to judge.

From: Adamagica
<christ-usch.grein@t-online.de>
Date: Thu, 23 Mar 2023 09:53:23 -0700

I do hope, this answers the question:

3.10(14/3) … The first subtype of a type defined by … an access_to_object_definition is unconstrained if the designated subtype is an ... discriminated subtype; otherwise, it is constrained.

4.8(6/3) If the designated type is composite, then … the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

From: Niklas Holsti
<niklas.holsti@tidorum.invalid>
Date: Thu, 23 Mar 2023 20:09:21 +0200

> I do hope, this answers the question:
>
> 3.10(14/3) … The first subtype of a type defined by … an access_to_object_definition is unconstrained if the designated subtype is an ... discriminated subtype; otherwise, it is constrained.

What do you infer from this, relating to Dmitry's original example code and the error? The "first subtype .. defined" here is the access subtype, and I don't see how that affects an assignment /via/ this access subtype to the accessed object.

(It is not clear to me how an access subtype that is constrained differs from one that is unconstrained. Can someone clarify?)

> 4.8(6/3) If the designated type is composite, then … the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

That rule applies to objects created by allocators, but the original example code has no allocators (some later variants do). The object in question is created by a declaration (which includes the "aliased" keyword), not by an allocator.

Also, AARM 3.10 contains the following notes on "Wording Changes from Ada 1995":

26.d/2 {AI95-00363-01} Most unconstrained aliased objects with defaulted discriminants are no longer constrained by their initial values. [...]

26.k/2 {AI95-00363-01} The rules about aliased objects being constrained

by their initial values now apply only to allocated objects, and thus have been moved to 4.8, “Allocators”.

This seems to mean that aliased objects created by declarations are /not/ constrained by the initial value, so it should be possible to change the discriminant. This seems to be a change from Ada 95 to Ada 2005. I don't see why that change could not be done via an access to the object.

I added some output to Dmitry's original code, with this result:

From: Niklas Holsti
<niklas.holsti@tidorum.invalid>
Date: Thu, 23 Mar 2023 20:55:48 +0200

```
   X'Constrained = FALSE
   P'Constrained = TRUE
   P.all'Constrained = TRUE
```

The first two values of 'Constrained (for X and P) are as expected by the RM rules, and the third value (for P.all) is consistent with the error, and seems valid for Ada 95, but the wording change quoted above suggests that it is wrong for Ada 2005 and later. This leads me to suspect that GNAT has not been fully updated for this RM change, so it would be a GNAT bug. Still, the addition of

```
   subtype Foo2_Ptr is Foo_Ptr (K => F2);
```

to Dmitry's original example provokes this error message:

```
fuf.adb:16:24: access subtype of
general access type not allowed

fuf.adb:16:24: discriminants have
defaults
```

which suggests that at least this part of AI95-00363 has been implemented, as noted in AARM 3.10:

14.b/2 Reason: {AI95-00363-01} [...] Constraints are not allowed on general access-to-unconstrained discriminated types if the type has defaults for its discriminants [...]

From: J-P. Rosen <rosen@adalog.fr>
Date: Thu, 23 Mar 2023 18:04:36 +0100

> I stumbled on a curious fact.

An access value is always constrained by its initial value; this is necessary because of constrained access subtypes. Here is a slightly modified version of your example:

```
procedure Test is
   type F is (F1, F2, F3);

   type Foo (K : F := F1) is record
      case K is
         when F1 =>
            X1 : Integer;
         when F2 =>
            X2 : Float;
         when F3 =>
            X3 : String (1..2);
      end case;
   end record;
   type Foo_Ptr is access all Foo;
   type Foo_Ptr2 is access Foo;
   X : aliased Foo;
   P : Foo_Ptr := X'Access;
   PF2: Foo_PTR2 (F2);
begin
   X := (F2, 1.0);   -- OK
   PF2 := new Foo (F2);
   P := PF2.all'Access;
   P.all := (F1, 3); -- Error!
end Test;
```

Without this rule, PF2.all would now designate a value whose discriminant is F1!

> An access value is always constrained by its initial value; this is necessary because of constrained access subtypes.

But constrained access subtypes are not allowed for general access types like Foo_Ptr in the example.

> Here is a slightly modified version of your example:

> [...]

> Without this rule, PF2.all would now designate a value whose discriminant is F1!

This error is understandable and valid, because now P.all is PF2.all which is an allocated object and therefore constrained by its initial value with K = F2.

But why should the same apply when P designates X, which is unconstrained? Is it just an optimization (in the RM) so that a general access value does not have to carry around a flag showing whether its designated object is constrained or unconstrained?

Perhaps it would be better to make the assignment P := PF2.all'Access illegal, because it in effect converts a constrained access value (PF2) to an unconstrained access subtype (P), and so in some sense violates the prohibition of constrained subtypes of general access types.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Thu, 23 Mar 2023 20:53:02 +0100

> Perhaps it would be better to make the assignment P := PF2.all'Access illegal [...]

Yes this is a substitutability violation. Such cases never go without a punishment. In this case it is an implementation overhead.

Consider:

```
procedure Set (Destination : in out Foo;
Source : Foo) is
begin
  Destination := Source;
end Set;
```

The compiler cannot implement Set in a natural way, because Destination might be arbitrarily constrained by the caller. E.g. when the actual for Destination is P.all. So, the constraint must be passed together with the actual. Quite a burden.

From: J-P. Rosen <rosen@adalog.fr>
Date: Fri, 24 Mar 2023 10:41:20 +0100

> But why should the same apply when P designates X, which isunconstrained? [...]

I didn't dig in the RM in all details, but I think this comes from the fact that being constrained (always) is a property of the pointer (more precisely, its subtype), not of the pointed-at object.

From: Randy Brukardt <randy@rrsoftware.com>
Date: Sat, 25 Mar 2023 03:51:07 -0500

The rule is question is 4.1(9/3):

If the type of the name in a dereference is some access-to-object type T, then the dereference denotes a view of an object, the nominal subtype of the view being the designated subtype of T. If the designated subtype has unconstrained discriminants, the (actual) subtype of the view is constrained by the values of the discriminants of the designated object, except when there is a partial view of the type of the designated subtype that does not have discriminants, in which case the dereference is not constrained by its discriminant values.

We have to do that so as otherwise the access value would have to carry a designation as to whether the object was allocated or not.

This rule was inherited from Ada 83.

IMHO, this rule is stupid. It's even more stupid with the hole for types that have partial views without discriminants. The *proper* solution is to get rid of the rarely used and mostly useless access constraints, and then have no extra restrictions on access values. But that's considered too incompatible.

Ada in Jest

Ada Lovelace Cosplay

From: Mockturtle <framefritti@gmail.com>
Subject: Ada Lovelace cosplay
Date: Mon, 16 Jan 2023 09:48:58 -0800
Newsgroups: comp.lang.ada

Well, yes, someone cosplayed Ada...

https://blog.adafruit.com/2013/10/24/from-scratch-ada-lovelace-costume/

Conference Calendar

Dirk Craeynest

KU Leuven, Belgium. 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.

The COVID-19 pandemic had a catastrophic impact on conferences world-wide. In general the situation seems to improve further, and only a few events are still planned to be held "virtually" or in "hybrid" mode. Where available, the status of events is indicated with the following markers: "(v)" = event is held online, (h)" = event is held in a hybrid form (i.e. partially online).

2023

April 05	**Eelco Visser Commemorative Symposium,** Delft, the Netherlands. Topics include: language engineering, program transformation, language workbenches, declarative language specification, name binding and scope graphs, type soundness and intrinsically-typed interpreters, language specification testing, language implementation generation, domain-specific programming languages, DSLs for software deployment, DSLs for web application development, tool-supported programming education.	
April 16-20	16th IEEE **International Conference on Software Testing, Verification and Validation** (ICST'2023), Dublin, Ireland. Topics include: manual testing practices and techniques, security testing, model-based testing, test automation, static analysis and symbolic execution, formal verification and model checking, software reliability, testability and design, testing and development processes, testing in specific domains (such as embedded, concurrent, distributed, ..., and real-time systems), testing for cyber-physical systems, testing/debugging tools, empirical studies, experience reports, etc.	
April 17-20	28th **International Working Conference on Requirements Engineering: Foundation for Software Quality** (REFSQ'2023), Barcelona, Catalunya, Spain. Theme: "Human Values in RE".	
April 22-27	26th **European Joint Conferences on Theory and Practice of Software** (ETAPS'2023), Paris, France. 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 registration: April 7, 2023 (VerifyThis competition grant), April 14, 2023 (VerifyThis competition).	
	☺ April 22	14th **Workshop on Programming Language Approaches to Concurrency- and communication-cEntric Software** (PLACES'2023). Topics include: general area of programming language approaches to concurrency, communication and distribution, ranging from foundational issues, through language implementations, to applications and case studies; design and implementation of programming languages with first class concurrency and communication primitives; models for concurrent and distributed systems; concurrent data types, objects and actors; verification and program analysis methods for safe and secure concurrent and distributed software; etc.
	April 22	2nd **International Workshop on Trusted Automated Decision-Making** (TADM'2023). Topics include: approaches for assuring security and safety of software created by LLMs, approaches for synthesis of interpretable or explainable models from specifications, metrics to assess trustworthiness and safety in emergent AI based systems, etc.
	April 26-27	29th **International Symposium on Model Checking of Software** (SPIN'2023). Topics include: automated tool-based techniques to analyze and model software for the purpose of verification and validation.

April 24-28 (h) — 9th **International Conference on Advances and Trends in Software Engineering** (SOFTENG'2023), Venice, Italy. Topics include: software designing and production; software reuse; software sustainability; software testing and validation; maintenance and life-cycle management; software reliability, robustness, safety; software security; challenges for dedicated software, platforms, and tools; etc.

April 24-28 — 26th **Ibero-American Conference on Software Engineering** (CIbSE'2023), Montevideo, Uruguay. Topics include: formal methods applied to software engineering (SE), mining software repositories and software analytics, model-driven SE, software architecture, software dependability, software ecosystems and systems-of-systems, SE education and training, SE for emerging application domains (e.g., cyber-physical systems, IoT, ...), SE in the industry, software maintenance and evolution, software processes, software product lines, software quality and quality models, software reuse, software testing, technical debt management, etc.

May 08-10 — 23rd **Annual High Confidence Software and Systems Conference** (HCSS'2023), Annapolis, Maryland, USA. Topics include: development of scientific foundations for assured engineering of software-intensive complex computing systems and transition of science into practice; approaches that integrate previous work on verified infrastructure, programming language technology, and platform elements to help form a vision for end-to-end, formally-supported, model-based development.

May 09-12 — 16th **Cyber-Physical Systems and Internet of Things Week** (CPS-IoT Week'2023), San Antonio, Texas, USA. Event includes: 5 top conferences, HSCC, ICCPS, IoTDI, IPSN, and RTAS, multiple workshops, tutorials, and competitions.

> May 09 — **Workshop on Time-Centric Reactive Software** (TCRS'2023). Topics include: automotive systems, compiler construction, cyber-physical systems, distributed systems, embedded systems, formal verification, programming languages, model-based design, modeling languages, middleware, real-time systems, etc.
>
> ☺ May 09-12 — 29th IEEE **Real-Time and Embedded Technology and Applications Symposium** (RTAS'2023). Topics include: systems research related to embedded systems and time-sensitive systems; original systems, applications, case studies, methodologies, and algorithms that contribute to the state of practice in design, implementation, verification, and validation of embedded systems or time-sensitive systems. Deadline for early registration: April 7, 2023.

May 14-20 — 45th **International Conference on Software Engineering** (ICSE'2023), Melbourne, Victoria, Australia. Topics include: the full spectrum of Software Engineering. Deadline for submissions: April 14, 2023 (childcare support applications).

> May 14-15 — 11th **International Conference on Formal Methods in Software Engineering** (FormaliSE'2023). Topics include: approaches, methods and tools for verification and validation; formal approaches to safety and security related issues; scalability of formal method applications; integration of formal methods within the software development lifecycle; model-based engineering approaches; correctness-by-construction approaches for software and systems engineering; application of formal methods to specific domains, e.g., autonomous, cyber-physical, intelligent, and IoT systems; formal methods in a certification context; case studies developed/analyzed with formal approaches; experience reports on the application of formal methods to real-world problems; guidelines to use formal methods in practice; usability of formal methods; etc.
>
> May 14-15 — 6th **International Conference on Technical Debt** (TechDebt'2023).

May 16-18 — 15th **NASA Formal Methods Symposium** (NFM'2023), Houston, Texas, USA. Topics include: challenges and solutions for achieving assurance for critical systems, such as formal verification, including theorem proving, model checking, and static analysis, advances in automated theorem proving including SAT and SMT solving, use of formal methods in software and system testing, techniques and algorithms for scaling formal methods (abstraction and symbolic methods, compositional techniques, parallel and/or distributed techniques, ...), etc.

May 23-25 — 15th **Software Quality Days** (SWQD'2023), Munich, Germany. Topics include: all topics about software and systems quality, such as improvement of software development methods and processes, testing and quality assurance of software and software-intensive systems, project and risk management, domain specific quality issues such as embedded, medical, automotive systems, novel trends in software quality, etc.

☺ May 23-25 — 26th IEEE **International Symposium On Real-Time Distributed Computing** (ISORC'2023), Nashville, Tennessee, USA. Topics include: all aspects of object real-time distributed computing (ORC) technology, such as Internet of Things (IoT), real-time scheduling theory, resilient cyber-physical systems, autonomous systems (e.g., autonomous driving), optimization of time-sensitive applications, real-time applications (for example, medical devices, intelligent transportation systems, industrial automation systems and industry 4.0, ...), etc.

May 23-25 — 21st IEEE/ACIS **International Conference on Software Engineering Research, Management and Applications** (SERA'2023), Orlando, Florida, USA. Topics include: software testing and analysis; empirical software engineering; software requirements, modeling and design; software security and privacy; parallel and distributed computing; software evolution and understanding; software engineering education; embedded, automotive, cyber-physical systems; Internet of Things (IoT); etc.

May 29 — ICRA2023 - **Workshop on Robot Software Architectures** (RSA'2023), London, UK. Topics include: tools and approaches for automatic validation and verification of robot software architectures, language- and model-based approaches for designing robot software architectures, etc.

☺ June 07-08 — 31st **International Conference on Real-Time Networks and Systems** (RTNS'2023), Dortmund, Germany. Topics include: real-time applications design and evaluation (automotive, avionics, space, railways, telecommunications, process control, ...), real-time aspects of emerging smart systems (cyber-physical systems and emerging applications, ...), real-time system design and analysis (real-time tasks modeling, task/message scheduling, mixed-criticality systems, Worst-Case Execution Time (WCET) analysis, security, ...), software technologies for real-time systems (model-driven engineering, programming languages, compilers, WCET-aware compilation and parallelization strategies, middleware, Real-time Operating Systems (RTOS), ...), formal specification and verification, real-time distributed systems, etc. Deadline for early registration: May 2, 2023.

♦ June 13-16 — 27th **Ada-Europe International Conference on Reliable Software Technologies** (AEiC 2023), Lisbon, Portugal. Sponsored by Ada-Europe. In cooperation with ACM SIGAda, SIGBED & SIGPLAN (pending), and the Ada Resource Association (ARA).

> ☺ June 16 — 8th **Workshop on Challenges and New Approaches for Dependable and Cyber-Physical System Engineering** (De-CPS'2023). Topics include: artificial intelligence for CPS; model-based system engineering for CPS; transport and mobility, vehicle of the future; Industry 4.0 / 5.0; IoT, edge and cloud continuum; digital twins; safety and (cyber)security; human/machine interaction; real-time computing; time-sensitive networking (TSN), 5G/6G networks. Deadline for submissions: April 30, 2023 (papers).
>
> June 16 — 2nd **ADEPT workshop, AADL by its practitioners** (ADEPT'2023). Topics include: current projects in the field of design, implementation and verification of critical systems where AADL is a first citizen technology. Deadline for submissions: April 30, 2023 (abstracts).

☺ June 18 (v) — 24th ACM SIGPLAN/SIGBED **International Conference on Languages, Compilers, Tools and Theory of Embedded Systems** (LCTES'2023), Orlando, Florida. Co-located with PLDI'2023. Topics include: programming language challenges (features to exploit multicore architectures; features for distributed and real-time control embedded systems; capabilities for specification, composition, and construction of embedded systems; language features and techniques to enhance reliability, verifiability, and security; ...), compiler challenges (support for enhanced programmer productivity; support for enhanced debugging, profiling, and exception/interrupt handling; optimization for low power/energy, code/data size, and real-time performance; ...), tools for analysis, specification, design, and implementation (hardware, system software, application software, and their interfaces; distributed real-time control, media players, and reconfigurable architectures; system integration and testing; run-time system support for embedded systems; support for system security and system-level reliability; ...), theory and foundations of embedded systems (validation and verification, in particular of concurrent and distributed systems; formal foundations of model-based design as the basis for code

	generation, analysis, and verification; ...), novel embedded architectures (architecture support for new language features, virtualization, compiler techniques, debugging tools; ...), etc. Deadline for submissions: May 1, 2023 (artifacts).
June 27	DSN2023 - 1st **International Workshop on Verification & Validation of Dependable Cyber-Physical Systems** (VERDI'2023), Porto, Portugal. Topics include: all aspects related to the dependability evaluation of safety-critical CPS using techniques such as fault/attack-injection, runtime verification, formal verification, semi-formal analysis, simulation, and testing.
July 02-06	23rd **International Conference on embedded computer Systems: Architectures, MOdeling and Simulation** (SAMOS'2023), Samos Island, Greece. Topics include: advances in systems efficiency in various domains; novel architectures and computing methodologies and solutions for accelerating applications in various embedded domains, such as next generation automotive and avionics, next generation (machine) learning systems for surveillance and recognition, ...; software tools, compilation techniques and optimizations, and code generation for reconfigurable architectures; embedded parallel systems and MultiProcessor Systems-on-Chip; application-level resource management of multi-core architectures; all design processes for embedded systems ranging from design languages, modeling and simulation, performance, reliability, ...; specification languages and models; system-level design, simulation, and verification; MP-SoC programming, compilers, simulation and mapping technologies; profiling, measurement and analysis techniques; (design for) system adaptivity; testing and debugging; etc.
July 11-14	35th **Euromicro Conference on Real-Time Systems** (ECRTS'2023), Vienna, Austria. Deadline for submissions: May 18, 2023 (Industrial Challenge full solutions), June 26, 2023 (Industrial Challenge early stage proposals).
July 17-21	**Software Technologies: Applications and Foundations** (STAF'2023), Leicester, UK. Topics include: practical and foundational advances in software technology. Deadline for submissions: May 21, 2023 (workshop papers).
	July 18-19 — 17th **International Conference on Tests And Proofs** (TAP'2023). Topics include: many aspects of verification technology, including foundational work, tool development, and empirical research; the connection between proofs (and other static techniques) and testing (and other dynamic techniques); verification and analysis techniques combining proofs and tests; program proving with the aid of testing techniques; formal techniques supporting the automated generation of test vectors and oracles, and supporting novel definitions of coverage criteria; specification inference by deductive and dynamic methods; testing and runtime analysis of formal specifications; verification of verification tools and environments; applications of test and proof techniques in new domains; combined approaches of test and proof in the context of formal certifications; case studies, tool and framework descriptions, and experience reports about combining tests and proofs; etc.
☺ July 17-21	37th **European Conference on Object-Oriented Programming** (ECOOP'2023), Seattle, USA. Topics include: all practical and theoretical investigations of programming languages, systems and environments; innovative solutions to real problems as well as evaluations of existing solutions. Deadline for submissions: April 15, 2023 (Student Research Competition).
July 18-21	19th **European Conference on Modelling Foundations and Applications** (ECMFA'2023), Leicester, UK. Co-located with STAF'2023. Topics include: all aspects of model-based engineering (MBE); foundations of MBE, including model transformations, domain-specific languages, verification and validation approaches, ...; application of MBE methods, tools, and techniques to specific domains, e.g., automotive, aerospace, cyber-physical systems, robotics, Artificial Intelligence or IoT; educational aspects of MBE; tools and initiatives for the successful adoption of MBE in industry; etc.
☺ Aug 28 – Sep 01	29th **International European Conference on Parallel and Distributed Computing** (Euro-Par'2023), Limassol, Cyprus. Topics include: all aspects of parallel and distributed processing, ranging from theory to practice, from small to the largest parallel and distributed systems and infrastructures, from fundamental computational problems to applications, from architecture, compiler, language and interface design and implementation, to tools, support infrastructures, and application performance aspects. Deadline for submissions: May 20, 2023 (posters, demos, PhD symposium).

September 06-08 49th **Euromicro Conference on Software Engineering and Advanced Applications** (SEAA'2023), Durres, Albania. Topics include: information technology for software-intensive systems; tracks on Cyber-Physical Systems (CPS), Emerging Computing Technologies (ECT), Model-Driven Engineering and Modeling Languages (MDEML), Software Engineering and Debt Metaphors (SEaDeM), Software Process and Product Improvement (SPPI), etc. Deadline for submissions: April 3, 2023 (papers).

September 11-13 16th **International Conference on the Quality of Information and Communications Technology** (QUATIC'2023), Aveiro, Portugal. Topics include: all quality aspects in ICT systems engineering and management. Deadline for registration: April 17, 2023 (full papers), May 29, 2023 (short papers, Journal First).

September 17-22 **Embedded Systems Week** 2023 (ESWEEK'2023), Hamburg, Germany. Includes CASES'2023 (International Conference on Compilers, Architectures, and Synthesis for Embedded Systems), CODES+ISSS'2023 (International Conference on Hardware/Software Codesign and System Synthesis), EMSOFT'2023 (International Conference on Embedded Software). Deadline for submissions: May 22, 2023 (Work-in-Progress track papers, Late-Breaking and Work-in-Progress papers).

☺ Sep 17-22 ACM SIGBED **International Conference on Embedded Software** (EMSOFT'2023). 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 integrate computation, networking, and physical dynamics; embedded distributed, networked systems (time-critical embedded systems, scheduling, resource allocation, and execution time analysis; ...); embedded software design and analysis (safety/mixed-critical embedded software, software design for cyber-physical systems, ...); resilience (embedded software security, robust implementation of control systems); process, methods (formal modeling and verification; testing, validation, and certification; model- and component-based approaches); empirical studies and their reproduction; application areas including automotive, avionics, energy, health care, mobile devices, multimedia, machine learning, and autonomous systems; etc. Deadline for submissions: May 22, 2023 (Work-in-Progress submissions).

Sep 17-22 **International Conference on Compilers, Architecture, and Synthesis for Embedded Systems** (CASES'2023). Topics include: latest advances in design, optimization, validation, and applications of embedded systems, Internet of Things (IoT), and the emergent trend of integrating Artificial Intelligence into IoT (AIoT); architecture, design, and compiler techniques for reliability, and aging; modeling, analysis, and optimization for timing and predictability; validation, verification, testing, and debugging of embedded software; etc. Deadline for submissions: May 22, 2023 (Work-in-Progress papers).

Sep 17-22 **International Conference on Hardware/Software Codesign and System Synthesis** (CODES+ISSS'2023). Topics include: system-level design, hardware/software co-design, modeling, analysis, and implementation of modern Embedded Systems, Cyber-Physical Systems, and Internet-of-Things, from system-level specification and optimization to synthesis of system-on-chip hardware/software implementations. Deadline for submissions: May 22, 2023 (Work-in-Progress papers).

September 18-23 34th **International Conference on Concurrency Theory** (CONCUR'2023), Antwerp, Belgium. Co-located with FORMATS, FMICS and QEST as part of CONFEST 2023 Topics include: semantics, logics, verification and analysis of concurrent systems; basic models of concurrency; verification and analysis techniques for concurrent systems such as abstract interpretation, model checking, race detection, run-time verification, static analysis, testing, 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. Deadline for submissions: April 24, 2023 (abstracts), May 2, 2023 (papers).

September 19-21	21st **International Conference on Formal Modeling and Analysis of Timed Systems** (FORMATS'2023), Antwerp, Belgium. Co-located with CONCUR, FMICS and QEST as part of CONFEST 2023 Topics include: fundamental and practical aspects of timed systems; modelling, design and analysis of timed computational systems; theoretical foundations of timed systems, languages and models; 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, scheduling in manufacturing and telecommunication, robotics, ...); etc. Deadline for submissions: April 21, 2023 (abstracts), April 28, 2023 (papers).
September 19-22	**42nd International Conference on Computer Safety, Reliability and Security** (SafeComp'2023), Toulouse, France. Topics include: development, assessment, operation and maintenance of safety-related and safety-critical computer systems; safety/security risk assessment; model-based analysis, design, and assessment; formal methods for verification, validation, and fault tolerance; validation and verification methodologies and tools; methods for qualification, assurance and certification; compositional verification and certification; cyber-physical threats and vulnerability analysis; safety guidelines, standards and certification; safety and security interactions and tradeoffs; etc. Domains of application include: railways, automotive, space, avionics, nuclear and process industries; autonomous systems, advanced robotics; telecommunication and networks; critical infrastructures; medical devices and healthcare; defense, emergency & rescue; logistics, industrial automation, off-shore technology; etc.
☺ September 20-22	28th **International Conference on Formal Methods for Industrial Critical Systems** (FMICS'2023), Antwerp, Belgium. Co-located with CONCUR, FORMATS and QEST as part of CONFEST 2023 Topics include: case studies and experience reports on industrial applications of formal methods, focusing on lessons learned or identification of new research directions; methods, techniques and tools to support automated analysis, certification, debugging, descriptions, learning, optimisation and transformation of complex, distributed, real-time, embedded, mobile and autonomous systems; verification and validation methods that address shortcomings of existing methods with respect to their industrial applicability (e.g., scalability and usability issues, tool qualification, and certification); impact of adoption of formal methods on development process and associated costs; application of formal methods in standardisation and industrial forums. Deadline for submissions: May 15, 2023 (papers).
September 20-22	22nd **International Conference on Intelligent Software Methodologies, Tools and Techniques** (SOMET'2023), Naples, Italy. Topics include: new directions in software development methodologies and related tools and techniques; software methodologies and tools for robust, reliable, non-fragile software design; software development techniques for legacy systems; software evolution techniques; agile software and lean methods; software optimization and formal methods for software design; software maintenance; software security tools and techniques; formal techniques for software representation, software testing and validation; object-oriented, aspect-oriented, component-based and generic programming, multi-agent technology; model driven development (DVD), code centric to model centric software engineering; etc. Deadline for submissions: April 1, 2023 (full papers).
October 03-06	23rd **International Conference on Runtime Verification** (RV'2023), Thessaloniki, Greece. Topics include: monitoring and analysis of runtime behaviour of software and hardware systems; program instrumentation; logging, recording, and replay; combination of static and dynamic analysis; monitoring techniques for concurrent and distributed systems; fault localization, containment, resilience, recovery and repair; etc. Deadline for submissions: May 15, 2023 (papers).
☺ October 10-12	**International Conference on Reliability, Safety and Security of Railway Systems** (RSSRail'2023), Berlin, Germany. Topics include: safety in development processes and safety management; combined approaches to safety and security; system and software safety analysis; formal modelling and verification techniques; system reliability; validation according to the standards; tool and model integration, tool chain; domain-specific languages and modelling frameworks; model reuse for reliability, safety and security; etc. Deadline for submissions: April 28, 2023 (abstracts, tutorials), May 5, 2023 (full papers), July 14, 2023 (posters).

☺ October 17 — **High Integrity Software Conference** (HISC'2023), Bristol, UK. Topics include: advanced software development for high-integrity and high-assurance systems, including programming languages, AI-assisted software development, verifiable code generation; verification of novel, high-integrity and high-assurance systems; assurance of high-integrity, high-assurance systems; infrastructure & ecosystem for high-integrity software. Deadline for submissions: May 31, 2023.

October 18-20 (h) — **16th International Conference on Verification and Evaluation of Computer and Communication Systems** (VECoS'2023), Marrakech, Morocco. Topics include: analysis of computer and communication systems, where functional and extra-functional properties are inter-related; cross-fertilization between various formal verification and evaluation approaches, methods and techniques, especially those developed for concurrent and distributed hardware/software systems. Deadline for submissions: May 15, 2023.

October 19-20 (v) — 19th **International Conference on Formal Aspects of Component Software** (FACS'2023), Internet. Topics include: applications of formal methods in all aspects of software components and services; formal methods, models, and languages for software-intensive systems, components and services: formal aspects of concrete software-intensive systems, including real-time/safety-critical systems, hybrid and cyber physical systems, components that use artificial intelligence, ...; tools supporting formal methods for components and services; case studies and experience reports over the above topics; special track on formal methods at large; etc. Deadline for submissions: July 3, 2023 (abstracts), July 10, 2023 (papers).

☺ October 21-25 — 32nd **International Conference on Parallel Architectures and Compilation Techniques** (PACT'2023), Vienna, Austria. Topics include: parallel architectures; compilers and tools for parallel computer systems; applications and experimental systems studies of parallel processing; computational models for concurrent execution; support for correctness in hardware and software; reconfigurable parallel computing; parallel programming languages, algorithms, and applications; middleware and run time system support for parallel computing; etc. Deadline for submissions: April 1, 2023 (papers), August 4, 2023 (artifacts).

October 22-24 — 30th **Static Analysis Symposium** (SAS'2023), Cascais (Lisbon), Portugal. Co-located with SPLASH'2023 Topics include: static analysis as fundamental tool for program verification, bug detection, compiler optimization, program understanding, and software maintenance. Deadline for submissions: April 24, 2023 (full papers), April 29, 2023 (artifacts).

☺ October 22-27 — ACM **Conference on Systems, Programming, Languages, and Applications: Software for Humanity** (SPLASH'2023), Lisbon, Portugal. Topics include: all aspects of software construction and delivery, at the intersection of programming, languages, and software engineering. Deadline for submissions: April 7, 2023 (SLE), April 14, 2023 (OOPSLA), April 24, 2023 (SAS), April 28, 2023 (Onward! essays, Onward! papers), May 15, 2023 (PPDP abstract), May 19, 2023 (LOPSTR abstract), May 22, 2023 (PPDP paper), May 26, 2023 (LOPSTR paper), June 19, 2023 (Doctoral Symposium), June 26, 2023 (MPLR), June 28, 2023 (DLS), July 7, 2023 (GPCE), July 12, 2023 (workshop papers).

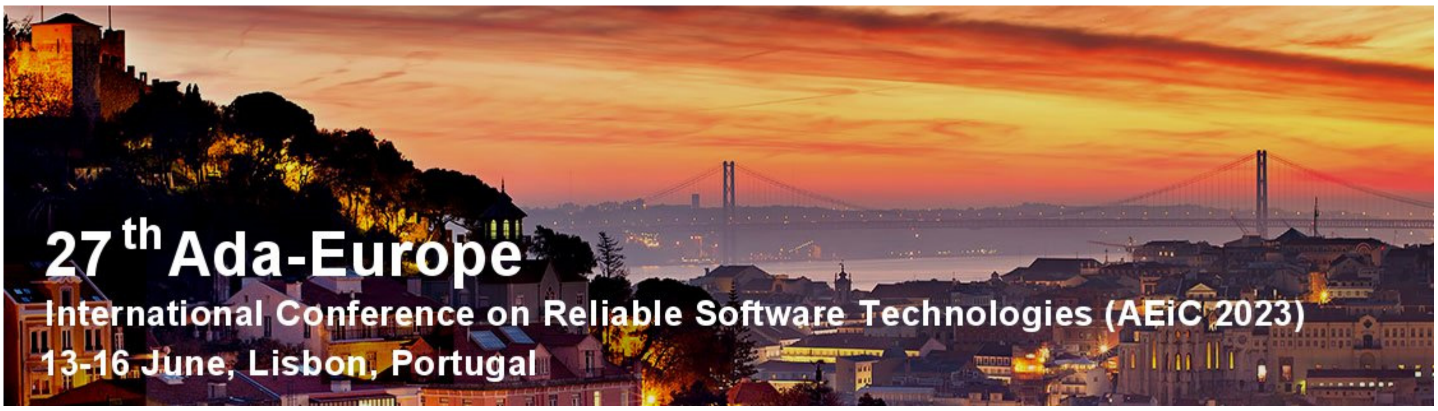
> Oct 22-27 — 16th ACM SIGPLAN **International Conference on Software Language Engineering** (SLE'2023). Topics include: software language engineering rather than engineering a specific software language; software language design and implementation; software language validation (verification and formal methods for languages, testing techniques for languages, simulation techniques for languages); software language integration and composition; software language maintenance (software language reuse, language evolution, language families and variability, language and software product lines); 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: April 7, 2023 (1st round papers), June 26, 2023 (2nd round abstracts), June 30, 2023 (2nd round papers), August 30, 2023 (artifacts).

> ☺ Oct 23-27 — **Conference on Object-Oriented Programming, Systems, Languages, and Applications** (OOPSLA'2023). Topics include: all practical and theoretical investigations of programming languages, systems and environments, targeting any stage of software development, including requirements, modeling, prototyping, design, implementation, generation, analysis, verification, testing, evaluation,

	maintenance, and reuse of software systems; development of new tools, techniques, principles, and evaluations. Deadline for submissions: April 14, 2023 (round 2).
October 24-27 (h)	28th IEEE **Pacific Rim International Symposium on Dependable Computing** (PRDC'2023), Singapore. Topics include: software and hardware reliability, resilience, safety, security, testing, verification, and validation; dependability measurement, modeling, evaluation, and tools; architecture and system design for dependability; reliability analysis of complex systems; dependability issues in computing systems (e.g. high performance computing, real-time systems, cyber-physical systems, ...); emerging technologies (autonomous systems including autonomous vehicles, human machine teaming, smart devices/internet of things); etc. Deadline for submissions: April 27, 2023 (abstracts), May 4, 2023 (papers).
October 24-27	21st **International Symposium on Automated Technology for Verification and Analysis** (ATVA'2023), Singapore. Topics include: theoretical and practical aspects of automated analysis, synthesis, and verification of hardware and software systems; program analysis and software verification; analytical techniques for safety, security, and dependability; testing and runtime analysis based on verification technology; analysis and verification of parallel and concurrent systems; analysis and verification of deep learning systems; verification in industrial practice; applications and case studies; etc. Deadline for submissions: April 27, 2023 (abstracts), May 4, 2023 (papers).
November 08-10	21st **International Conference on Software Engineering and Formal Methods** (SEFM'2023), Eindhoven, the Netherlands. Topics include: software development methods (formal modelling, specification, and design; software evolution, maintenance, re-engineering, and reuse), design principles (programming languages; abstraction and refinement; ...), software testing, validation, and verification, security and safety (security, privacy, and trust; safety-critical, fault-tolerant, and secure systems; software certification), applications and technology transfer (real-time, hybrid, and cyber-physical systems; intelligent systems and machine learning; education; ...), case studies, best practices, and experience reports. Deadline for submissions: June 2, 2023 (abstracts), June 9, 2023 (papers).
November 13-15	18th **International Conference on integrated Formal Methods** (iFM'2023), Leiden, the Netherlands. Topics include: recent research advances in the development of integrated approaches to formal modelling and analysis; all aspects of the design of integrated techniques, including language design, verification and validation, automated tool support and the use of such techniques in software engineering practice. Deadline for submissions: May 25, 2023 (abstracts), June 1, 2023 (papers).
December 10	Birthday of Lady Ada Lovelace, born in 1815. Happy Programmers' Day!

2024

January 17-19	19th **International Conference on High Performance and Embedded Architecture and Compilation** (HiPEAC'2024), Munich, Germany. Topics include: computer architecture, programming models, compilers and operating systems for general-purpose, embedded and cyber-physical systems. Areas include safety-critical dependencies, cybersecurity, energy efficiency and machine learning. Deadline for submissions: June 1, 2023.
March 02-06	IEEE/ACM **International Symposium on Code Generation and Optimization** (CGO'2024), Edinburgh, UK. Deadlines for paper submissions: May 19, 2023 (1st round), September 1, 2023 (2nd round).


27th Ada-Europe
International Conference on Reliable Software Technologies (AEiC 2023)
13-16 June, Lisbon, Portugal

Advance Information

AEiC 2023
Lisboa

The 27th Ada-Europe International Conference on Reliable Software Technologies (AEiC 2023) returns to Lisbon from the 13th to the 16th of June, five years after the 2018 edition. After the hybrid-mode edition in Ghent, Belgium, last year, AEiC 2023 returns to in-presence only modality.

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 conference is an established international forum for providers, practitioners and researchers in reliable software technologies. The conference presentations will illustrate current work in the theory and practice of developing, running and maintaining challenging long-lived, high-quality software systems for a variety of application domains including manufacturing, robotics, avionics, space, transportation.

The program features a keynote, a panel discussion, technical presentations 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. The conference program includes two core days with special sessions featuring presentations of invited experts, peer-reviewed academic papers, industrial presentations, and work-in-progress talks and posters.

Overview of the Conference Program

	Morning	Before Lunch	After Lunch	Afternoon	Evening
Tuesday, June 13th *Tutorials*	**Tutorial 1:** *The HAC Ada Compiler*		**Tutorial 2:** *Controlling I/O Devices with Ada and the Linux Simple I/O Library*		**Welcome Reception**
	Tutorial 3: *Everything you always wanted to know about characters and strings*		**Tutorial 4:** *Introduction to the development of safety critical software*		
	Tutorial 5: *Rust Fundamentals*		**Tutorial 6:** *Concurrency and Parallelism in Rust*		
Wednesday, June 14th *Technical Presentations*	**Keynote Talk**	**Session 1:** *Verification and Validation 1*	**Session 2:** *Advanced Systems*	**Session 3:** *Reliability and Performance*	**Conference Banquet**
	WiP posters shown during breaks		**WiP posters shown during breaks**		
Thursday, June 15th *Technical Presentations*	**Panel**	**Session 4:** *Verification and Validation 2*	**Session 5:** *Reliable Programming*	**Session 6:** *Real-Time Systems*	
	WiP posters shown during breaks		**WiP posters shown during breaks**		
Friday, June 16th *Satellite Events*	**Workshop 1:** *DeCPS 2023 (Challenges and New Approaches for Dependable and Cyber-Physical System Engineering)*				
	Workshop 2: 2nd *ADEPT (AADL by its practitioners)*				

Invited Speakers

- Wednesday, June 14th, Keynote Talk on *"Applications of Liquid Types for More Reliable Software",* by **Alcides Fonseca**, LASIGE, University of Lisbon, Portugal
- Thursday, June 15th, Panel on "Promises and Challenges of AI-enabled Software Development Tools for Safety-Critical Applications", with **Douglas Schmidt**, Vanderbilt University, USA, **Jochen Quante**, Robert Bosch GmbH, Germany, **Jon Pérez Cerrolaza**, Ikerlan, Spain, and **Björn Andersson**, SEI - Carnegie Mellon University, USA.

Tutorials

The following 6 half-day tutorials will take place on Tuesday, June 13th:

- The HAC Compiler, **Gautier de Montmollin**, Ada Switzerland
- Controlling I/O Devices with Ada and the Linux Simple I/O Library, **Philip Munts**, Sweden
- Everything you Always Wanted to Know About Characters and Strings, **Jean-Pierre Rosen**, Adalog, France
- Introduction to the Development of Safety Critical Software, **Jean-Pierre Rosen**, Adalog, France
- Rust Fundamentals, **Luis Miguel Pinho** and **Tiago Carvalho**, ISEP, Portugal
- Concurrency and Parallelism in Rust, **Luis Miguel Pinho** and **Tiago Carvalho**, ISEP, Portugal

Co-Located Workshops

On Friday, June 16th there will be 2 workshops: the 8th DeCPS workshop on "Challenges and new Approaches for Dependable and Cyber-Physical Systems Engineering" and the "2nd ADEPT: AADL by its practitioners" workshop.

Social Events

The program includes 1-hour long coffee breaks, providing the opportunity for participants to discuss their work, to view the WiP posters, and to socialise. Lunches will be served at the hotel restaurant, from Tuesday to Friday, providing further interaction opportunities. Furthermore, there will be a Welcome Reception event and a Conference Banquet.

The Welcome Reception event will be in the gardens of the National Museum of Science & Natural History. A selection of drinks and appetizers will be served and participants will have the opportunity to taste port wine while walking in the gardens.

The Conference Banquet will be at the "Casa do Alentejo" restaurant, downtown Lisbon. The dinner will include cod fish baked in olive oil, which is a very typical Portuguese dish. The restaurant is located in a XVII century building in downtown Lisbon, which served as a casino about a century ago and features several attractively decorated rooms.

Conference Venue

The conference takes place at Fénix Lisboa, a four-star hotel in the centre of Lisbon, just across the "Marquês de Pombal" metro station. The hotel meeting area will host all the meetings, tutorials, workshops and conference sessions. The hotel restaurant, where participants will have their buffet lunch, is in the same area. Coffee breaks will be in the atrium outside the meeting rooms, just in front of the room where WiP posters will be displayed during the two core days.

As Lisbon is now a very touristic venue, there are plenty of restaurants near the hotel and downtown. The city center also features many historical places, landmarks and museums that you may want to visit, like the Lisbon medieval Castle (Castelo de São Jorge), the Santa Justa elevator, the National Museum of Contemporary Art at Chiado, or the Roman Theatre near the castle. Simply walking the streets is most enjoyable, going to the old neighbourhoods of Alfama and Bairro Alto, or along the river front.

AEiC 2023
Lisboa

Achieving 100% Availability in the ERAM Air Traffic Control System

Howard Ausden

ERAM Deputy Software Architect Leidos Corporation; email: Howard.Ausden@leidos.com

Abstract

Fault tolerance is a key requirement for En Route Automation Modernization (ERAM), the FAA's system that manages En Route air traffic over the USA. A system failure could lead to hundreds of flights being delayed or cancelled. Using experience from earlier systems a set of techniques were built into ERAM at inception, including a hot standby copy of each executable and the latest state checkpointed in disk files. As the system matured through formal testing and operational experience at the first sites (2010 – 2015), the goal of 100% availability was not achieved so additional techniques were added. These included exception safety, runaway process protection, and proactive monitoring of the system to detect defects and often resolve them without the air traffic controllers being aware. With the implementation of these additional techniques the FAA has measured ERAM as 100% available from October 2016 at all 20 operational sites. Software fault tolerance techniques have been well documented [2]; this extended abstract describes the specific techniques that have led to ERAM achieving continuous 24x7 availability for 6 years.

1 Introduction

The ERAM software as of July 2022 is more than 2 million lines, of which the operational software is mostly Ada 95 and C++. The software is constantly changing and growing. As code units are developed, they are tested with 100% decision coverage, loops are tested with minimum and maximum iterations, and special cases are tested. Code units are then combined in software testing of entire paths through an executable. The code is then delivered to the formal test organization, where it is tested in a replica ERAM system against requirements and use cases and for regressions. Further levels of testing are performed by the FAA with the system release, and each site is equipped with a replica ERAM system for its unique testing and training. Each level of testing is reviewed periodically to ensure an appropriate number of defects is detected.

The software includes a great deal of code to handle invalid data (explicit checks for things we know to look for), but the problem space of 4-dimensionsal air traffic control is very complex. For example, the international aircraft route standard [6] has complex semantics to accommodate any kind of flight, from helicopters to military flights, and routes are input from a variety of sources that are sometimes manual and sometimes more automated. Not every condition and/or test case can be foreseen, much less executed during system test. Fault tolerance kicks in for those rare things that weren't explicitly planned for.

2 Initial Fault Tolerance Design

At ERAM inception the fault tolerance design comprised techniques that had been used successfully on earlier programs. The intent was to provide enough fault tolerance to meet the availability requirements, and then later to add additional fault tolerance if needed.

1. Services that have persistent state data are implemented as 'Operational Units (OUs)'. An OU comprises a primary executable that services requests from its clients, and a secondary copy of the executable that is kept up to date by the primary and can quickly take over if the primary fails [2] [3] [5]. The secondary protects against software failures in the primary and against hardware failures since the secondary is loaded on a different machine.
2. State data is checkpointed to disk each time it changes. If both the primary and secondary executables fail, the data can be read from the disk. This is a slower recovery but protects against scenarios such as power failures.
3. A second copy of the ERAM system runs in backup mode and is kept up to date by the active system. Controllers have an A/B switch to let them use either system. Promotion of the backup system to active requires an operator command. The active channel runs continuously, sometimes for a month or more before the backup, which may be running a new release, is promoted to active, providing uninterrupted availability.
4. Executables report their 'heartbeat' frequently and are killed if there is no heartbeat for a short interval. This protects against, for example, endless loops.
5. Servers are designed with a specific order of processing:
    a. Validate inputs
    b. If input is valid, apply changes to databases
    c. At this point the change has been successfully digested, so checkpoint the changes to disk, expecting they can be reloaded successfully.
    d. Propagate the changes to clients, including the secondary copy of the executable and the backup copy of the system. Since the change was successful in the server's database, it can

be expected to succeed in the replicated databases in clients, secondary and backup.

6. Different kinds of issues exist in C++ and Ada. For example, because C++ doesn't have Ada's protection for walking off the end of the array we use the compiler options -qcheck (AIX) and -fsanitize (Linux) during integration builds to be more stringent and fail faster.
7. All problems are logged with sufficient contextual data to allow for later debugging.

3 Testing and Early Operations

ERAM initial operations began in 2010 at 2 sites. Availability problems encountered were:

1. Unhandled exceptions caused occasional executable failures. The secondary would take over but by design if the problem continues to recur over and over the executables are not reloaded, and manual action is needed. This causes a loss of the service for all clients, but the exception is almost always associated with unique circumstances for one object (e.g., a flight). Prior systems that used the primary/secondary executable technique had smaller real-world interfaces than ERAM (they were fed from other systems), so they didn't encounter the same volume of unexpected exceptions.
2. Runaway processes in some of the many systems ERAM interfaces with would flood ERAM with messages, leading to denial of service.
3. There were one or two occurrences of algorithmic code failing to exit from a loop.

4 Techniques Added to Improve Availability

The required availability is 99.999% but in 2013, after experience at the initial sites, the FAA asked us to make some key parts of the system (such as the flight server) bulletproof. The techniques listed below were used, in places that justified the need. Either standard industry design patterns were followed, or patterns were developed that can be reused elsewhere in ERAM and other systems.

4.1 Strong Exception Safety [1]

Since the unhandled exceptions were almost always due to some unique circumstances involving one or a few flights, it was better to confine the problem than to let the executable fail and risk the problem repeating in the secondary copy of the executable, perhaps leading to loss of service. Hence strong exception safety, also known as ‘commit or rollback’ semantics, was implemented: Operations can fail, but failed operations are guaranteed to have no side effects, leaving the original values intact. This allows the system to continue processing all the other flights that are not impacted by the unique circumstances.

The order of operations is essential:

1. Validate inputs: Exceptions often occur here, where the real world meets the system.
2. If the input is valid, apply changes to databases: Exceptions sometimes occur here, and database entries can be rolled back to their initial state and a rejection returned to the client.
3. When all databases have been updated we reach the commit point, when roll back is no longer possible. At a high-level view the server has a single path after this point (doing what servers do – checkpoint the changes to disk, propagate them to clients, and respond to the input indicating success) and exceptions are vanishingly rare.

Since the addition of exception safety, the mean time between failure of executables has continuously increased. The other changes were targeted at more specific scenarios, while exception safety protects against most software faults; other kinds of software faults, such as illegal memory accesses, are rare.

Exception safety should be considered for any non-trivial program. After all, if a program is not exception-safe, it implies either that it will sometimes fail in an unmanaged way, or that it has been proved not to raise exceptions. If the latter isn’t true, buyer beware!

4.2 Repetitive Application Failure Protection

Repetitive Application Failure Protection handles the case where, despite exception safety, an executable fails repeatedly, triggered by a problem with the same object each time. If an executable fails twice processing the same object then that object is quarantined, and the system continues to provide service for all other objects. This has prevented failures at operational sites on several occasions and should be considered for future high availability systems.

Sometimes a restart alone clears a problem. A restart resets an executable’s state as it is brought up to date by other servers (e.g., if a server deleted an object but a client failed to delete the object, re-start of the client will clear out the stuck object). A restart also clears any memory fragmentation or leakage issues. If the problem was due to some timing issue, it will be unlikely to recur – perhaps the flight has moved on, or a conflicting process has finished.

4.3 Runaway Message Cycle Protection

Runaway processes in interfacing systems would drive ERAM applications at high loading – in short, spamming, leading to denial of the service to all clients. The solution was:

- Close to the interface, track the number of messages of a particular kind per second.
- If the message rate exceeds a threshold, reject messages back to the sender until the rate falls below the threshold.
- Additionally, where appropriate ERAM servers monitor their rate of publications and, if too rapid, space out the sends to protect clients from being overwhelmed. This protects against cases like a timed event repeatedly expiring.

4.4 Monitoring of system health

Since inception ERAM has engineered a comprehensive recording capability that continuously captures all necessary context for the system, including in error cases, without requiring human interaction by the system users. This capability allowed us to understand why 100% availability was not initially achieved and allowed us to design the appropriate enhancements. Additionally, automated monitoring of data recorded at sites allows the development team to receive notifications quickly when a problem occurs. The fix is often delivered without the site being aware a problem existed. Sometimes the site can be notified when they need to take quick action (e.g., deal with a problematic flight, or periodically restart an executable that is leaking memory).

Not all problems need to be fixed. Some exceptions and spam occurrences have been understood and judged not worth either the cost of fixing or the risk of a fix causing other unintended actions, such as when the fault is adequately contained and the impact to the controller workforce insignificant. We are still striving for ways to automatically tag the recording of an exception or spam occurrence as a known problem, to reduce the burden of manually analyzing daily reports from sites.

4.5 Other Protections

One repetitive modeling subsystem bounds loops at 500 iterations, and this has been effective in preventing failures.

Code that uses recursion generally bounds the depth of the recursion.

An attempt was made to provide a simplified fallback implementation of the flight server. It was not successful because no compromise on the requirements was acceptable, so no simplification was possible.

No attempt was made to use N-version programming on ERAM (our organization did use N-version programming for the space shuttle's in-flight computers [4]).

5 Conclusion

Since the FAA's Operational Readiness Date (Spring 2015) ERAM has used a variety of techniques, with defense in depth and scaling of the number of layers to the difficulty and impact of failure and recovery from the failure. New programs and new ERAM developments have taken advantage of the common frameworks that are now mature and can be built-in from the beginning. Ordered from preventive care to life support, the techniques include:

a. Proactive Monitoring

b. Strong Exception Safety

c. Spam Protection

d. Primary / Secondary Executables ("Operational Unit")

e. Checkpointed State Data on Disk

f. Repetitive Application Failure Protection

g. Active System / Backup System

There are remaining challenges ("it's all fun until someone divides by zero") but they occur infrequently. Future systems that need to be highly available should build in these techniques at the start.

References

[1] A good description of exception safety is in https://en.wikipedia.org/wiki/Exception_safety

[2] See the section on Process Pairs in https://ntrs.nasa.gov/api/citations/20000120144/downloads/20000120144.pdf

[3] Our implementation of process pairs followed the work of Dr. Flaviu Cristian. See the section on failure masking in server groups in http://csis.pace.edu/~marchese/CS865/Papers/cristian93understanding.pdf

[4] See for example http://www.ganssle.com/blog/blog/on-n-version-programming.html

[5] Fault-Tolerance in the Advanced Automation System (Cristian, Dancey, Dehn).

[6] See the route (item 15) description in https://flightcrewguide.com/wiki/rules-regulations/flight-plan/.

Hardware/Software Co-assurance for the Rust Programming Language Applied to Zero Trust Architecture Development

David Hardin
Collins Aerospace, Iowa, USA; email: david.hardin@collins.com

Abstract

Zero Trust Architecture requirements are of increasing importance in critical systems development. Zero trust tenets hold that no implicit trust be granted to assets based on their physical or network location. Zero Trust development focuses on authentication, authorization, and shrinking implicit trust zones to the most granular level possible, while maintaining availability and minimizing authentication latency. Performant, high-assurance cryptographic primitives are thus central to successfully realizing a Zero Trust Architecture. The Rust programming language has garnered significant interest and use as a modern, type-safe, memory-safe, and potentially formally analyzable programming language. Our interest in Rust particularly stems from its potential as a hardware/software co-assurance language for developing Zero Trust Architectures. We describe a novel environment enabling Rust to be used as a High-Level Synthesis (HLS) language, suitable for secure and performant Zero Trust application development. Many incumbent HLS languages are a subset of C, and inherit many of the well-known security shortcomings of that language. A Rust-based HLS brings a single modern, type-safe, memory-safe, high-assurance development language for both hardware and software. To study the benefits of this approach, we crafted a Rust HLS subset, and developed a frontend to the hardware/software co-assurance toolchain due to Russinoff and colleagues at Arm, used primarily for floating-point hardware formal verification. This allows us to leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. In this paper, we describe our Rust subset, detail our prototype toolchain, and describe the implementation, performance analysis, formal verification and validation of representative Zero Trust algorithms and data structures written in Rust, emphasizing cryptographic primitives and common data structures.

Keywords: hardware/software co-assurance, Rust, theorem proving, ACL2.

1 Introduction

Zero Trust Architecture [1] requirements are increasingly becoming adopted in critical systems development. Zero trust tenets state that there is no implicit trust granted to assets based on their physical or network location. All communications should be conducted “in the most secure manner available, protect confidentiality and integrity, and provide source authentication” [1]. Zero trust architectures shrink implicit trust zones to the most granular level possible, while maintaining availability and minimizing authentication delays. Performant, high-assurance cryptographic technologies are thus central to successfully realizing a Zero Trust Architecture, as are “leak-free” data structures, and other high-assurance components.

We have developed several zero trust primitives as part of the DARPA CASE program [2]. As the Zero Trust Architecture specification [1] was not created until the CASE program was well underway, this was not an explicit goal of the program, but similar cyber-assurance concerns informed both efforts, resulting in convergent technologies. A major guiding principle of CASE is the need for verified and validated automated synthesis of security-enhancing components from high-level architectural specifications, including input filters [3], safety monitors [4], remote attestation and measurement [5], as well as trustworthy interprocess communications [6]. Our research and development effort on CASE has emphasized the value of modern, type-safe, memory-safe, and formally analyzable languages for use in automated synthesis [3, 5, 6], and has identified the value of automated high-assurance synthesis to hardware, software, or a combination of the two, from architectural level specifications [7].

In this paper, we describe the development, formal verification, and validation of a number of zero trust architecture primitives in a High-Level Synthesis (HLS) subset of Rust, suitable for software and/or hardware implementation. Along the way, we introduce Rust, outline our HLS subset, describe a prototype hardware/software co-assurance toolchain for this subset, present case studies of zero trust primitives, and detail verification and validation efforts. It is hoped that this explication will convince the reader of the practicality of Rust as a high-assurance hardware/software co-design language, as well as the feasibility of performing full functional correctness proofs of code written in this Rust subset. We describe related work, then provide concluding remarks.

2 The Rust Programming Language

The Rust Programming Language [8] is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. A distinguishing feature of Rust is a non-scalar object may only have one owner. For example, one cannot assign a reference to an object in a local variable, and then pass that reference to a function. The Rust runtime performs array bounds checking, as well as arithmetic overflow checking (the latter can be disabled by a build environment setting). In most other ways, Rust is a fairly conventional modern programming language, with interfaces (called traits), lambdas (termed closures), and pattern matching, as well as a macro capability. Also in keeping with other modern programming language ecosystems, Rust features a build and package management system, named cargo.

Rust has garnered significant interest and use as a modern, type-safe, memory-safe language, with compiled code performance approaching that of C/C++. Google [9] and Amazon [10] make significant use of Rust, and Linus Torvalds has commented positively on the near-term ability of the Rust toolchain to be used in Linux kernel development [11]. The latter capability comes none too soon, as use of C/C++ continues to spawn a seemingly never-ending parade of security vulnerabilities, which continue to manifest at a high rate [12] despite the emergence and use of sophisticated C/C++ analysis tools.

Our interest in Rust additionally stems from its (until now, unrealized) potential as a hardware/software co-assurance language that can be used to create high-assurance systems, including those that must meet zero trust architecture requirements. We are particularly motivated by new autonomous and semi-autonomous platforms that require sophisticated algorithms and data structures, are subject to stringent accreditation/certification, and encourage hardware/software co-design approaches. (For an unmanned aerial vehicle use case illustrating a formal methods-based systems engineering environment, please consult [4].) In this paper, we explore the use of Rust as a High-Level Synthesis (HLS) language [13]. Most incumbent HLS languages are a subset of C, e.g. Mentor Graphics' Algorithmic C [14], or Vivado HLS by Xilinx [15]. A Rust-based HLS would bring a single modern, type-safe, and memory-safe expression language for both hardware and software realizations, with very high assurance.

Another keen research topic is reasoning about application logic written in the imperative style favored by industry. Much progress has been made in this area in recent years, and we can now verify the correctness of algorithm and data structure code that utilizes idioms such as records, loops, modular integers, and the like; and verified compilers can guarantee that such code is compiled correctly to binary [16, 17]. Progress has also been made in the verification of hardware/software co-design algorithms, where array-backed data structures are common [7, 18]. (NB: This style of programming addresses one of the shortcomings of Rust, namely its lack of support for cyclic data structures.)

3 Hardware/Software Co-assurance at Scale

In order to begin to realize our aspirational vision for hardware/software co-assurance at scale, we have conducted several experiments employing a state-of-the-art toolchain, due to Russinoff and O'Leary, and originally designed for use in floating-point hardware verification [19], to determine its suitability for the creation of safety-critical/security-critical applications in various domains.

Algorithmic C [14] is a High-Level Synthesis (HLS) language, and is supported by hardware/software co-design environments from Mentor Graphics, *e.g.*, Catapult [20]. Algorithmic C defines C++ header files that enable compilation to both hardware and software platforms, including support for the peculiar bit widths employed, for example, in floating-point hardware design. Restricted Algorithmic C (RAC) imposes several restrictions beyond those of Algorithmic C. The most significant of these is that pointers are not allowed, all loops must terminate, and all functions must be side-effect-free.

An ACL2 translator converts imperative RAC code to functional ACL2 code. Loops are translated into tail-recursive functions, with automatic generation of measure functions to guarantee admission into the logic of ACL2 (RAC subsetting rules ensure that loop measures can be automatically determined). Structs and arrays are converted into functional ACL2 records. The combination of modular arithmetic and bit-vector operations of typical RAC source code is faithfully translated to functions supported by Russinoff's RTL theorem library. ACL2 is able to reason about non-linear arithmetic functions, so this usual concern is not an issue. Finally, the RTL theorem library in ACL2 is capable of reasoning about a combination of arithmetic and bit-vector operations, which is a very difficult feat for most automated solvers.

Recently, we have investigated the synthesis of Field-Programmable Gate Array (FPGA) hardware directly from high-level architecture models, in collaboration with colleagues at Kansas State University. The goal of this work is to enable the generation of high-assurance hardware and/or software from high-level architectural specifications expressed in the Architecture Analysis and Design Language (AADL) [21], with proofs of correctness in ACL2.

4 Restricted Algorithmic Rust

As a study of the suitability of Rust as an HLS, we have crafted a Rust subset, inspired by RAC, which we have imaginatively named Restricted Algorithmic Rust, or RAR [22]. In fact, in our first implementation of a RAR toolchain, we merely "transpile" (perform a source-to-source translation of) the RAR source into RAC. By so doing, we leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. By transpiling RAR to RAC, we gain access to existing HLS compilers (we can generate code for either the Algorithmic C or Vivado HLS toolchains via some simple C preprocessor directives). We are also able to leverage the RAC-to-ACL2 translator that Russinoff and colleagues at Arm have successfully utilized in industrial-strength floating point hardware verification.

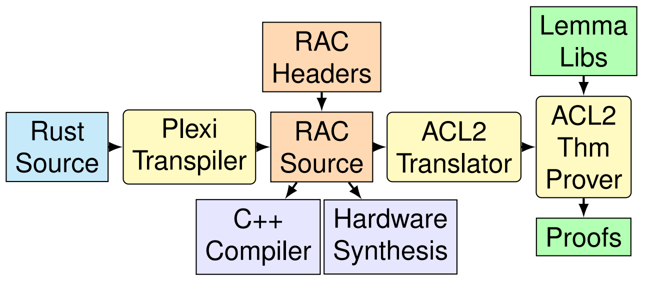


Figure 1: Restricted Algorithmic Rust (RAR) prototype toolchain.

As we wish to utilize the RAC toolchain as a backend in our initial work, we adopt the same semantic restrictions for RAR as described in Russinoff's book. Additionally, in order to ease the transition to/from C, we support a commonly used macro that provides a C-like *for* loop in Rust. Note that, despite the restrictions, RAR code is proper Rust; it compiles to binary using the standard Rust compiler.

RAR is transpiled to RAC via a source-to-source translator, as depicted in Fig. 1. Our transpiler is based on the *plex* parser and lexer generator [23] source code. We thus call our transpiler *Plexi*, a nickname given to a famous (and now highly sought-after) line of Marshall guitar amplifiers of the mid-1960s. Plexi performs lexical and syntactic transformations that convert RAR code to RAC code. This RAC code can then be compiled using a C/C++ compiler, fed to an HLS-based FPGA compiler, as well as translated to ACL2 via the RAC ACL2 translator, as illustrated in Fig. 1.

We have implemented several representative algorithms and data structures in RAR, including:

- a suite of array-backed algebraic data types, previously implemented in RAC [18, 22];
- a significant subset of the Monocypher [24] modern cryptography suite, including XChacha20 and Poly1305 (RFC 8439) encryption/decryption, Blake2b hashing, and X25519 public key cryptography; and
- a DFA-based JSON lexer, coupled with an LL(1) JSON parser. The JSON parser has also been implemented using Greibach Normal Form (previously implemented in RAC, as described in [3]).

The RAR examples created to date are similar to their RAC counterparts in terms of expressiveness, and we deem the RAR versions somewhat superior in terms of readability (granted, this is a very subjective evaluation). Additionally, RAR support of basic Rust syntax gives embedded developers an "on-ramp" to a more modern development language, as opposed to being forced to stay with C in order to achieve high performance and low latency. Further, the benefits of using the Rust compiler in RAR code development should not be discounted: its enforcement of type safety and memory safety, coupled with its efficient code generation capability, encourages performant, high-quality code development.

5 Examples

5.1 Circular Queue

High-assurance data structures are necessary building blocks for any zero-trust architecture realization. In this section, we present an example of a verified circular queue implemented using RAR. Circular queues can be found in both hardware and software realizations, making it an ideal example. First, we declare the basic queue structure, as shown below. The maximum queue size can be changed by modifying the `CQ_SZ` constant; ACL2 can reason about arrays of any size.

```
#[derive(Copy, Clone)]

struct CQ {
  front: usize,
  rear: usize,
  arr: [i64; CQ_SZ],
}
```

A typical circular queue operator is the head-of-queue accessor:

```
fn CQ_hd(CObj: CQ) -> (u8, i64) {
  if (CQ_isEmpty(CObj)) {
    return (CQ_EMPTY, 0);
  } else {
    return (CQ_OK, CObj.arr[CObj.front]);
  }
}
```

The enqueue function is as follows:

```
fn CQ_enqueue(value: i64, mut CObj: CQ) -> (u8, CQ) {
  if (CQ_isFull(CObj)) {
    return (CQ_FULL, CObj);
  } else {
    if (CObj.front == CQ_SZ) { // Insert First Element
      CObj.front = 0;
      CObj.rear = 0;
    } else
      if (CObj.rear == CQ_MAX_NODE) {
      CObj.rear = 0;
    } else {
      CObj.rear += 1;
    }
    CObj.arr[CObj.rear] = value;
    return (CQ_OK, CObj);
  }
}
```

The circular queue source comprises some 300 lines of RAR code. We use `Plexi` to transpile the RAR source to RAC (not shown), then use the RAC tools to convert the RAC source to ACL2. An example of the translation to ACL2 is shown below for the `CQ_hd` function:

```
(DEFUN CQ_HD (COBJ)
  (IF1 (CQ_ISEMPTY COBJ)
     (MV (BITS 254 7 0) (BITS 0 63 0))
     (MV (BITS 0 7 0)
        (AG (AG 'FRONT COBJ) (AG 'ARR COBJ)))))
```

In this automatically translated function, `AG` is an ACL2 record "get" operation, `MV` provides multi-value return, and `BITS` provides a bit-width specification for a given value.

Similarly, the enqueue function is automatically translated to ACL2 as follows, where AS is an ACL2 record "set" operation:

```
(DEFUN CQ_ENQUEUE (VALUE COBJ)
  (IF1 (CQ_ISFULL COBJ)
    (MV (BITS 255 7 0) COBJ)
    (LET ((COBJ (IF1 (LOG= (AG 'FRONT COBJ) 8191)
             (LET ((COBJ (AS 'FRONT 0 COBJ)))
                (AS 'REAR 0 COBJ))
             (IF1 (LOG= (AG 'REAR COBJ) 8190)
                (AS 'REAR 0 COBJ)
                (AS 'REAR
                  (+ (AG 'REAR COBJ) 1)
                    COBJ)))))
        (MV (BITS 0 7 0)
          (AS 'ARR
            (AS (AG 'REAR COBJ)
              VALUE (AG 'ARR COBJ))
            COBJ)))))
```

At this point, we can prove theorems about the data structure implementation. We first define a well-formedness predicate cqp for the queue in ACL2. We can then prove functional correctness theorems for the circular queue operations, of the sort stated below:

```
(defthm dequeue-of-enqueue-from-empty
  (implies
   (and
    (cqp CObj)
    (= 1 (CQ_isempty CObj)))
   (= (nth 1 (CQ_dequeue (nth 1 (CQ_enqueue v CObj))))
      v)))
```

ACL2 proves the 35 correctness lemmas and theorems that we have formulated for the circular queue example automatically.

5.2 Crypto Primitives

As noted previously, cryptographic methods are commonly used to enforce zero trust architecture tenets. We have thus ported a majority of the Monocypher cryptography suite [24] (approximately 2300 lines of source code) to RAR. Monocypher is a simple, yet performant and well-maintained set of modern crypto primitives implemented in C. This porting effort was accomplished in two phases: first, the Monocypher C sources were modified to conform to the RAC subset; then that code was ported to Rust/RAR. The initial goal of these first modifications was to ensure that the Monocypher sources were amenable to the use of fixed-size arrays; and if so, to see if there was any appreciable negative performance impact. As it happened, for the selection of crypto primitives that we chose (a cross-section of Monocypher capabilities, from hashing to Encryption/Decryption to Elliptic Curve-based public key functions), the fixed-size array modifications were not that difficult, and as one can observe from columns two and three of Table 1, perfomance was nearly identical after these changes were made. (All results were obtained on one core of a 2020 MacBook Pro with a 2 GHz Intel i5 CPU, 32 GB of RAM, and running MacOS Monterey version 12.0.1.) Importantly, the last column of Table 1 reveals the translation to Rust does not negatively impact execution speed, compared to the baseline.

One additional performance measure we can readily make is to compare the results of the Monocypher speed tests under

Function	*Baseline*	*FixedArr/ PassByRef*	*FixedArr/ PassByVal*	*Rust*
Chacha20	423	437	360	389
Poly1305	1157	992	1054	1213
AuthEncrypt	309	328	264	322
Blake2b	636	631	487	735
X25519	9259	10638	7874	9433

Table 1: Monocypher performance comparisons. Higher numbers are better. X25519 results are exchanges/sec; all other results are MB/sec.

pass-by-reference vs. pass-by-value (made possible due to some C preprocessor cleverness). As one can see by comparing columns three and four of Table 1, pass by value reduces C execution speed by 20-30% for most tests.

We were able to translate the Monocypher primitives from RAR to RAC (and thence to both hardware and software synthesis), as well as to ACL2, using the toolchain of Figure 1. Test vectors from the Monocypher regression suite were then used to validate the translation to ACL2, but no significant proof efforts have been undertaken on the translated functions thus far, beyond the necessary loop termination proofs. Future verification plans are discussed in the sections that follow.

6 Related Work

Our work is inspired by, and builds upon, the pioneering work of Russinoff's team at Arm on Restricted Algorithmic C for floating-point hardware verification at scale [19]. Floating-point hardware verification utilizing theorem proving technology has a notable history (*e.g.* [25], [26], [19]). Many of these efforts have focused on engineering artifacts expressed using traditional Hardware Description Languages, such as Verilog; Russinoff's work using an HLS is a notable exception.

A number of domain-specific languages targeting both hardware and software realization have been created. Cryptol [27], for example, has been employed as a "golden spec" for the evaluation of cryptographic implementations, in which automated tools perform equivalence checking between the Cryptol spec for a given algorithm, and the VHDL implementation.

EverCrypt [28] provides a comprehensive verified implementation of modern cryptographic algorithms written in F*, then transpiled to lower-level languages, eventually producing C and/or assembly. We successfully used EverCrypt for our Remote Attestation work on CASE. We initially wished to tie in to the EverCrypt toolchain for our current work, but the lower-level forms produced by the EverCrypt transpilers did not allow us to produce solely fixed-size arrays that we needed for hardware generation. In future, we hope to modify this transpiler machinery to allow us to generate RAC or RAR code, thus allowing us to leverage the significant formal verification work produced by the EverCrypt team.

Rod Chapman recently translated the TweetNaCl compact cryptographic source code suite to the SPARK Ada subset

[29], motivated by a similar desire to ours to produce a cryptographic suite written in a higher-assurance language subset with proof support. Chapman did not, however, contemplate possible hardware implementation. We considered the TweetNaCl sources as a starting point for our work, but Monocypher exhibits superior performance, provides regression and performance testing, and is better written.

Formal verification systems for Rust include Creusot [30], based on WhyML; Prusti [31], based on the Viper verification toolchain; and RustHorn [32], based on constrained Horn clauses. And recently, AWS has announced a model-checker for Rust, Kani [33]. It will be interesting to attempt the sorts of correctness proofs achievable on our system using these verification tools.

7 Conclusion

We have developed a prototype environment to enable the Rust programming language to be used as a hardware/software co-design and co-assurance language for critical systems, focusing on systems that implement Zero Trust Architecture tenets. We have demonstrated the ability to establish the correctness of several practical data structures commonly employed in high-assurance systems through automated formal verification, enabled by automated source-to-source translation from Rust to RAC to the ACL2 theorem prover. We have also successfully applied our toolchain to cryptography and data format filtering examples typical of the algorithms and data structures employed in zero trust architecture applications.

We presented two case studies in the development, verification, and validation of zero trust primitives. For the case of an array-based circular queue, we presented the results of full functional verification after automated translation from Rust to ACL2. This was supplemented by test vectors executed using the ACL2 read/eval/print loop, thus providing validation of the translation process. For the case of cryptographic primitives, we detailed how we ported the Monocypher suite first to Russinoff's RAC, and then to the RAR Rust subset. We demonstrated that translation to a fixed-array-size formulation, needed for RAC, had no negative impacts on performance. We then exercised the RAR toolchain on a significant subset of the Monocypher suite, demonstrating the feasibility of expressing cryptographic primitives under the datatype and iterative form restrictions necessary to achieve both hardware and software synthesis, as well as automated translation to the ACL2 theorem prover.

In future work, we will continue to develop the RAR toolchain, increasing the number of Rust features supported by the RAR subset, as well as continuing to improve the ACL2 verification libraries in order to increase the ability to discharge RAR correctness proofs automatically. We will also continue to work with our colleagues at Kansas State University on direct synthesis from architectural models. Finally, we will pursue a connection to the EverCrypt work, as described earlier.

8 Acknowledgments

This work was partly funded by DARPA contract HR00111890001[1]. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Many thanks to David Russinoff of Arm for answering questions about the RAC toolchain; to Loup Vaillant for developing the excellent Monocypher crypto package; to Geoffry Song for the plex tool; and to John Hatcliff and Robby at Kansas State University for their ongoing pioneering efforts in the area of high-assurance synthesis from architectural models. Thanks also go to the anonymous reviewers for their insightful comments.

References

[1] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, *NIST Special Publication 800-207: Zero Trust Architecture*. National Institute of Standards and Technology, August 2020.

[2] D. Cofer, I. Amundson, J. Babar, D. Hardin, K. Slind, P. Alexander, J. Hatcliff, Robby, G. Klein, C. Lewis, E. Mercer, and J. Shackleton, "Cyberassured systems engineering at scale," in *IEEE Security & Privacy*, May/June 2022 (to appear).

[3] D. S. Hardin and K. L. Slind, "Formal synthesis of filter components for use in security-enhancing architectural transformations," in *Proceedings of the Seventh Workshop on Language-Theoretic Security, 42nd IEEE Symposium and Workshops on Security and Privacy (LangSec 2021)*, May 2021.

[4] E. Mercer, K. Slind, I. Amundson, D. Cofer, J. Babar, and D. Hardin, "Synthesizing verified components for cyber assured systems engineering," in *24th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021)*, October 2021.

[5] A. Petz, G. Jurgensen, and P. Alexander, "Design and formal verification of a Copland-based attestation protocol," in *ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE 2021)*, November 2021.

[6] J. Hatcliff, J. Belt, Robby, and T. Carpenter, "HAMR: An AADL multi-platform code generation toolset," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, vol. 13036 of *LNCS*, pp. 274–295, 2021.

[7] D. S. Hardin, "Verified hardware/software co-assurance: Enhancing safety and security for critical systems," in *Proceedings of the 2020 IEEE Systems Conference*, 2020.

[8] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018.

[1]Distribution Statement A (Approved for Public Release, Distribution Unlimited)

[9] J. V. Stoep and S. Hines, "Rust in the Android platform," April 2021.

[10] S. Miller and C. Lerche, "Sustainability with Rust," February 2022.

[11] R. Amadeo, "Google is now writing low-level Android code in Rust," April 2021.

[12] M. Miller, "A proactive approach to more secure code," July 2019.

[13] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[14] Mentor Graphics Corporation, *Algorithmic C (AC) Datatypes*, 2016.

[15] Xilinx, Inc., *Vivado Design Suite User Guide: High-Level Synthesis*, December 2018.

[16] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[17] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014* (S. Jagannathan and P. Sewell, eds.), pp. 179–192, ACM, 2014.

[18] D. S. Hardin, "Put me on the RAC," in *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-20)*, pp. 142–145, May 2020.

[19] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, second ed., 2022.

[20] Mentor Graphics Corporation, *Catapult High-Level Synthesis*, 2020.

[21] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st ed., 2012.

[22] D. S. Hardin, "Hardware/software co-assurance using the Rust programming language and ACL2," in *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-22)*, May 2022.

[23] G. Song, *plex: a parser and lexer generator as a Rust procedural macro*, 2020.

[24] L. Vaillant, *Monocypher: Boring Crypto that Simply Works*, 2022.

[25] J. Harrison, "Floating-point verification using theorem proving," in *Formal Methods for Hardware Verification* (M. Bernardo and A. Cimatti, eds.), pp. 211–242, Springer Berlin Heidelberg, 2006.

[26] W. A. Hunt, S. Swords, J. Davis, and A. Slobodova, "Use of formal verification at Centaur Technology," in *Design and Verification of Microprocessor Systems for High-Assurance Applications* (D. S. Hardin, ed.), pp. 65–88, Springer, 2010.

[27] S. Browning and P. Weaver, "Designing tunable, verifiable cryptographic hardware using Cryptol," in *Design and Verification of Microprocessor Systems for High-Assurance Applications* (D. S. Hardin, ed.), pp. 89–143, Springer, 2010.

[28] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy*, IEEE, May 2020.

[29] R. Chapman, "SPARKNaCl: A verified, fast re-implementation of TweetNaCl," in *Proceedings of FOSDEM'22*, February 2022.

[30] X. Denis, *Creusot*, September 2022.

[31] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust (invited)," in *NASA Formal Methods (14th International Symposium)*, pp. 88–108, Springer, 2022.

[32] Y. Matsushita, T. Tsukada, and N. Kobayashi, "Rusthorn: Chc-based verification for rust programs," *ACM Trans. Program. Lang. Syst.*, vol. 43, oct 2021.

[33] Amazon Web Services, *Announcing the Kani Rust Verifier Project*, May 2022.

Beyond Model Checking of Idealized Lustre in KIND 2

Daniel Larraz, Arjun Viswanathan, Cesare Tinelli
The University of Iowa, USA

Mickaël Laurent
IRIF, CNRS — Université de Paris, France

Abstract

This paper describes several new features of the open-source model checker KIND 2*. Its input language and model checking engines have been extended to allow users to model and reason about systems with machine integers. In addition,* KIND 2 *can now provide traceability information between specification and design elements, which can be used for several purposes, including assessing the quality of a system specification, tracking the safety impact of model changes, and analyzing the tolerance and resilience of a system against faults or cyber-attacks. Finally,* KIND 2 *is also able to check whether a component contract is realizable or not, and provide a deadlocking computation and a set of conflicting guarantees when the contract is unrealizable.*

Keywords: Machine-precise Model Checking, Safety Analysis, Realizability Checking.

1 Introduction

KIND 2 [1] is an SMT-based model checker for safety properties of finite- and infinite-state synchronous reactive systems. It takes as input models written in an extension of the dataflow Lustre language [2]. The extension allows the specification of assume-guarantee-style contracts for the modeled system and its components which enables modular and compositional reasoning and considerably increases scalability. KIND 2's contract language [3] is expressive enough to allow one to represent any (LTL) regular safety property by recasting it in terms of invariant properties. One of KIND 2's distinguishing features is its support for modular and compositional analysis of hierarchical and multi-component systems. KIND 2 traverses the subsystem hierarchy bottom-up, analyzing each system component, and performing fine-grained abstraction and refinement of the sub-components. At the architectural level, KIND 2 runs concurrently several model checking engines which cooperate to prove or disprove contracts and properties. In particular, it combines two induction-based model checking techniques, k-induction [4] and IC3 [5], with various auxiliary invariant generation methods. All the engines are fully automated and logic-based, relying on external SMT solvers for satisfiability/entailment checks and other relevant logical operations such as quantifier elimination.

KIND 2 is open-source and distributed in binary and source-code form under a liberal license.[1] This paper focuses on its most recent features, in particular, reasoning about models with machine integer values, providing traceability information between specification and design elements, and checking component contracts are realizable.

2 Machine-precise Verification

Lustre is a synchronous dataflow language that operates on infinite streams of values of three basic types: `bool`, `int` (finite precision integers), and `real` (floating point numbers). In contrast, KIND 2 considers an idealized version of Lustre, which treats `int` as the type of mathematical integers, and `real` as the type of real numbers. Idealized Lustre programs can be faithfully encoded as state transition systems $S = \langle \mathbf{s}, I[\mathbf{s}], T[\mathbf{s}, \mathbf{s}'] \rangle$ where $\mathbf{s}$ is a vector of typed state variables, I is the initial state predicate, and T is a two-state transition predicate (with $\mathbf{s}'$ being a renamed version of $\mathbf{s}$). Then, instances of I and T can be expressed as quantifier-free first-order formulas over the combined theory of equality with uninterpreted functions and integer/real arithmetic. SMT solvers implement efficient decision procedures for the fragment of this theory that limits arithmetic constraints to linear ones. Although using idealized Lustre is often adequate for proving and disproving a wide range of properties of programs of interest, sometimes it is important to reason with respect to the original semantics of the numeric types (for instance, to capture accurately the modulo n behavior of arithmetic operators over machine integers). For the latter cases, we have extended KIND 2 to support both signed and unsigned versions of C-style machine integers of size 8, 16, 32, and 64.

Semantics, declaration, and value construction. The standard semantics of machine integers of size w is binary numbers of width w, with signed machine integers represented using 2's complement. We effectively adopt the same semantics by representing machine integers internally as (signed or unsigned) bit vectors of width w. KIND 2 currently supports signed machine integers of width 8, 16, 32 and 64, allowing expressions of types `int8`, `int16`, `int32`, and `int64`, respectively, and their unsigned versions, with types `uint8`, `uint16`, `uint32`, and `uint64`. Machine integers values

[1]Kind 2 is available at `http://kind.cs.uiowa.edu`.

can be constructed using explicit conversion functions applied to integer constants, with a conversion functions for each possible destination type. For example, `uint8` converts any numeral n to the unsigned 8-bit value corresponding to the integer value $(n \mod 8)$. This means, for instance that `uint8 0` and `uint8 256` are both converted to the 8-bit zero value. Conversions in the opposite direction are also possible, with the expected inclusion semantics. Conversions between machine integers of different widths are also allowed as long as the types are both signed or both unsigned. Values are adjusted modulo the range of the destination type when converted to a smaller width, and remain unchanged when converted to a larger width.

Operations. Kind 2 supports C-style arithmetic, logical, shift, and comparison operations over machine integers. Lustre's integer operators `+`, `-`, `*`, `div`, and `mod` are overloaded to apply also to two machine integers of the same type and return a machine integer of that type.

The integer comparison operators `>`, `<`, `>=`, `<=`, `=` are overloaded to the corresponding binary operations over machine integers of the same type. They all output a boolean value.

There are new machine integer operators for bit-wise conjunction (`&&`), disjunction (`||`), and negation (`!`), all with the expected arity and type, as well as left shift (`lsh`) and right shift (`rsh`) operators. The last two are both binary: the two inputs must have the same width but only the first can be signed. The output is signed if the first input is signed, and is unsigned otherwise; it is obtained by shifting the first input by the number of positions indicated by the second input. Right-shifting when the first operand is signed results in an arithmetic right shift, where the sign bit is preserved. A left-shift is equivalent to multiplication by 2 (modulo the width), and a right-shift is equivalent to division by 2. In other words, the left shift operator shifts towards the most-significant bit and the right shift operator shifts towards the least-significant bit.

To check safety properties of Lustre models with machine integers Kind 2 relies on off-the-shelf SMT solvers by leveraging their support for the theory of bit vectors of fixed width. Currently, only the SMT solvers Z3 [6] and cvc5 [7] support logics that allow the *combined* use of mathematical integers and machine integers. To use any of the other supported SMT solvers, the Lustre input must contain only boolean and machine integer types.

Kind 2's manual [8] provides more detailed information on machine integer support and on which SMT solvers are recommended for different combinations of data types in the input model.

In future work, we plan to extend Kind 2 to support floating point types as well.

3 Realizability Checking of Contracts

Contract-based software development is a major methodology for the rigorous construction of component-based reactive systems, embedded systems in particular. Contracts provide a mechanism for capturing the information needed to specify and reason about component-level properties at a desired level of abstraction. In this paradigm, a component C can be associated with a contract specifying its input-output behavior in terms of guarantees provided by C when its environment satisfies certain assumptions. Contracts are an effective way to establish boundaries between components and can be used to facilitate proofs of global properties of a complex system prior to its construction. Such proofs capitalize on the fact that complex components are typically specified simply as the composition of lower-level components. However, they are also built upon the implicit assumption that each leaf-level component contract in the system hierarchy is *realizable*. Roughly speaking, this means that it is possible to construct a component that, for any input allowed by the contract assumptions, can produce an output satisfying the contract guarantees. Unfortunately, without tool support it is all too easy for system designers to write leaf-level contracts that are unrealizable.

In Kind 2 the behavior of each component, or *node* in Lustre terminology, can be specified by providing either a set of equations that define the component's output in terms of its input and internal state (a *low-level* specification), or an assume-guarantee contract (a *high-level* specification), or both. The syntax restrictions and semantics of the Lustre language ensure that every low-level specification of a component is *executable* in the sense that for each possible input and internal state for the component there is a *unique* output and next state for the component to move to. Hence low-level specifications in Lustre are realizable by construction. When both specifications are provided in Kind 2, the low-level specification is expected to be a refinement of the high-level one. Kind 2 checks this by verifying that every execution that satisfies the former also satisfies the latter. Informally, we say that the set of equations *satisfy* the contract. In compositional reasoning, when only a contract is provided for a subcomponent C, Kind 2 assumes the existence of such a component when checking the properties of components that use C. This, however, may lead to bogus compositional proof arguments when C's contract is unrealizable.

Kind 2 now provides an option to check whether the contract of a component with no low-level specification is realizable. When a contract is unrealizable, the only way to fully explain why the contract is impossible to satisfy is to provide a *counter-strategy*, a (temporal) description of an *environment* for the component that prevents any potential realization of that component.

A user can examine a counter-strategy to try understand the reasons the contract is unrealizable, and fix it accordingly. However, as pointed out by Könighofer et al [9], a counter-strategy may be very large and complex, especially if it was generated automatically. For this reason, instead of a counter-strategy, Kind 2 provides examples of execution scenarios that lead to impossible conditions. Specifically, it outputs a single, finite computation path all of whose transitions satisfy the contract but whose end state has no outgoing transitions that satisfy the contract. In other words, Kind 2 provides concrete evidence for the existence of a *reachable deadlocking* state d for any putative realization of the contract. In addition, to facilitate the comprehension of the deadlocked state further, KIND 2 also provides a state d' such that transitioning from d to d' would minimize the number of violated contract guarantees. The (non-empty) set of the violated guarantees in question is returned as well.

When the contract is proven unrealizable, the user has also the option of invoking a sanity check on whether the contract is *satisfiable* at all, i.e., whether it is possible to construct a component such that for *at least* one input sequence allowed by the contract assumptions, there is some output value that the component can produce to satisfy the contract guarantees.

The realizability check implemented in KIND 2 is largely based on a synthesis procedure for infinite-state reactive systems, called JSYN-VG, by Katis et al. [10]. The main difference is that while the original work relies on a dedicated solver to implement the functionality provided by the AE-VAL procedure [11] of JSYN-VG, our implementation only requires a generic quantifier elimination procedure for the underlying data theories supported by KIND 2 (Booleans, linear integer arithmetic, and linear real arithmetic). Such quantifier elimination capabilities are provided by state-of-the-art SMT solvers such as Z3 [6] and cvc5 [7].

A detailed description of Kind 2's realizability checking functionality and an experimental evaluation comparing our implementation and the original implementation of JSYN-VG in the JKIND model checker is available in a technical report [12].

4 Merit and Blame Assignment

One clear strength of model checkers is their ability to return precise error traces witnessing the violation of a given safety property. In addition to being invaluable in helping identify and correct bugs, error traces also represent a checkable unsafety certificate. Similarly, some model checkers are able to return some form of corroborating evidence when they declare a safety property to be satisfied by a system under analysis. For instance, KIND 2 can produce an independently checkable proof certificate for each of the properties it claims to hold [13]. Since these proof certificates are meant for automated proof checkers, however, they usually provide limited user-level insights on what elements of the system model contribute to the satisfaction of a property.

KIND 2 now offers two new diagnostic features that provide additional information on a chosen set of verified properties [14]: (i) the identification of minimal sets of model elements that are *sufficient* to prove the properties together with the subset of model elements that are *necessary* to prove those properties; (ii) the computation of minimal sets of model constraints whose violation leads the system to falsify one of more of the chosen properties.

Although these two pieces of information are closely related, they can be naturally mapped to difference typical use cases in model-based software development: respectively, *merit assignment* and *blame assignment*. With the former the focus is on assessing the quality of a system specification, tracking the safety impact of model changes, and assisting human users in the synthesis of optimal implementations. With the latter, the goal is to determine the tolerance and resilience of a system against faults or adversarial environments due to natural causes or cyber-attacks. In general, proof-based traceability information can be used to perform a variety of engineering analyses, including vacuity detection [15]; coverage analysis [16, 17]; impact analysis [18], design optimization; and robustness analysis [19, 20].

The merit assignment functionality relies on the concept of inductive validity core introduced by Ghassabani et al. [21]. Generally speaking, given a set of *model elements* M and an invariance property P, an *inductive validity core* (IVC) for P is a subset of M that is enough to prove P invariant. Kind 2 allows the user to choose among four sets of model elements: assumptions/guarantees in contracts, node calls, equations in node definitions, and assertions[2]. Note that M itself is an IVC, although not a very interesting one. In practice, for complex enough models, smaller IVCs exist. In fact, it is often possible to compute efficiently a smaller IVC that contains few or no irrelevant elements. We can ensure that the elements of an IVC for a property P are necessary by requiring the IVC to be *minimal*, that is, have no proper subsets that are also an IVC for P. KIND 2 offers the option to compute a *small* but possibly non-minimal IVC, *a minimal* IVC (MIVC), or *all minimal* IVCs.

IVCs for coverage and change impact analysis. If a property P of a system S has multiple MIVCs, inspecting all of them provides insights on the different ways S satisfies P. Moreover, given all the MIVCs for P, it is possible to partition all the model elements into three sets [18]: a set of *MUST* elements which are required for the satisfiability of P in every case, a set of *MAY* elements which are optional, and a set of elements that are irrelevant. This categorization provides complete traceability between specification and design elements, and can be used for coverage analysis [17] and for tracking the safety impact of model changes [14]. For instance, a change to an element e in the MAY set for P will not affect the satisfaction of P but will definitely impact some other property Q if e occurs in the $MUST$ set for Q.

IVCs for fault-tolerance or cyber-resiliency analysis. Another use of IVCs, is in the analysis of a system's tolerance to faults [20] or resiliency to cyber-attacks [19]. For instance, an empty MUST set for a system S and one of its invariants P indicates that the property is satisfied by S in various alternative ways, making the system tolerant to faults or resilient against cyber-attacks as far as P is concerned. In contrast, a large MUST set suggest a more brittle system, with multiple points of failure or a big attack surface.

Quantifying a system's resilience. To help quantify the resilience of a system, KIND 2 also supports the computation of minimal cut sets (aka, *minimal correction sets*) for an invariance property P. Given a set of model elements M, a *cut set* C for P is a subset of M such that P is no longer invariant for $M \setminus C$. A *minimal cut set* (MCS) for P is a cut

[2]Assertions are *unchecked* assumptions on a node's input. They are deprecated in KIND 2, in favor of contract assertions, but still supported for being part of Lustre.

set none of whose proper subsets is a cut set for P. A *smallest cut set* is an MCS of minimum cardinality. KIND 2 provides options to compute a (single) smallest cut set, all the MCSs, and all the MCSs up to a given cardinality bound. In the context of fault or security analyses, the cardinality of an MCS for a property P represents the number of design elements that must fail or be compromised for P to be violated. The smaller the MCS, or the higher the number of MCSs of small cardinality, the greater the probability that the property can be violated.

We refer the interested reader to a related publication [14] and technical report [22] for implementation details and experimental results on merit and blame assignment with KIND 2.

References

[1] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II* (S. Chaudhuri and A. Farzan, eds.), vol. 9780 of *Lecture Notes in Computer Science*, pp. 510–517, Springer, 2016.

[2] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE," *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 785–793, 1992.

[3] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli, "Cocospec: A mode-aware contract language for reactive systems," in *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings* (R. D. Nicola and eva Kühn, eds.), vol. 9763 of *Lecture Notes in Computer Science*, pp. 347–366, Springer, 2016.

[4] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings* (W. A. H. Jr. and S. D. Johnson, eds.), vol. 1954 of *Lecture Notes in Computer Science*, pp. 108–125, Springer, 2000.

[5] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings* (R. Jhala and D. A. Schmidt, eds.), vol. 6538 of *Lecture Notes in Computer Science*, pp. 70–87, Springer, 2011.

[6] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[7] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (D. Fisman and G. Rosu, eds.), vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.

[8] K. . Developers, *The Kind 2 user manual*. The University of Iowa, 2022.

[9] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 563–583, 2013.

[10] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen, "Validity-guided synthesis of reactive systems from assume-guarantee contracts," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II* (D. Beyer and M. Huisman, eds.), vol. 10806 of *Lecture Notes in Computer Science*, pp. 176–193, Springer, 2018.

[11] G. Fedyukovich, A. Gurfinkel, and A. Gupta, "Lazy but effective functional synthesis," in *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings* (C. Enea and R. Piskac, eds.), vol. 11388 of *Lecture Notes in Computer Science*, pp. 92–113, Springer, 2019.

[12] D. Larraz and C. Tinelli, "Realizability checking of contracts with kind 2," *CoRR*, vol. abs/2205.09082, 2022.

[13] A. Mebsout and C. Tinelli, "Proof certificates for smt-based model checkers for infinite-state systems," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016* (R. Piskac and M. Talupur, eds.), pp. 117–124, IEEE, 2016.

[14] D. Larraz, M. Laurent, and C. Tinelli, "Merit and blame assignment with kind 2," in *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings* (A. Lluch-Lafuente and A. Mavridou, eds.), vol. 12863 of *Lecture Notes in Computer Science*, pp. 212–220, Springer, 2021.

[15] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *Int. J. Softw. Tools Technol. Transf.*, vol. 4, no. 2, pp. 224–233, 2003.

[16] H. Chockler, D. Kroening, and M. Purandare, "Coverage in interpolation-based model checking," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010* (S. S. Sapatnekar, ed.), pp. 182–187, ACM, 2010.

[17] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. E. Heimdahl, and L. G. Wagner, "Proof-based coverage metrics for formal verification," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017* (G. Rosu, M. D. Penta, and T. N. Nguyen, eds.), pp. 194–199, IEEE Computer Society, 2017.

[18] A. Murugesan, M. W. Whalen, E. Ghassabani, and M. P. E. Heimdahl, "Complete traceability for requirements in satisfaction arguments," in *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pp. 359–364, IEEE Computer Society, 2016.

[19] K. Siu, A. Moitra, M. Li, M. Durling, H. Herencia-Zapana, J. Interrante, B. Meng, C. Tinelli, O. Chowdhury, D. Larraz, *et al.*, "Architectural and behavioral analysis for cyber security," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pp. 1–10, IEEE, 2019.

[20] D. Stewart, J. J. Liu, M. W. Whalen, D. Cofer, and M. Peterson, "Safety annex for the architecture analysis and design language," 2020.

[21] E. Ghassabani, A. Gacek, and M. W. Whalen, "Efficient generation of inductive validity cores for safety properties," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (T. Zimmermann, J. Cleland-Huang, and Z. Su, eds.), pp. 314–325, ACM, 2016.

[22] D. Larraz, M. Laurent, and C. Tinelli, "Merit and blame assignment with kind 2," *CoRR*, vol. abs/2105.06575, 2021.

An AADL Contract Language Supporting Integrated Model- and Code-Level Verification

John Hatcliff
Kansas State University, Department of Computing and Information, USA; email: hatcliff@ksu.edu
Danielle Stewart
Galois, Inc., USA; email: danielle@galois.com
Jason Belt
Kansas State University, Department of Computing and Information, USA; email: belt@ksu.edu
Robby
Kansas State University, Department of Computing and Information, USA; email: robby@ksu.edu
August Schwerdfeger
Galois, Inc., USA; email: august.schwerdfeger@galois.com

Abstract

Model-based systems engineering approaches support the early adoption of a model – a collection of abstractions – of the system under development. The system model can be augmented with key properties of the system including formal specifications of system behavior that codify portions of system and unit-level requirements. There are obvious gaps between the model with formally specified behavior and the deployed system. Previous work on component contract languages has shown how behavior can be specified in models defined using the Architecture Analysis and Design Language (AADL) – a SAE International standard (AS5506C). That work demonstrated the effectiveness of model-level formal methods specification and verification but did not provide a strong and direct connection to system implementations developed using conventional programming languages. In particular, there was no refinement of model-level contracts to programming language-level contracts nor a framework for formally verifying that program code conforms to model-level behavioral specifications.

To address these gaps and to enable the practical application of model-contract languages for verification of deployed high-integrity systems, this paper describes the design of the GUMBO AADL contract language that integrates and extends key concepts from earlier contract languages. The GUMBO contract language (GCL) is closely aligned to a formal semantics of the AADL run-time framework, which provides a platform- and language- independent specification of AADL semantics. We have enhanced the HAMR AADL code generation framework to translate model-level contracts to programming language-level contracts in the Slang high-integrity language. We demonstrate how the Logika verification tool can automatically verify that Slang-based AADL component implementations conform to contracts, both at the code-level and model-level. Slang-based implementations of AADL systems can be executed directly or compiled to C for deployments on Linux or the seL4 verified microkernel.

Keywords: Program verification, model-based system engineering, formal methods.

1 Introduction

Over the last few decades, significant advancements in Model-Based Systems Engineering (MBSE) approaches, including modeling languages, model-level analyses, simulation, and code generation have improved the ability to design and deploy complex critical systems. Within the broader modeling space, the Architecture Analysis and Design Language (AADL) is distinguished by its relatively strong semantic emphasis (compared to other modeling languages like UML and SysML) and by its ecosystem of analysis tools that leverage that semantics. Tools that generate code from AADL models such as Ocarina [1], RAMSES [2], and HAMR [3] are helping fulfill the AADL community's MBSE visions by supporting the deployment of critical systems derived directly from models.

Due to in part to the AADL's strong semantics, researchers have developed formal model-based behavior specification and verification techniques. In particular, AADL component contract languages such as AGREE [4] and BLESS [5] have illustrated how system requirements can be refined into system and component-level formal specifications based on propositional and first-order logic.

Despite this progress, there are still gaps in the AADL MBSE vision related to deeply integrating behavioral specifications

[6] at the model and code level. In recent years there has been significant progress on developing highly automated *code-level* verification tools including those for high-integrity languages such Spark Ada [7] and Frama-C [8], as well as program checking tools for general purpose languages including Java, C, C#, Rust, etc. However, both AGREE and BLESS analyses apply to the *model-level* and only address thread behavior specifications based on rather abstract notations (Lustre-based notations for AGREE, and state transition notations for BLESS). Given that the AADL has a significant vision related to code generation in the standard, including a code generation annex, descriptions of thread code organization (thread entry points), and descriptions of run-time libraries that implement foundational thread dispatching and communication steps, a stronger connection between model-level contract languages and code-level contract languages would be a significant step forward in further developing the AADL MBSE vision.

In this paper, we address these gaps by presenting an AADL contract language that can be utilized by AADL code generation to produce application source code with code-level contracts derived from model-level contracts. This required us to reorient concepts from both AGREE and BLESS to better align with notions of AADL thread entry points and AADL Application Programming Interfaces (APIs) associated with AADL's run-time services.

The contributions of this paper are as follows.

- We defined and implemented the AADL GUMBO[1] contract language (GCL) as an AADL annex, and we implemented full editing support for the GUMBO Contract Language (GCL) in the AADL OSATE IDE.

- We extended HAMR's multi-platform AADL code generator to automatically extract contracts from AADL models and weave them into the HAMR-generated code skeletons in the Slang high-integrity subset of Scala [9].

- We illustrated how the Logika-powered Sireum Integrated Verification Environment (IVE) for Slang can support contract reasoning at the code level, and that Logika can be used by engineers to verify that their thread implementations conform to contracts both at the code and model-level [10].

The framework described in this paper is included in the publicly available, open source High Assurance Modeling and Rapid engineering framework (HAMR) distribution [11]. A GitHub repository [12] provides the examples discussed in this paper, along with additional examples that illustrate the features of the contract language at both model- and source-code levels.

[1]This material is based upon work supported by the U.S. Army Combat Capabilities Development Command, Aviation and Missile Center, under Contract No. W911W6-20-C-2020: Grand Unified Modeling of Behavioral Operators.

2 Background Concepts

AADL: SAE International standard AS5506C [13] defines the AADL core language for expressing the structure of embedded real-time systems via definitions of software and hardware components, their interfaces, and their communication. The AADL provides a precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements using a component-based approach [14]. The AADL standard also describes Run-Time Services (RTS) – a collection of run-time libraries that provide key aspects of threading and communication behavior. A major subset of the Run-Time Services has been formalized and a reference implementation has been developed [15], and we have designed our contract language and associated translation to code-level contracts with these definitions in mind.

HAMR: The HAMR framework generates code from AADL models for multiple execution platforms [3]. This includes generating threading, port communication, and scheduling infrastructure code that conforms to AADL run-time semantics as well as application code skeletons that engineers fill in to complete the behavior of the system. For the JVM platform, HAMR generates code in Slang [9], a high-integrity subset of Scala, which can be integrated with support code written in Scala and Java. Mixed Slang/Scala-based HAMR systems can also be translated to JavaScript (e.g., for simulation and prototyping) and run in a web browser or on the NodeJS platform. HAMR generates C infrastructure and application skeletons when targeting Linux and the seL4 microkernel [16]. Slang can be transpiled to C, and HAMR factors its C code through a Slang-based "reference implementation" of the AADL run-time and application code skeletons. Using the Logika verification framework for Slang (described below), Slang code can be verified with a high-degree of automation. This provides a basis for developing high-assurance AADL-based systems using Slang directly or via translation of Slang to C. C code transpiled from Slang can be compiled using standard C compilers, as well as the CompCert Verified C compiler [17].

Logika: Logika is a highly automated program verifier for Slang [10]. Slang's integrated contract language enables developers to formally specify method pre/post-conditions, data type invariants, and global invariants for global states. Verification of code conformance to contracts is performed compositionally and employs multiple back-end solvers in parallel, including Alt-Ergo [18], CVC4 [19], CVC5 [20], and Z3 [21]. The scalability of Logika is complemented by using incremental and parallel (distributable) verification algorithms. For situations where automated solvers cannot provide full verification, Slang includes an extensible proof language directly integrated with the programming language that Logika checks. Verification results, developer feedback on verification status, and contract/proof editing are supported in the Sireum Integrated Verification Environment (IVE) – a customization of the popular IntelliJ Integrated Development Environment (IDE).

The code-level contracts for the method include a *Requires* clause (pre-conditions), an *Ensures* clause (post-conditions), and a *Modifies* clause (frame conditions). Within the IVE, the developer can access program state/verification conditions and solver interactions.

The Logika verification engine uses asynchronous communications between the plugin client and tool server. This enables a seamless, on-the-fly integration similar to static type checking analysis usually offered by IDEs. Logika's main usability features include an as-you-type well-formedness analysis and verification of Slang programs by sending the checking tasks to a background server process, and visualizations of various helpful feedback propagated from the server as responses of the verification requests.

Example: This section presents a small example that we use for illustration.

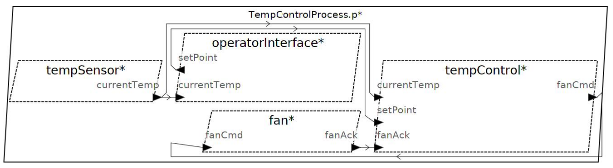


Figure 1: Temperature Control Example – AADL Graphical View

Figure 1 presents the AADL graphical view for a simple temperature control system that maintains a temperature according to a set point containing high and low bounds for the target temperature. The periodic `tempSensor` thread measures the temperature and transmits the reading on its `currentTemp` data port. It sends a notification on its `tempChanged` event port if it detects that the temperature has changed since the last reading. When the sporadic (event-driven) `tempControl` thread receives a `tempChanged` event, it reads the value on its `currentTemp` data port and compares it with the most recent set point. If the current temperature exceeds the high set point or drops below the low set point, the fan is turned on or off respectively. In turn, the fan acknowledges these commands.

Typical Workflow: Given a collection of system requirements, AADL is used to develop a system architecture. As system requirements are refined to component-level requirements, the GCL is used to formally specify behavioral properties on component interfaces as well as system-level properties. Construction of the GCL specifications is interleaved with other AADL analyses for error modeling (hazard analysis), timing and schedulability analysis, and information flow analysis. Logika is used to verify compatibility of contracts at composition points as well as system model properties. When the architecture stabilizes, HAMR is used to generate the initial build infrastructure, AADL run-time code, and application code skeletons with model-level contracts translated down to code contracts. The IVE supports development of the AADL thread components using Slang, which also provides conventional unit and system testing. Logika is applied to verify that thread implementation conform to interface contracts. Verified Slang-based thread implementations can be executed with the HAMR AADL run-time on the JVM or translated to Javascript for simulation/visualization. As requirements and model-level GCL specifications change, HAMR can be re-run to update application code skeletons and contracts while preserving existing code. Slang implementations can be transpiled to C and deployed on Linux or the seL4 microkernel (to support strong spatial and temporal partitioning). The example used in this paper was completed using this workflow and deployed on the JVM, Javascript, Linux, and seL4. Similar HAMR workflows were used, e.g., on the DARPA CASE program for mission control software for military helicopters.

3 Modeling Language Elements

There are four categories of contracts in the GCL: *integration constraints*, *data invariants*, *entry point contracts*, and *state variables*. Space constraints do not permit a detailed explanation of each, but important concepts of these categories are presented in Figure 2, which we refer to in the subsequent sections.

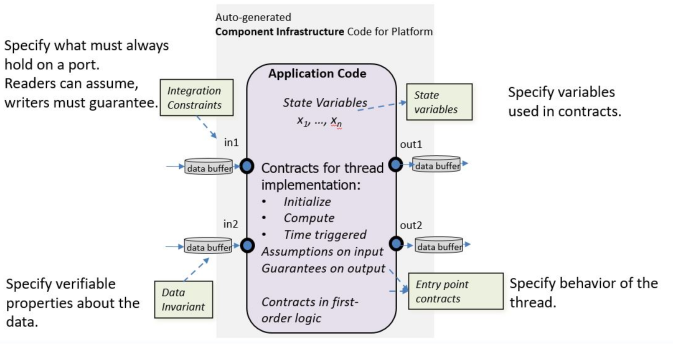

Auto-generated
Component Infrastructure Code for Platform
Specify what must always hold on a port. Readers can assume, writers must guarantee.
Integration Constraints
Application Code
State Variables
$x_1, \ldots, x_n$
State variables
Specify variables used in contracts.
in1
out1
data buffer
data buffer
Contracts for thread implementation:
• Initialize
• Compute
• Time triggered
Assumptions on input
Guarantees on output
in2
out2
data buffer
data buffer
Specify verifiable properties about the data.
Data Invariant
Contracts in first-order logic
Entry point contracts
Specify behavior of the thread.

Figure 2: Categories of contracts and how they fit into infrastructure code.

Integration Constraints: Integration constraints are the most simple conceptually and are likely the first to be used by engineers in typical workflows. The purple blocks in the concept graphic of Figure 2 indicate that each integration constraint pertains to a single port and specifies properties that must hold for any value passing through the port. They also enable engineers to specify constraints on port values that must be satisfied when components are integrated by connecting one port to another. The integration constraint shown below specifies a requirement on the usage of the `currentTemp` input data port of the temperature control thread: for any sender component S with output port p_o that is connected to (uses) the `currentTemp` port, the `degrees` field of all `Temperature.i` values flowing into the port must lie within the indicated range. `TC-Assume-01` is a unique label within the declaring component (`TempControl`) that is used for traceability. The optional string `"Current temperature range"` provides a longer human-readable description of the constraint that can be used in reporting. The `f32".."` is the current syntax within our prototype for specifying typed literals.[2]

```
integration
    assume TC-Assume-01 "Current temperature range":

        currentTemp.degrees >= f32"-70.0"
        & currentTemp.degrees <= f32"180.0";
```

[2]Ideally, the expression syntax used in contracts would be aligned with the envisioned AADL V3 expression language syntax. However, since the AADL V3 expression language and type system has not been developed yet, for simplicity of implementation, we have chosen to use Slang expressions and types in the contract grammar. Our implementation pipeline is designed to easily change the front-end concrete syntax for contracts as plans for the future of the AADL become more clear.

The listing below shows an integration constraint on the TempSensor currentTemp output port. This specifies the valid operating temperature range for the sensor.

```
integration
    guarantee Sensor_Temperature_Range:
       currentTemp.degrees >= f32"-50.0"
       & currentTemp.degrees <= f32"150.0";
```

Intuitively, the integration of TempSensor to TempControl for the respective currentTemp ports is valid (i.e., the connected output and input ports are compatible) because any temperature value flowing out of the TempSensor that satisfies the Sensor_Temperature_Range constraint will satisfy the TempControl TC-Assume-01 constraints.

In the underlying verification framework, the connection of any output port to input port gives rise to a verification obligation requiring evidence that any value flowing from a connected output port will satisfy any stated integration constraints on the input port. From a work flow perspective, this can be understood as part of the *component integration* activity. When the application logic of components are coded and verified as part of the *component development* activity, the verification framework requires evidence that all values placed in the output port by the code satisfy the output port integration constraints (i.e., guarantee). For input ports, the verification activity can assume any value flowing in from a component port will satisfy the integration constraints on the input port (since this must be guaranteed in the component integration activity as described above).

The concepts described above represent conventional "assume/guarantee" reasoning for component frameworks, and both AGREE and BLESS have analogous concepts. We have adopted the syntactic style of AGREE with the assume and guarantee keywords. For BLESS, integration constraints are a restricted form of BLESS port assertions. A key difference in the GCL is that integration constraints are distinguished from entry point contracts that specify input/output relationships – functional behavior of component implementations. These distinct concepts are handled using the same syntactic form in AGREE (assume/guarantee clauses) and BLESS (port assertions). Our rationale for having them in separate syntactic categories is explained in the discussion of entry point contracts.

Data Invariants: Requirements often specify invariants to ensure that data flowing through a system is well-formed — e.g., a temperature cannot be less than absolute zero, a timestamp that must be in 24-hour format. An invariant can also specify a well-formedness condition on a composite data structure. To support formal specification, what is needed in all these situations is the ability to associate a constraint representing an invariant with a datatype. The GCL supports the ability to add an invariant to any user-defined type specified using the Data Modeling Annex, a conventional notation in the AADL for defining data types.

The code snippet below shows a GCL invariant defined for the SetPoint data type. Bounds are defined for the low and high temperature values. Then a relational constraint specifies that the set point degrees of low is always less than or equal to that of high.

```
data SetPoint
  properties Data_Model::Data_Representation =>
       Struct;
end SetPoint;
data implementation SetPoint.i
  subcomponents
    low: data TempSensor::Temperature.i;
    high: data TempSensor::Temperature.i;
  annex GUMBO {**
   invariants
     inv SetPoint_Data_Invariant:
      (low.degrees >= f32"50.0") &  (high.degrees <
          = f32"110.0")
       &  (low.degrees <= high.degrees);
  **};
end SetPoint.i;
```

In the underlying verification framework, everywhere a data value whose type has an associated invariant is passed or accessed, the fact that the value satisfies the invariant can be accepted as a premise. For this to be sound, at each point where a value of the type is created or updated, there is a verification obligation to show that the invariant holds. In relation to the other contract categories, whereas an integration constraint applies to a specific port, a data invariant on type T is relevant for any port whose type uses or includes type T. As indicated by Figure 2, the effective constraints on ports include both integration constraints and data invariants associated with types on the ports. For any such input port, the invariant can be assumed for values retrieved from the port; for any such output port, there is a verification obligation to show that the invariant holds before sending the data through the port.

AGREE and BLESS specification languages do not support the notion of a data invariant on Data Model Annex specifications. However, it seems straightforward to add this useful feature both to the specification languages and the underlying verification frameworks.

Entry Points Contracts: The AADL standard specifies that a thread's application code is organized into entry points as illustrated in Figure 2. The *Initialize* entry point is called once during the system's initialization phase and the *Compute* entry point is called repeatedly for the thread's normal dispatching.[3]

In addition to reading from and writing to ports, the thread application code may use local state variables whose values persist between entry point executions to perform computations and save previous port readings. Not every local state variable is relevant for a component's externally specified behavior. For those that are, the GCL provides a declaration clause to make the variables available for reference in entry point contracts (center of Figure 2). For example, a state variable currentFanState can be defined in the tempControl thread to store the most recent command sent to the fan:

```
state:
  currentFanState: CoolingFan::FanCmd;
  currentSetPoint: SetPoint.i;
  latestTemp: TempSensor::Temperature.i;
```

[3]AADL includes other entry points for finalization, performing mode changes, etc. These are not yet supported in HAMR code generation or our contract language.

A thread's initialize entry point typically includes code to initialize thread local variables and to put initial values on output ports. The listing below shows excerpts of a GCL contract for the TempControl initialize entry point.

```
initialize
  modifies currentSetPoint, currentFanState,
       latestTemp;
  guarantee defaultSetPoint:
      (currentSetPoint.low.degrees == f32"70.0") &&
            (currentSetPoint.high.degrees == f32
          "80.0");
  ...
```

A modifies clause is used to specify the variables that may be modified during entry point execution. A guarantee clause is used to specify properties that a variable value must satisfy at the completion of the entry point (similar clauses for currentFanState and latestTemp are omitted). In the contract for the TempSensor initialize entry point below, the value that the currentTemp output port must have at the completion of the entry point is specified.

```
initialize
   guarantee currentTempPortInitialVal:
       currentTemp.degrees == f32"72.0";
```

Initialize entry point contracts cannot have assume clauses since the purpose of the entry point is to initialize (no aspects of pre-state, including input ports or variable values, are allowed to be read).

A periodic thread component's compute entry point is invoked at intervals corresponding to the thread's declared period, whereas a sporadic thread's is invoked upon arrival of a message to event or event data input ports.[4] The GCL provides several contract variants for compute entry points. The most significant departure from AGREE or BLESS is that sporadic threads may have clauses that apply to all dispatches of the thread (i.e., they apply no matter what event triggers a dispatch), and then additional clauses can be added to specify behaviors that apply on the arrival of messages on specific ports. Excerpts from the TempControl compute entry point contract illustrate two clauses that constrain the post-state values of the thread's latestTemp, currentSetPoint, and currentFanState state variables regardless of whether the entry point is triggered by the arrival of, e.g., the tempChanged event or the setPoint message. (Note: in the listing below, the symbol ->: is an implication.)

```
compute
    modifies currentSetPoint, currentFanState,
         latestTemp;

    guarantee  TC_Req_01:
          latestTemp.degrees < currentSetPoint.low.
               degrees
             ->: currentFanState == CoolingFan::
               FanCmd.Off;

    guarantee TC_Req_02:
          latestTemp.degrees > currentSetPoint.high.
               degrees
             ->: currentFanState == CoolingFan::
               FanCmd.On;
  ...
```

[4]An AADL model can be configured to allow exceptions to this general rule, but we omit discussions of these special cases since they do not impact the design of our overall approach.

Using the GCL's handle clause, one can specify additional constraints that apply only when the thread is dispatched by the arrival of an event on a particular port. The contract excerpt below specifies that at the completion of the compute entry point when the thread is dispatched due a tempChanged event, the latestTemp local variable will be equal to the value of the currentTemp input data port at the time of dispatch.

```
handle tempChanged:
    modifies latestTemp;

    guarantee tempChanged:
       latestTemp == currentTemp;
```

The GCL also introduces a contract variant that supports case-based reasoning. Consider a variant of the temperature control system in which the TempControl thread is periodic with currentTemp and setPoint input data ports and a fanCmd output data port.

```
compute
  modifies latestFanCmd;
  cases
     case TC_Req_01:
        assume currentTemp.degrees < setPoint.low.
             degrees;
        guarantee (latestFanCmd == CoolingFan::
             FanCmd.Off)
                        & (fanCmd == CoolingFan::
             FanCmd.Off);
     case TC_Req_02:
        assume currentTemp.degrees > setPoint.high.
             degrees;
        guarantee (latestFanCmd == CoolingFan::
             FanCmd.On)
                        & (fanCmd == CoolingFan::
             FanCmd.On);
     ...
```

The contract excerpt above uses the cases clauses to specify, e.g., that when the currentTemp is less than the low set point at the time of dispatch, then at the completion of the entry point execution, the state variable latestFanCmd and the output port fanCmd are Off.

In the underlying semantics, due to the AADL dispatch semantics and notion of input port freezing, the compute entry point can be understood as a function from port input values and local state variables to output port values and possibly updated local state variables. Assume clauses place constraints on the input state, and guarantee clauses state constraints on the output state, sometimes referencing the input state. For example, guarantee clauses can reference the input values of state variables using an In(<varName>) construct as well as the values of input ports. The GCL entry point contracts can also include invariant clauses for state variables that hold for both initialize and compute entry points. Constructs are also available to reason about the absence or presence of events/messages on event and event-data ports. There are several minor well-formedness conditions that space constraints prevent us from enumerating completely. For example, assume clauses cannot appear in an Initialize entry point contract because there is no valid pre-state to refer to before the initialization occurs, and assume clauses cannot refer to output ports.

The GCL entry point contract notation was designed to be similar to AGREE, however, GCL includes a number of new concepts. AGREE was originally designed to support synchronous systems whose foundations were expressed in Lustre, and thus it does not have distinct initialize and compute contract forms aligned with AADL standard's entry point concepts. BLESS does not have distinct Initialize and Compute contracts due to its ties to behavioral specifications written in the AADL Behavioral Annex, which does not separate state machine behavior into separate entry points. The convenience notation for `cases` is not present in AGREE or BLESS.

4 Model Contracts to Code Contracts

4.1 Code Generation Overview

A detailed overview of HAMR code generation and how code for component application logic is integrated with HAMR's AADL run-time libraries is given in [3] (focusing on Slang code generation) and [22] (focusing on C code generation).

For each thread component, HAMR generates code that provides an execution context for a real-time task. This includes: (a) infrastructure code for linking application code to the platform's underlying scheduling framework, implementing storage associated with ports, and realizing the semantics associated with event and event-data ports, and (b) templates for developer-facing code including port APIs. Representations of the GCL contracts are woven into code (b).

To support semantic consistency for code generation across multiple languages and platforms, HAMR generates code in stages. A platform-independent implementation of the AADL Run-Time Services (RTS) is generated. HAMR specifies the API and aspects of RTS in Slang and then uses Slang extensions in Scala and C to implement platform-dependent aspects. As mentioned previously, there are multiple supported back-end targets including seL4 and Linux.

For the work in this paper, we focus on Slang. The GCL contracts are translated to Slang contracts, and Slang implementations of thread component application logic are verified to conform to the contracts. In addition to enabling verified Slang-based AADL system implementations, HAMR can already translate formally verified Slang-based component implementations to C, yielding high-assurance C-based deployments which can be compiled and executed on the seL4 formally verified microkernel. We believe that the same strategies that we use to inject representations of the GCL contracts into Slang could be used to inject contracts into the generated C code, e.g., for verification using a tool like Frama-C [8]. This could be used to provide further assurance of the Slang-to-C transpiler correctness. Alternatively, it can be used to support verification of HAMR systems in workflows that focused on writing C application code directly instead of coding at the Slang level.

At the front-end of the translation pipeline, we have extended the HAMR AADL Intermediate Representation (AIR) to include representations of the GCL contracts. This JSON-based representation enables us to support multiple modeling languages and environments. Currently, the GCL is implemented with full IDE support (type-checking, error reporting, code completion, etc.) as a plug-in for the AADL Eclipse-based Open Source AADL Tool Environment (OSATE) plug-in. Although the GCL is currently implemented as an AADL annex, it is external to the HAMR framework itself, and using the language-independent AIR gives us a mechanism to support multiple modeling languages, e.g., SysMLv2.

4.2 Translated Contracts

Integration Constraints: According to the principles introduced in Section 3, any value moving through a port must satisfy the integration constraints associated with the port. HAMR code generation generates dedicated APIs for putting and getting values to/from each port. Thus, a natural strategy for realizing integration constraints at the code level is to add code contracts that: (a) require the value to be placed in an output port satisfies the associated constraints as preconditions to the `put` method, and (b) ensure the value to be retrieved from an input port satisfies the associated constraints as post-conditions.

For example, for the `currentTemp` input port in the `TempControl` thread, HAMR auto-generates: (a) a Slang representation of the predicate corresponding to the declared integration constraint, and (b) a contract on the `get` API that guarantees that the value retrieved satisfies the predicate.

```
// Auto-generated Slang predicate corresponding to
// currentTemp input port's Integration Constraint
@strictpure def currentTemp_ICPred(x:TempSensor.
     Temperature_i): B =
    -70.0f <= x.degrees & x.degrees <= 180.0f

// Auto-gen API for retrieving value from
     currentTemp input port
def get_currentTemp(): Option[TempSensor.
     Temperature_i] = {
  Contract(
    Ensures(currentTemp_ICPred(currentTemp),
            Res == Some(currentTemp))
  )
  ...(infrastructure code)...
  return value
}
```

For the corresponding sender side in the `TempSensor`, an analogous predicate is generated for the output port integration constraint, and the `put` method uses that in a precondition to ensure that all values placed on the port satisfy the constraint. Behind the scenes, the translation introduces Logika `spec` variables to provide a logical representation for the state of each port. Contract aspects that operate on ports (including the `get` and `put` methods above) use special `spec` clauses to read and update the port state abstractions during the flow of deductions in the verification.

```
// Auto-gen Slang predicate corresponding to
// currentTemp output port's Integration Constraint
@strictpure def currentTemp_ICPred(x:TempSensor.
     Temperature_i): B =
  x.degrees >= -50.0f & x.degrees <= 150.0f

// Auto-generated API for putting value on
     currentTemp output port
def put_currentTemp(value : TempSensor.
     Temperature_i) : Unit = {
  Contract(
    Requires(currentTemp_ICPred(value))
  )
  ...(infrastructure code)...
}
```

A key concept in this strategy is that the API methods serve as an abstraction for the underlying inter-component communication (specified via the AADL run-time services [15]).

Verifying that the infrastructure code is correct *is not* the focus of the verification being presented here. Rather, when a HAMR back-end is developed for a new platform, there is an obligation within the overall assurance case to demonstrate that the platform implementation correctly implements the AADL run-time communication services. This assurance effort is carried out once, and then each application built using the platform relies on the previously developed assurance. This allows component developers and system integrators to focus on verifying that component implementations and their abstract connections conform to component and system level requirements (formalized in part, using the GCL contracts). We have other lines of work on formalizing the semantics of the AADL run-time services, establishing the conformance of platform implementations to those semantics, establishing the soundness of the contract framework verification conditions against the semantics, etc. Therefore, when the contract verification framework is applied to a HAMR code base, the implementations of the APIs presented above are not verified against their contracts. Instead, the contracts are used in the checking of client code (calls to the APIs) following the usual approach for dealing with library methods in contract verification frameworks.

The verification framework applied to the client code verifies that for calls to `put` methods, the argument (e.g., `temp`) satisfies the precondition of `put` which is derived from the port's integration constraint.

```
api.put_currentTemp(temp)
```

Correspondingly, in the receiver client code of `TempControl` thread,

```
latestTemp = api.get_currentTemp().get // .get
    always succeeds
// latestTemp inherits constraint assumptions
// from currentTemp port integration constraints
```

the return value from the `get` inherits constraints on the input `currentTemp` port based on the `get` post-condition (i.e., `-70.0f <= tempControl.degrees` and `tempControl.degrees <= 180.0f`).

To verify the compatibility of integration constraints of connected ports, a Slang script is generated that uses Slang proof constructs to establish the associated entailment. For example, for the connection above, the following script fragment states the desired integration property: for all values `v` of type `Temperature_i`, `v` satisfies integration constraint of the receiving port, under the assumption that it satisfies the integration constraint of the sending port. These proof scripts are verified by Logika “behind the scenes”. More complicated forms of scripts are generated to reflect the composition of component entry points within a particular scheduling regime.

```
//  tempSensor.currentTemp --> tempControl.
    currentTemp
@pure def SensorCurrentTemp_TempControlCurrentTemp(
                                            v:
                                                Temperature_i
                                                ): Unit
                                                 = {
  Deduce(TempSensor_i_Api.currentTemp_ICPred(v) |-
          TempControl_i_Api.currentTemp_ICPred(v))
}
```

Data Invariants: HAMR translates each model-level datatype defined according to the AADL Data Model Annex into a Slang datatype. GCL datatype invariants are automatically translated to corresponding Slang datatypes using Slang's type invariant mechanism.

```
@datatype class SetPoint_i(val low: TempSensor.
    Temperature_i,
                          val high: TempSensor.
                              Temperature_i) {
  // Slang datatype invariant
  @spec def SetPoint_Data_Invariant = Invariant(
    low.degrees >= 50.0f & high.degrees <= 110.0f &
      low.degrees <= high.degrees)
                                         }
```

Slang `@datatype` structures are immutable, and compiler optimizations allow them to be used efficiently for embedded code. In the Logika framework, each time the datatype constructor is used, there is a verification obligation to show that the supplied fields satisfy the invariant. Whenever the datatype is used, the verification framework can safely assume that the invariant holds. This achieves the GCL model-level semantics for datatype invariants.

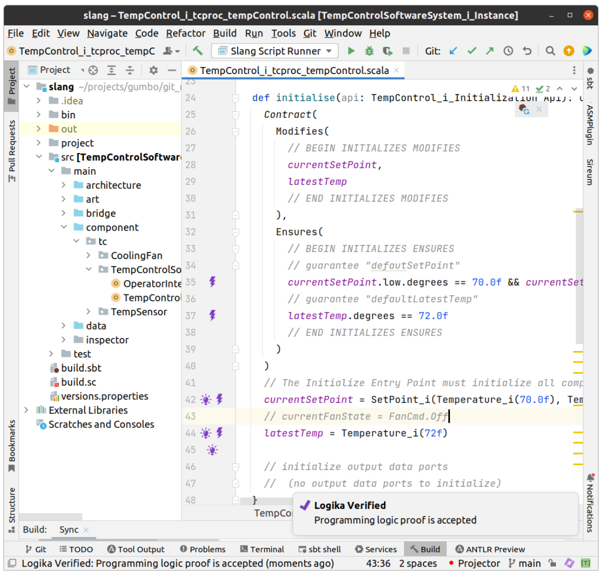
Entry Point Contracts: Local state variables, state variable invariants, and all entry point contracts are automatically inserted into the thread component application code. Due to space constraints, we show only excerpts of the contract for the `tempChanged` handler of the `TempControl` thread.

```
def handle_tempChanged(api:
     TempControl_s_Operational_Api): Unit = {
    Contract(
      Modifies(
        // BEGIN COMPUTE MODIFIES tempChanged
        currentSetPoint, currentFanState,
             latestTemp,
        // END COMPUTE MODIFIES tempChanged
      ),
      Ensures(
        // BEGIN COMPUTE ENSURES tempChanged
        // guarantee TC_Req_01
        (latestTemp.degrees < currentSetPoint.low.
             degrees)
             ->: (currentFanState == CoolingFan.
             FanCmd.Off),
        // guarantee TC_Req_02
        (latestTemp.degrees > currentSetPoint.high.
             degrees)
             ->: (currentFanState == CoolingFan.
             FanCmd.On),
         ...
        latestTemp == api.currentTemp
        // END COMPUTE ENSURES tempChanged
      )
    )
    ...(user-supplied application logic)...
  }
```

For sporadic threads, the HAMR structure for the Compute entry point is a collection of event handlers. The listing above shows that the GCL handler-independent `TempControl` compute `modifies` declarations and `guarantees` are injected directly into the Slang `handle_` method. Then the handler-specific constraint on `latestTemp` is added as appropriate for this handler. This last clause constrains the `latestTemp` state variable in the post-state to be equal to the Logika spec variable `api.currentTemp` representing the abstract state of the `currentTemp` input data port. There is an implicit conjunction for the `Ensures` clauses.

Contract Weaving: As the model goes through iterations of development, HAMR supports this by partial code-regeneration. One may regenerate into the same directory and HAMR will only make the changes in the code reflecting changes in the model and contracts. To avoid overwriting developer code upon contract changes at the model level, HAMR inserts comments to indicate where contracts begin and end (e.g., `// BEGIN INITIALIZES MODIFIES`). When code is regenerated, HAMR will parse the target code file, locate markers indicating contract blocks, and weave in updated contracts within the delimited regions. This supports iterative model and code development without the cost of overwriting developer-added application code in the thread implementations.

Code-level Verification: For our example system, Logika is able to verify that the Slang code for the thread entry points conforms to the contracts (including data invariants, etc.) in just a few seconds.


slang – TempControl_i_tcproc_tempControl.scala [TempControlSoftwareSystem_i_Instance]
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
TempControl_i_tcproc_tempC
Slang Script Runner
Git:
Project
TempControl_i_tcproc_tempControl.scala
slang ~/projects/gumbo/git_
.idea
bin
out
project
src [TempControlSoftwar
main
architecture
art
bridge
component
tc
CoolingFan
TempControlSo
OperatorInte
TempControl
TempSensor
data
inspector
test
build.sbt
build.sc
versions.properties
External Libraries
Scratches and Consoles
Pull Requests
Bookmarks
Structure
sbt
ASMPlugin
Sireum
Notifications
Build: Sync
Git TODO Tool Output Problems Terminal sbt shell Services Build ANTLR Preview
43:36 2 spaces Projector main

```
def initialise(api: TempControl_i_Initialization_Api): U
  Contract(
    Modifies(
      // BEGIN INITIALIZES MODIFIES
      currentSetPoint,
      latestTemp
      // END INITIALIZES MODIFIES
    ),
    Ensures(
      // BEGIN INITIALIZES ENSURES
      // guarantee "defautSetPoint"
      currentSetPoint.low.degrees == 70.0f && currentSet
      // guarantee "defaultLatestTemp"
      latestTemp.degrees == 72.0f
      // END INITIALIZES ENSURES
    )
  )
  // The Initialize Entry Point must initialize all comp
  currentSetPoint = SetPoint_i(Temperature_i(70.0f), Tem
  // currentFanState = FanCmd.Off
  latestTemp = Temperature_i(72f)

  // initialize output data ports
  //  (no output data ports to initialize)
}
```

Logika Verified
Programming logic proof is accepted

Logika Verified: Programming logic proof is accepted (moments ago)

Figure 3: Slang implementation of Initialize entry point code in Logika IVE

Figure 3 shows the code for the `tempControl` thread Initialize entry point and associated IVE verification annotations. The developer has filled in the implementation code, and the auto-generated contracts for the example are verified as evidenced by the *Logika Verified* banner at the bottom right of the figure. Logika incremental checking provides on-the-fly verification response as code is typed/edited. These capabilities were demonstrated in a video presentation (e.g., that shows checking of systems using a served-based parallelized checking using an 80-core server) at an industrial engagement event in January 2022 [5]. In addition to the example discussed in this paper, we have specified and verified contracts for a simple Isolette infant incubator medical device control system as well as high-integrity-oriented system components including voter-based sensor banks.

5 Related Work

The most closely related works are the AGREE and BLESS AADL contract frameworks. AGREE originally targeted AADL models that emphasized dataflow with synchronous communication based on Lustre as an underlying computational model. Thus, AGREE most naturally handles thread components with data ports and with timing and state aspects aligned with Lustre. Component behaviors can be specified using equations relating inputs to outputs as well as more complex behaviors based on Lustre-inspired "node" blocks. Model-level verification is supported by the JKind model-checking framework, with the most common notions of verification being compatibility of guarantee/assume clauses on port connections, conformance of composed contracts on networks of sub-components to contracts on an enclosing component and verification of composed component behavior against equation-oriented property specifications. Recently AGREE has added support for events and static scheduling (similar to the strategy that we use, as needed to support seL4). In addition, "fold"-operations were added to support processing of inductively defined data types. AGREE specification and checking is supported by an OSATE plug-in that includes detailed reporting of verification status of claims and counter-examples. All fixed-width data types in AGREE are currently approximated using unbounded integers and reals. AGREE has had a strong influence within the AADL community, particularly in getting industrial users to understand the benefits of assume/guarantee specifications for component interfaces and the usefulness of automated formal methods. In our language design, we try to maintain these qualities while shifting the focus from the dataflow paradigm to the general tasking/communication primitives of AADL and to supporting integrated code level checking.

The BLESS contract language focuses on AADL thread components whose implementations are defined using an extension of AADL's Behavior Annex (BA) state transition notation. Typical verification activities include: (a) proving that a BLESS transition system for a thread (perhaps annotated with assertions) conforms to its interface specification, and (b) proving that the composition of thread components satisfies end-to-end system properties. The BLESS OSATE plug-in supports editing of BLESS artifacts and coordination of verification activities. BLESS emphasizes a custom-built manual proof framework. Verification conditions are generated from interface specifications and transition systems, and then BLESS proof scripts apply proof rules to discharge the verification conditions. Building proof scripts requires significant additional effort on the part of developers compared with AGREE, but this approach supports more expressive specifications, verification of stronger properties, and detailed auditable evidence of property satisfaction. The more expressive specifications include rich notions of timing. Due to the ties to the AADL BA, compared to AGREE, BLESS more naturally supports AADL notions of event-based communication and complex event-oriented dispatch conditions. BLESS has been used to verify properties of complex embedded systems including pacemakers [5] and PCA infusion pumps [23]. GCL is less expressive than BLESS with respect to timing. In our language design, we have adopted many of BLESS's general verification condition principles and deductive structuring while emphasizing verification automation using automated solvers (using Slang manual proof steps only

[5]Video available at `https://bit.ly/tccoe22-logika`

as necessary to help the automated solvers), and direct integration with source code verification and notions of AADL entry points.

6 Conclusion

We have presented a contract language for the AADL that supports integrated component contract specification and verification at both model and code levels. The model-to-code contract translation has been validated via an implementation within an industrial-relevant code generation framework. The feasibility of the verification strategy has been validated using Logika contract checking for the Slang high-integrity language.

Our contract language design builds on concepts from earlier AADL contract languages (AGREE and BLESS) and suggests directions for more closely aligning with code generation concepts called out in the AADL standard and AADL runtime service semantics [15]. While we have illustrated code-level concepts using Slang and Logika, we believe these same principles can be supported by other code-level verification frameworks that have been used in conjunction with AADL such as SPARK Ada [24] and Frama-C [8].

There are some limitations to our current implementation. First, the approach for entry point composition and end-to-end reasoning is specialized for the static scheduling approach [22] used in our most recent industrial project that emphasized system implementations for the seL4 verified microkernel. Second, the constructs for reasoning about AADL's event and event data ports apply to ports with buffer size one. While this aligns with buffer restrictions found from HAMR and other AADL code generation frameworks [1], moving beyond this restriction will broaden the applicability of the contracts to richer event-based systems that build on widely-used message-oriented middleware frameworks. We also hope to add greater support for timing-related specifications, aligned with AADL's standard timing properties and associated real-time scheduling frameworks [25].

Acknowledgments: The authors wish to thank Todd Carpenter for his valuable inputs on this research project and other GUMBO team members from Adventium Labs for the feedback they provided on the toolset.

References

[1] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications," in *International Conference on Reliable Software Technologies*, pp. 237–250, Springer, 2009.

[2] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, pp. 67–76, IEEE, 2012.

[3] J. Hatcliff, J. Belt, Robby, and T. Carpenter, "HAMR: an AADL multi-platform code generation toolset," in *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings* (T. Margaria and B. Steffen, eds.), vol. 13036 of *Lecture Notes in Computer Science*, pp. 274–295, Springer, 2021.

[4] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)* (A. E. Goodloe and S. Person, eds.), vol. 7226, (Berlin, Heidelberg), pp. 126–140, Springer-Verlag, April 2012.

[5] B. Larson, P. Chalin, and J. Hatcliff, "BLESS: Formal specification and verification of behaviors for embedded systems with software," in *Proceedings of the 2013 NASA Formal Methods Conference*, vol. 7871 of *Lecture Notes in Computer Science*, (Berlin Heidelberg), pp. 276–290, Springer-Verlag, 2013.

[6] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson, "Behavioral interface specification languages," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 16:1–16:58, 2012.

[7] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman, "SPARK 2014 and GNATprove," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, 2015.

[8] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C, a software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, 2015.

[9] Robby and J. Hatcliff, "Slang: The sireum programming language," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 253–273, Springer, 2021.

[10] S. Laboratory, "Sireum logika." `https://logika.v3.sireum.org/index.html`, 2022.

[11] S. Laboratory, "HAMR project website." `https://hamr.sireum.org`, 2022.

[12] S. Laboratory, "GCL case studies." `https://github.com/santoslab/hilt22-case-studies/`, 2022.

[13] S. A. R. C, "Architecture analysis and design language (AADL)," 2017.

[14] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2013.

[15] J. Hatcliff, J. Hugues, D. Stewart, and L. Wrage, "Formalization of the AADL run-time services," in *Leveraging Applications of Formal Methods, Verification and Validation - 11th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2022, Rhodes, Greece (To Appear)*, 2022.

[16] "sel4 microkernel," 2015. `sel4.systems/`.

[17] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert-a formally verified optimizing compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[18] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-ergo 2.2," in *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[19] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*, pp. 171–177, Springer, 2011.

[20] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, *et al.*, "cvc5: a versatile and industrial-strength SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442, Springer, 2022.

[21] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[22] J. Belt, J. Hatcliff, Robby, J. Shackleton, J. Carciofini, T. Carpenter, E. Mercer, I. Amundson, J. Babar, D. Cofer, D. Hardin, K. Hoech, K. Slind, I. Kuz, and K. Mcleod, "Model-driven development for the seL4 microkernel using the HAMR framework," *Journal of Systems Architecture*, p. (to appear), 2022.

[23] J. Hatcliff, B. R. Larson, T. Carpenter, P. L. Jones, Y. Zhang, and J. Jorgens, "The open PCA pump project: an exemplar open source medical device as a community resource," *SIGBED Rev.*, vol. 16, no. 2, pp. 8–13, 2019.

[24] B. Carré and J. Garnsworthy, "Spark—an annotated ada subset for safety-critical programming," in *Proceedings of the conference on TRI-ADA'90*, pp. 392–402, 1990.

[25] R. Edman, H. Shackleton, J. Shackleton, T. Smith, and S. Vestal, "A framework for compositional timing analysis of embedded computer systems," in *IEEE International Conference on Embedded Software and Systems*, August 2015.

Correctness-by-Construction: An Overview of the CorC Ecosystem

Tabea Bordis, Tobias Runge, Alexander Kittelmann, Ina Schaefer
Karlsruhe Institute of Technology (KIT), Germany; email: {first name}.{last name}@kit.edu

Abstract

Correctness-by-Construction (CbC) is an incremental software development technique in the field of formal methods to create functionally correct programs guided by a specification. In contrast to post-hoc verification, where the specification and verification take part after implementing a program, with CbC the specification is defined first, and then the program is successively created using a small set of refinement rules that define side conditions preserving the correctness of the program. This specification-first, refinement-based approach as pursued by CbC has the advantage that errors are likely to be detected earlier in the design process and can be tracked more easily. Even though the idea of CbC emerged over 40 years ago, CbC is not widespread and is mostly used to create small algorithms. We believe in the idea of CbC and envision a scaled CbC approach that contributes to solving problems of modern software verification. In this short paper, we give an overview of our research regarding CbC in four different lines of research. For all of them, we provide tool support for building the CorC *ecosystem that even further enables CbC-based development for different fields of application and size of software systems. Furthermore, we give an outlook on future work that extends on our concepts for CbC.*

Keywords: correctness-by-construction, information flow control, software product lines, architecture, program verification.

1 Introduction

The amount of software in safety-critical systems increases, and, therefore, functional correctness of programs is an important concern. While most verification approaches rely on post-hoc verification, where a program is only verified *after* it is implemented, the incremental approach of *Correctness-by-Construction* (CbC) as imagined by Dijkstra [1], Gries [2], or Kourie and Watson [3] offers an alternative.[1] CbC starts with defining a formal specification in an abstract Hoare triple `{P} S {Q}` consisting of a precondition `P`, an abstract statement `S`, and a postcondition `Q`. Hoare triples represent total correctness assertions that are only `true` if, starting from the precondition, the postcondition is met after executing the eventually defined concrete program. With CbC, a Hoare triple is successively refined using a set of refinement rules to a concrete implementation, which satisfies the specification. To guarantee the correctness of the refinement steps, each rule defines specific side conditions for its applicability.

The underlying idea of this specification-first, refinement-based approach is that better programs can be constructed when the developer must think about their construction more thoroughly rather than hacking them into correctness. As a result, when applying CbC compared to classical post-hoc verification, errors are more likely to be detected earlier in the design process [3]. Additionally, programmers and users gain trust because a formal methodology was used to create the program.

To further investigate these claims and spread the idea of correct-by-construction software development, we implemented CbC in the tool CorC [5]. CorC is a graphical and textual IDE to construct algorithms following CbC. It supports developers to refine a program by a sequence of refinement steps and to verify the correctness of these refinement steps using the theorem prover KeY [6]. First evaluation results show a decreased verification effort compared to post-hoc verification [5, 7].

Besides CbC as we pursue it, there are also other tools that implement different refinement-based approaches. For example, Event-B [8] uses automata-based system descriptions instead of source code as done by CorC, ArcAngel [9] is based on Morgan's refinement calculus and comprises a very large number of refinement rules compared to minimal set in CorC, and SOCOS [10] uses invariants as specifications instead of pre- and postcondition pairs as used in CorC.

In this short paper, we give an overview of the CorC ecosystem[2] that combines existing research on improving and extending the applicability of CbC to emphasize the benefit of the ecosystem as a whole. We present four lines of research where CbC is either considered for different fields of application or integrated into software engineering processes to scale its applicability from single algorithms to object-oriented software systems and component-based architectures. All of these lines of research are implemented as extensions of CorC constituting the CorC ecosystem to benefit the community. Last, we give an outlook on our vision for future

[1] CbC as we pursue it is different from correctness-by-construction (CbyC) as promoted by Hall and Chapman [4]. CbyC is a software development process where formal modeling techniques are used to make it difficult to introduce defects and to detect and remove any defects that do occur as early as possible.

[2] The CorC ecosystem is available at https://github.com/TUBS-ISF/CorC

research on CbC for quantitative information flow control by construction.

2 The CorC Ecosystem

In the past years, we have been investigating different fields of application for CbC that benefit from the idea of a structured development process guided by specifications and refinements. First, we investigated how CbC that has been designed to create single, small algorithms can be used in modern software engineering processes. We therefore integrated object-oriented programming as a commonly used paradigm into CorC and improved the development process of CorC to enable the development using CbC with other verification techniques in concert. A natural follow-up is to develop concepts and tool support that make CbC available at scale. Currently, we aim to address this in two further directions. First, we study the role of correct-by-construction implementations in software architectures with ArchiCorC, where the main goal is to bundle CorC programs into reusable software components. Second, VarCorC is a framework for CbC-based development of software product lines. Instead of developing monolithic programs, the goal of software product lines is to systematically construct a family of similar software programs following the CbC paradigm. Last, we applied CbC-style refinement rules to ensure security of programs. Therefore, we introduced an information flow policy to CbC (IFbC) and implemented it in an extension of CorC. We briefly present all these lines of research in the following sections.

Object-Oriented Development using Correctness-by-Construction

The size and complexity of software rapidly increases. Therefore, software engineering paradigms need to adapt to these requirements. Guaranteeing correctness for these complex systems is still a challenge. To scale the applicability of CbC, we extended CorC to support object-oriented programming and investigated a software engineering process in CorC to use CbC in concert with other verification strategies or classical testing [11]. Object-oriented programming introduces classes with fields and class invariants to CbC, which allows to develop more complex projects including inheritance and interfaces. At the same time, CbC may not be the ideal method to guarantee correctness for a large software project. Therefore, we support a roundtrip engineering process from Java code to CbC development to correct Java code. We integrated these concepts and further usability-features that simplify the development using CbC in the successor of CorC, CorC 2.0.[3]

Correct-by-Construction Software Architectures

Component-based architectures allow to establish a set of reusable, correct-by-construction components. This is equally interesting for libraries, where implementations are accessed through interfaces, and for third-party developments that are easier to integrate into individual projects. Most important, creating components that modularize correct implementations allows developers to think about how to compose software systems instead of how to program a monolithic software system from scratch. We argue that this is the foundation for building large and complex systems that are based on CbC. As an extension to CorC, we propose a framework called ArchiCorC [12] that connects UML-style component modeling, formal specification, and code generation. ArchiCorC[4] comprises the following key ingredients. First, a component and interface description language is used to interconnect provided and required interfaces of components. Second, developers can either refine method signatures of provided interfaces to correct implementations using CorC, or map signatures to already existing CorC programs. Third, analyses and algorithms to check compatibility between components are provided. Finally, ArchiCorC allows to generate Java code from the correct-by-construction components.

Correctness-by-Construction for Software Product Lines

Software product lines provide systematic reuse paired with variability mechanisms to realize whole product families [13]. The commonalities and differences of the product variants are communicated as features, whose relationships are often modeled in feature models. Guaranteeing the correctness of a product line is especially challenging because of the number of possible product variants resulting from the number of feature configurations and the variable code structures [14]. To create a correct product line using CbC, we extended the original CbC approach with a new refinement rule for a variability mechanism that allows to call different implementations of a method depending on the distinct feature configuration [7, 15]. Additionally, we combined this mechanism with contract composition for variability in the pre- and postcondition [16]. We call this extension *variational Correctness-by-Construction*. VarCorC[2] uses FeatureIDE [17] and variational CbC to support the development of correct-by-construction software product lines.

Information Flow Control-by-Construction

Besides verifying functional correctness, it is also important to consider non-functional properties of a program, such as dependability, reliability, resource/energy consumption, or security [18]. For security, an information flow policy can be used to define how information may flow in a program (e.g., a flow from public to secret data is allowed, but the other way is prohibited to ensure confidentiality and integrity of the data). Our extension of CbC to ensure this type of security-by-design is called *Information Flow Control-by-Construction* (IFbC) [19]. Programs are constructed incrementally using refinement rules to follow an information flow policy. In every refinement step, security and functional correctness of the program is guaranteed, such that insecure programs are prohibited by construction. The information flow policy can be specified in any bounded upper semi-lattice (i.e., security levels are arranged in a lattice representing the allowed direction of information flow). IFbC is implemented in an extension of CorC.[5]

[3]https://github.com/TUBS-ISF/CorC/tree/CorC2.0, CorC 2.0 supports object-oriented development and development for software product lines using CbC (VarCorC)

[4]https://github.com/TUBS-ISF/ArchiCorC

[5]https://github.com/TUBS-ISF/CorC/tree/CCorC

Implementation of the CorC Ecosystem

The core of the CorC ecosystem is the tool CORC [5] which is an open-source Eclipse plug-in supporting the development of programs with CbC. It stores the structure of a CbC program including the refinement rules through a meta-model that is modeled using the Eclipse Modeling Framework.[6] CORC comes with a graphical and textual editor. The graphical editor is implemented using Graphiti[7] and visualizes the underlying meta-model in a tree structure. The textual editor is implemented using XText.[8] The beginning of a CbC program is a Hoare triple, which can then be refined by applying CbC refinement rules until there are no more abstract statements. In the background, the deductive verification tool KeY [6] is used to prove the correct application of each refinement rule.

The presented lines of research each extend the core functionality of CORC. For example, VARCORC uses FeatureIDE [17] to integrate feature models and calculations on them [15], while CORC 2.0 introduces another graphical view for classes with fields and implements a roundtrip engineering process that generates correct Java code from CbC programs [11]. Further implementation details are provided in the referenced papers and on GitHub.

3 Correctness-by-Construction - Next Steps

Driven by our research in the past years that we conducted on fields of application and extensions of CORC as tool, we can see our vision of scaling CbC as necessary practice in modern software engineering come together. We are convinced that CbC in combination with good tool support is an underestimated approach that is worth exploring. This belief is also substantiated by the participants of two user studies that agree that CORC is good tool to develop correct software [20, 21]. Besides the extensions that we presented in the previous section, there are still some open ideas that we want to explore in future work on CbC.

Feature Interactions in CbC Product Lines.

Naturally, features in a software product line interact with each other using shared variables or by calling methods defined in other features. While most of these feature interactions are wanted or even needed to implement functionality, sometimes there are also unwanted feature interactions that lead to malfunctions, unexpected behavior, or security leaks. An example of an unwanted feature interaction in an Email product line is when a feature that automatically forwards incoming mails to a certain address and a feature that decrypts incoming mails are combined together in a software product. In that case, an email may first be decrypted and afterwards forwarded in plain text to the forward receiver which violates a security property of encrypting mails. In future work, we will examine unwanted feature interactions and develop concepts to integrate safety and security constraints to the functional specification used in VARCORC, our extension of CbC for software product lines. Our goal is to give a guarantee that there are no unwanted feature interactions in a product line constructed with CbC.

[6]https://eclipse.org/emf/
[7]https://eclipse.org/graphiti/
[8]https://www.eclipse.org/Xtext/

Quantitative Information Flow Control

Confidentiality and integrity requirements on data, as well as privacy concerns can be expressed using information flow control policies, specifying how data may flow through a program and which observations an attacker may make about the data that is being processed. For future work, we will work on our vision of CbC to enable security-by-design with an extension of IFbC for quantitaive information flow. We will extend IFbC with quantitative and probabilistic information flow specifications, and develop refinement rules for correct program construction using appropriate program annotations and side conditions. We will further investigate how probabilistic programming constructs can be used to capture uncertainty of program execution and what influence they have on the information flow specifications, both classically and quantitative. To enable scalability of the correctness-by-construction engineering process, we will extend existing work on the correct construction of component-based systems with classical and quantitative information flow policies to obtain larger functionally correct systems incorporating security-by-design.

References

[1] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall PTR, 1st ed., 1976.

[2] D. Gries, *The Science of Programming*. Springer, 1st ed., 1981.

[3] D. G. Kourie and B. W. Watson, *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[4] A. Hall and R. Chapman, "Correctness by Construction: Developing a Commercial Secure System," *IEEE software*, vol. 19, no. 1, pp. 18–25, 2002.

[5] T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson, "Tool Support for Correctness-by-Construction," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 25–42, Springer, 2019.

[6] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, *Deductive Software Verification – The KeY Book*. Springer, 2016.

[7] T. Bordis, T. Runge, A. Knüppel, T. Thüm, and I. Schaefer, "Variational Correctness-by-Construction," in *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 1–9, 2020.

[8] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[9] M. Oliveira, A. Cavalcanti, and J. Woodcock, "ArcAngel: A Tactic Language for Refinement," *Formal Aspects of Computing*, 2003.

[10] R.-J. Back, J. Eriksson, and M. Myreen, "Testing and Verifying Invariant Based Programs in the SOCOS Environment," in *International Conference on Tests and Proofs*, Springer, 2007.

[11] T. Bordis, L. Cleophas, A. Kittelmann, T. Runge, I. Schaefer, and B. W. Watson, "Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY," in *The Logic of Software. A Tasting Menu of Formal Methods*, pp. 80–104, Springer, 2022.

[12] A. Knüppel, T. Runge, and I. Schaefer, "Scaling Correctness-by-Construction," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 187–207, Springer, 2020.

[13] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Citeseer, 2000.

[14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys*, 2014.

[15] T. Bordis, T. Runge, and I. Schaefer, "Correctness-by-Construction for Feature-Oriented Software Product Lines," in *International Conference on Generative Programming: Concepts and Experiences*, pp. 22–34, 2020.

[16] T. Bordis, T. Runge, D. Schultz, and I. Schaefer, "Family-based and Product-based Development of Correct-by-Construction Software Product Lines," *Journal of Computer Languages*, p. 101119, 2022.

[17] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," *Science of Computer Programming*, vol. 79, no. 0, pp. 70–85, 2014.

[18] M. H. t. Beek, L. Cleophas, I. Schaefer, and B. W. Watson, "X-by-Construction," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 359–364, Springer, 2018.

[19] T. Runge, A. Knüppel, T. Thüm, and I. Schaefer, "Lattice-based Information Flow Control-by-Construction for Security-by-Design," in *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pp. 44–54, 2020.

[20] T. Runge, T. Thüm, L. Cleophas, I. Schaefer, and B. W. Watson, "Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study," in *Refine*, Springer, 2019.

[21] T. Runge, T. Bordis, T. Thüm, and I. Schaefer, "Teaching Correctness-by-Construction and Post-hoc Verification–The Online Experience," in *Formal Methods Teaching Workshop*, pp. 101–116, Springer, 2021.

AADL Modelling with SysML v2

Jean-Charles Roger, Pierre Dissaux

Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France; email: jean-charles.roger@ellidiss.com and pierre.dissaux@ellidiss.com

Abstract

This paper introduces a new way to implement a bridge between SysML models used for the System engineering activities and AADL models relevant for more detailed specifications of the Software Sub-Systems architecture. Proposed approach takes profit of the features offered by SysML v2, the new standard candidate of the OMG.

Keywords: AADL, SysML v2

1 Introduction

Ensuring digital continuity during the engineering steps of Software intensive Systems is a key issue. To move forwards in this direction, several SysML v1 to AADL [1] model transformations have been implemented already. They are based either on the definition of a dedicated UML Profile specifying AADL stereotypes for SysML constructs, or by the implementation of fully automated transformation rules that do not require a change of the original SysML model and an extension of the used SysML tool. Ellidiss chose the second approach and provides such a transformation with AADL Inspector. Transformation rules are expressed in Prolog language and can be customized by the user within a dedicated LAMP annex [5].

The arrival of SysML v2 [2] brings a set of new attractive features that deserve to be analysed in depth. Own their own, the three following changes represent a significant positive move to better reach the goal of making SysML to AADL model interchange easier:

- a normalized textual syntax,
- support of instance models and
- extension by Domain Libraries instead of UML Profiles.

Moreover, it appears that a significant number of topics that are on the table for the preparatory work for future v3 of AADL have been addressed by SysML v2

- unified type system with SI units,
- consistent automata description and
- direct SW/HW binding construct.

The article first introduces the approach by comparing an example (generated using the AADL tools developed by Ellidiss [3]) and its SysML v2 counterpart (written by hand). It then presents the SysML v2 Domain Library for AADL and the motivation behind the choices we made.

2 An illustrative example

The proposed solution consists of a conceptual mapping between AADL v2 and SysML v2 modelling elements and its implementation under the form of a SysML v2 Domain Library for AADL. Such a Domain Library is expressed in SysML v2 itself and is thus supposed to be portable across any SysML v2 compliant tool. This conceptual mapping can also act as the list of requirements for model transformations between SysML v2 and AADL v2 and reverse.

This first example shows a software process containing two threads connected by an immediate data port connection and bound to a single processor. Figure 2 presents the graphical representation of the AADL model, edited with the Stood for AADL tool to generate the corresponding AADL text.

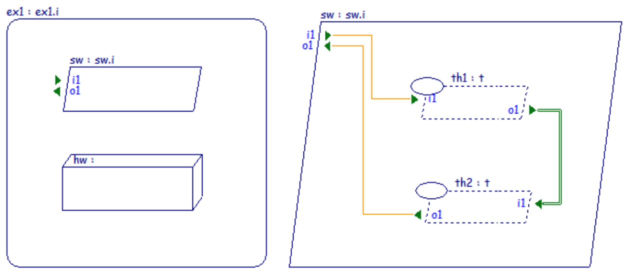


Figure 1. AADL graphical representation of the example

Using the conceptual mapping, we then manually transformed the AADL text into its SysML v2 textual version. To close the loop, Figure 2 presents the automatically generated graphical view from the SysML v2 text, using the SysML v2 pilot implementation [4].

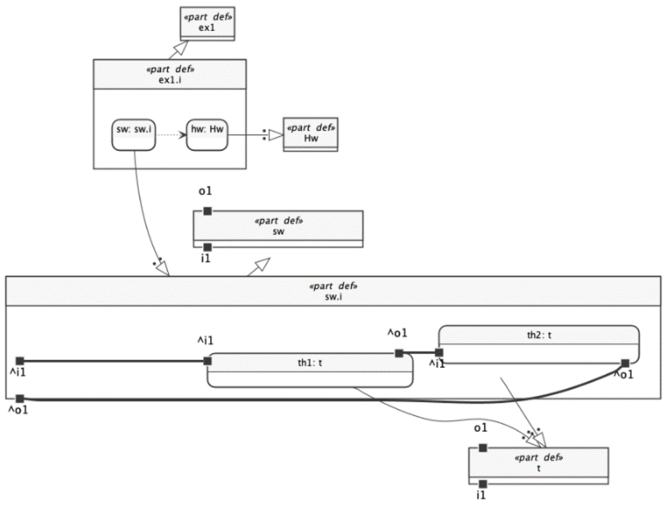


Figure 2. SysML v2 graphical representation of the illustrative example

To compare the AADL model to its counterpart in SysML v2 using our mapping, we present at first the example in its AADL textual representation.

AADL

```
package ex1_pkg public
with Base_Types;
renames data Base_Types::Integer;
system ex1 end ex1;
system implementation ex1.i
subcomponents
  hw: processor Hw {
    Scheduling_Protocol =>
    (Rate_Monotonic_Protocol);
  };
  sw: process sw.i;
properties
  Actual_Processor_Binding =>
  (reference(hw)) applies to sw;
end ex1.i;
processor Hw end Hw;
process sw
features
  i1: in data port Integer;
  o1: out data port Integer;
end sw;
process implementation sw.i
subcomponents
  th1: thread t {
    Dispatch_Protocol => Periodic;
    Period => 50ms;
    Deadline => 50ms;
    Compute_Execution_Time =>
    2ms ..2ms;
  };
  th2: thread t {
    Dispatch_Protocol => Periodic;
    Period => 50ms;
    Deadline => 50ms;
    Compute_Execution_Time =>
    2ms ..2ms;
  };
connections
  c1 : port i1 -> th1.i1;
  c2 : port th1.o1 -> th2.i1 {
    Timing => Immediate;
  };
  c3 : port th2.o1 -> o1;
end sw.i;
thread t
features
  i1: in data port Integer;
  o1: out data port Integer;
end t;
end ex1_pkg;
```

The SysMLv2 textual representation of this model is shown in the next column.

A special formatting effort has been made to highlight the similarities between the two views. Note that tokens **`:>>`** and **`redefines`** are synonymous.

SysML v2

```
package ex1_pkg {
import AADL::**;
import ScalarValues::Integer;
part def ex1 specializes System;
part def 'ex1.i'
specializes ex1, SystemImplementation {
  part hw: Hw {
    redefines Scheduling_Protocol =
    Rate_Monotonic_Protocol;
  }
  part sw: 'sw.i';
  allocation sw_to_hw:
    Actual_Processor_Binding
    allocate sw to hw;
}
part def Hw specializes Processor;
part def sw specializes Process {

  in port i1: IntegerPort;
  out port o1: IntegerPort;
}
part def 'sw.i'
specializes sw, ProcessImplementation {
  part th1: t {
    :>> Dispatch_Protocol = Periodic;
    :>> Period = 50 [ms];
    :>> Deadline = 50 [ms];
    :>> Compute_Execution_Time =
    2 [ms] .. 2 [ms];
  }
  part th2: t {
    :>> Dispatch_Protocol = Periodic;
    :>> Period = 50 [ms];
    :>> Deadline = 50 [ms];
    :>> Compute_Execution_Time =
    2 [ms] .. 2 [ms];
  }

  connection c1 connect i1 to th1.i1;
  connection c2 connect th1.o1 to th2.i1 {
    redefines Timing = Immediate;
  }
  connection c3 connect (th2.o1, o1);
}
part def t specializes Thread {

  in port i1: IntegerPort;
  out port o1: IntegerPort;
}
}
```

3 SysML v2 Domain library for AADL

This section presents the AADL domain library that allows to build AADL models in SysML v2. It defines the AADL package containing all the SysML defined AADL elements representing the AADL SAE Standard.

3.1 SysML v2 definition of AADL Components

AADL components are defined as SysML part definitions that specialize Component part definitions. AADL features (ports, access, ...) are defined as SysML port definitions that specialize Feature port definitions.

```
private abstract port def Feature;
abstract part def SubComponent;

abstract part def Component;
abstract part def ComponentType
  :> Component {
  port features: Feature[0..*]
    :> portsOnPart;
}
abstract part def ComponentImplementation
  :> ComponentType {
    part subcomponents: SubComponent[0..*]
    :> subparts;
}
```

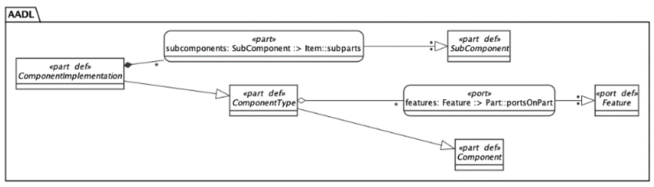


Figure 3. Graphical notation for AADL components

As an example, here are the AADL process type and process implementation SysML part definitions in the library.

```
abstract port def ProcessFeature
  :> Feature;
abstract part def ProcessSubComponent
  :> SubComponent;

part def Process
  :> ComponentType,
     ProcessorBindable,
     SystemSubComponent,
     AbstractSubComponent {
  port :>> features: ProcessFeature[0..*];
}

part def ProcessImplementation
  :> Process,
     ComponentImplementation {
  part :>>
  subcomponents: ProcessSubComponent[0..*];
}
```

`ProcessSubComponent` is an abstract part definition used as a category to constrain sub-components to those allowed for a `Process`. Consequently, `Data`, `Subprogram`, `SubprogramGroup`, `Thread`, `ThreadGroup` and `Abstract` part definitions all specialize `ProcessSubComponent`.

`ProcessFeature` is an abstract port definition used as a category to constrain features to those allowed for a `Process`. Consequently, `Port`, `DataAccess`, `SubProgramAccess` all specialize `ProcessFeature`.

To constrain subcomponent categories in an AADL component implementation, each SysML component category part definition specializes the SysML subcomponent abstract part definition of each of its possible containers (shown in Figure 4).

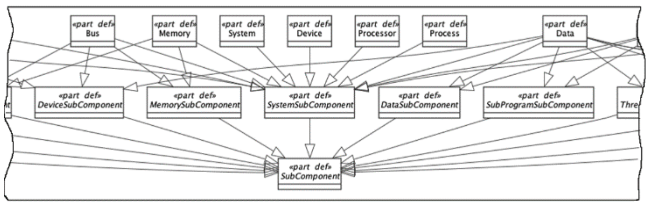

«part def» Bus
«part def» Memory
«part def» System
«part def» Device
«part def» Processor
«part def» Process
«part def» Data
«part def» DeviceSubComponent
«part def» MemorySubComponent
«part def» SystemSubComponent
«part def» DataSubComponent
«part def» SubProgramSubComponent
«part def» SubComponent

Figure 4. The diagram of the hierarchy for the sub-components

This allows fine control over which category of subcomponent can be included into another, provided that the used tool supports the SysML type system. An alternate solution could be to use SysML constraint elements.

3.2 SysML v2 definition of AADL Features

Interactions between AADL components are defined by connections between their features. AADL features are described using SysML port definitions used to type port usages in component part usages and connected using SysML connections. Figure 5 presents the specialization hierarchy for the AADL features in SysML v2.

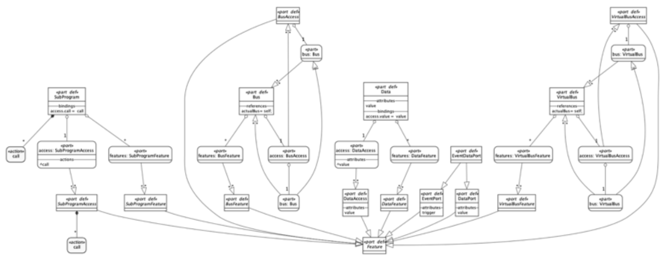


Figure 5. The graphical notation for the features

3.3 SysML v2 definition of AADL Hw/Sw Binding

The deployment properties defined in AADL express the way software components are deployed onto the run time hardware (i.e., Processors, Busses, Memories and Devices). They can be described as an allocation of a logical component to a physical one in SysML. The SysML allocation definition of the `Actual_Processor_Binding` AADL property is defined as follows:

```
abstract part def AbstractProcessor;
abstract part def ProcessorBindable;
allocation def Actual_Processor_Binding {
    end logical : ProcessorBindable;
    end physical : AbstractProcessor;
}
```

3.4 SysML v2 definition of AADL Properties

AADL defines several sets of standard properties applicable to restricted categories of components. To offer a similar capability in SysML v2, we have defined a series of abstract parts that declare a group of consistent properties that the component type part definition can specialize if needed.

As an example, the `Schedulable` SysML part definition declares aliases to standard AADL properties applicable for components interacting with a run-time scheduler (e.g., Threads and Devices).

```
abstract part def Schedulable {

    abstract attribute
      Period: DurationValue[0..1];
    attribute Deadline: DurationValue;
    attribute Priority: Integer;

    abstract attribute
      Compute_Execution_Time:
        DurationInterval;

    abstract attribute
      Dispatch_Protocol:
        Supported_Dispatch_Protocols;
    …
}
```

All properties defined by the AADL specifications need to be categorized by abstract parts to allow the concerned parts to specialize them. Only a few AADL properties have been defined in the SysML v2 Domain Library for now. This will need to be discussed and completed in the future.

4 Next steps

Several aspects of the AADL language have not been addressed at all yet. This is the case for Modes, Flows, Prototypes and Call Sequences.

For the AADL constructs that have been considered in this paper, more experiments and interactions with other stakeholders will be needed to consolidate the approach.

Regarding the Behavior and Error Annex standard sub-languages**,** it is likely that some existing SysML v2 features could be reused, such as the state elements.

A simple mapping such as the one proposed below could be used as a first step to embed existing AADL Behavior annexes inside SysML v2 models without introducing any syntactic change.

```
part def ALU :> Thread {
  …
  state behavior : Behavior {
  language "AADL::Behavior_Specification"
  /*
   * states s : initial complete final
state;
   * transitions
   *   t1 : s -[on dispatch e1]-> s
   *     { o := i1 + i2 };
   *   t2 : s -[on dispatch e2]-> s
   *     { o := i1 - i2 };
   */
  }
}
```

5 Conclusion

This paper describes the current state of a study conducted internally at Ellidiss. Our goal is twofold:

- Quickly implement a bidirectional SysML v2 to AADL transformation prototype within AADL Inspector, to evaluate the feasibility of reusing the same existing analysis tools (static, timing, safety, security) for both textual representations.
- Contribute to any collaborative action within the AADL committee, or a wider community, to specify a SysML v2 annex for AADL v2 and reactivate the discussions about AADL v3 if needed.

The first outcomes of this preliminary study are promising but the work will need to be updated according to the final definition of the SysML v2 language that was not available when this paper was written, and the feedback of the AADL and SysML communities regarding the relevance of the approach.

References

[1] Architecture Analysis & Design Language (AADL) - https://www.sae.org/standards/content/as5506c/.

[2] OMG System Modeling Language™ (SysML®) v2 - https://github.com/Systems-Modeling/SysML-v2-Release.

[3] Used AADL tools: https://www.ellidiss.fr/

[4] SysML v2 pilot: https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/releases

[5] LAMP: A new model processing language for AADL, P. Dissaux, ERTS 2020.

Unified Graphical Co-modelling, Analysis and Verification of Cyber-physical Systems by Combining AADL and Simulink/Stateflow

Xiong Xu, Shuling Wang, Bohua Zhan, Xiangyu Jin, Naijun Zhan
SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China; email: {xux, wangsl, bzhan, jinxy, znj}@ios.ac.cn

Jean-Pierre Talpin
Inria, Rennes, France; email: jean-pierre.talpin@inria.fr

Abstract

The design of safety-critical cyber-physical systems (CPSs) involve several dimensions, including physics, hardware rchitecture and software functionality. It is desirable to design CPSs by taking these issues into account uniformly and yet, few existing design workflows support this aim. For instance, AADL is an architecture-centric modelling formalism for CPSs, which focuses on modelling architecture and prototyping real-time hardware platforms, but it delegates physical and software behavioral models to so-called annexes. By contrast, Simulink/Stateflow (S/S) focuses on modelling interacting physical and software behaviors, but does not render the non-functional characteristics of their hardware platforms. To address this issue, in [1], we proposed the combination of AADL and S/S, called AADL⊕S/S, to co-model CPSs and presented a method to uniformly analyse and verify them. AADL⊕S/S provides a unified graphical co-modelling environment for CPS design and supports simulation through C code generation. Also, [1] presented a formal semantics of AADL⊕S/S by translation to Hybrid Communicating Sequential Processes (HCSP), yielding a deductive verification framework of the combined models using Hybrid Hoare Logic (HHL). Additionally, [1] proved the correctness of the translation of AADL⊕S/S to HCSP.

Keywords: Simulink/Stateflow, AADL, HCSP, formal semantics, simulation and verification.

1 Introduction

Cyber-physical systems (CPSs) tightly couple hardware and software to sense and actuate a physical environment. To correctly model them, it is paramount to take the three perspectives of their software functionalities, physical environment, hardware platform and system architecture into account, uniformly, Figure 1.

Unfortunately, according to the commonly accepted design principle of "separation of concerns", most of existing design methodologies and workflows do not support all three

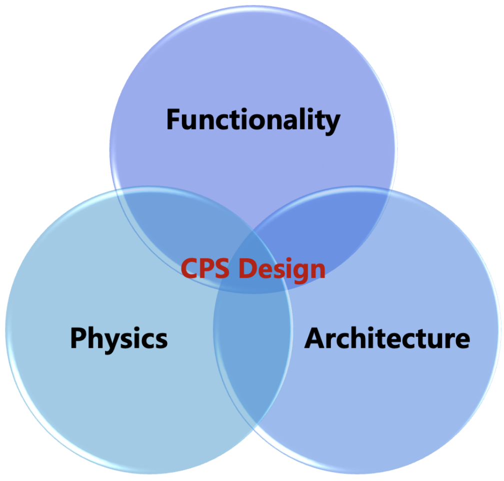


Figure 1: The three perspectives of CPS design

design aspects uniformly. For example, AADL [2, 3] features strong capabilities for describing the architecture of a system due to the pragmatic (and practice-inspired) effectiveness of combining software and hardware component models. However, the core of AADL only supports modelling of embedded system hardware and abstraction of its relevant discrete behavior, and does not support the description of the continuous physical processes to be controlled and its combination with software. By contrast, Simulink/Stateflow (S/S) [4, 5], the de-facto industry standard for model-based analysis and design of embedded systems, is best-suited for modelling and analysing continuous physical processes, discrete computations and their combination. However, S/S cannot naturally model system architectures and hardware platforms.

To address the above issue, we presented a combination of AADL and S/S [1], named AADL⊕S/S, that provides a unified graphical modelling formalism to represent all three perspectives of CPS design. An overview of AADL⊕S/S is given in Figure 2. Using AADL⊕S/S, a CPS is modelled with the following three layers:

Architecture layer The system architecture and its hardware

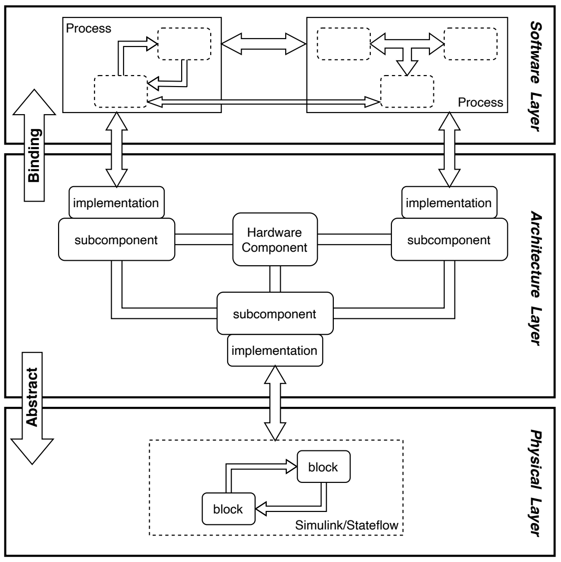


Figure 2: An overview of AADL⊕S/S

platform are described by AADL components that define the structure, type and characteristics of composed hardware and software components.

Software layer The software behavior can be modelled either through AADL behavioral annexes or S/S diagrams.

Physical layer The physics of the cyber-physical system and its interaction with the hardware/software platform are modelled by S/S diagrams.

Paper Organization. This paper is an extended abstract of a talk presented at ADEPT 2022, based on earlier works reported in [1]. The rest of this paper is organized as follows: Section 2 presents an overview of AADL⊕S/S. Section 3 briefly reviews the co-simulation of AADL⊕S/S. Section 4 introduces the formal analysis for AADL⊕S/S. Section 5 introduces a toolchain that supports the design of CPSs with AADL⊕S/S. Finally, we conclude in Section 6.

2 Overview of AADL⊕S/S

In the combined framework, AADL is used to define the overall architecture of the system, including connections between the software, hardware, and physical components. The software components define the discrete behavior of the system, either as behavior annex within AADL, or S/S diagrams. The physical components define the continuous plants of the system as S/S diagrams.

The architecture layer, described as AADL system composite components, specifies the types of components, and (part of) their implementation (an abstraction of their actual implementation), as well as their composition. It usually consists of a central processor unit classifier with several subcomponent devices (like sensor, controller, and actuator etc). Each of these classifiers has its own type and implementation. For software functionality and physical processes, the architecture layer usually needs their abstractions, i.e., the type classifiers of these software and physical components. The type classifier of a component declares the set of input and output ports, specifies the contract of its behavior, that are accessible from outside. By contrast, the implementation classifier of a component binds its type classifier with a concrete implementation in the software and physical layers.

3 Co-simulation of AADL⊕S/S

3.1 AADL⊕S/S to C

In order to simulate AADL⊕S/S models, we presented a way to translate AADL⊕S/S to C code [1, 6], allowing co-simulation of the combined models. It relies on the Real Time Workshop (RTW) toolbox of Matlab, which permits code generation from S/S diagrams. The C code generation is divided into three parts:

1. for the AADL part, we implement AADL2C translator to generate C code following the execution semantics of AADL;
2. for the S/S part, we use the existing code generation facility in Matlab, to produce C code that can simulate this part of the model step-by-step;
3. for the architecture part, we implement a library in C that includes thread scheduling protocols, interaction between components and combination of AADL and S/S.

To realise co-simulation, the three parts are integrated together to form an executable C code that simulates the combined model. The translation of the combined model amounts to coordinating code generated from AADL and S/S through port communications specified in the architecture layer. The result of the simulation is then displayed visually for analysis.

3.2 AADL⊕S/S to HCSP

In [1], we proposed an approach that translates AADL⊕S/S to HCSP [7, 8]. In order to test the correctness of the translation from AADL⊕S/S into HCSP, we also implemented a simulator for HCSP with a graphical user interface. Additionally, this allows us to quickly obtain the result of running an HCSP process, in order to check that its behavior is as expected.

While there are non-deterministic elements in HCSP, they are not used often. In particular, the result of translation described in this paper is essentially deterministic. Our aim in the simulator is to compute an execution path of the process and visualise it in a graphical interface. The computation follows closely the small-step operational semantics of HCSP.

The simulator is implemented in Python. In addition to real numbers, the state of the system may contain strings and lists. Operations on lists as stack, queue, or priority queue are supported. Solving of ODEs is done using Python's `scipy` package (function `solve_ivp`), which is also able to accurately calculate the time at which the boundary of the domain is reached using a root-finding algorithm. Finally, the simulator is linked to a web interface which is able to show the HCSP process in pretty-printed form, the steps of execution, and a plot of the variables in the process against time. This allows us to not only view the result of running an HCSP process, but also find out what went wrong if the process does not execute as expected.

4 Formal verification

However, guaranteeing the reliability of a safety-critical CPS developed using AADL⊕S/S remains challenging, as simulation-based techniques are inherently incomplete, and therefore cannot ensure reliability of safety-critical CPS rigorously. To address this problem, we further developed an HCSP-based deductive verification approach for AADL⊕S/S, including

- A formal semantics of AADL in terms of transition systems, including thread dispatch, scheduling, execution, and bus connections with latency.
- A translation from graphical AADL⊕S/S models to HCSP [9]. HCSP is an extension of CSP with ordinary differential equations (ODEs) for modelling hybrid systems. It contains flexible constructs, including communications, continuous evolution, interrupt, and it is also extendable, to fully express behaviors of CPSs.
- The correctness of the translation is proved by building a weak bisimulation relation between the transition semantics of the source and target models. The translated HCSP model can be formally verified using HHL and its proof assistant [10, 11, 12] and the results are preserved by the original AADL⊕S/S model.

5 The MARS toolchain

In order to provide tool support for AADL⊕S/S, we developed a toolchain, named MARS, for modelling, analysis, verification and code generation of CPSs. The toolchain is implemented upon the previous version [9, 13], by combining the AADL part.

The overall structure of MARS is shown in Figure 3. In MARS, modelling a CPS can be given graphically with AADL⊕S/S [1], or formally with HCSP. Both AADL and S/S support the analysis of their models through simulation, but they cannot be used for AADL⊕S/S models directly. In [6], the co-simulation of AADL⊕S/S models is implemented by translating both AADL and S/S to C code and then defining their integration for simulation execution. The simulation of formal HCSP models is supported by an HCSP simulator (Section 3.2). The HCSP simulator executes a given HCSP process according to its semantics and visualises the execution path, including the values of variables, communication status etc, at given times in a graphical interface.

To complement incomplete simulation, in MARS, AADL⊕S/S models can be transformed into HCSP processes for further verification. The translation from AADL⊕S/S to HCSP and the correctness of the translation have been considered in [1, 14]. The verification of HCSP is then conducted by HHL Prover, which has been implemented in Isabelle/HOL. The HHL was first presented in [10, 11]. It is compositional such that an HCSP process can be decomposed into smaller components to be verified. It is able to handle synchronized communications, parallel compositions, and so on, but on the other hand, it needs more proof effort in the proof assistant.

The verified HCSP is then transformed automatically into final implementation of CPSs. The transformation requires to discretize the continuous evolutions of HCSP and realise the channel communications between parallel components. The code generation to SystemC, as well as its correctness proof, was presented in [15].

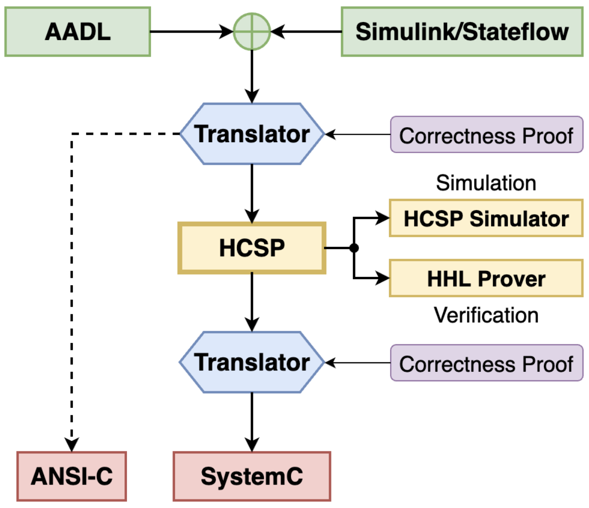


Figure 3: The overall architecture of MARS

6 Conclusion

We presented AADL⊕S/S, a combination of AADL and S/S, and developed a simulation tool for it. Moreover, to verify AADL⊕S/S models, we defined an operational semantics and an HCSP-based denotational semantics, and proved that there exists a weak bisimulation between the transition system of any AADL⊕S/S model and the transition system of the translated HCSP process. This makes all AADL⊕S/S models can be verified with HHL Prover. In addition, we also developed a simulator for HCSP for testing the correctness of the translation by comparing the simulation results before and after translating, and providing the possibility that one can design a CPS starting with HCSP. The details of our approach can be found in [1, 6].

There are some limitations to our approach. First, AADL provides a plenty of components and functions, while we only consider its core functionalities, which limits the practicality of our framework for case-studying realistic CPSs. Second, at present, our verifier only scales to small HCSP models, as means of model abstraction, verification automation and modular verification would be needed to improve.

Acknowledgement

This work is partly supported by National Key R&D Program of China under grant No. 2022YFA1005101 and 2022YFA1005100, NSFC under grant No. 62192732, and CAS Project for Young Scientists in Basic Research, Grant No. YSBR-040.

References

[1] X. Xu, S. Wang, B. Zhan, X. Jin, J.-P. Talpin, and N. Zhan, “Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow,” *Theoretical Computer Science*, vol. 903, pp. 1–25, 2022.

[2] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[3] SAE International Standards, "Architecture analysis & design language (AADL), Revision C," 2017.

[4] MathWorks Inc., *Simulink User's Guide*, 2013. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.

[5] MathWorks Inc., *Stateflow User's Guide*, 2013. http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf.

[6] H. Zhan, Q. Lin, S. Wang, J.-P. Talpin, X. Xu, and N. Zhan, "Unified graphical co-modelling of cyber-physical systems using AADL and Simulink/Stateflow," in *UTP*, vol. 11885 of *LNCS*, pp. 109–129, 2019.

[7] J. He, "From CSP to hybrid systems," in *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pp. 171–189, Prentice Hall International (UK) Ltd., 1994.

[8] C. Zhou, J. Wang, and A. P. Ravn, "A formal description of hybrid systems," in *Hybrid Systems*, vol. 1066 of *LNCS*, pp. 511–530, 1996.

[9] N. Zhan, S. Wang, and H. Zhao, *Formal Verification of Simulink/Stateflow Diagrams: A Deductive Approach*. Springer, 2017.

[10] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou, "A calculus for hybrid CSP," in *APLAS*, pp. 1–15, 2010.

[11] S. Wang, N. Zhan, and L. Zou, "An improved HHL prover: an interactive theorem prover for hybrid systems," in *ICFEM*, vol. 9407 of *LNCS*, pp. 382–399, Springer, 2015.

[12] L. Zou, N. Zhan, S. Wang, M. Fränzle, and S. Qin, "Verifying Simulink diagrams via a hybrid Hoare logic prover," in *EMSOFT*, pp. 1–9, IEEE, 2013.

[13] M. Chen, X. Han, T. Tang, S. Wang, M. Yang, N. Zhan, H. Zhao, and L. Zou, "MARS: A toolchain for modelling, analysis and verification of hybrid systems," in *Provably Correct Systems*, pp. 39–58, Springer, 2017.

[14] X. Xu, B. Zhan, S. Wang, J.-P. Talpin, and N. Zhan, "Semantics foundation for cyber-physical systems using higher-order UTP," *ACM Trans. Softw. Eng. Methodol.*, 2023. https://doi.org/10.1145/3517192.

[15] G. Yan, L. Jiao, S. Wang, L. Wang, and N. Zhan, "Automatically generating SystemC code from HCSP formal models," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 4:1–4:39, 2020.

C2AADL_Reverse: A Model-Driven Reverse Engineering Approach for Development and Verification of Safety-Critical Software

Zhibin Yang, Zhikai Qiu, Yong Zhou, Zhiqiu Huang
School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China; email: yangzhibin168@163.com, {2427153594, zhouyong, zqhuang}@nuaa.edu.cn

Jean-Paul Bodeveix, Mamoun Filali
IRIT-Université de Toulouse, Toulouse, France; email: {bodeveix, filali}@irit.fr

Abstract

The safety-critical system communities have been struggling to manage and maintain their legacy software systems because upgrading such systems has been a complex challenge. To overcome or reduce this problem, reverse engineering has been increasingly used in safety-critical systems. This paper proposes `C2AADL_Reverse`, a model-driven reverse engineering approach for safety-critical software development and verification. `C2AADL_Reverse` takes multi-task C source code as input, and generates AADL (Architecture Analysis and Design Language) model of the legacy software systems. Compared with the existing works, this paper considers more reversed construction including AADL component structure, behavior, and multi-threaded run-time information. Moreover, two types of activities are proposed to ensure the correctness of `C2AADL_Reverse`. First, it is necessary to validate the reverse engineering process. Second, the generated AADL models should conform to desired critical properties. We propose the verification of the reverse-engineered AADL model by using UPPAAL to establish component-level properties and the Assume Guarantee REasoning Environment (AGREE) to perform compositional verification of the architecture. This combination of verification tools allows us to iteratively explore design and verification of detailed behavioral models, and to scale formal analysis to large models. In addition, the prototype tool and the evaluation of `C2AADL_Reverse` using a real-world aerospace case study are presented.

Keywords: Safety-Critical Software, Model-Driven Reverse Engineering, AADL, Compositional Verification.

1 Introduction

Safety-critical systems (SCS) are the systems whose failure could result in loss of life, substantial economic loss, or damage to the environment [1]. There are many well-known examples in different domains such as aircraft flight control, space missions, and nuclear systems. The SCS communities have been struggling to manage and maintain their legacy software systems because upgrading such systems has been a complex challenge. As surveyed by FAA (Federal Aviation Administration), reverse engineering (RE) has been increasingly used in many industries, including aircraft applications [2].

In contrast with forward engineering, reverse engineering can be defined as the process of examining an already implemented software system to create a higher abstraction level representation in a different form. Reverse engineers typically start with a low-level representation of a system (such as source code, or execution traces), and try to build more abstract representations from these (such as architectural models, or use cases, respectively) [3]. The main objective of RE is to provide a better understanding of the software system's current state, which can be used to correct (e.g. fix bugs), update (e.g. alignment with updated user requirements), upgrade (e.g. add new capabilities), or even completely re-engineer the system under study [4].

Generally, reverse engineering a software system is a time-consuming and error-prone process. It's difficult to predict how much time RE will require and there are no standards to evaluate the quality of the result of RE [4]. To overcome these difficulties, model driven reverse engineering (MDRE) [4,5,6] has been proposed to enhance the traditional reverse engineering processes. MDRE is the application of model driven engineering (MDE) principles and techniques to RE in order to generate relevant model-based views on legacy systems, thus facilitating their understanding and manipulation.

There have been several past works on MDRE which can be classified as two categories: *specific* and *general* solutions. This is determined depending on whether they aim to reverse engineer the system from a single technology and/or with a predefined scenario in mind (e.g., a concrete kind of analysis), or to be the basis for any other type of manipulation in later steps of the reverse engineering process [7]. Manev et al. [8] propose a tool, called ITACG (IoT software Analysis and Code-Generation tool), for performing reverse engineering and extraction. This is accomplished by scanning the source code of the target system and extracting architectural information from it, which is stored into a UML model. Umair Sabir et al. [9] present a MDRE framework named Src2MoF to generate UML structural and behavioral diagrams from the Java source code. In order to address several kinds of scenarios relying on different legacy technologies, Hugo Bruneliere et al. [7] give an extensible and generic model driven reverse engineering: MoDisco. MoDisco has three layered architecture i.e. infrastructure, technologies and use case layers. It defines a basic meta-model approach for MDRE based on Knowledge Discovery Meta-model (KDM) specification to provide support for XML, JSP and Java. MoDisco only deals with structural aspects and does not support the MDRE for behavioral aspects from source artifacts.

Most of the existing works of MDRE mainly consider general domains such as desktop or business applications. In this paper, we consider MDRE in the domain of complex embedded systems, especially the safety-critical systems. Complex embedded software systems are typically special-purpose systems developed for control of a physical process with the help of sensors and actuators. They are often the systems requiring a deep combination of software, runtime operational system and hardware platform. Typical non-functional analysis of the requirements in this domain, such as safety, schedulability, and so on, needs the modeling of architecture, functional behaviors and runtime. These characteristics already make it apparent that complex embedded systems differ from desktop and business applications. Compared with the modeling languages used in the existing works of MDRE such as UML, AADL (Architecture Analysis and Design Language) [10] is a powerful modeling language for complex embedded system, which provides a unified formalism for the modeling of architecture, functional behaviors, and runtime.

This paper proposes `C2AADL_Reverse`, a MDRE approach for safety-critical software development and verification. `C2AADL_Reverse` takes multi-task C source code as input, and generates AADL model of the legacy software systems. Moreover, when MDRE exists in the domain of safety-critical systems, validation of the MDRE process and verification of the resulted models are highly desirable because such software systems have to undergo development regulations and certification restrictions. Therefore, the reverse-engineered AADL components become the basis for applying MDD development in the same application domain, and should be analyzed and verified.

2 Research problems

Currently, there are several researches on AADL automatic code generation (i.e. forward engineering). For instance, OCARINA [11] and RAMSES [12] support automatic code generation from AADL to C, Ada and Java. Regarding the reverse generation of AADL models, Wang et al. [13] propose an approach for extracting AADL models from existing embedded software in order to reduce maintenance costs. In an effort to bridge the semantic and syntactic gaps between the two languages, they have defined a set of mapping rules from C to AADL models. For Integrated Modular Avionics (IMA) systems, Lesovoy et al. [14] present an approach to extract the AADL models from source code of ARINC 653-compatible application software. They apply the ideas of counterexample and path feasibility check to the task of extracting the architectural information from source code. As mentioned before, safety-critical software often run on various embedded platforms, reverse engineering needs to deal with the information such as static structure, dynamic run-time, and functional behavior. However, the existing approaches mainly deal with structural aspects instead of behavioral and run-time aspects of source artifacts. Safety-critical software systems are large and intricate, often constituting hundreds of components. Thus, the challenge is to be able to derive the information about the functional behaviors and the runtime dynamics of a system. In particular, as multi-core processors are widely used in safety-critical software [15], the reverse engineering of multi-task synchronization, mutex, communication, and task scheduling has become an important problem.

Moreover, how to evaluate or measure a MDRE effort? On the one hand, we can use the generated model of MDRE to produce another version of the original software and make the comparison between the two versions to validate the MDRE process. On the other hand, automatic formal verification techniques such as model-checking can be used to analyze the behaviours of the generated model. Since the increasingly size of the source code, formal verification of reverse-engineered AADL models often faces the so-called state-explosion problem. An approach to deal with the state-explosion problem is the use of compositional verification [16, 17, 18] which leverages the structure of the system. The basic idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components.

To overcome the above-mentioned research problems, we have implemented a complete framework for the proposed approach, that is `C2AADL_Reverse`, as shown in Fig.1. It includes five phases, (1) analysis of the original source code, (2) extraction of an intermediate model, (3) generation of an AADL model, (4) validation of the C2AADL process, and (5) formal verification of the generated AADL model. In our case, the original source code is structured, i.e., it conforms with the coding/programming rules in aerospace industry, such as strict development patterns with clear separation of communications, data types, components types, etc. Compared with the existing AADL RE method, this paper considers more reverse constructions including AADL component structure, behavior, and multi-threaded run-time information. For the validation of the reverse process, we generate a second version of the original software and compare the two versions of code. Moreover, we propose the verification of the generated AADL model by using UPPAAL to establish component-level properties and the Assume Guarantee REasoning Environment (AGREE) [19, 20] to perform the compositional verification of the architecture.

3 Main contributions

The main contributions of the paper can be summarized as follows:

- *A new MDRE approach named C2AADL_Reverse*: The transformation from multi-task C source code to AADL is divided into three parts:

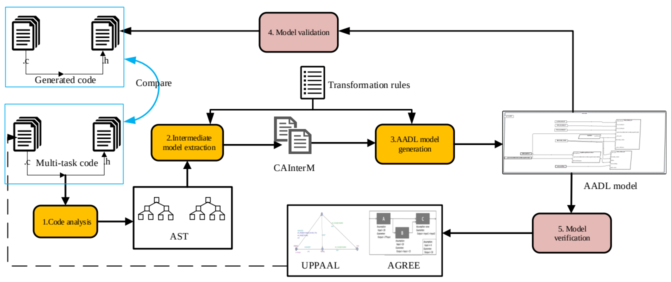


Figure 1: The framework of C2AADL_Reverse

- Structural aspect: the transformation from global variables, local variables, data types, function definitions and multi-task structures to AADL components;
- Behavioral aspect: the transformation from function and task execution behavior to AADL behavior annex [21], which involve various types of branch statements, assignment statements, and function call statements;
- Run-time aspect: the transformation from multi-task communication, multi-task synchronization and mutex, and task scheduling to AADL execution-model properties.

- *Validation and verification approach of C2AADL_Reverse*: Two types of activities are proposed to ensure the correctness of `C2AADL_Reverse`. First, it is necessary to validate the reverse engineering process. Second, the generated AADL models should conform to desired critical properties. We propose the verification of the generated AADL model by using the model checker UPPAAL to establish component-level properties and the AGREE environment to perform the compositional verification of the architecture. This combination of verification tools allows us to iteratively explore design and verification of detailed behavioral models, and to scale formal analysis to large models.
- *The prototype tool*: The `C2AADL_Reverse` prototype tool adopts a modular architecture, which is implemented based on the AADL open source environment OSATE [22], in which an intermediate model is proposed to facilitate the transformation from C source code to AADL.
- *Case study*: A real-world aerospace industrial case, the rocket launch control subsystem, is used to show the feasibility of the method presented in the paper.

4 Conclusion and future work

This paper has presented a model-driven reverse engineering approach for safety-critical software development and verification, namely `C2AADL_Reverse`. Compared with the existing works, `C2AADL_Reverse` considers more reversed construction including AADL component structure, behavior, and multi-threaded run-time information. Moreover, when MDRE exists in the domain of safety-critical systems, validation of the MDRE process and verification of the resulted models are highly desirable because such software systems have to undergo development regulations and certification restrictions. We use reverse reverse engineering to validate the reverse engineering process, and verify the generated AADL models by using the model checker UPPAAL to establish component-level properties and the AGREE environment to perform the compositional verification of the architecture. To the best of our knowledge, this paper presents a first effort on the validation and verification of the reverse process from C to AADL. Finally, the effectiveness of `C2AADL_Reverse` is demonstrated using a real-world aerospace case study.

We will further carry out the following future work:

- The source code cannot explicitly express non-functional properties of the software system (such as period, execution time, resource consumption, and so on). At present, we apply third-party dynamic tools (such as WCET analysis tools) to measure timing properties and add them to the corresponding AADL model.
- Inspired by the restricted natural language approach proposed in our previous work [23], the automatic transformation from natural language requirements into AGREE contracts is currently being developed. As well, the translation from AGREE contracts to TCTL properties in UPPAAL will be also automated.
- We are considering the extension of AGREE to support for modeling components that execute asynchronously (or quasi-synchronously), and formalizing the reasoning rules in the theorem prover Coq [24].

References

[1] N. G. Leveson, *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016.

[2] M. D. George Romanski, "Reverse engineering for software and digital systems," tech. rep., 2016.

[3] A. van Deursen and E. Burd, "Software reverse engineering," *Journal of Systems and Software*, vol. 77, no. 3, pp. 209 – 211, 2005. Software reverse engineering.

[4] S. Rugaber and K. Stirewalt, "Model-driven reverse engineering," *IEEE software*, vol. 21, no. 4, pp. 45–53, 2004.

[5] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14516–14542, 2017.

[6] H. Bruneliere, *Generic Model-based Approaches for Software Reverse Engineering and Comprehension*. PhD thesis, Nantes, 2018.

[7] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[8] D. Manev and A. Dimov, "Facilitation of IoT software maintenance via code analysis and generation," in *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*, pp. 1–6, IEEE, 2017.

[9] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, and A. Amjad, "A model driven reverse engineering framework for generating high level UML models from java source code," *IEEE Access*, vol. 7, pp. 158931–158950, 2019.

[10] SAE, "Architecture Analysis & Design Language (AADL), AS5506C," 2017.

[11] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the ocarina aadl tool suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 1–25, 2008.

[12] S. Rahmoun, E. Borde, and L. Pautet, "Multi-objectives refinement of AADL models for the synthesis embedded systems (mu-RAMSES)," in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 21–30, IEEE, 2015.

[13] G. Wang, X. Zhou, Y. Dong, and H. Zhao, "Studying on AADL-based architecture abstraction of embedded software," in *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, pp. 14–19, IEEE, 2009.

[14] S. L. Lesovoy, "Extracting architectural information from source code of ARINC 653-compatible application software using CEGAR-based approach," *Trudy ISP RAN/Proc*, vol. 30, no. 3, 2018.

[15] S. M. Salman, A. V. Papadopoulos, S. Mubeen, and T. Nolte, "A systematic methodology to migrate complex real-time software systems to multi-core platforms," *Journal of Systems Architecture*, vol. 117, p. 102087, 2021.

[16] E. Posse and J. Dingel, "Contract-based specification and analysis of aadl models," in *ACVI 2014–Architecture Centric Virtual Integration Workshop Proceedings*, p. 4, Citeseer, 2014.

[17] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 64–79, Springer, 2008.

[18] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods Symposium*, pp. 126–140, Springer, 2012.

[19] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. Heimdahl, and L. Wagner, "Proof-based coverage metrics for formal verification," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 194–199, IEEE, 2017.

[20] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *International Conference on Computer Aided Verification*, pp. 20–27, Springer, 2018.

[21] SAE, "Architecture Analysis and Design Language (AADL) Annex D: Behavior Model Annex," 2017.

[22] "OSATE: Plug-ins for front-end processing of AADL models," tech. rep., The Software Engineering Institute, 2013.

[23] F. Wang, Z. Yang, Z. Huang, C. Liu, Y. Zhou, J. Bodeveix, and M. Filali, "An approach to generate the traceability between restricted natural language requirements and AADL models," *IEEE Trans. Reliab.*, vol. 69, no. 1, pp. 154–173, 2020.

[24] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.

COMPASTA: Integrating COMPASS Functionality into TASTE

A. Bombardelli, A. Bonizzi, M. Bozzano, R. Cavada, A. Cimatti, A. Griggio, M. Nazaria, E. Nicolodi, S. Tonetta, G. Zampedri

Fondazione Bruno Kessler, Via Sommarive 18, 38123 Trento, Italy; email: bozzano@fbk.eu

Abstract

TASTE is a tool chain dedicated to the design and implementation of embedded, real-time systems, developed under the initiative of the European Space Agency (ESA). It consists of various tools, which support model-based design of embedded systems, automatic code generation, deployment and simulation. TASTE is based on several specification languages, in particular it uses AADL for the architectural design, whereas the behavior of SW components can be specified in SDL and other languages.

TASTE currently lacks a comprehensive support for performing early verification and assessment of the design models. COMPASTA is an ESA study that aims at filling this gap, by integrating into TASTE the formal verification functionality of COMPASS, a tool for model-based HW-SW co-Engineering developed in a series of ESA studies. COMPASTA extends TASTE by providing the possibility to model the behavior of HW components using SLIM, a dialect of AADL supported by COMPASS. Moreover, it offers capabilities such as library-based specification of HW faults, automatic fault injection, contract-based design, functional verification and safety assessment, fault detection and identification analysis.

Keywords: AADL, SDL, TASTE, COMPASS.

1 Introduction

TASTE [1,2] is a model-based software engineering design environment dedicated to embedded, real-time systems, which has been actively developed by ESA since 2008. Specifications are written in different languages, such as AADL [3] (for the architectural specification) and SDL [4] (for the behavioral specification). TASTE includes various other tools, such as editors, viewers, and code generators. TASTE has been adopted as a glue technology and for system deployment in several projects, see e.g. [5, 6, 7].

COMPASS [8,9,10] is a tool for System-SW Co-Engineering developed in a series of ESA studies from 2008 to 2016. Specifications are written in SLIM, a dialect of AADL. COMPASS supports model-based verification techniques, based on model checking, such as requirements analysis, contract-based analysis, fault specification, functional verification, safety and dependability assessment, fault detection and identification analysis. COMPASS is based on the ocra [11], nuXmv [12] and xSAP [13] verification back-ends.

COMPASTA is an ESA study (2021-2022) that aims at integrating the formal verification functionalities of COMPASS [8, 9, 10] into TASTE [1, 2]. COMPASTA extends TASTE by supporting model-based specification of both SW and HW components, fault injection, and a full set of formal analyses, based on model checking. The goal of the analyses is to formally validate the system model, before the system is deployed to the target HW. Thus, COMPASTA makes TASTE a comprehensive and coherent end-to-end tool chain, that covers system design and development SW implementation, deployment and testing.

2 The COMPASTA Workflow Exemplified

COMPASTA extends the TASTE workflow by providing additional functionalities which are complementary with respect to the ones available in TASTE. TASTE is a tool for model-based SW engineering, focusing on SW design, deployment and implementation. COMPASTA, on the other hand, extends TASTE by providing the possibility to model HW components and their faults, to perform fault injection, and to carry out several formal analyses (e.g., requirements validation, contract-based design, functional verification, safety and dependability assessment) on the complete formal model (including both HW and SW). The goal of COMPASTA is to enable early validation of the design model, before the SW is implemented and deployed to the target HW.

We illustrate the COMPASTA workflow in a simple running example, shown in Fig. 1, modeling a redundant power system.

The example consists of HW components (batteries, generators, sensors, and switches) and SW components (the FDIR components). Generators provide power to batteries, which in turn provide power to sensors. In case of a fault of a generator or a battery, the lines connecting generators, batteries and sensors can be reconfigured. For instance, in case of a fault of one battery, the remaining battery can be used to power both sensors. FDIR components perform a re-configuration by sending a command to the corresponding switch component.

The system is modeled using the graphical user interface of TASTE. Fig. 1 shows the Interface View (architecture) of the system, i.e. the blocks corresponding to the components, and the connections (provided and required interfaces) between

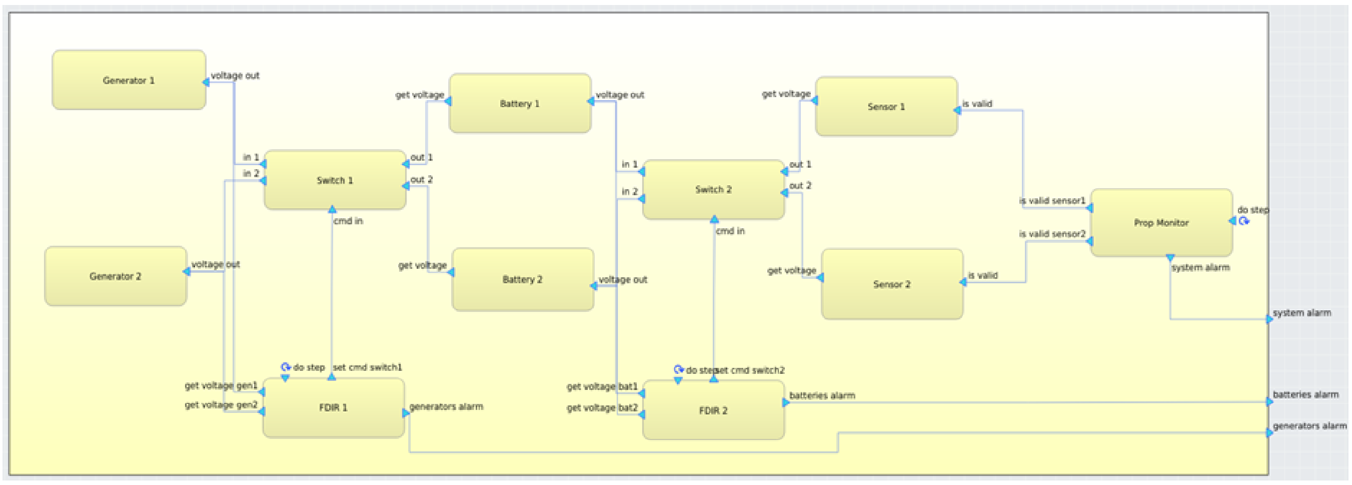


Figure 1: A power system example.

the components. TASTE uses AADL to generate an internal representation of the Interface View.

SW components (FDIRs in our example) can be modeled using the SDL language. For instance, Fig 2 shows an excerpt of the code for FDIR_2. It periodically reads the input voltages of the two batteries and, in case the output voltage of either of them is under a given threshold, it sends a command to the Switch_2 component to change from primary mode to a secondary mode.

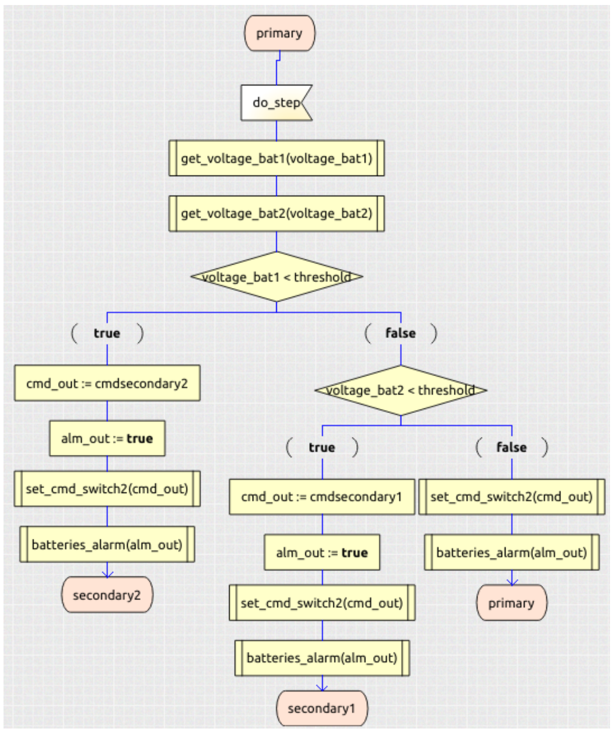


Figure 2: Sample SDL code for FDIR_1.

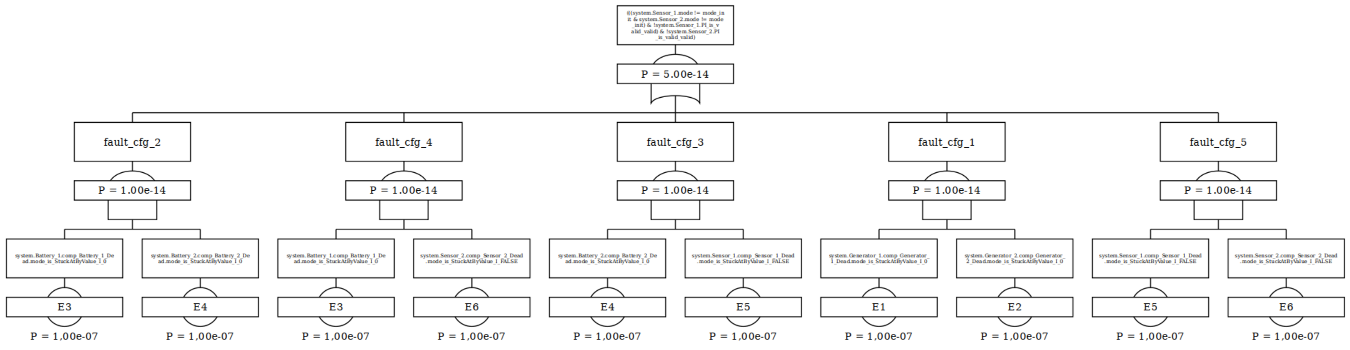
HW components can be modeled in SLIM, a dialect of AADL, which extends AADL by providing the possibility, among other things, to specify behavioral models in the form of state machines, and to specify faults and fault injections. We show below some sample code for the Battery_1 component. The given transition causes the output voltage of the battery to decrease by 1, when the input voltage is below 10.

```
system implementation Battery_1.others
-- BATTERY SUBCOMPONENTS
subcomponents
  -- DELAY FOR TIMESTEPS
  delay: data clock;
-- BATTERY STATES
states
   init :  initial  state;
  base: state while (delay <= 1);
-- BATTERY STATE TRANSITIONS
transitions
  -- INIT
   init  -[
     then voltage_out.voltage :=  12
  ]-> base;
  -- BATTERY DISCHARGES BY 1V
  base -[
     when delay >= 1
      and get_voltage.voltage < 10
      and voltage_out.voltage >= 1
     then delay := 0;
           voltage_out.voltage := voltage_out.voltage − 1
  ]-> base;
 [...]
end Battery_1.others;
```

The SDL and SLIM models are translated by COMPASTA into the language supported by the verification back-ends, which are run to carry out the formal analyses. The translation performed by COMPASTA is based on the definition of the semantics of the input languages (based on the standards [3, 4] and on the COMPASS semantics for SLIM), and of the semantics of the communication between HW and SW (defined in COMPASTA, and compatible with the TASTE communication semantics).

Fault definitions can be picked from a library, and automatically injected into the system model, e.g., a fault injection can model a permanent "stuck-at-zero" fault of the "voltage_out" signal of a battery. This is specified via the following fault injection specification:

```
system implementation Battery_1.others
properties
```


(((system.Sensor_1.mode != mode_init & system.Sensor_2.mode != mode_init) & !system.Sensor_1.PI_is_valid_valid) & !system.Sensor_2.PI_is_valid_valid)
P = 5.00e-14
fault_cfg_2
fault_cfg_4
fault_cfg_3
fault_cfg_1
fault_cfg_5
P = 1.00e-14
system.Battery_1.comp_Battery_1_Dead.mode_is_StuckAtByValue_I_0
system.Battery_2.comp_Battery_2_Dead.mode_is_StuckAtByValue_I_0
system.Sensor_2.comp_Sensor_2_Dead.mode_is_StuckAtByValue_I_FALSE
system.Sensor_1.comp_Sensor_1_Dead.mode_is_StuckAtByValue_I_FALSE
system.Generator_1.comp_Generator_1_Dead.mode_is_StuckAtByValue_I_0
system.Generator_2.comp_Generator_2_Dead.mode_is_StuckAtByValue_I_0
E3
E4
E3
E6
E4
E5
E1
E2
E5
E6
P = 1,00e-07

t

Figure 3: An example Fault Tree.

```
    FaultInjections  => (
      [
        Description => "Dead";
        Fault_Model => "StuckAtByValue_I";
        Fault_Dynamics => "Permanent";
         Probability  => "1.e−7";
        DataInput => "voltage_out.voltage";
        DataVarout => "voltage_out.voltage";
        DataTerm => "0";
      ]
    );
end Battery_1.others;
```

Model checking can be used to verify functional properties. For instance, the following property (specified using a property pattern from COMPASS):

> "Globally, it is always the case that sensor1.valid and sensor2.valid holds"

states that the outputs of both sensors are always valid (which is the case when the sensors are powered and they are not failed. COMPASTA can automatically generate and display artifacts such as traces (e.g., a counterexample trace, when a property is violated). Moreover, COMPASTA can automatically generate artifacts such as Fault Trees. Fig. 3 shows an example Fault Tree for the property corresponding to the outputs of both sensors being invalid.

When the formal model has been validated, the standard TASTE workflow can be used for the implementation of the SW components. To this aim, first HW components must be replaced with corresponding interface components, that represents the SW layer realizing the communication between SW and HW. Then, the deployment of the SW components (binding of the SW to the target HW platform(s)) is specified. Finally, TASTE can then be used to generate the executable code for the target platform(s) and to test and simulate the final implementation.

3 Conclusions and Future Work

COMPASTA is an ESA-funded study whose goal is to extend the TASTE toolset with formal verification and assessment functionality, creating a comprehensive and coherent tool chain that covers architectural and functional design, system-level safety assessment, and deployment of the embedded software. In the proposed workflow, system, safety, and software engineers share the same models and use them in an iterative process, supported by various analyses that increase the confidence in the internal and external consistency of the system, and the overall quality of the final product.

The integration is based on a view where the COMPASS back-ends are split from the COMPASS front-end and integrated in other model-based design environments such as TASTE. On the same lines, ocra, nuXmv, and xSAP have been integrated into CHESS for a SysML-based design [14], while FBK is working on the integration of such back-ends into CAMEO and on the prototype support for SysML-V2.

Acknowledgments

This work was funded by ESA-ESTEC under Contract No. 4000133700/21/NL/GLC/kk.

References

[1] J. Hugues, L. Pautet, B. Zalila, P. Dissaux, and M. Perrotin, "Using AADL to build critical real-time systems: Experiments in the IST-ASSERT project," in *Proc. ERTS*, 2008.

[2] "TASTE web page." `https://taste.tools/`.

[3] SAE, "Architecture Analysis & Design Language (AADL)," 2022. SAE document AS5506D.

[4] International Telecommunication Union, "ITU-T Z.100. Specification and Description Language – Overview of SDL-2010," 2021.

[5] "ADE: Autonomous Decision Making in Very Long Traverses."

[6] "MOSAR: Modular Spacecraft Assembly and Reconfiguration."

[7] R. Cavada and A. Cimatti and L. Crema, and M. Roccabruna and S. Tonetta, "Model-Based Design of an Energy-System Embedded Controller Using Taste," in *Proc. FM 2016*, vol. 9995 of *LNCS*, pp. 741–747, 2016.

[8] M. Bozzano, H. Bruintjes, A. Cimatti, J.-P. Katoen, T. Noll, and S. Tonetta, "COMPASS 3.0," in *Proc. TACAS 2019*, 2019.

[9] M. Bozzano, A. Cimatti, J.-P. Katoen, P. Katsaros, K. Mokos, V. Nguyen, T. Noll, B. Postma, and M. Roveri, "Spacecraft Early Design Validation using Formal Methods," *Reliability Engineering & System Safety*, vol. 132, pp. 20–35, 2014.

[10] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, and M. Roveri, “Safety, Dependability and Performance Analysis of Extended AADL Models,” *Computer Journal*, vol. 54, no. 5, pp. 754–775, 2011.

[11] “ocra web page.” `https://ocra.fbk.eu`.

[12] “nuXmv web page.” `https://nuxmv.fbk.eu`.

[13] “xSAP web page.” `https://xsap.fbk.eu`.

[14] A. Debiasi, F. Ihirwe, P. Pierini, S. Mazzini, and S. Tonetta, “Model-based Analysis Support for Dependable Complex Systems in CHESS,” in *MODELSWARD*, pp. 262–269, SCITEPRESS, 2021.

Experiences Modeling a OPC UA / DDS Gateway in AADL in the Context of Fog Computing

P. Denzler, D.Ramsauer, D.Scheuchenstuhl, W.Kastner
Institute of Computer Engineering, TU Wien, Vienna; email:
{patrick.denzler}, {daniel.ramsauer},{daniel.scheuchenstuhl},{wolfgang.kastner}@tuwien.ac.at

Abstract

The legacy protocols still used in industrial automation are an obstacle to interoperability. In the meantime, while newer protocols are slowly replacing gateways, they can provide a bridge between new and legacy protocols. The Architecture Analysis & Design Language (AADL) was used in the Fog Computing for Robotics and Industrial Automation (FORA) project to model a Fog Computing Platform (FCP). Part of the FCP is an OPC Unified Architecture (OPC UA) / Data Distribution Service (DDS) gateway. The main goal was to develop an AADL model that allows the creation of platform-specific instances of such a gateway and the creation of other gateways. While the gateway model is incomplete, it formed the basis for several gateway prototypes. Challenges included complex data structures and tooling issues related to code generation. Nevertheless, the experiences with the AADL modeling of the gateway and the FCP were positive overall.

1 Summary of the Talk

The presentation was structured as an experience report and gave an overview of the following topics and results. As the research was conducted as part of the European Union's Horizon 2020 research and innovation programme (FORA—Fog Computing for Robotics and Industrial Automation [1]), the introduction included some essential information about the project and its members. Another part of the opening was focused on current interoperability issues with legacy systems within industrial automation.

A fog computing platform (FCP) is one of the outcomes of FORA [1,2]. During the development, the parts of the FCP were modelled in AADL as a means for documentation and evaluation. One part of the FCP was concerned with an OPC Unified Architecture (OPC UA) / Data Distribution Service (DDS) gateway as a possible solution to ease legacy protocol issues. Both middlewares are the new defacto standard in industrial automation. The main goal was to develop an AADL model that allows the creation of platform-specific instances of such a gateway and the creation of other gateways.

The talk provided insights into the various AADL models focusing on the OPC UA / DDS gateway model within this context. In the foreground of the report were the experiences made during the modelling activities. Some of the encountered challenges were:

- Difficulties with modelling dynamic complex data types required in the gateway.
- Problems encountered with the code generation tools (parsers) as part of the AADL tool environment.
- The lack of documentation and examples.

Some of the positive results were:

- All gateway parts could be modelled in AADL in full conformity to the OMG specification.
- A reduced version of the AADL model was implemented based on the model and used in other projects [3,4].

In summary, the experiences were positive and were published in more detail in Denzler et al. [5].

References

[1] P. Pop, B. Zarrin, M. Barzegaran, S. Schulte, S. Punnekkat, J. Ruh, and W. Steiner, "The FORA fog computing platform for industrial IoT," *Information Systems*, vol. 98, p. 101727, 2021.

[2] P. Denzler, J. Ruh, M. Kadar, C. Avasalcai, and W. Kastner, "Towards Consolidating Industrial Use Cases on a Common Fog Computing Platform," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 172–179, 2020.

[3] P. Denzler, D. Ramsauer, and W. Kastner, "Tunnelling and Mirroring Operational Technology Data with IP-based Middlewares," in *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, vol. 1, pp. 1205–1210, 2021.

[4] P. Denzler, D. Ramsauer, T. Preindl, and W. Kastner, "Communication and container reconfiguration for cyber-physical production systems," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, 2021.

[5] P. Denzler, D. Scheuchenstuhl, D. Ramsauer, and W. Kastner, "Modelling protocol gateways for cyber-physical systems using Architecture Analysis & Design Language," *Procedia CIRP*, vol. 104, pp. 1339–1344, 2021. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0.

Modelling Robot Architectures with AADL

Gianluca Bardaro, Matteo Matteucci
Politecnico di Milano, Piazza Leonardo Da Vinci 32, Milano (IT); email: {name.surname}@polimi.it

Abstract

Robots are complex systems composed of many interacting subsystems, each coordinating a multitude of hardware and software components. In the last twenty years, robotic frameworks helped accelerate the development process by providing a reference structure and publicly available software components. However, current practices are not sustainable for modern robotic systems. In this work, we present a modelling approach based on the Architecture Analysis and Design Language (AADL) to define robotic systems and enhance software development practices for robotics. Additionally, we exploit the model to perform automatic code generation, which reduces development time and guarantees a more reliable and robust implementation.

Keywords: AADL, robotics, automatic code generation.

1 Introduction

Robots have become increasingly popular in recent years, with a wide range of applications in fields such as manufacturing, healthcare, and retail. This increase in popularity is also thanks to the introduction and consolidation of many frameworks for software robot development. In particular, the Robot Operating System (ROS) [1] is now the de facto standard for robot software in academia, and it is becoming more popular in industry since the introduction of ROS 2 [2].

Nonetheless, the process of developing and programming a robot is complex and time-consuming. Robots are composed of many interacting subsystems, and each of them requires the expertise of a specific domain expert. This complexity makes the development of software for a robot a task that cannot be tackled by a single expert but requires cooperation and significant integration effort.

Robotic software is well-suited for a model-based design approach, especially when combined with automatic code generation [3]. Given its nature as a system of systems, robot software is easily decomposed into components and supports a decentralised development approach. This approach allows domain experts to focus on implementing specific functionality without having to worry about the overall structure of the software. Additionally, the use of a model allows for the identification of potential problems and errors at the design stage, as the complete architectural view is available from the hardware configuration to the high-level software implementation. This can be particularly useful for robots, which have complex processing pipelines and can experience errors that propagate through multiple components before causing disruptions.

In this work, we present an approach based on the Architecture Analysis and Design Language (AADL) [4] to model robotic architectures and a toolchain to automatically generate boilerplate code and deployment configurations. The work is structured as follows. In section 2, we provide an overview of our abstraction to generalise robot components. Next, in Section 3, we show how AADL can be used to model robot components following our abstraction. In Section 4, an overview of our automatic code generation toolchain is presented. Section 5 summarises a practical use case of our approach. Finally, in Section 6, we draw relevant conclusions.

2 Component and connector paradigm

In robotics, the most popular frameworks are based on a component-connector paradigm, and while different approaches implement it in different ways, the underlying structure is the same. In ROS, it is the computation graph, a peer-to-peer network of processes managing and exchanging data. Here, following the terminology of graphs, the components are called nodes, while asynchronous topics or synchronous services represent the connections. In both cases, communication happens by exchanging messages.

The popularity of the component-connector paradigm is not coincidental. In their structure, robots are a system of systems, a hierarchical collection of components interconnected to create a working apparatus. Physically, a robot is a collection of sensors and actuators, and the same goes for the behaviour where simple low-level independent functionalities are not enough to implement even the simplest robot. Given all these needs, the most natural approach is to decompose the system into different and simpler subsystems and to simplify and characterise their interactions by the use of interfaces. The result is a component-connector paradigm.

In an effort to provide a general and flexible representation that can be used to compose robotic architectures, we identified four recurring design patterns that we called component behaviours. Each one is characterised by how data is received and processed. These component behaviours are composable and can be assembled to create complex components.

Source. A component expresses a source behaviour when it is a generator of data or events. In ROS, a node implementing a publisher that generates messages has a source behaviour. This type of behaviour is used, for example, for device drivers since they create and circulate a digital version of the analogue input they detect.

Sink. A sink is a component that consumes data or events. In ROS, a node is a sink when it implements a subscriber that receives and consumes messages. A component that controls actuators implements this type of behaviour since it receives commands from other components and consumes them to operate a physical device.

Filter. The most common behaviour for a component is the filter. This type of component receives messages or events as input and processes or relays them to create an output. The component does not store the data received since they are processed and directly re-circulated in the system. This approach is common when doing simple conversions or when it is necessary to resample the data. In ROS, it is implemented by processing the received message directly in the subscriber callback and publishing it before leaving the callback environment.

Reactive. A component has a reactive behaviour when its functionalities are synchronously triggered by a message or an event, and it is usually implemented by using a remote function call. In ROS, this kind of behaviour is exemplified by services. They offer a public interface that can be called by external components and react with synchronous execution of a function that may return a value.

3 Using AADL

AADL is the perfect candidate to describe architectures based on the component-connector paradigm because one of the main design principles of the language is the interaction of different components through connections. AADL components, at any level (e. g., system, process, device, subprogram, etc.), support some form of feature (i. e., ports and accesses) to communicate and interact with other components [5].

Following how AADL models are structured, we use the AADL system to encapsulate the entire architecture and, if necessary, to define any specific subsystem inside the robot. Going one step further in the structure of a robotic architecture, we encounter components or nodes, as defined in ROS. With the exception of some niche implementations, ROS nodes are all independent processes. Using this approach simplifies the deployment on multiple machines. Given this, the most straightforward candidate to model ROS nodes is the process category.

3.1 Component behaviours

To model component behaviours, we use threads. They represent an execution path through code, and their behaviour is periodic with various characteristics or triggered by an external input. Moreover, multiple threads can coexist in the same process and execute in parallel. All these characteristics make the thread a suitable category to model component behaviours. However, as we described in the previous section, each behaviour is related to a specific external interaction. We achieve this by combining threads and ports.

First, we exploit AADL inheritance, and we define a generic component behaviour that only includes data access to access the internal state of the component. Additionally, this generic component behaviour includes one subcomponent: a subprogram that will contain the implementation as a property. Specific component behaviours are then created by using inheritance and specialised ports. In particular, the sink behaviour is characterised by a single inbound event data port. Its counterpart is the source, which exposes an outbound data port. The source behaviour also specifies through a property that the thread is executed periodically. The two combined in a single thread represent the filter behaviour, with one inbound and one outbound port. Lastly, the reactive behaviour is obtained using a subprogram access that triggers the execution of the functionality specified as the property of the subprogram subcomponent.

Again using inheritance, these component behaviours are extended into building blocks for ROS nodes. Ports are refined to target a specific data component, modelling the fact that communication in ROS is strongly typed. Moreover, ports are added to target ROS-specific functionalities and APIs.

3.2 ROS architectural elements

When describing models closer to the implementation level, it is necessary to take into account all the elements that compose a robotic architecture. First of all, physical devices and hardware. For elements like processors or memories, it is possible to bind a software component (e.g., processes or data) to its physical counterpart (e.g., processor and memory) to specify the hardware implementation of the system. In ROS, this feature of AADL can be used to model distributed architectures by binding components to different physical platforms, and this specifies where each node will be executed at runtime. Devices are connected to processes using ports or accesses.

Sensors and actuators are an integral part of a robotic architecture, and to model them, it is possible to use AADL devices. A device represents an interface between the physical world and the architecture, and it can be modelled as a simple interface or include the inner functionalities and characteristics of the physical component. Devices can connect to processes using ports or accesses. When modelling a ROS architecture, physical devices and software components communicate using the same interface. This creates the issue of differentiating between a topic-based connection and other types of connections. To solve this problem, we exploit AADL physical and virtual buses. A virtual bus can be used to model abstract communication channels, like ROS topics, while a bus can be used for physical connections, like Ethernet and USB.

One of the most important features of ROS and the main reason for its popularity is the large library of already implemented and readily available packages and nodes. When creating a modelling approach for ROS, it is essential to include the possibility of modelling existing nodes, and we achieve it by exploiting the dual representation provided by AADL of component type and implementation. Existing ROS packages are modelled directly as AADL packages, while existing nodes are modelled using the component type only. We provide an interface that appears and behave in the same way as the already existing component, but we do not detail in any way the internal functioning.

Lastly, a key feature of ROS is the package managing all the different reference frames and transformations, known as *tf*.

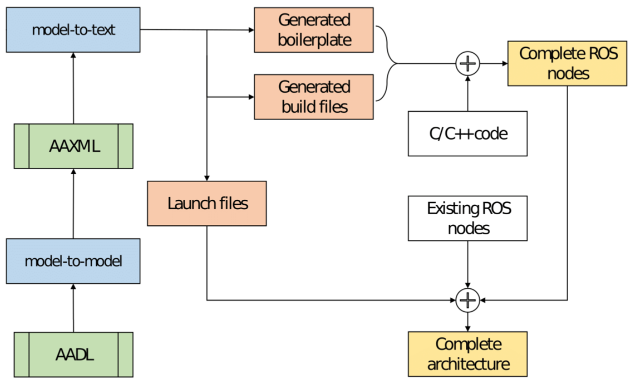


Figure 1: Overview of the automatic code generation toolchain.

Differently from other ROS features, the access to *tf* does not go through any established communication channel (i. e., topics or services). It is a centralised resource where all the coordinate frames of the robot and their evolution in time are stored, and it is possible to read or update the content of this shared resource by using specialised APIs. Given this description, the most suitable way to model *tf* is to use a single data component at system level that all the nodes can access through data accesses when necessary.

4 Automatic code generation

In our toolchain [6], we adopted a two-steps approach, first a model-to-model transformation that converts the input AADL model to an intermediate XML-based representation, then a model-to-text transformation to automatically generate ROS-compatible C++ code.

Figure 1 summarises the complete process. The automatic code generation requires as input a model defined in AADL, completed by a data description, and specialised via properties to include functionality-specific source code. When all these conditions are met, the process provides as an output a collection of automatically generated and compilation-ready ROS nodes, their associated communication files (i.e., messages, service and action files) and the necessary launch files to run the architecture. A fully complete model creates an architecture that only needs to be compiled and run.

The model-to-model transformation going from AADL to XML is the first step of the code generation approach and preserves the original structure of the model when it is converted to an XML-based representation. This transformation is achieved using a custom backend for Ocarina [7]. The output of this process, combined with data models describing the configuration of the components, is the input of the next step of the automatic code generation toolchain.

First, the automatic programming system creates the source code for the new custom nodes in C++. Since C++ is a compiled language, the system will automatically generate all the necessary files to build the node executables. If the model contains all the necessary information (i.e., source code of the functionalities), the final output of the automatic programming process will be ready to compile with no intervention required. The automatically generated code will be placed in the correct package structure expected by ROS, together with any custom message, service or action file.

To organise the generated nodes into an architecture, it is necessary to create launch files. The topology of the architecture can be automatically extracted from the model and converted into launch files, and the parametrisation defined using a data modelling language can be converted in the YAML description used by ROS. Moreover, in launch files, existing nodes are included in the architecture and connected to the rest of the system.

5 Use case

In this section, we present a test use case [8] where a model-based approach has been used to replicate and reimplement an existing architecture developed with traditional techniques. We started from an already implemented and fully functional system to show that it is possible to achieve the same level of functionalities as the original application by combining a model-based design with automatic programming. The target robot is an electric wheelchair modified to be controlled with a computer and equipped with various sensors to achieve levels of autonomy and teleoperation. The wheelchair used as the starting platform is a commercial model (Twist T4 2x2) produced by Degonda Rehab SA. It is suitable for both indoor and outdoor usage, and it has high manoeuvrability thanks to the two-wheeled dynamics. The conversion from a traditional electric wheelchair to an autonomous robotic platform is achieved by installing encoders on the wheels to provide odometry information and two Sick TiM 561 laser scanner distance sensors, which are used for mapping, localisation, and obstacle avoidance. The robotic wheelchair supports three modes of operation: manual, controlled with the on-board joystick or through a radio controller, assisted driving, controlled by the user but obstacle-aware, and fully autonomous, the user specifies a destination on the map.

Figure 2 shows an overview of the complete architecture of the robotic wheelchair using AADL graphical representation. This overview is already a significant advantage with respect to traditional development in ROS since this kind of representation of the computation graph is available only at runtime when connections between nodes are established.

From this architectural overview, it is possible to see many of the elements mentioned in the previous sections. The entire architecture is encapsulated in a system component, and more systems are used to aggregate nodes with similar functionalities. For example, we separate the teleoperation subsystem in the top left corner from the autonomous navigation in the bottom right corner. This division is also useful when automatically generating launch files. Moreover, differently than the runtime computation graph, this overview also includes hardware components such as sensors, actuators, and communication channels.

The figure also highlights the three categories of components we have to manage during automatic code generation. Already existing nodes, defined only as interfaces, are not generated and included directly in launch files. Custom nodes are fully modelled, and therefore, it is possible to process

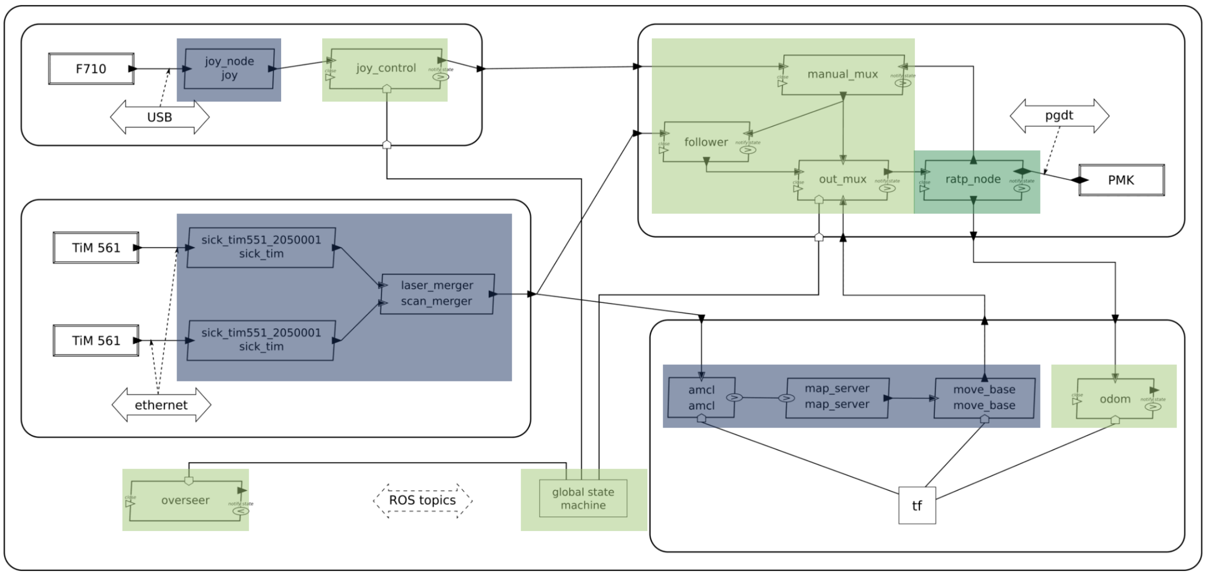


Figure 2: Graphical representation of the robotic wheelchair's architecture. In blue the existing ROS nodes, in light green the custom nodes automatically generated, and in dark green the partially generated *rapt_node*.

them with the code generation toolchain to generate a ready-to-compile component. Finally, special custom nodes are managed by the toolchain but then require direct intervention by the component developer to finalise their implementation.

6 Conclusions

In this work, we presented a complete solution to model robotic architectures. Our approach span from the description of a generic component-based robotic system to the refinement into a model of a ROS architecture and, finally, automatic code generation and deployment.

With AADL, it is possible to detail both hardware and software components, a feature that is impactful in robotic systems. Additionally, by exploiting the language's built-in inheritance and extensibility, we can minimise the amount of code both the designer and the developer need to write. Thanks to the definition of component behaviours, the system designer can easily assemble the model of complex architectures. While the component developer can work in well-defined boundaries and avoid the burden of writing boilerplate code.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] G. Bardaro, A. Semprebon, and M. Matteucci, "Aadl for robotics: a general approach for system architecture modeling and code generation," in *IRC 2017-IEEE International Conference on Robotic Computing*, 2017.

[4] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[5] G. Bardaro and M. Matteucci, "Using aadl to model and develop ros-based robotic application," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, pp. 204–207, IEEE, 2017.

[6] G. Bardaro, A. Semprebon, A. Chiatti, and M. Matteucci, "From models to software through automatic transformations: An aadl to ros end-to-end toolchain," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 580–585, IEEE, 2019.

[7] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications," in *International Conference on Reliable Software Technologies*, pp. 237–250, Springer, 2009.

[8] G. Bardaro, A. Semprebon, and M. Matteucci, "A use case in model-based robot development using aadl and ros," in *Proceedings of the 1st International Workshop on Robotics Software Engineering*, pp. 9–16, 2018.

Modeling ROS Based Applications with AADL

E. Senn, L. W. J. Bourdon
Lab-STICC, Université de Bretagne Sud, Lorient, France; email: {eric.senn, lucie.bourdon}@univ-ubs.fr

Abstract

This paper presents a library of AADL models for ROS based applications. AADL models are provided for every ROS services, namely nodes, and for complete robots hardware and embedded computer boards. The model of the complete system describes the binding of software onto hardware components, and allows for CPU and bus load analysis. To this purpose, dedicated AADL properties are included into the models, that come from actual measurements onto the hardware platforms.

1 Introduction

Developing robot software is a difficult task. Beside being complex, robots are often critical systems, when people safety is at stake. The work that we present here is directly motivated by the need we had in common while developing such software, and which led us to formal modeling. Formal modeling has been successfully used for years to ease the design of complex, and critical, systems. This approach has multiple purposes, and we are here particularly interested in checking the correctness of the specification and guaranteeing performances. One of the performance that is directly related to safety is the reaction time, e.g. the time needed for a robot to brake then to stop when a human being steps into his way. As a matter of fact, the reaction time of a robot is greatly impacted by the load of its processing units. We indeed observe that when every service in the robotic application meets its deadline, and that means, is not slowed down by an overloaded CPU, or communication bus, the robot react as expected : its reaction time is in the fixed tolerances. Hence, the language that we need to support formal verification from our models must allow to describe, beside the application itself, the hardware on which it is executed, and the deployment of every software components on the different parts of the hardware. In addition, the models of the complete robotic system must include dedicated properties and mechanisms to check those two features as mentioned above : CPU and bus load. The *Architecture Analysis and Design Language* (AADL) [1] is particularly well fitted to those needs. It is supported by a set of tools, among which OSATE2, that allows to write models for the software, the hardware, and the binding of software to hardware components. From there, the models rightness is checked, and performances analysis is performed. Both textual description and graphical representation are defined in AADL.

The Robot Operating System, ROS, has been designed to ease the writing of robot software. Now widely used by developers, it provides a set of libraries and tools for building, debugging, and running code possibly across multiple computers. To model our ROS based applications with AADL, we need a library of ROS components carrying dedicated properties for further performance analysis. Those properties, intended to predict CPU and bus load on different hardware targets, come from a set of different measurements on actual embedded computer boards. Those measurements have been presented in [2] for CPU load, and [3] for bus load analysis. This paper focuses on the building of the library, on the architecture of the models for every ROS component, and presents how to use it to describe a complete robotic system.

2 State of the art

Different works have recently studied the use of AADL for checking robotic applications. Latencies are analyzed in [4], but regardless of CPU or bus load, which actually impact a robot reaction time. Beside, the approach is not dedicated to ROS based application. The authors in [5] concentrates on ROS components, but have to develop their own modeling language. They propose a modeling of tasks chains to evaluate response times, but tasks are only assigned to one CPU core and communication needs are note checked against bus capacities. The benefit from modeling and developing ROS-based robotic application with AADL has been presented in [6] where an automatic generation of ROS code from the model [7] is proposed. The AADL model is only build for the software and hardware performances are not considered here. In [8], the author tackled, as we do, AADL modeling, hardware profiling, and deployment analysis at the same time. His approach for measuring compute execution time is however far more complex since it involves modification and rebuilding of the code. This is something we definitely do not want for our own code, and even less for entire ROS on-the-shelf packages as we use in our applications. Beside being time consuming, this is also an intrusive approach whose impact on performances should be evaluated. Nodes are also considered independently, whereas we observe a modification of performance when they are connected to others. Hence the necessity to profile a node in its proper usage context. Moreover, MIPS budget properties are not determined which prevent for checking MIPS demand at the process level.

3 Modeling ROS with AADL

ROS might be defined as different things. It is often referred to as a set tools and libraries for obtaining, building, writing, and running robot code across one or several computers. As such, it provides implementation of commonly-used functionality, for compilation, debugging, feedback and control over the running software, together with management of the many software packages that may be found in a ROS distribution.

ROS is also commonly defined as a middleware that provides hardware abstraction, low-level device control, and above all a structured communications layer on top of the host operating system(s) of a computer (a heterogeneous computer cluster). Indeed, a typical robotic application involves several services : some are deployed on the robot with its embedded computer board, or boards. Other might be deployed on a remote computer for control purpose. In ROS, a service is implemented as a *node*. Nodes communicate through virtual channels called *topics*. A node may publish on a topic, subscribe to a topic, or do the both at the same time.

To begin with, the model of a ROS based application must reflect this set of nodes with their interconnections. Hence we provide a set of packages with models for different robotic services, aka nodes in ROS. Since we want to analyze performances from our model, and because those performances actually depend on the way nodes are deployed on the computer cluster that supports the robot software, we need models of the different computer boards in this cluster and their connections. In many cases, only one, or two, single computer boards, specifically dedicated to ROS nodes, are used in a robot. In those models of the actual hardware, specific AADL *properties* are used in conjunction with analysis tools to predict future performances of the complete system.

Our library is organized as follows.

- A central *ros* package : our central ROS package contains the declaration of every component type and associated implementation that will be used in our library of ROS nodes. Indeed, according to the AADL modeling style, inputs and outputs of a any component are declared in the component definition, and the internal structure is declared in its implementation.
- A collection of packages for different ROS nodes : one AADL package is written for each node that we want to include in our library. A node package includes the node declaration as an AADL *process*, defining its inputs and outputs, and its platform independant implementation where its internal structure is defined. In this implementation, every thread in the node is declared as an AADL *subcomponents*, together with its connections to other threads and to the node's input and outputs.

  A node package also includes several platform dependent implementations for every thread in the node. Those implementations carries AADL *properties* definition related to the thread performances measured on different embedded computer boards. For one computer board, properties might differ depending on the actual CPU cores on which the thread is running, begetting as much different implementations as needed.
- A package for ROS messages : This package will contains the declaration of types and associated implementations for every kind of messages that the nodes in our library could exchange.
- A set of packages for modeling the hardware : here we will find the packages for different robots, as well as for single computer boards and the system on chips or multiprocessor CPU they carry.

We feel compeled to issue a warning here : our AADL library does not include a model for all nodes that can be found in a given ROS distribution. In order to obtain such an exhaustive library, tenth of nodes should be modeled, when we only use a part of them in our applications. Our approach is incremental: we add a new AADL component to our library every time we use a new node. It is the same for new hardware parts or single board computers.

3.1 Modeling ROS nodes

A robot software based on ROS is a constellation of nodes communicating through topics. A dedicated package is written for every node that we want to include in our library. Every node in those packages will inherit from the high-level node component type and implementation defined in the central *ros* package. A ROS node is modeled as an AADL *process* component which includes several threads, mimicking the actual ROS node code. The main threads that are found running in a ROS node are :

- The main thread in every node.
- A thread type to implement subscribers in nodes. A subscriber thread is associated to a callback function that is called to deal with the data received on the topic listened to. In fact, the callback function is placed in the FIFO callback queue of the node.
- The spinner thread keeps taking the callbacks from the callback queue and executing them one by one in an infinite loop acting as a "spinning" thread. One ROS node has one spinner thread by default. However, several spinners could be spawned to allow for multiple concurrent executions of callback functions [9].
- A thread type to implement callback functions in nodes (from [7]).
- A thread type to implement publishers in nodes.
- A thread to provide a ROS *service* to a requesting node.
- A timer thread to implement a ROS *timer* in nodes.
- A thread type to implement a broadcaster of transformation of frames (TF) in nodes.

A tf node will publish on the /tf topic with many others in general in a robot application. Unlike in [7], where /tf is considered an independent bus to which different nodes require an access, we consider it a regular topic which will be bound to the ROS virtual bus.

Like for AADL processes modeling ROS nodes, an AADL thread component has one declaration defining its inputs and outputs, with an associated implementation defining its inner structure. As many implementations as needed will inherit from this last one, to reflect the performances of the thread on different hardware targets and CPU cores. The AADL *extends* mechanism is used there. New AADL *properties* values will be declared with a new set of performances for a thread. Those properties, namely *MIPSBudget* and *Compute_exectuion_time*, are directly issued from the measurements presented in [2] and [3].

3.2 Modeling ROS communications

Communications between nodes in ROS are seen as happening on a virtual bus called *TCPROS*. A publisher node warns, through XML/RPC, the master node that it is ready to send messages on a topic. This topic is then registered by the master and advertised in the ROS middleware. A subscriber node warns the master that it wants to listen to this topic. If the topic exists, the master gives the subscriber the address of the publisher. Then the publisher and subscriber directly talk and establish a TCP (or UDP) communication between the two of them. Unlike [8], we finally use a regular bus to model the ROS virtual bus because the bandwidth capacity property does not apply to AADL virtual bus. Our measurements show that the virtual bus speed is impacted by the hardware on which the communicating nodes are running [3]. Several implementation of the bus are thus provided. For instance, the Odroid XU4 computer board begets two implementation of the ROS bus model: one for A15 cores and the other for A7 cores. Speed differs when ROS communications use wired Ethernet or WiFi and associated AADL components are provided in the library.

4 Modeling the hardware

A robot commonly gathers many electronic components: sensors, actuators, interfaces, interconnections, and one or several single computer boards. The purpose of our models here is not to provide a detailed view of every parts inside a robot. It is rather to provide a simple view, including only the minimum set of components needed for performance analysis. Our hardware models might appear almost simplistic then. For instance, the AADL model of a multi-core SoC would only contains the CPU cores inside it, but would discard any interconnection between them or the memory. However, it would carry dedicated AADL properties to be used by the performance analysis tools. In the AADL model of the Exynos 5422 SoC from Samsung which is a heterogeneous 8 cores chip (4 A15 and 4 A7 cores), we chose to gather big and little cores in two respective clusters. Indeed, it might be interesting to bound a thread to a cluster rather than to unique core, which is necessary when the thread MIPS demands exceeds the core capacity, and afterwards to compute the load of the whole cluster, while adding the load contribution of any thread also bound to the cluster. However, the analysis of resource budget allocation is only performed by OSATE when a thread is bound to a processor. and the *Actual_Processor_Binding* property of a thread can only target ONE processor.

5 Performance analysis

The underlying principle in model based design and AADL is to have a view of the software, with software components (processes, threads ...) connected together, a view of the hardware, including standard hardware components (processors, buses, memories ...), and a model describing the deployment, *binding* in AADL, of software onto hardware components. Instantiation of this *binding* model produces an AADL *instance* of the system from which analysis tools are called. Three different analysis tools from OSATE2 are used.

Analyze Resource Allocations (Bound): resource allocations analyse can be done when a thread is bound to a processor, like this :

```
Actual_Processor_Binding => (reference (p3DX.OdroidXU4.Exynos_SOC.
        big_procs_cluster.big_proc1)) applies to rem_trk_sw.usbcam;
```

Several properties have to be defined: the Compute_execution_time and Period properties in the thread model, and the MIPSCapacity in the processor model.

The tool then computes:

- the load for a thread is the compute execution time divided by the period.
- the total processor load is the sum of every load per thread bound to the processor.
- the MIPS demand for the processor is its MIPS capacity multiplied by its total load.

Whenever the MIPS demand exceeds the MIPS capacity for a processor, an error is reported for the instance. As an example, we provide an extract of the AADL model for the usb_cam node found in many ROS distributions. This node takes the video stream from a RGB camera pluged in the USB bus of the computer board, and publish the video frames on a dedicated ROS topic. The first part of our nd_usb_cam AADL package include the PIM definition: the node input and output in a process component, and its internal structure in the associated implementation.

```
process usb_cam_nd extends ros::node
  features
    rgb_stream_in: in event data port ros_data::video_stream.rgb;
    rgb_image_raw_out: out event data port ros_data::Image.rgb;
  end usb_cam_nd;

process implementation usb_cam_nd.impl
  subcomponents
    image_broadcaster: thread imagePublisher.impl;
    usbSpinner: thread usbcam_spinner.impl;
  connections
    con1: port image_broadcaster.pub_msg -> rgb_image_raw_out;
    con2: port rgb_stream_in -> usbSpinner.rgb_stream_in;
  end usb_cam_nd.impl;

thread imagePublisher extends ros::publisher
  features
    pub_msg: refined to out event data port ros_data::Image.rgb;
  end imagePublisher;

thread implementation imagePublisher.impl
  properties
    Period => 33333 us;--@ 30 images/s
  end imagePublisher.impl;
```

The second part (PDM) shows the platform and CPU dependent implementation for the Odroid XU4 (from HardKernel) board, which includes ARM A15 and A7 cores. We chose to put the node Period on the PIM implementation of the node, since it does not depends on the hardware target, but on the RGB camera we use. The Compute_execution_time property is set on the PDM side.

```
process implementation  usb_cam_nd.xu4_a15 extends usb_cam_nd.impl
  subcomponents
    image_broadcaster: refined to thread imagePublisher.xu4_a15;
  properties
    SEI::MIPSBudget => 141.0 MIPS;--(197 MIPS)/(1.4 IPC)
  end usb_cam_nd.xu4_a15;

thread implementation imagePublisher.xu4_a15 extends imagePublisher.
      impl
  properties
    Compute_execution_time => 2319 us ..  2319 us;
    Queue_Size => 512 applies to pub_msg;
  end imagePublisher.xu4_a15;
  end imagePublisher.i7;
```

Analyze Resource Budgets (Not Bound): the additional MIPSBudget (SEI standard) property must be set for the process. The tool adds the MIPS demands for every thread inside the process and check if the total does not exceed the budget. An error is reported if so.

Analyze Bus Load: bus load analysis is performed from the the size of the message to be transmitted, and its frequency. The first one is the Data_Size property, which is defined in our *ros_data* package. We give below the definition for the 640x480 RGB images, 8bits per channel, 3 channels, message stream from the RGB camera that the usb_cam node of our former example is processing. In the PIM model of the thread imagePublisher, given earlier, we find that the format of the *out event data port* is *refined to* this particular data size:

```
data implementation Image.rgb extends Image.impl
  properties
  --640x480x3=921600 Bytes #0.92MBytes
  Data_Size => 921 KByte;
  end Image.rgb;
```

The frequency of the message is the Period property of the thread component that issues the message. From the period and data size, the tool calculates the bandwidth demand for a publisher thread on the bus onto which its output connection is bound. In our example, that would be $921\ \mathtt{KBytes} \times {}^{1}/_{33333\mu s} = 27.63\ \mathtt{MBytes/s}$. The next step is to compare this bandwidth demand with the bandwidth capacity of the bus. For this we need to add this specific property to the AADL model of our hardware platform, where buses are declared, and to the ROS library where a bus component will be added for every implementation of the TCPROS virtual bus. Hence, our ROS core package includes the following components, with the bandwidth capacities reported from the profiling work presented in [3]:

```
bus ros_bus end ros_bus;
bus implementation ros_bus.no_taskset extends ros_bus.impl
  properties
  SEI::BandWidthCapacity => 122.0 MBytesps;
  end ros_bus.no_taskset;

bus implementation ros_bus.A15 extends ros_bus.impl
  properties
    SEI::BandWidthCapacity => 166.0 MBytesps;
  end ros_bus.A15;
```

Obviously, in order for any bus load analysis to take place, software connections ought to be bound to busses. This is done thanks to the Actual_Connection_Binding property in the AADL model of the deployed implementation of our complete system; e.g. for the output of our usb_cam node :

```
Actual_Connection_Binding => (reference (ROSbus)) applies to
        rem_trk_sw.con6;-- usbcam->color_tracking OK
```

6 Conclusion

Our library of AADL components for ROS based applications allows to model complete robotic software. It is organized in a set of AADL packages to reflect the ROS middleware basic structure and underlying mechanism. ROS nodes are modeled as processes including threads as spawn by ROS at run time. ROS communications via topics are modeled as buses, to allow for bus load analysis. Dedicated properties are set into the models of components to allow for CPU and bus load analysis, using standard tools from OSATE2. Those analysis stem from the binding of software components onto hardware components which are provided in the AADL model of the hardware of the robot. Different AADL packages for robots and embedded computer boards are also included in our library. Together with the continuous building of models for new hardware and software parts, especially integrating components from the last ROS2 distributions, one short term perspective include automatic code generation for ROS from our AADL models, by developing an extension of the RAMSES (Refinement of AADL Models for Synthesis of Embedded Systems) [10] automatic code generation tool.

References

[1] P. H. Feiler, B. A. Lewis, and S. Vestal, *The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems*, pp. 1206–1211. 2006.

[2] E. Senn and L. Bourdon, “Introducing CPU load analysis from AADL models for ROS applications : a use case,” in *FDL 2021, Forum on specification & Design Languages*, September 2021.

[3] E. Senn, “ROS communications profiling for bus load analysis from AADL,” in *ERTS 2022, 11th European Congress on Embedded Real Time Systems*, March 2022.

[4] G. Biggs, K. Fujiwara, and K. Anada, “Modelling and analysis of a redundant mobile robot architecture using AADL,” in *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

[5] A. Lotz, A. Hamann, R. Lange, C. Heinzemann, J. Staschulat, V. Kesel, D. Stampfer, M. Lutz, and C. Schlegel, “Combining robotics component-based model-driven development with a model-based performance analysis,” in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pp. 170–176, 2016.

[6] G. Bardaro and M. Matteucci, “Using AADL to model and develop ROS-based robotic application,” in *2017 First IEEE International Conference on Robotic Computing (IRC)*, pp. 204–207, 2017.

[7] G. Bardaro, A. Semprebon, A. Chiatti, and M. Matteucci, “From models to software through automatic transformations: An AADL to ROS end-to-end toolchain,” in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 580–585, 2019.

[8] M. Larsen, “Modelling field robot software using AADL,” Electrical and Computer Engineering Technical report ECE-TR-25, Aarhus University, april 2016.

[9] N. Valigi, “Concurrency in ros1 and ros2,” in *ROSCon 2019, ROS developers CONference*, October - November 2019.

[10] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, “Design patterns for rule-based refinement of safety critical embedded systems models,” in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, pp. 67–76, 2012.

An Introduction to ALISA and Its Usage for an Industrial Railway System Case Study

Dominique Blouin
LTCI, Telecom Paris, Institut Polytechnique de Paris, Palaiseau, France; email: dominique.blouin@telecom-paris.fr
Paolo Crisafulli, Cristian Maxim
Institut de Recherche Technologique SystemX, Palaiseau, France; email: {first name}.{last name}@irt-systemx.fr
Francoise Caron
Eiris Conseil, Jouy en Josas, France, Palaiseau, France; email: francoise.caron@eiris.fr

Abstract

This paper presents an overview of ALISA (Architecture-Led Incremental System Assurance) and its evaluation for a case study of the railway domain as presented during the ADEPT workshop collocated with the 26th Ada-Europe International Conference on Reliable Software Technologies.

Keywords: ALISA, AADL, Model-Based Engineering, Safety-Critical Systems, Cyber-Physical Systems.

1 Introduction

Model-Based Engineering is a paradigm where models are used to represent the system to be developed, at the appropriate level(s) of abstraction, so that analyses can be performed early on these models, before the real system is implemented. In doing so, design errors can be detected early to reduce development costs. This is realized by augmenting the standard V-cycle development process model with another V where validation and verification activities occur at each phase (figure 1). In this Architecture-Centric Virtual Integration (ACVIP) process [1], models and tools are used to support such validation and verification activities. While the left-hand side of the V-cycle is concerned with *building* the system, the right-hand side deals with *assuring* that the system meets its requirements. This is particularly important for safety critical systems that must demonstrate their safety to certification authorities.

The SAE Architecture Analysis & Design Language (AADL) standard[1] has been developed to support the system development phases of figure 1, from design down to code development. However, the other phases not supported by AADL could also greatly benefit from modelling. To cover these phases, ALISA (Architecture-Led Incremental System Assurance) [2, 3] has been developed, complementing AADL with notations to model requirements, requirements verification activities and assurance case. In this short paper, we briefly introduce ALISA and present an overview of its application to a safety-critical system of the railway domain, covering both sides of the development activities of figure 1.

[1]https://www.sae.org/standards/content/as5506c/

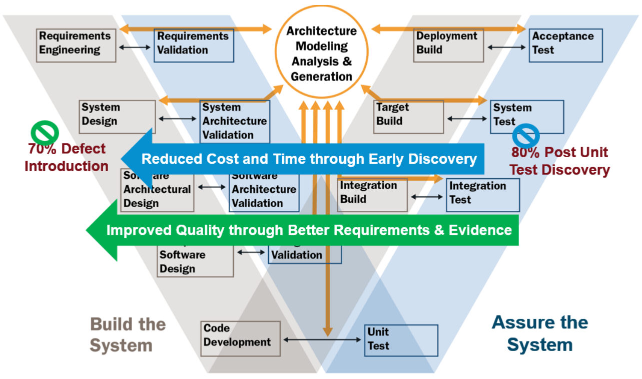

Requirements Engineering
Requirements Validation
Architecture Modeling Analysis & Generation
Deployment Build
Acceptance Test
System Design
System Architecture Validation
Target Build
System Test
70% Defect Introduction
Reduced Cost and Time through Early Discovery
80% Post Unit Test Discovery
Software Architectural Design
Software Architecture Validation
Integration Build
Integration Test
Improved Quality through Better Requirements & Evidence
Software Design
Validation
Build the System
Code Development
Unit Test
Assure the System

Figure 1: V-cycle development process model with virtual integration activities (from [2])

2 The safety-critical European Train Control System

This work was produced during the PST project[2] where ALISA and AADL were exercised for an industrial safety-critical system from the railway domain [4]. This system consists of the on-board equipment of the European Train Control System (ETCS). The ETCS is a system of systems for the signalling and control of the European Rail Traffic Management System (ERTMS)[3]. The European Vital Computer (EVC) is the core component of the train on-board equipment, being the computing platform hosting the train control functions. Some of these functions are safety-critical, such as the emergency breaking function.

The engineering process of the EVC is typical of the railway industry because it is constrained by many non-functional requirements on safety, reliability and availability. For instance, a safety requirement defines that the hazard rate of the EVC shall not exceed a threshold of 0.6×10^{-9} failures per hour of operation (ERA requirement). This lead to choose a Triple Modular Redundant (TMR) architecture to implement the EVC, which is very typical of safety-critical embedded

[2]https://www.irt-systemx.fr/en/projets/pst/
[3]https://www.ertms.net/

systems.

The EVC is composed of three identical computing platforms intended to perform the same computation based on a single input message. Extra functions are added including a majority-voting process executed by each of the computers. In case different outputs are produced, the faulty platform will be turned off. The three execution platforms are viewed by the train applications as a single computing unit, the EVC.

The TMR architecture of the EVC has a strong impact on the response-time and schedulability properties due to the middleware functions. This can be adverse to the performance requirements. The more CPU consuming the middleware is, the less resources the application will have to execute. This leads to a design requirement stating that at least 50% of the CPU utilization is available for the application. Besides, requirements exist at the braking system level for the overall delay between receiving the emergency break signal and applying the break command to be less than 1 second (ERA requirement). In addition to performance requirements, design requirements state that all threads of the EVC shall be periodic and that all CPUs of the EVC hold the same functions and shall be of same make and model.

3 Using ALISA and AADL to model the ETCS

3.1 Overview of ALISA

ALISA takes its origins from the Requirements Definition and Analysis Language (RDAL) [5], first developed as a fragment language that could be combined with an existing Architecture Description Language (ADL) to provide Requirements Engineering (RE) support. RDAL was inspired from several existing RE approaches including, among others, the requirements diagram of SysML[4] and KAOS [6, 7]. KAOS defines four complementary and interrelated views on a system:

1. Goals from stakeholders (owners, users, business managers, regulations, etc).
2. Responsible agents (humans, automated systems or environment)
3. Problem domain (concepts and their relationships)
4. System behaviours to achieve the goals

The KAOS language to specify these views is implemented as a single monolitic language, therefore requiring to use the KAOS generic problem domain view. However, for ACVIP, the problem domain is already captured in AADL making the use of KAOS cumbersome since it requires translation between KAOS and AADL. This justified the development of RDAL, from which ALISA was later developed and extended to cover the verifications and assurance domains.

The architecture of ALISA in terms of its sublanguages and covered concepts is depicted in figure 2. In the following, we briefly illustrate AADL and each of the ALISA sublanguages using the ETCS system.

[4]http://www.omgsysml.org/

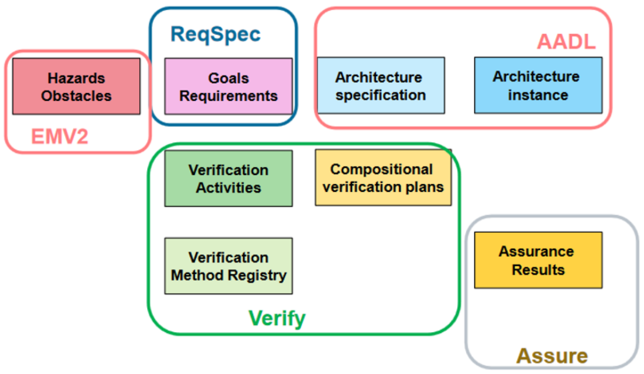


Figure 2: ALISA unified concepts (from [8])

3.2 AADL

AADL is a component-based Architecture Description Language developed for safety-critical real-time embedded and cyber-physical systems. It allows for modeling both the software and hardware parts of a system. Several analyses tools are readily available for estimating properties such as latency and schedulability, as well as resource consumption. Automatic code generation is also available.

The ETCS braking system has been modeled in AADL following a process similar to that of [9], where the system is decomposed into four layers:

- A functional view where system functions are represented as *abstract* components interacting via abstract features and connections. Other abstract components are also declared to represent system variable types. Abstract components in AADL are similar to SysML blocks.
- A software view derived from the above functional components converted to *subprograms*. Those subprograms are called by *threads* to be deployed on the execution platform according to performance requirements. Such threads are grouped in to *processes* representing dedicated memory space. System variable types represented as abstract components in the functional view are refined as data components to which properties such as data representation and data size can be set to capture their discrete nature.
- An execution platform view where hardware components such as processors, memory, buses and devices are composed to model computing platforms.
- A deployment view where components of the software view are mapped to the selected execution platform. Threads are mapped to processors, connections to buses and processes and data components to memories.

Figure 3 shows the overall software view for the EVC, where each redounded software application is modeled as a system including several processes for the middleware and the application functions.

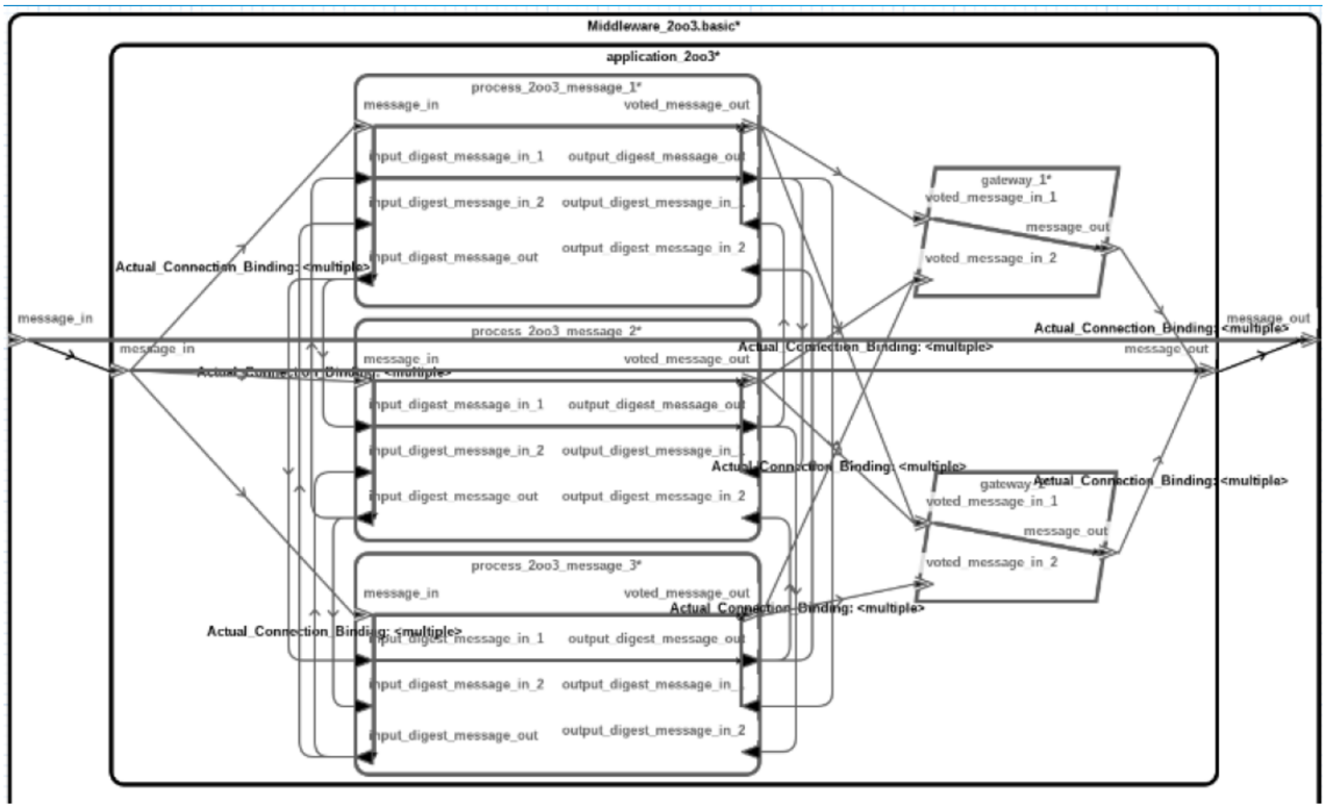


Figure 3: The AADL software view of the EVC

3.3 ReqSpec

The *ReqSpec* notation allows capturing stakeholder goals and their organization, and their corresponding verifiable requirements. An example requirement for the ETCS is shown in the top of figure 5. Those verifiable requirements are gradually decomposed into subrequirements, allocated to architecture subcomponents responsible for satisfying them via a *for* construct. In this iterative process, requirements decomposition is therefore led by design decisions on the system architecture.

Requirements at the leaf of such decomposition must be verifiable, either by expressing them with a predicate of the Resolute language [10], or linked to a set of verification activities via a *claim* element, whose name is identical to the corresponding requirement.

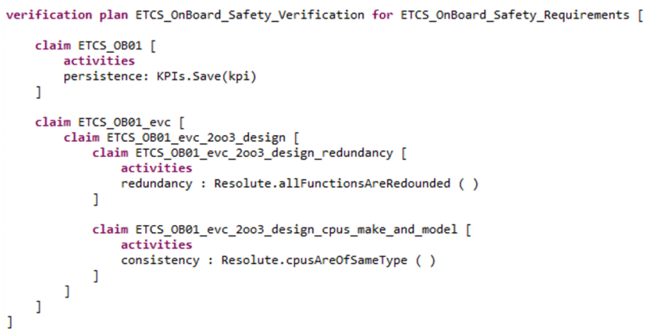
3.4 EMV2

The *EMV2* (Error Model Annex Version 2) notation is used to annotate AADL components with error propagations and behaviours to mitigate those errors. ReqSpec requirements, can then be traced to some EMV2 elements via the *mitigates* construct to indicate that a safety requirement has been created to handle a hazard modeled in EMV2.

3.5 Verify

The aforementioned *claim* notion is part of the *Verify* notation, which allows modeling sophisticated *verification activities* (figure 4). A verification activity can be of different kinds such as Resolute claim functions or Python Scripts. In addition, when the reference tool for AADL OSATE[5] is used, its analysis plugins, external Java methods or JUnit-based code tests can also be declared as verification activities. Those can be registered in the development environment to be used by claims. A combination of such claims then constitutes a *verification plan*, which can be assigned to a requirement set via the *for* construct.

[5]https://osate.org/

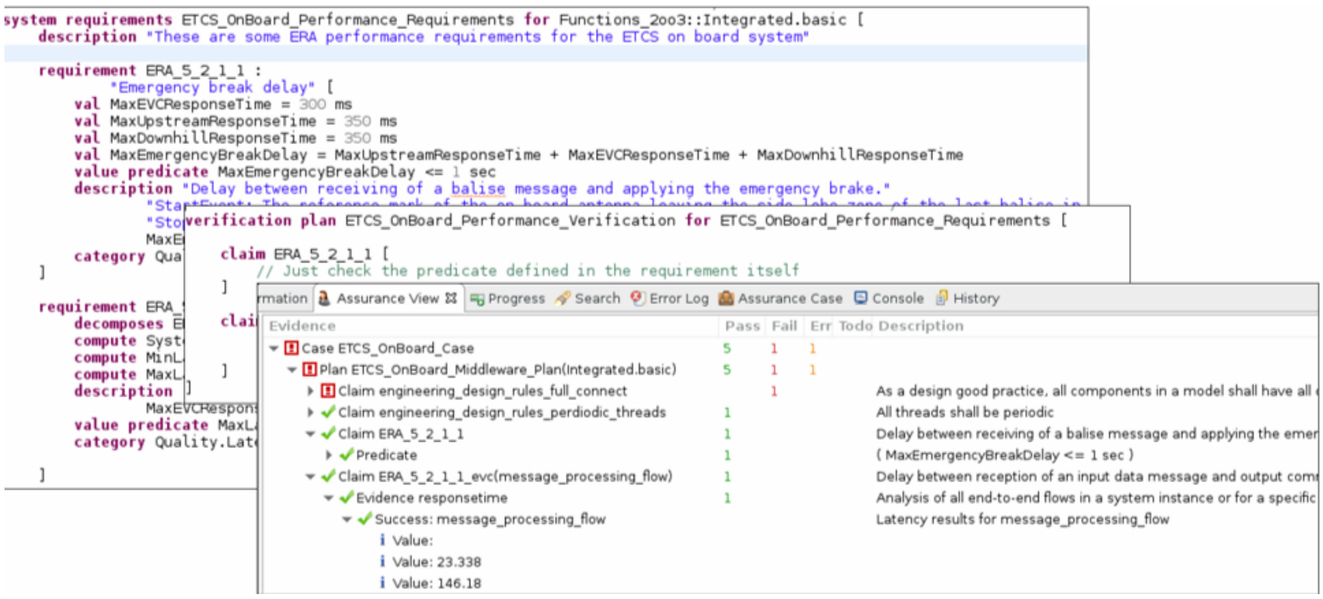


```
verification plan ETCS_OnBoard_Safety_Verification for ETCS_OnBoard_Safety_Requirements [

    claim ETCS_OB01 [
        activities
        persistence: KPIs.Save(kpi)
    ]

    claim ETCS_OB01_evc [
        claim ETCS_OB01_evc_2oo3_design [
            claim ETCS_OB01_evc_2oo3_design_redundancy [
                activities
                redundancy : Resolute.allFunctionsAreRedounded ( )
            ]

            claim ETCS_OB01_evc_2oo3_design_cpus_make_and_model [
                activities
                consistency : Resolute.cpusAreOfSameType ( )
            ]
        ]
    ]
]
```

Figure 4: A verification plan for the ETCS.

3.6 Assure

Finally, *assurance cases* can be modelled with the *Assure* notation (bottom part of figure 5). An assurance case consists of a set of verification plans whose execution results for a system design (and potentially implementation artifacts such as tests) can be used as evidence for system certification. In OSATE, assurance cases can be automatically executed and results are displayed in a dedicated *assurance view* where the number of *Success*, *Failed*, *Error* or *TBD* verification results are shown.



```
system requirements ETCS_OnBoard_Performance_Requirements for Functions_2oo3::Integrated.basic [
  description "These are some ERA performance requirements for the ETCS on board system"

  requirement ERA_5_2_1_1 :
        "Emergency break delay" [
    val MaxEVCResponseTime = 300 ms
    val MaxUpstreamResponseTime = 350 ms
    val MaxDownhillResponseTime = 350 ms
    val MaxEmergencyBreakDelay = MaxUpstreamResponseTime + MaxEVCResponseTime + MaxDownhillResponseTime
    value predicate MaxEmergencyBreakDelay <= 1 sec
    description "Delay between receiving of a balise message and applying the emergency brake."
        "Sta
        "Sto
        MaxE
    category Qua
  ]

  requirement ERA_
    decomposes E
    compute Syst
    compute MinL
    compute MaxL
    description
        MaxEVCRespon
    value predicate MaxL
    category Quality.Lat

  ]
```

```
verification plan ETCS_OnBoard_Performance_Verification for ETCS_OnBoard_Performance_Requirements [

  claim ERA_5_2_1_1 [
      // Just check the predicate defined in the requirement itself
  ]

  clai

  ]
]
```

```
rmation | Assurance View | Progress | Search | Error Log | Assurance Case | Console | History

Evidence                                                Pass  Fail  Err  Todo  Description
▼ Case ETCS_OnBoard_Case                                 5     1     1
  ▼ Plan ETCS_OnBoard_Middleware_Plan(Integrated.basic)  5     1     1
    ▶ Claim engineering_design_rules_full_connect              1                 As a design good practice, all components in a model shall have all
    ▶ ✓ Claim engineering_design_rules_perdiodic_threads   1                     All threads shall be periodic
    ▼ ✓ Claim ERA_5_2_1_1                                  1                     Delay between receiving of a balise message and applying the emer
      ▶ ✓ Predicate                                        1                     ( MaxEmergencyBreakDelay <= 1 sec )
    ▼ ✓ Claim ERA_5_2_1_1_evc(message_processing_flow)     1                     Delay between reception of an input data message and output com
      ▼ ✓ Evidence responsetime                            1                     Analysis of all end-to-end flows in a system instance or for a specific
        ▼ ✓ Success: message_processing_flow                                     Latency results for message_processing_flow
              i Value:
              i Value: 23.338
              i Value: 146.18
```

Figure 5: Requirements, verification plan and assurance case for the ECTS.

3.7 Towards an agile engineering process

The automated verification of requirements and assurance case modeling in OSATE have been integrated into an agile engineering process making use of the well-known Jenkins build server. The ALISA verification results were used to compute and display the evolution of Key Performance Indicators (KPI) during continuous integration. Coupled with a Git versionning server, the KPI computation results can be used to chart qualify system performance evolution of design alternatives over time to evaluate their impact on performances.

4 Conclusion and Perspectives

We have demonstrated how AADL and ALISA are well-suited to model, analyse, verify and assure safety-critical systems supporting an agile architecture-centric engineering process. This process includes continuous verification to maintain the design within the solution space shaped by the set of requirements. However, our experience showed that ALISA is not yet completely mature. Scalability and multi-organization issues still need to be addressed. Moreover, the development of the *design goal* construct, which can be used to define system performance objectives to support the computation of KPIs was not completed at the time of our experiment.

Nevertheless, the AADL ecosystem of companion languages and development environment is very promising since it opens the way to agile engineering of highly constrained systems requiring certification.

References

[1] A. Boydston and P. H. Feiler, "Architecture centric virtual integration process (acvip) : A key component of the dod digital engineering strategy," 2019.

[2] J. D. McGregor, D. P. Gluch, and P. H. Feiler, "Analysis and design of safety-critical, cyber-physical systems," *Ada Lett.*, vol. 36, pp. 31–38, may 2017.

[3] J. Delange, P. H. Feiler, and N. A. Ernst, "Incremental life cycle assurance of safety-critical systems," 2016.

[4] P. Crisafulli, D. Blouin, F. Caron, and C. Maxim, "Engineering Railway Systems with an Architecture-Centric Process Supported by AADL and ALISA: an Experience Report," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, (Toulouse, France), Jan. 2020.

[5] D. Blouin and H. Giese, "Combining requirements, use case maps and aadl models for safety-critical systems design," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 266–274, 2016.

[6] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley Publishing, 1st ed., 2009.

[7] A. van Lamsweerde and E. Letier, "From object orientation to goal orientation: A paradigm shift for requirements engineering," in *Radical Innovations of Software and Systems Engineering in the Future* (M. Wirsing, A. Knapp, and S. Balsamo, eds.), (Berlin, Heidelberg), pp. 325–340, Springer Berlin Heidelberg, 2004.

[8] P. Feiler, "Architecture-led incremental system assurance (alisa) tutorial," 2014.

[9] D. Blouin and E. Borde, *AADL: A Language to Specify the Architecture of Cyber-Physical Systems*, pp. 209–258. Cham: Springer International Publishing, 2020.

[10] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen, "Resolute: An assurance case language for architecture models," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT 2014, (New York, NY, USA), pp. 19–28, Association for Computing Machinery, 2014.

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

Ada-Deutschland

Dr. Hubert B. Keller CEO
ci-tec GmbH
Beuthener Str. 16
76139 Karlsruhe
Germany
+491712075269
Email: h.keller@ci-tec.de
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

Ada-Europe Sponsors

27 Rue Rasson
B-1030 Brussels
Belgium
Contact:Ludovic Brenta
ludovic@ludovic-brenta.org

46 Rue d'Amsterdam
F-75009 Paris
France
sales@adacore.com
www.adacore.com

506 Royal Road
La Caverne, Vacoas 73310
Republic of Mauritius
Contact: David Sauvage
david.sauvage@adalabs.com

2 Rue Docteur Lombard
92441 Issy-les-Moulineaux Cedex
France
Contact: Jean-Pierre Rosen
rosen@adalog.fr
www.adalog.fr/en/

Jacob Bontiusplaats 9
1018 LL Amsterdam
The Netherlands
Contact: Wido te Brake
wido.tebrake@deepbluecap.com
www.deepbluecap.com

24 Quai de la Douane
29200 Brest, Brittany
France
Contact: Pierre Dissaux
pierre.dissaux@ellidiss.com
www.ellidiss.com

In der Reiss 5
D-79232 March-Buchheim
Germany
Contact: Frank Piron
info@konad.de
www.konad.de

3271 Valley Centre Drive,Suite 300
San Diego, CA 92069
USA
Contact: Shawn Fanning
sfanning@ptc.com
www.ptc.com/developer-tools

Enterprise House
Baloo Avenue, Bangor
North Down BT19 7QT
Northern Ireland, UK
enquiries@sysada.co.uk
sysada.co.uk

1090 Rue René Descartes
13100 Aix en Provence
France
Contact: Patricia Langle
patricia.langle@systerel.fr
www.systerel.fr/en/

Tiirasaarentie 32
FI 00200 Helsinki
Finland
Contact: Niklas Holsti
niklas.holsti@tidorum.fi
www.tidorum.fi

Beckengässchen 1
8200 Schaffhausen
Switzerland
Contact: Ahlan Marriott
admin@white-elephant.ch
www.white-elephant.ch

Ada
europe