The journal for the international Ada community

Ada User Journal

Volume 43
Number 3
September 2022

Produced by Ada-Europe

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 43
Number 3
September 2022

Contents

Editorial Policy for Ada User Journal

Publication

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

Aims

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

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

The Journal publishes the following types of material:

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

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

Original Papers

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

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

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

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

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

Proceedings and Special Issues

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

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

News and Product Announcements

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

Reprinted Articles

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

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

Commentaries

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

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

Announcements and Reports

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

Reviews

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

Submission Guidelines

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

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

Editorial

This editorial starts with two notes related to the annual event organized by Ada-Europe. Firstly, I would like to highlight the fact that the next edition of the conference will be held in Lisbon, Portugal, featuring several tracks, as usual (Journal-track, Industrial-track, Work-in-Progress-track). A Call for Submissions is included in the pages of this issue. Secondly, I would like to note that since the 2019 edition of the Ada-Europe Conference, the papers presented at the conference (in the main or, now called, journal track), are published by Elsevier with open access, and links to all these papers are provided on the Ada-Europe website (under the ”Conferences” tab).

As for the contents of this issue, we conclude the publication of the proceedings of the AEiC 2022 Work-in-Progress Session, and we provide four papers derived from AEiC 2022 Industrial presentations. Four WiP papers are included. The first one is a brief abstract authored by Ivan Kolesnikov, from IRIT, France, proposing the use of Bounded Model Checking (BMC) to improve the performance of simulation and testing of Cyber-Physical Systems. Then, another paper related to verification and testing is provided. B. Kempa, C. Johannsen and K. Y. Rozier, from the Iowa State University, describe their on-going work on extending the R2U2 real-time verification tool to provide validation transparency without sacrificing performance in deployment time. The third paper describes joint work by Q. Dauprat and J. P. Rosen, from Adalog, and P. Dorbec and G. Richard, from the university of Caen Normandie, on the exploration of graph databases for code analysis. The objective is to not only reduce the time required to perform the analysis, but also to make it more effective. Finally, these proceedings are closed with a paper by T. Carvalho and L. M. Pinho, from ISEP, Portugal, presenting on-going work on the integration of GPU tracing in the AMALTHEA framework for automotive system design and development.

The papers derived from industrial presentations at AEiC 2022 start with one authored by P. van de Laar and A. Mooij, from TNO, The Netherlands, describing Renaissance-Ada. Renaissance-Ada is a set of tools for Ada code analysis and transformation/improvement. The paper describes the tools and provides examples on how to use them. Then, several authors from the Barcelona Supercomputing Center, Spain, and from Bosch, Germany, describing work on the improvement of the AMALTHEA domain-specific modelling language to handle the incorporation of software redundancy in the system design. The third paper, authored by A. Medaglini and S. Bartolini from the University of Siena, V. Di Massa from Thales, and F. Dini from Magenta, presents a suite of tools for the analysis of ADAS systems. The tools allow the generation of tests to find corner cases in the systems under test, and hence support the safety assessment tasks. The last paper, authored by C. Dross, from AdaCore, provides insights on how the SPARK tool works, for the verification of contracts in Ada.

As usual, the issue also includes the Quarterly News Digest, prepared by Alejandro R. Mosteo, and the Calendar section, prepared by Dirk Craeynest.

Antonio Casimiro
Lisboa
September 2022
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,

This number is packed full with works from the last Ada-Europe conference and so the News section is slim and focused on announcements. Our regular contents will pick up where they left in the next number.

Even so, there are a few news items of note: There is a new kid on the block of Ada websites, fully community-driven and intended to ease the onboarding of Ada newcomers [1]. Will it stand the test of time? Let us hope so.

And, on the front of the final arrival of Ada 2022, an exhaustive overview of its new features has been posted at the Ada Auth website [2]; great stuff for those of us eager for the next iteration of the language.

[1] "Yet another Ada website", in Ada-related Resources.

[2] "Ada 2022 Overview", in Ada-related Resources.

Sincerely,
Alejandro R. Mosteo.

Ada-related Events

SIGAda Awards Nominations

From: Tucker Taft <tucker.taft@gmail.com>
Subject: SIGAda Awards Nominations -- Due Sept. 20, 2022
Date: Tue, 13 Sep 2022 123800 -0700
Newsgroups:comp.lang.ada

[Past call, for the record. Stay tuned for the winner! —arm]

Dear Members of the Ada Community:

We welcome your nominations for the 2022 Robert Dewar Award for Outstanding Ada Community Contributions and the 2022 ACM SIGAda Distinguished Service Award.

We hope this message finds you and your family safe and healthy. The SIGAda meeting in 2022 will be a HILT workshop of the 37th IEEEACM International Conference on Automated Software Engineering, ASE'2022. The workshop will be held on October 14th 2022. See ASE'2022 for details on the venue and registration.

This year's award winners will be announced as part of the SIGAda Workshop.

Award nominations are due on September 20th.

The ACM SIGAda Awards recognize individuals, teams, and organizations that have made outstanding contributions to the Ada community and to SIGAda. The two categories of awards are

(1) Robert Dewar Award for Outstanding Ada Community Contributions
-- For broad, lasting contributions to Ada technology & usage.

(2) ACM SIGAda Distinguished Service Award
-- For exceptional contributions to SIGAda activities & products.

If there are individuals or teams who you feel have made contributions that satisfy these criteria, please consider nominating them. You may nominate a person or a team of people for either or both awards, and as many people as you think worthy.

Please visit the SIGAda Awards page http://www.sigada.org/exec/awards/awards.html
and peruse the names of past winners. This may help you think about the measure of accomplishment that is appropriate. You may be aware of people who have made substantial contributions that have not yet been acknowledged. Nominate them. Consider what you believe to be the best developments in the Ada community or SIGAda in the last year; the last 5 years; since Ada's inception. Who was responsible? Nominate them.

Please note that anyone who has received either of the two awards remains eligible for the other. Perhaps there is an outstanding SIGAda volunteer who has won our Distinguished Service Award and who has also made important contributions to the advance of Ada technology, or vice versa. Nominate him or her!

The nomination form is available on the SIGAda Awards page: http://www.sigada.org/exec/awards/awards.html

Submit your nomination as an e-mail or e-mail attachment to sigada-awards-comm@acm.org
From your nominations, the recipients of the awards are determined by a poll of previous award winners.

Call our attention to the people who are most deserving, by nominating them. And please nominate by September 20th!

Your participation in the nominations process will help maintain the prestige and honor of these awards.

Thank you,

Drew Hamilton
Chair ACM SIGAda Awards Committee
ACM SIGAda Past Chair

Drew Hamilton, Ph.D. '96
Professor of Computer Science & Engineering
Director, Texas A&M Center for Cybersecurity

ACM SIGAda HILT'22

From: Tucker Taft <tucker.taft@gmail.com>
Subject: ACM SIGAda HILT'22 Workshop on Supporting Rigorous SW Development -- Oct 14, 2022
Date: Fri, 16 Sep 2022 122628 -0700
Newsgroups: comp.lang.ada

The seventh ACM workshop on High Integrity Language Technology (ACM HILT 2022) is being held on October 14, 2022 in Detroit, MI in conjunction with the 2022 Automated Software Engineering conference (ASE'22), sponsored by SIGAda. This year's HILT theme is Language and Tool Support for

Rigorous Software Development. We have 9 presentations plus two keynotes related to this theme. Our keynote speakers are K. Rustan M. Leino, the creator of the Dafny verifiable language and the Boogie system supporting major industrial uses of formal methods, and Niko Matsakis, one of the original members of the Rust design team, talking about a-mir-formality, a more formal model of Rust. For more information see

https://conf.researchr.org/track/ase-2022/ase-2022-workshop-hilt-22

#formalmethods #softwareengineering #ada #rust #spark #dafny #ACM #ASE

Ada-related Resources

[Delta counts are from July 18th to November 13th. —arm]

Ada on Social Media

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Ada on Social Media
Date: 13 Nov 2022 1127 CET
To: Ada User Journal readership

Ada groups on various social media

- Reddit: 8_200 (+122) members	[2]
- LinkedIn: 3_400 (+72) members	[1]
- Stack Overflow: 2_273 (+35) questions	[3]
- Telegram: 153 (+10) users	[6]
- Gitter: 140 (+17) people	[5]
- Libera.Chat: 77 (+2) concurrent users	[4]
- Ada-lang.io: 50 (new) users	[8]
- Twitter: 37 (+7) tweeters	[7]
85 (+10) unique tweets	[7]

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

[2] http://www.reddit.comrada

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

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

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

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

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

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

Repositories of Open Source Software

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Repositories of Open Source software
Date: 13 Nov 2022 1128 CET
To: Ada User Journal readership

Rosetta Code: 919 (+4) examples	[1]
39 (=) developers	[2]
GitHub: 763* (=) developers	[3]
Alire: 309 (+49) crates	[6]
Sourceforge: 238 (-6) projects	[4]
Open Hub: 214 (=) projects	[5]
Codelabs: 53 (=) repositories	[8]
Bitbucket: 31 (-56**) repositories	[7]
AdaForge: 8 (=) repositories	[9]

*This number is unreliable due to GitHub search limitations.

**This large drop may be related to the extinction of Mercurial repositories, see https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket.

[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://sourceforge.net/directory/language:ada/

[5] https://www.openhub.net/tagsnames=ada

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

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

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

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

Language Popularity Rankings

From: Alejandro R. Mosteo <amosteo@unizar.es>
Subject: Ada in language popularity rankings
Date: 13 Nov 2022 1128 +0100
To Ada User Journal readership

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

- TIOBE Index: 27 (+3) 0.48% (+0.1%)	[1]
- PYPL Index: 17 (=) 0.81% (-0.05%)	[2]
- IEEE Spectrum (general): 35 (-4) Score 1.16	[3]
- IEEE Spectrum (jobs): 33 (new) Score 0.79	[3]
- IEEE Spectrum (trending): 32 (new) Score 3.95	[3]

The Spectrum ranking has been revamped, no longer using the same categories and rating methodology. Thus, historic trends are omitted for this issue except for the default category.

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

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

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

Yet Another Ada Website?

From: Maxim Reznik <reznikmm@gmail.com>
Subject: yet another Ada web site
Date: Thu, 25 Aug 2022 030129 -0700
Newsgroups: comp.lang.ada

I wonder if the Ada community needs yet another web site?

My idea is here:

https://www.reddit.com/r/ada/comments/wx9zp1/yet_another_ada_web_site/

From: Paul Rubin <no.email@nospam.invalid>
Date: Fri, 26 Aug 2022 115814 -0700

>I wonder if the Ada community needs yet another web site

Adahome.com is sort of like that, but it is run by some company and hasn't been updated in forever. Maybe what you want is a wiki (like forthfreak.net used to be), but you'd have to do a lot of work getting it initially populated, before it became interesting enough to attract more contributors. It's very easy to suggest work for other people to do, but they all have their own projects already.

I don't have much trouble finding any information that I want about Ada, e.g. with web searches. The challenge is in digesting and using the information, not in finding it. I don't see the proposed new web site as being much help. More helpful would be a systematic effort to reproduce or at least supply Ada bindings for the main toolsets that exist for other languages, to target popular microcontrollers, etc.

From: Rene <rehartmann@t-online.de>
Date: Sat, 27 Aug 2022 111209 +0200

Maybe a web forum would be a good idea, because many people nowadays see Usenet Newsgroups: as an outdated thing. So the fact that the community mostly relies on comp.lang.ada may turn them off. (I Don't want to discuss whether Usenet is actually outdated or not, but I guess many people feel this way)

From: Nasser M. Abbasi nma@12000.org
Date: Sat, 27 Aug 2022 045317 -0500

Some are starting to use discord for such things. For example, the main Julia forum is at discord

https://discourse.julialang.org/t/julialang-official-discord-server/45499

From: Simon Wright <simon@pushface.org>
Date: Sun, 28 Aug 2022 082148 +0100

Would be better than Telegram or Gitter - at any rate for actual discussions.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 28 Aug 2022 095047 +0200

> Some are starting to use discord for such things.

Indeed. It is quite uncomfortable I must say from my experience. (I participate there because I maintain Ada Julia bindings)

P.S. They just killed Firefox support keeping it listed as a supported browser...

P.P.S. Clearly, how anybody could implement a discussion board without making it dependent on petabytes of browser-specific scripts. Right (-))

From: Maxim Reznik <reznikmm@gmail.com>
Date: Fri, 16 Sep 2022 082540 -0700

I'm happy to announce a new Ada website

https://ada-lang.io

Thank people who make it real!

I'm asking the community to send their updates and make it even better.

Here is the Paul Jarrett's original message:

> Hi folks, @onox and me have been working on something for a few weeks, and we need your help. We've been building an open source, Ada community site to share with everyone. The intent is an open source community hub that will persist for a long time. There's a Github organization set up for people to contribute to and my intent is to hand off the domain to some existing Ada group.

> Right now, I've migrated some of my old programming with Ada content over, and I've built on Maxim's work to output a fancy version of the AARM for it. If you have content elsewhere you'd like to add, feel free to submit it. You can use plain Markdown (.md files) or Markdown with React (.mdx files). Some things which I haven't found time to write, which other people could help with, would be an Alire introduction, patterns for when binding to C, how to make a memory allocator, etc.

From: Luke A. Guest <laguest@archeia.com>
Date: Fri, 16 Sep 2022 180725 +0100

Looks decent, especially the non-yellow RM.

From: Stephen Leake <stephen_leake@stephe-leake.org>
Date: Fri, 16 Sep 2022 103403 -0700

> https:// ada-lang.io

It would be nice if comp.lang.ada was listed under community; this newsgroup is far older than all those flash-in-the-pan wannabes).

From: Jere <jhb.chat@gmail.com>
Date: Fri, 16 Sep 2022 114556 -0700

If you do add it, I would recommend NOT using a link to the google groups interface given the porn spam problem. It would stink if someone at a work computer followed it and got hammered by their IT department (speaking from experience). Perhaps someone has a tutorial webpage on how to set up a mail reader for comp.lang.ada that could be linked to under the community section

From: Luke A. Guest <laguest@archeia.com>
Date: Sat, 17 Sep 2022 104547 +0100

The link would be news://comp.lang.ada

From: Emmanuel Briot <briot.emmanuel@gmail.com>
Date: Sun, 18 Sep 2022 233955 -0700

> https:// ada-lang.io

Well done Maxim and Paul, the new site looks nice.

One area that could be nice is a blog aggregator, which would monitor various Ada-related blogs on the Internet and help people find those resources.

I am sure you guys already have plenty of ideas on what to add, so maybe not looking for more -)

Ada 2022 Overview

From: Randy Brukardt <randy@rrsoftware.com>
Subject: Ada 2022 Overview posted
Date: Fri, 23 Sep 2022 024828 -0500
Newsgroups: comp.lang.ada

The Ada 2022 Overview is now available on the Ada-Auth.org website at:

http://www.ada-auth.org/standards/overview22.html

It is available in HTML and PDF versions.

This is an extensive update of Jeff Cousins' Ada 2022 Overview that was published two years ago this month in the Ada User Journal. It expands upon many topics, adds a few missing topics, and corrects many errors found in the original article. An index also has been added, and the HTML version includes links to all of the mentioned AIs and RM subclauses.

The overview tries to cover all of the significant changes and enhancements found in Ada 2022. It includes many examples, and helps to illustrate how the new features could be used.

This version was built partially in response to some complaints here on comp.lang.ada about the lack of Ada 2022 material (outside of the new edition of John Barnes' book, which many be too expensive for many purposes).

Randy Brukardt, ARG Editor.

Ada-related Tools

GNU Emacs Ada Mode 7.3.beta.

From: Stephen Leake <stephen_leake@stephe-leake.org>
Subject: Gnu Emacs Ada mode 7.3.beta released.
Date: Tue, 12 Jul 2022 073115 -0700
Newsgroups: comp.lang.ada

Gnu Emacs Ada mode 7.3.beta is now available in GNU ELPA devel for beta testing.

ada-mode and wisi are now compatible with recent GNAT versions. The grammar is updated to the proposed Ada 2022 version.

Incremental parse is provided. It still has some bugs, so it is not enabled by default. To try it:
(setq-default wisi-incremental-parse-enable t).

Incremental parse often gets confused; to recover, use M-x wisi-reset-parser. That does a full parse of the entire buffer, which can be noticeably slow in large buffers.

To access the beta version via Gnu ELPA, add the devel archive to package-archives:
(add-to-list 'package-archives (cons “gnu-devel” “https://elpa.gnu.org/devel”))

Then M-x list-packages; the beta release shows as ada-mode version 7.3beta1.0.20220711.185004, wisi version 4.0beta1.0.20220711.185552.

See the NEWS files in ~.emacs.delpaada-mode-7.3beta and wisi-4.0beta, or at https://elpa.gnu.org/packages/ada-mode.html, for more details.

Please report success and issues to the Emacs ada-mode mailing list https://lists.nongnu.org/mailman/listinfo/ada-mode-users.

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

```
cd ~.emacs.delpaada-mode-7.3beta
.build.sh
.install.sh
```

There's a bug in install.sh; it looks for WISI_DIR with the old version. Copy the equivalent code from build.sh to fix it.

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

build.sh will take longer than in previous releases, up to several minutes; the ada-mode LR1 parse table is now too big to store in ELPA, so build.sh generates it.

Strings Edit v3.8

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Subject: ANN Strings Edit v3.8
Date: Fri, 5 Aug 2022 143618 +0200
Newsgroups: comp.lang.ada

The library provides text editing and I/O:

- Generic axis scales support;
- Integer numbers (generic, package Integer_Edit);
- Integer sub- and superscript numbers;
- ISO 8601 representations of time and duration;
- Floating-point numbers (generic, package Float_Edit);
- Roman numbers (the type Roman);
- Strings;
- Ada-style quoted strings;
- Base64 encoding;
- Object identifiers and distinguished names;
- RFC 8439 (ChaCha20 cipher, Poly1305 digest, AEAD);
- UTF-8 encoded strings and conversions to older encoding standards;
- Unicode maps and sets;
- Wildcard pattern matching.

http://www.dmitry-kazakov.de/adastrings_edit.htm

Changes to the version previous 3.7:

- Minor bug fixes in Strings_Edit-ISO_8601.

Simple Components v4.63

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Subject: ANN Simple Components v4.63
Date: Fri, 5 Aug 2022 143807 +0200
Newsgroups: comp.lang.ada

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

http://www.dmitry-kazakov.de/adacomponents.htm

Changes to the previous version:

- Code cleanup;
- SQL_Show and Close added to the package SQLite;
- Python dynamic bindings added.

Simple Components v4.64

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Subject: ANN Simple Components v4.64
Date: Fri, 19 Aug 2022 114416 +0200
Newsgroups: comp.lang.ada

The release is focused on B-trees. The B-tree represents a more performant and easy to use alternative to SQLite in Ada applications.

The release fixes bugs and adds tagging B-tree buckets with user data. Tags can be used for effective (e.g. logarithmic) search for values rather than for keys only, e.g. for points of entering or leaving an interval of values etc.

A B-tree based implementation of waveforms (x,y) provides means to store render and analyze large sets of measurement data.

http://www.dmitry-kazakov.de/adacomponents.htm

Changes to the previous version:

- Persistent.Memory_Pools.Streams.Generic_Float_Waveform was added to provide waveform implementation;
- The implementation of B-trees was modified to support tagging buckets of the three. For this the packages Generic_B_Tree, Generic_Indefinite_B_Tree, Persistent.Memory_Pools.Streams.Generic_External_B_Tree, Persistent.Memory_Pools.Streams.Generic_External_Ptr_B_Tree provide subprograms Get_Tag and Set_Tag;
- The package Generic_B_Tree now has additional generic formal parameters Tag_Type and Initial_Tag;
- Subprograms to navigate tree buckets Get_Item, Get_Left_Child, Get_Left_Parent, Get_Right_Child, Get_Right_Parent, Get_Root were added to the implementations of B-Trees in the listed above packages;
- Functions Get_First and Get_Last were added to the implementations of B-Trees in the listed above packages;
- Procedures Store and Restore were added to the implementations of B-Trees in the listed above packages;
- The generic procedure Generic_Traverse and non-generic Travers were added to the implementations of B-Trees in the listed above packages to shallow and deep traversal of the tree items and buckets;
- Persistent.Memory_Pools lock is made reentrant;
- Image function was added to Persistent.Blocking_Files;
- Bug fix in encodings in Persistent.Blocking_Files.Transactional and Persistent.Memory_Pools;
- Bug fix in persistent B-tree implementations;
- Documentation extended.

Zip-Ada V.58

From: Gautier Write-Only Address <gautier_niouzes@hotmail.com>
Subject: Ann Zip-Ada v.58
Date: Sat, 27 Aug 2022 005822 -0700
Newsgroups: comp.lang.ada

* New in '58', 20-Aug-2022 [rev. 922]

- Support for Zip_64 archives.
The Zip_64 format extension is needed when there are more than more than 65535 entries or more than 4 GiB data for a single entry's compressed or uncompressed size, or for a whole archive.

***

Zip-Ada is a pure Ada library for dealing with the Zip compressed archive file format. It supplies:

- compression with the following sub-formats (methods) Store, Reduce, Shrink (LZW), Deflate and LZMA
- decompression for the following sub-formats (methods) Store, Reduce, Shrink (LZW), Implode, Deflate, Deflate64, BZip2 and LZMA
- encryption and decryption (portable Zip 2.0 encryption scheme)
- unconditional portability - within limits of compiler's provided integer types and target architecture capacity
- input archive to decompress can be any kind of indexed data stream
- output archive to build can be any kind of indexed data stream
- input data to compress can be any kind of data stream
- output data to extract can be any kind of data stream
- cross format compatibility with the most various tools and file formats based on the Zip format 7-zip, Info-Zip's Zip, WinZip, PKZip, Java's JARs, OpenDocument files, MS Office 2007+, Google Chrome extensions, Mozilla extensions, E-Pub documents and many others
- task safety this library can be used ad libitum in parallel processing
- endian-neutral IO

Main site & contact info:
http://unzip-ada.sf.net

Project site & subversion repository: https://sf.net/projects/unzip-ada/

GitHub clone with git repository: https://github.com/zertovitch/zip-ada

Azip V.2.50

From: Gautier Write-Only Address <gautier_niouzes@hotmail.com>
Subject: Ann AZip v.2.50
Date: Sun, 28 Aug 2022 003334 -0700
Newsgroups: comp.lang.ada

The version 2.5 of AZip is out!

URL: http://azip.sf.net/

AZip is a Zip archive manager.

Some features:

- Multi-document
- Flat view / Tree view toggle
- Simple to use (at least I hope so ;-))
- Useful built-in tools
  - Text & name search function through an archive, without having to extract files
  - Archive updater
  - Integrity check
  - Archive recompression, using an algorithm-picking approach for improving a Zip archive's compression.
- Encryption
- Zip compression formats supported Reduce, Shrink, Implode, Deflate, Deflate64, BZip2, LZMA
- Free, open-source
- Portable (in the sense no installation needed, no DLL, no configuration file)

Summary of latest changes since 2.15:

2.50: Support for Zip_64 archives.

2.40: Optional Windows Explorer context menu integration.

2.38: AZip is its own installer (if desired).

2.20: Drag & Drop for extracting Zip archive data. Stealth mode.

Full list: https://sourceforge.net/p/azip/news

Under the hood features

- AZip is from A to Z in Ada -)
- Uses the highly portable Zip-Ada library - all in Ada.
- (regarding Windows skin) Uses the GWindows library - all in Ada.

Conference Calendar

Dirk Craeynest

Department of Computer Science, 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. Although the situation seems to improve, a few events are still planned to be held "virtually", and some others in "hybrid" mode, also for wider dissemination. 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).

2022

October 03-06 (v) — 29th IEEE **Software Technology Conference** (STC'2022), Internet. Topics include: software engineering for emerging systems; software testing, testability, and assurance; cybersecurity and information assurance; agile software development; challenges and opportunities in SW & systems development processes; etc.

October 07-14 (h) — **Embedded Systems Week 2022** (ESWEEK'2022), Shanghai, China. Includes CASES'2022 (International Conference on Compilers, Architectures, and Synthesis for Embedded Systems), CODES+ISSS'2022 (International Conference on Hardware/Software Codesign and System Synthesis), EMSOFT'2022 (International Conference on Embedded Software).

- October 07-14 (h) — **International Conference on Compilers, Architecture, and Synthesis for Embedded Systems** (CASES'2022). Topics include: latest advances in compilers and architectures for high-performance, low-power embedded systems; software security for embedded systems, IoT, and CPS; 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.

- October 07-14 — **International Conference on Hardware/Software Codesign and System Synthesis** (CODES+ISSS'2022). 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 system synthesis of multi-processor hardware/software implementations.

- October 07-14 (h) — ACM SIGBED **International Conference on Embedded Software** (EMSOFT'2022). 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.

October 10-14 — 37th IEEE/ACM **International Conference on Automated Software Engineering** (ASE'2022), Oakland Center, Michigan, USA. Events include: ACM SIGAda's HILT workshop (High Integrity Language Technology) on Tools and Languages in support of a Rigorous Approach to Software Development.

☺ October 14 (h) — ACM SIGAda **High Integrity Language Technology International Workshop on Supporting a Rigorous Approach to Software Development** (HILT'2022), Ann Arbor, Michigan, USA. Co-located with ASE'2022. Organized by ACM SIGAda, in cooperation with Ada-Europe. Topics include: practical use of High Integrity languages, technologies, and methodologies that enable expedited design and development of software-intensive systems; practical use of formal methods at industrial scale; IDE-support for formal methods; model-level analysis tools for systems like SysML, AADL, Lustre, or Simulink; continuous integration and deployment based on advanced static analysis tools; safety-oriented programming language features; qualification of language tools for critical systems use; etc.

October 16-20 (h)
17th **International Conference on Software Engineering Advances** (ICSEA'2022), Lisbon, Portugal. Topics include: trends and achievements; advances in fundamentals for software development; advanced mechanisms for software development; advanced design tools for developing software; software performance; software security, privacy, safeness; advances in software testing; specialized software advanced applications; open source software; agile and Lean approaches in software engineering; software deployment and maintenance; software engineering techniques, metrics, and formalisms; software economics, adoption, and education; etc.

October 25-28 (v)
20th **International Symposium on Automated Technology for Verification and Analysis** (ATVA'2022), Beijing, China. Topics include: theoretical and practical aspects of automated analysis, synthesis, and verification of hardware, software, and machine learning (ML) systems; specifications and correctness criteria for programs and systems decision procedures and solvers for verification and synthesis program analysis and software verification analysis and verification of parallel and concurrent systems analysis of cyber-physical systems analysis and verification of machine learning algorithms and systems formal models and methods for security and privacy testing and runtime analysis based on verification technology applications and case studies verification in industrial practice; etc.

November 8-9
Ada-France at Paris Open Source Experience, Paris, France.

November 10-11 (v)
18th **International Conference on Formal Aspects of Component Software** (FACS'2022), Oslo, Norway. Topics include: applications of formal methods in all aspects of software components and services; formal methods, models, and languages for components and services, including formal aspects of concrete component-based systems, including real-time/safety-critical systems, hybrid and cyber physical systems, ...; tools supporting formal methods for components and services; case studies and experience reports over the above topics; etc.

November 14-18
30th ACM **Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering** (ESEC/FSE'2022), Singapore.

November 15-17
24th **International Symposium on Stabilization, Safety, and Security of Distributed Systems** (SSS'2022), Clermont-Ferrand, France. Topics include: concurrent and distributed computing (foundations, fault-tolerance, and security); distributed, concurrent, and fault-tolerant algorithms; synchronization protocols; formal methods, validation, verification, and synthesis; etc.

November 21-23
23rd **International Conference on Product-Focused Software Process Improvement** (PROFES'2022), Jyväskylä, Finland. Topics include: experiences, ideas, innovations, as well as concerns related to professional software development and process improvement driven by product and service quality needs.

Nov 28 – Dec 01 (v)
27th **Pacific Rim International Symposium on Dependable Computing** (PRDC'2022), Beijing, China. 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; dependability issues in computing systems (e.g. high performance computing, real-time systems, cyber-physical systems, ...); etc.

December 05-07
29th **Static Analysis Symposium** (SAS'2022), Auckland, New Zealand. In conjunction with SPLASH'2022 Topics include: static analysis as fundamental tool for program verification, bug detection, compiler optimization, program understanding, and software maintenance.

☺ December 05-08 (h)
43rd IEEE **Real-Time Systems Symposium** (RTSS'2022), Houston, Texas, USA. Topics include: addressing some form of real-time requirements such as deadlines, response times or delay/latency. Deadline for submissions: October 25, 2022 (Industry Challenge). Deadline for early registration: November 18, 2022.

Dec 05
5th **Workshop on Security and Dependability of Critical Embedded Real-Time Systems** (CERTS'2022). Topics include: security and dependability of cyber-physical and other real-time and embedded systems, vulnerabilities and protective

measures of CPS infrastructure, fault and intrusion tolerant distributed real-time systems, system architectures encompassing combinations of distribution, security, dependability and timeliness; etc. Deadline for submissions: October 3, 2022.

December 05-09 (h) 22nd IEEE **International Conference on Software Quality, Reliability and Security** (QRS'2022), Guangzhou, China. Topics include: reliability, security, availability, and safety of software systems; software testing, verification, and validation; program debugging and comprehension; fault tolerance for software reliability improvement; modeling, prediction, simulation, and evaluation; metrics, measurements, and analysis; software vulnerabilities; formal methods; operating system security and reliability; benchmark, tools, industrial applications, and empirical studies; etc. Deadline for submissions: October 1, 2022 (workshop papers), October 10, 2022 (fast abstracts, industry track, posters).

☺ December 05-10 (h) ACM **Conference on Systems, Programming, Languages, and Applications: Software for Humanity** (SPLASH'2022), Auckland, New Zealand. Topics include: all aspects of software construction and delivery, at the intersection of programming, languages, and software engineering. Deadline for early registration: November 15, 2022.

Dec 05-10 15th ACM SIGPLAN **International Conference on Software Language Engineering** (SLE'2022). 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: October 11, 2022 (artifacts).

☺ Dec 05-10 **Conference on Object-Oriented Programming, Systems, Languages, and Applications** (OOPSLA'2022). 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.

December 06-09 (v) 29th **Asia-Pacific Software Engineering Conference** (APSEC'2022), Japan. Topics include: agile methodologies; component-based software engineering; cyber-physical systems and Internet of Things; debugging and fault localization; embedded real-time systems; formal methods; middleware, frameworks, and APIs; model-driven and domain-specific engineering; open source development; parallel, distributed, and concurrent systems; programming languages and systems; refactoring; reverse engineering; security, reliability, and privacy; software architecture, modeling and design; software comprehension and traceability; software engineering education and training; software engineering tools and environments; software maintenance and evolution; software product-line engineering; software reuse; software repository mining; testing, verification, and validation; etc. Deadline for submissions: October 14, 2022 (posters), October 14-31, 2022 (workshop papers). Deadline for early registration: November 30, 2022.

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

2023

January 16-18 18th **International Conference on High Performance and Embedded Architecture and Compilation** (HiPEAC'2023), Toulouse, France. Topics include: software development for high performance parallel systems; tools for compilation, evaluation, optimization of high performance parallel systems (compiler support, tracing, and debugging for parallel architectures, ...); embedded real-

	time systems, mixed criticality system support, dependable systems, ...; software support for embedded architectures (tracing and real-time analysis of embedded applications, runtime software); etc.
☺ February 04-05	**Free and Open source Software Developers' European Meeting** (FOSDEM'2023), Brussels, Belgium. FOSDEM 2023 is an international two-day event (Sat-Sun 4-5 Feb), held in Brussels, Belgium. After our 11th Ada DevRoom earlier this year there won't be an Ada DevRoom in 2023, but Ada-related proposals can be submitted to several of the 53(!) accepted DevRooms...
March 06-10 (h)	25th **International Symposium on Formal Methods** (FM'2023), Lübeck, Germany. Topics include: development and application of formal methods in a wide range of domains including trustworthy AI, software, computer-based systems, systems-of-systems, cyber-physical systems, security, human-computer interaction, manufacturing, sustainability, energy, transport, smart cities, healthcare and biology; techniques, tools and experiences in interdisciplinary settings; experiences of applying formal methods in industrial settings; design and validation of formal method tools; formal methods in practice (industrial applications of formal methods, experience with formal methods in industry, tool usage reports, experiments with challenge problems); tools for formal methods (advances in automated verification, model checking, and testing with formal methods, tools integration, environments for formal methods, and experimental validation of tools); formal methods in software and systems engineering (development processes with formal methods, usage guidelines for formal methods, and method integration); special FM 2023 session on "Formal methods meets AI" (focused on formal and rigorous modelling and analysis techniques to ensuring safety, robustness etc. (trustworthiness) of AI-based systems); etc. Deadline for submissions: November 20, 2022 (artefacts), November 21, 2022 (Doctoral Symposium).
March 13-17	20th IEEE **International Conference on Software Architecture** (ICSA'2023), L'Aquila, Italy. Topics include: architecture evaluation and quality aspects of software architectures; model-driven engineering for architecture; component-based software engineering; automatic extraction and generation of software architecture descriptions; refactoring and evolving architecture design decisions and solutions; architecture frameworks and architecture description languages; linking architecture to requirements and/or implementation; architecture & continuous integration/delivery, and DevOps; training, soft skills, coaching, mentoring, education, and certification of software architects; architecture for legacy systems and systems integration; architecting families of products; roles and responsibilities for software architects; etc. Deadline for submissions: October 28, 2022 (technical track abstracts), November 4, 2022 (technical track full papers).
Mar 27 – Apr 02	38th ACM/SIGAPP **Symposium on Applied Computing** (SAC'2023), Tallinn, Estonia.
	Mar 27 – Apr 02 **Embedded Systems Track** (EMBS'2023). Topics include: the application of both novel and well-known techniques to the embedded systems development. Deadline for submissions: October 24, 2022 (full papers).
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. Deadline for submissions: October 28, 2022 (papers).
April 15-19	14th ACM/SPEC **International Conference on Performance Engineering** (ICPE'2023), Coimbra, Portugal. Deadline for submissions: October 14, 2022 (research track abstracts), October 21, 2022 (research track papers, industry track papers, (SPEC Kaivalya Dixit Distinguished Dissertation award nominations), January 7, 2023 (artifact track submission).
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. Deadline for submissions: October 20, 2022.

April 17-20	28th **International Working Conference on Requirements Engineering: Foundation for Software Quality** (REFSQ'2023), Barcelona, Catalunya, Spain. Theme: "Human Values in RE". Deadline for submissions: November 11, 2022 (research paper abstracts), November 18, 2022 (research papers).
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 submissions: October 13, 2022 (papers), November 10, 2022 (TACAS artefact submissions), January 5, 2023 (ESOP, FASE, FoSSaCS artefact submissions), January 16, 2023 (Doctoral Dissertation Award nominations.
	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. Deadline for submissions: January 9, 2023 (abstracts), January 16, 2023 (papers).
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 09-12 — 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 submissions: October 31, 2022 (papers), February 10, 2023 (brief presentations).
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. Deadline for submissions: December 9, 2022 (abstracts), December 16, 2023 (papers).
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. Deadline for submissions: October 21, 2022.
☺ 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 submissions: January 13, 2023 (abstracts 2nd round), January 17, 2023 (papers 2nd round).
♦ June 13-16	27th **Ada-Europe International Conference on Reliable Software Technologies** (AEiC 2023), Lisbon, Portugal. Sponsored by Ada-Europe.
December 10	Birthday of Lady Ada Lovelace, born in 1815. Happy Programmers' Day!

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

13-16 June, Lisbon, Portugal

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

13 February 2023	Submission deadline for journal-track papers
27 February 2023	Submission deadline for industrial-track papers, work-in-progress papers, tutorial and workshop proposals
20 March 2023	First round notification for journal-track papers, and notification of acceptance for all other types of submissions
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.

Conference Chair

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

Journal-track Chair

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

Industrial-track Chairs

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

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

Work-In-Progress-track Chairs

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

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

Tutorial and Education Chair

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

Workshop Chair

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

Exhibition & Sponsorship Chair

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

Publicity Chair

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

Webmaster

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

http://www.ada-europe.org/conference2023

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).

General information for submitting to the JSA can be found at the *Journal of Systems Architecture* website. The submission link will be available on the conference web page. Contributions must be submitted by **13 February 2023**. JSA has adopted the Virtual Special Issue model to speed up the publication process, where Special Issue papers are published in regular issues, but marked as SI papers. Acceptance decisions are made on a rolling basis. Therefore, authors are encouraged to submit papers early, and need not wait until the submission deadline. 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. Prospective authors may direct all enquiries regarding this track to the corresponding chair, Elena Troubitsyna (elenatro@kth.se).

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@fr.thalesgroup.com) and Sara Royuela (sara.royuela@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@sei.cmu.ed) and José Cecílio (jmcecilio@fc.ul.pt).

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@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@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.

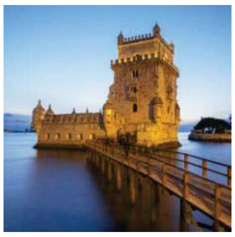
Call for Exhibitors

The conference will include a vendor and technology exhibition. Interested providers should direct inquiries to the Exhibition & Sponsorship Chair.

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!

AEiC 2023
Lisboa



Boosting Simulation and Debugging of Cyber-physical Systems with Symbolic Exploration

Ivan Kolesnikov
IRIT; email: ivan.kolesnikov@irit.fr

Cyber-physical systems (CPS) often have a mission critical nature; it is therefore mandatory to ensure their correct functionality at runtime. Simulation and testing are quite common approaches, however, for the most of real-life CPS they fall short to explore all possible execution scenarios, due to the very large or even infinite size of their state space. This well-known state-explosion problem is mainly due to two factors, namely, (i) the number of components and (ii) the number and the domain (i.e., type) of data variables manipulated within the CPS. Several techniques allowing to cope effectively with state-explosion have been developed in the past; in particular, symbolic exploration methods rely on specific encoding of system executions avoiding explicit enumeration of states, and therefore, provide opportunities to boost the performance of classical simulation and testing techniques.

One of these methods is *Bounded Model Checking* (BMC) [1, 2, 3] which allows to effectively represent *all* system executions consisting of a specific number of consecutive steps and to verify related reachability properties. System executions are typically encoded as SMT formulae [4] or MILP formulae [5]). First, such formulae are obtained by replicating constraints of the form $T^{step}(x_i, y_i, \ldots, x_{i+1}, y_{i+1}, \ldots)$ encoding the i-th execution step on variables like x_0, x_1, x_2 which represent the value of a model variable *x* on each step 0, 1, 2, etc. For the first and the last step some extra constraints are added e.g., defining initial and final states of interest. Second, SMT solvers (such as Z3, CVC4, etc) or MILP solvers (such as CPLEX, Gurobi, etc) are used to check for satisfiability of such formula. If the formula is satisfiable the solver provides concrete values for each variable, representing one possible execution trace from a state in the initial set to a state in the final set. Otherwise, the solver provides a subset of contradictory constraints (i.e., an unsatisfiable core), that is, a minimum set of constraints which cannot be met together.

The use of BMC on real systems has also some known limitations. For example, the number of steps to consider may need to be limited, the types and operations on data variables may need to be restricted in order to obtain formula manageable by existing solvers. Also, the counter-examples extracted when formulas are not satisfied can be counter-intuitive. So it might be difficult to explore a real scale system just using BMC. In this work, we propose the development of the theoretical approach and tools to reach two goals:

1. use BMC as a tool for local search of the state space, helping the user to reach states of interest during explicit state simulation, or to verify that certain conditions are unsatisfiable when starting from the current simulation state.
2. whenever a property is not satisfied, leverage the unsatisfiable core to provide user with some explanation of the counter-example to help him understanding the reason of failure and guiding towards finding a fix, if possible.

We plan to practically implement our approach on the top of TASTE [6] a tool-chain targeting heterogeneous embedded systems, using a model-based development approach. We are currently working on supporting BMC for TASTE models, more precisely for encoding the semantics of TASTE components defined using the SDL graphical programming language as SMT constraints.

Acknowledgement

The paper is written as a part of PhD thesis "Boosting simulation and debugging of Cyber-Physical Systems (CPS) models with symbolic analysis methods" under the supervision of Iulian Ober (ISAE-Supaero, Université de Toulouse) and Marius Bozga (CNRS-VERIMAG, Université Grenoble Alpes).

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207, Springer, 1999.

[2] M. Sorea, "Bounded model checking for timed automata," *Electronic Notes in Theoretical Computer Science*, pp. 116–134, 01 2003.

[3] M. Fränzle and C. Herde, "Efficient Proof Engines for Bounded Model Checking of Hybrid Systems," *Electronic Notes in Theoretical Computer Science*, vol. 133, pp. 119–137, May 2005.

[4] A. Cimatti, S. Mover, and S. Tonetta, "Smt-based scenario verification for hybrid systems," *Formal Methods in System Design*, vol. 42, pp. 46–66, 02 2013.

[5] I. Ober, "Revisiting bounded reachability analysis of timed automata based on milp," *Formal Methods for Industrial Critical Systems*, p. 269–283, 2018.

[6] "A tool-chain targeting heterogeneous embedded systems, using a model-based development approach." `https://taste.tools/`.

Improving Usability and Trust in Real-Time Verification of a Large-Scale Complex Safety-Critical System*

Brian Kempa, Chris Johannsen, Kristin Yvonne Rozier
Iowa State University, Ames, Iowa, USA; email: {bckempa,cgjohann,kyrozier}@iastate.edu

Abstract

Large-scale complex safety-critical systems are inherently difficult to both verify in real-time and transparently validate. The iterative specification development process is challenging when the performance and reliability demands of target systems (e.g., flight software) require strict behavior of verification tools which often trade off usability for performance and conformance. Providing both strict behavioral guarantees and efficiency of this iterative process allows specification authors and engineers to more quickly deploy their systems and have more confidence in their verification efforts.

Our on-going work addresses this challenge by providing validation transparency for specification authors during system development while maintaining necessary performance during deployment by extending R2U2, a real-time verification tool specifically designed for resource-constrained systems. We also strengthen the trust in R2U2 by providing a robust suite of tests to show adherence to the strict requirements of safety-critical flight software. These tasks are efforts toward transitioning R2U2 from a research-grade tool to a flight-software-grade tool suitable for use by real-time safety-critical systems and thereby answer the calls for expanded developmental-to-operational verification by, e.g., the Vehicle System Management (VSM) team of the NASA Lunar Gateway.

Keywords: Real-Time and Safety-Critical Systems, Runtime Verification, Developmental Contract Verification, Assume-Guarantee Contracts.

1 Introduction

Complex autonomous real-time systems such as robots, rovers, satellites, and unmanned aerial systems (UAS) must operate reliably for extended periods without human intervention. Runtime verification is a family of techniques that enable such systems to check themselves during operation by identifying and correcting problems as they occur. The legacy approach to runtime verification in software is to use custom ad-hoc algorithms that are difficult to implement, susceptible to errors, and extremely difficult to verify [1]. Large-scale complex safety-critical systems require both real-time verification during system operation but also transparent requirement validation during system development that can carry through to runtime.

After an extensive survey of all currently-available verification tools, the NASA Lunar Gateway Project selected the R2U2 runtime verification engine for use in developing and monitoring autonomous spacecraft software, starting with the Vehicle System Manager (VSM) [1]. The choice was based primarily on R2U2's unobtrusive, flight-certifiable architecture, proven capacity for real-time runtime verification on-board safety-critical systems, and an open-source, extensible C codebase that integrates into the NASA core Flight System/core Flight Executive (cFS/cFE) [2] environment [3]. A hardware version of R2U2 that implements the same algorithms as the C version previously embedded in the space left over on the FPGA controlling NASA's Robonaut2's knee to provide real-time fault disambiguation [4]. The three implementations of R2U2 (hardware/FPGA, C, and C++) have verified many previous safety-critical systems; see [5] for a tool overview and summary of previous case studies. R2U2's underlying specification-monitoring algorithms were originally created specifically to fulfill NASA's needs for a Responsive, Reliable, Unobtrusive Unit (hence the name R2U2) [6], and optimized (with accompanying proofs of correctness) for the Robonaut2 study [4].

While design-by-contract systems like SPARK have provided formal verification in this domain [7], VSM focused on standalone monitors for their verification efforts because they sought runtime visibility of system status instead of design time prescription of component correctness, therefore their selected tool needed to be independent from the flight software implementation [1]. The VSM team has an established verification workflow that includes extensive requirement elicitation in the form of Assume-Guarantee Contracts (AGCs), and the design-time verification technique of model checking to verify AGCs against state-machine models of various sub-systems. However, specification (of models and their requirements) is the biggest bottleneck to verification of autonomy [8]. Developing a system model of required fidelity to fully leverage model checking involves significant effort and expertise, so only some AGCs are suitable for model checking

*This work was supported in part by NASA Cooperative Agreement Grant #80NSSC21M0121 and NSF CAREER Award CNS-1552934.

and others instead undergo a lighter-weight validation analysis with a modified form of runtime verification via R2U2 during development phase instead [9]. The end-goal is to validate AGCs by simulating the possible system executions that satisfy them during system development time, then verify them over simulated system runs (developmental verification) and eventual mission-time execution (operational verification); see [10] for a description of the distinction between developmental runtime verification and simulation.

We describe ongoing work extending the open-source, publicly-available runtime verification engine R2U2, to enable its use for public purposes that are relevant to NASA, including enabling system designers to transparently capture their desired requirements, and making verification results accessible to users. We aim to enhance R2U2 to make it more accessible to software developers and to make R2U2 output tie transparently to the input AGCs. Since the goal is to measurably increase R2U2's usability, user documentation/example uses, and both input and output interfacing, we are evolving R2U2 with regular feedback from a representative NASA mission, in this case, the Lunar Gateway Vehicle System Manager (VSM) team, as an outside check that these goals are being accomplished.

This report previews ongoing work expanding R2U2's usability and trust as a runtime verification engine optimized for a minimal resource footprint running on-board safety-critical systems. Figure 1 displays the features discussed here bounded by the dashed box. The contributions of this paper are (1) extensions to the input and output formats of R2U2 to improve usability when validating and verifying complex specifications and (2) increasing trust in the underlying runtime monitor of R2U2 while maturing research software for flight. Section 2 explores extensions to R2U2's input and output syntax to facilitate specification authorship and validation. Our approach to preparing research software for flight is highlighted in Section 3. Finally, Section 4 summarizes work in progress and next steps.

2 Usability

R2U2 utilizes a minimal input/output syntax to meet real-time deadlines in resource-constrained embedded systems. This low-level syntax makes authoring specifications, validating specifications against requirements, and interpreting verification results more difficult for system engineers. As systems scale in complexity (and often criticality) the mental overhead quickly becomes untenable; however, intuitive naming and reporting schemes can improve specification transparency, reducing iteration time during specification and system development. We extend R2U2's specification syntax with three enhancements to facilitate human validation: Assume-Guarantee Contracts (AGCs), set aggregation operations, and more readable syntax. These ergonomic improvements impact efficiency in authoring and validating specifications, as shown in the "develop" path in Figure 1.

2.1 Assume-Guarantee Contracts

VSM uses AGCs to capture requirements [1]. AGCs are a simple yet powerful requirement framework, but encoding them

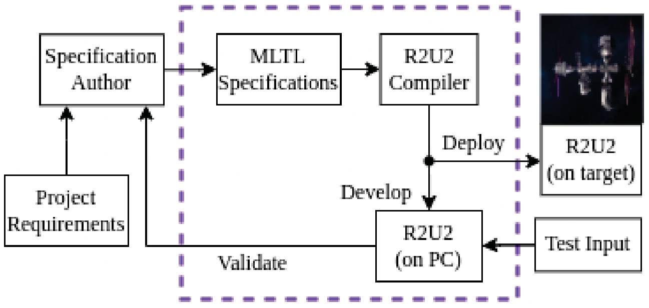


Figure 1: Workflow of the design, validation, and deployment of R2U2 on board the Lunar Gateway where the components in the dashed box are those that the contributions of this paper refer to. A specification author translates the set of AGC project requirements to a set of Mission-time Linear Temporal Logic (MLTL) specifications, compiles these specifications into an R2U2 configuration, then tests and validates the configuration locally. Once the author validates a configuration, engineers then deploy the configuration onto the target platform.
(Photo of Lunar Gateway, `https://flic.kr/p/2mHPaLg`, by NASA/Alberto Bertolin CC BY-NC-ND 2.0 / Cropped)

using logical implication ($assumption \rightarrow guarantee$) conflates inactive and verified contracts because R2U2's verdicts are Boolean.

The "Old Syntax" in Figure 2 demonstrates a workaround encoding the three conditions as separate formulas in a "one-hot" pattern such that one and only one is true at each timestep. In contrast, we implmented an AGC operator "$\Rightarrow$" in R2U2, which efficiently computes the trinary status and outputs a single clear verdict. The new syntax is easier to write, read, and validate.

2.2 Set Aggregation

We also implement set-aggregation operations as first suggested in [11], permitting succinct requirements over sets of expressions. The set-aggregation operators are syntactic sugar and eliminate long repetitive structures that may hide typos. They accept a set of expressions and evaluate a set property such as "exactly one of these" and are simpler for system engineers to write, interpret, and validate.

For example, a requirement may state that exactly one task shall be active simultaneously. The "Old Syntax" in Figure 2 demonstrates writing this as a disjunction of conjunctions that grow exponentially with the number of tasks, while the "New Syntax" captures this succinctly with a set-aggregation operation.

2.3 Syntax Readability

Internally, R2U2 addresses all values by their index positions in internal vectors. Accordingly, interpreting the output of the old syntax seen in Figure 2 requires knowing the formula number, and mentally mapping that back though the atomic number to the input signal number, increasing the complexity of writing, reading and validating specifications.

Our new syntax and tooling support human-readable labels for formulas, variables, and subexpressions. Named subexpressions allow specifications to resemble the requirements they monitor more closely, while formula names carry through

Old Syntax

```
a0 && ((a1 && !a2 && !a3) || // AGC:
       (!a1 && a2 && !a3) || // TRUE
       (!a1 && !a2 && a3));
!a0; // AGC: INACTIVE
a0 && !((a1 && !a2 && !a3) || // AGC:
        (!a1 && a2 && !a3) || // FALSE
        (!a1 && !a2 && a3));

a0 = bool(s0) == 1;
a1 = bool(s1) == 1;
a2 = bool(s2) == 1;
a3 = bool(s3) == 1;
```

New Syntax

```
RVALID: resRactive => resRvalid;

taskAactive = bool(Aactive) == 1;
taskBactive = bool(Bactive) == 1;
taskCactive = bool(Cactive) == 1;
resRactive = bool(Ractive) == 1;
resRvalid =
  exactly-one-of(active_tasks) == 1;

active_tasks = {taskAactive,
                taskBactive,
                taskCactive};
```

Input

Time	s0 resRactive	s1 taskAactive	s2 taskBactive	s3 taskCactive
0	T	T	F	F
1	T	F	T	F
2	F	F	F	F
3	T	T	T	F

Old Output	**New Output**
0:0 T 1:0 F 2:0 F	RVALID:0 TRUE
0:1 T 1:1 F 2:1 F	RVALID:1 TRUE
0:2 F 1:2 T 2:2 F	RVALID:2 INACTIVE
0:3 F 1:3 F 2:3 T	RVALID:3 FALSE

Figure 2: An example usage of R2U2 with the old and new syntaxes. The specification shown captures the system behavior that when a shared resource R is active, exactly one task is using that resource. There is no native support for AGCs and variable names in the old syntax so the specification must be written without these features i.e., each case of the AGC must be explicitly written out and each variable uses a generic name. The new syntax adds these features and as such is more human-readable and easier to validate.

to the output stream, both easing validation. Because the VSM team selected R2U2 for its real-time performance and bounded resource guarantees under flight software restrictions [1], these ergonomic improvements cannot impact the deployed monitor performance. Most of these features only affect the formula compiler, but human-readable output like formula names requires auxiliary information and runtime actions. While development and deployment workflows utilize the same specification files, R2U2 now stores auxiliary data like formula names separately. Deployment monitors do not compile the auxiliary output hooks or read the auxiliary data files, leaving them strictly more performant than development builds, under equivalent conditions.

Additionally, we added an option to dynamically map input signals by the name used in the specification. This added input robustness decouples specification authorship from engineering decisions until target integration, i. e., changing structure definitions no longer requires specification modification.

3 Trust

Academic research software ("gradware") is developed under different motivations than projects targeting third-party use, and unpublishable custodial tasks (e.g., documentation, testing) are often are not attended to beyond what is required for peer acceptance. Software deployed in critical applications, however, must meet a higher bar than standard software best practices. As we convert R2U2 from a research tool to a flight-certified component, we establish trust in R2U2's output with a hierarchical testing campaign, automated analysis-guided peer-review, and adherence to open-source best practices.

3.1 Testing

Our new R2U2 test suite design supports fast iteration as we react to VSM's needs and meet established flight software verification standards, bridging traditional and formal methods.

Unit Tests: Following NASA's standards for VSM flight software, unit tests verify individual functions (e. g., queue operations) and must exercise every line and branch. We parameterize tests over the Cartesian product of the input parameters, covering the input space without repeated code.

R2U2's 66 unit tests currently cover 98.1% of the 577 executable lines and 52.3% of the 2276 branches. The low branch coverage results from a standard C macro idiom for debug print statements that create a do-while structure that can never repeat, generating an unreachable jump instruction. Crucially, these spurious branches do not appear in deployment binaries.

Integration Tests: These black-box tests confirm implementation correctness by comparing the output of R2U2 with a slower but simpler Python oracle over a benchmark set with 2000+ combinations of formulas and input signals. We curate this collection to exercise all logical operators in varied compositions, including published and randomly-generated benchmark specifications. A core set of 50 acceptance tests that cover common cases and check for regressions of previous issues runs in under a minute on consumer hardware. Although the total space of formulas and inputs is infeasible to cover exhaustively, we "fuzz" for edge cases beyond the curated set with randomized inputs and formulas.

3.2 Automated Analysis and Review

GitLab provides version control; all changes automatically trigger the *Continuous Integration* (CI) server, which scans the source with linters and static analysis tools, builds a debug binary with maximum compiler warnings, and runs both test suites with the sanitizer runtimes linked to catch memory mistakes not detectable at compile time. We use CodeChecker[2] to aggregate analysis results from Clang Tidy, CLang Static Analyzer, Cppcheck, Infer, and cpplint. The CI report assists in finding potential defects during code reviews. CI does not measure performance since benchmarks are highly sensitive to environmental context (e. g., working directory, cache alignment, etc.) [12]. Instead, profile-guided optimization is performed on integration target hardware.

3.3 Release Best Practices

Though R2U2 is already open source, code availability is insufficient to ensure the project remains maintainable and accessible for developers of R2U2 and projects incorporating using it. Popular open-source libraries solve this problem with a series of best practices R2U2 is adopting: an open Git repository with full version history, public issue tracking, an established open license, and documentation targeting both users. These tasks are vital to transitioning any research-grade software to software suitable for flight. Beyond the existing in-line comments, we are preparing three documents: 1) a user's guide detailing the use of R2U2 (e.g., formula syntax, output format, target platform integration), 2) a developer's guide with architectural decisions, code style, and algorithm descriptions with proofs, and 3) an API reference autogenerated from the source using Doxygen.

4 Conclusion

NASA's VSM team is actively developing specifications for the Lunar Gateway using our tool chain. The new usability and trust features are crucial for the transition of R2U2 from a research-grade academic tool to one suitable for safety-critical flight-software systems. We continue to collaboratively evaluate user needs, modify the tool accordingly, and monitor the effectiveness of delivered solutions. We are looking forward to insightful experience reports and technical evaluations at the end of the project.

Additionally we are working on: 1) Adding an optimization pass to formula compilation that removes unnecessary instructions (e. g., double negations) and improves partial result reuse to improve performance and reduce resource requirements. 2) Building a visual configuration utility for tuning the static memory bound parameters that provides statistics on formula resource usage. This is also useful when designing new formula sets for a monitor with existing bounds.

[2]https://github.com/Ericsson/codechecker

References

[1] J. B. Dabney, J. M. Badger, and P. Rajagopal, "Adding a verification view for an autonomous real-time system architecture," in *AIAA Scitech 2021*, p. 0566, 2021.

[2] NASA, "core Flight System (cFS) Background and Overview." Online: https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf, 2014.

[3] J. B. Dabney, P. Rajagopal, and J. M. Badger, "Using assume-guarantee contracts in autonomous spacecraft." Flight Software Workshop (FSW) Online: https://www.youtube.com/watch?v=zrtyiyNf674, February 2021.

[4] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, "Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2," in *FORMATS, Proc. 18th*, vol. 12288 of *LNCS*, (Vienna, Austria), pp. 196–214, Springer, September 2020.

[5] K. Y. Rozier and J. Schumann, "R2U2: Tool Overview," in *RV-CUBES*, vol. 3, (Seattle, WA, USA), pp. 138–156, Kalpa Publications, September 2017.

[6] T. Reinbacher, K. Y. Rozier, and J. Schumann, "Temporal-logic based runtime observer pairs for system health management of real-time systems," in *TACAS, Proc. 20th*, vol. 8413 of *LNCS*, pp. 357–372, Springer, April 2014.

[7] P. Neto, J. Tojal, J. Veríssimo, and S. M. de Sousa, "Towards a formally verified space mission software using spark.," *Ada User Journal*, vol. 40, no. 4, pp. 243 – 246, 2019.

[8] K. Y. Rozier, "Specification: The biggest bottleneck in formal methods and autonomy," in *VSTTE, Proc. 8th*, vol. 9971 of *LNCS*, (Toronto, ON, Canada), pp. 1–19, Springer-Verlag, July 2016.

[9] J. B. Dabney, P. Rajagopal, and J. M. Badger, "Using assume-guarantee contracts for developmental verification of autonomous spacecraft." Flight Software Workshop (FSW) Online: https://www.youtube.com/watch?v=HFnn6TzblPg, February 2022.

[10] K. Y. Rozier, "From simulation to runtime verification and back: Connecting single-run verification techniques," in *SpringSim*, (Tucson, AZ, USA), pp. 1–10, Society for Modeling & Simulation Int'l, April 2019.

[11] A. Hammer, M. Cauwels, B. Hertz, P. Jones, and K. Y. Rozier, "Integrating runtime verification into an automated uas traffic management system," *Innovations in Systems and Software Engineering: A NASA Journal*, July 2021.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 265–276, 2009.

Use of Graph Databases for Static Code Analysis

Q. Dauprat[1,2] ***, P. Dorbec***[1] ***, G. Richard***[1] ***, JP. Rosen***[2]
1: Université de Caen Normandie, Campus 2, Boulevard Maréchal Juin, 14000 Caen, France;
2: Adalog, 2 rue du Docteur Lombard, 92130 Issy les Moulineaux, France; email: 1:
{quentin.dauprat,paul.dorbec,gaetan.richard}@unicaen.fr; 2: {dauprat,rosen}@adalog.fr

Abstract

This paper deals with static code analysis. With analysis needs becoming more and more complex, and code volumes getting bigger and bigger, scalability of code analysis tools is becoming one of the current challenges. We explore the use of recent technologies, like graph databases, to represent the source code and pattern matching to find information into a graph. We hope that this will reduce the time to analysis a source code and improve the effectiveness of the analysis. When trying to answer the same query compared to AdaControl, we manage to find results that were missed by the programatic approach. We expect further improvement on future benchmarking.

Keywords: Ada, Static analysis, ASIS, graph databases, Neo4J, AST

1 Introduction

In some activity sectors like industry (railway, avionics, space), the life cycle of the programs can extend over several decades. Over time, programs become huge and complex. With many engineers working on the code, the needs of coding guidance, and more generally, code analysis, is at the heart of the concerns.

It is in this objective that code analysis tools were created. This kind of software allows to check the quality of the code, to validate the implementation, to find performance issues, etc. Since their emergence in the 70s, static code analysis tools kept working on the same structure to analyze the source code, namely an Abstract Syntax Tree (AST). The problem with AST is that we have often needs to re-explore previously visited nodes (sub-tree) to find related information, a variable to this declaration, a type to his definition, etc. We cannot store the information when we firstly explore it, because we do not know in advance what we really need, and this can take a lot of memory. Therefore, AST is not perfectly suited to the needs of static code analysis.

Furthermore, more and more needs for advanced analysis have emerged [1,2,3], and, consequently, the complexity of analysis induces a long analysis time. The scalability of static analysis tools become one of the current challenges [4,5].

Ada programming language is an interesting subject of study, since all its complexity has been delegated to the compiler. As a result, it is very difficult to compile, and the resulting AST is heavy and complex, making it complicated to understand and to query.

In this work-in-progress paper, we try to take advantage of recent technologies (graph databases) to represent the source code, with Code Property Graph (CPG) [6], and pattern matching to find information into a graph. Our goal is to decrease the time of analysis of a source code and improve the effectiveness of the analysis. Lastly, Ada is a good starting point for our study, and we hope that will lead us to provide a general approach for static analysis of other programming languages.

2 Background

Code analysis is the process of analyzing a program (its source code or its execution). Here, we focus on static analysis that only analyzes the source code. It is mainly used for the computation of code metrics (cyclomatic complexity, etc.) and checking coding rules.

Nearly all static code analysis software works with a structure which is globally the same: an Abstract Syntax Tree (AST). This structure is generally provided by a compiler, or hand built by this software. The resulting AST can be complex and hard to query, especially in the case of Ada. Unfortunately, we cannot provide a simple but complex (in terms of analysis) example of such structures.

3 Approach

The main goal of this research is to take advantage of a new way to represent a source code for static code analysis, with the main objective of reducing the time of analysis, notably for scalability.

3.1 Structures and definitions

Our approach is to use a directed graph to represent the code. We call it a Code Property Graph (CPG) [6]. A property graph is a directed graph where nodes and edges can have properties, whereas in an AST only nodes have them. The following figure shows an example of a property graph:

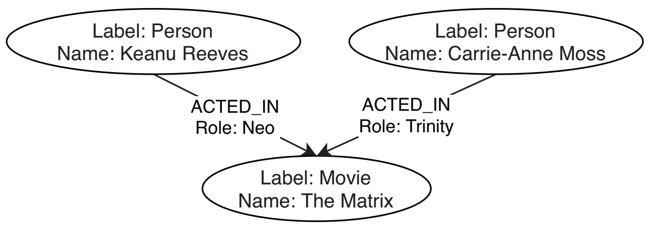


Figure 1: Example of a Property Graph

We can see the CPG as a structure that encompasses AST, the call graph (calls of subprograms), the dependency graph, and many more, simply by adding relations between nodes of the previous AST. The basic intuition is that we use additional edges (with properties) to represent semantic and non-local information. One simplest example is that we can add an (usage) edge from any reference of a variable to the definition of the variable. Enriching the graph with semantic information allows thus easier checking properties. However, the main challenge is the trade-off between the gain induced by using the new edges and the cost resulting from the increase in size of the graph. For the former, we identify below some edges concepts that seems to be the most promising. For the latter, the recent development of graph databases (see next section) gives high hopes of keeping efficient queries. In fact, our experiment search to give a realistic proof of concept to validate the benefits of using CFG.

Furthermore, where in an AST we have a fixed starting node (root of the node), that can be possibly split by compilation units, with a graph, there is no fixed starting node. So we have to choose from which node of the graph we want to start our queries. This can be very convenient to target some kind of nodes by using indexes.

By taking advantage of our experience at Adalog, we have introduced new relationships that we found relevant from a query point of view. These relationships are based on common usage we have to often deal with in static analysis, and from which require deep search to find the corresponding information. The added relationships are:

- **CORRESPONDING_NAME_DEFINITION**: Given an expression of type *Identifier*, *Operator Symbol*, *Character Literal* or *Enumeration Literal*, this refers to where its name has been defined.

- **IS_OF_TYPE**: Given a *component, constant (dereferenced), discriminant specification, formal object, parameter (loop) specification, constant/variable return specification, simple tasks/variable protected object, element iterator specification, object renaming declaration*, this refers to the type associated with this declaration.

- **CORRESPONDING_INSTANCIATION**: this indicates whether an element is part of a generic instantiation.

- **CORRESPONDING_PARAMETER_-SPECIFICATION**: Given *A_PARAMETER_-ASSOCIATION* this points out the corresponding *A_PARAMETER_SPECIFICATION*. This is useful to get all information about a parameter, notably since the order of parameters can differ between the user and the specification.

Nevertheless, we have to keep a number of kinds of relationships as few as possible in order to not increase the size and the time of creation of the database in a disastrous way.

3.2 Graph databases

In addition to this emerging structure, we can take advantage of recent technologies for the scalability. With the rise of NoSQL databases, one type of database appears to be particularly suited to our needs : Graph Databases. A graph Database (GDB) is a kind of non-relational database where data is stored in a graph structure. This kind of database rely on pattern matching to find information. Unlike relational databases, there is no costly joins or foreign keys in a GDB. Here, we can see relations as a pointer. Furthermore, where an AST is stored in files or RAM, a GDB stores the graph in files and RAM but in a way that makes it efficient for querying, especially thanks to indexes.

We believe that the use of Graph Databases could solve current problems of static code analysis, notably for the scalability, and thus, its ability to quickly analyze a large volume of code. Regarding the scalability, some studies [7, 8] demonstrate that GDB is easily scalable for representing the source code.

Nevertheless, GDB lack a standardized query language. Indeed, there is no universal query language like SQL to query any Graph Database Management System (GDBMS). However, a movement of standardization has been launched with GQL (Graph Query Language) [9, 10] and an ISO normalization is on the way.

In this study, we decided to use the Neo4j database. **Neo4j** is a GDBMS written in Java. It is the world leader in directed graph databases. It has its own query language, Cypher, allowing expressing queries in ASCII art (visual) form, like `(node)-[RELATED_TO]->(anotherNodes)`, where parentheses are used for nodes and braces for relationships. We do not believe that the use of one graph database rather than another has a significant impact on this research.

3.3 Thinking about the use of GDB versus the AST approach

We think that the graph approach is interesting because, on the one hand, we have to query the whole AST to get some information, on the other hand, only a specific portion of the graph is queried, thanks to indexes, which only contains the nodes we are interested. To illustrate that the use of GDB comes across as to be a promising approach, we introduce the following example: Given a program, with multiple compilations units, we want to list every declaration of a function that returns a specific type A. With an AST, we have to traverse the whole AST to find every function declaration, and apply a filter to find every function that returns the A type. With a GDB, we can directly target any function declaration nodes, and apply a filter to find every function that returns the A type.

Now, a user may want to make another query, to control that none of the function of a program returns a specific type B. With an AST, we can make sure that the controls follow each other, and thus, only traverse the AST once. Using GDB, we can take advantage of another useful feature, the cache. Since we have to start with the function declaration (like the previous query), this result is stored in the cache and can be reused with the new query.

4 Experiments

4.1 Example of Ada code into Neo4j

As an illustration of our research, one example can be given by the code below that is stored in Neo4j as depicted in the figure:

```
package Pack is
   A1, A2 : Integer;
   A3 : Integer range 1..10 := 1;
end Pack;

-- Later in the program

Pack.A1 := Pack.A3;
```

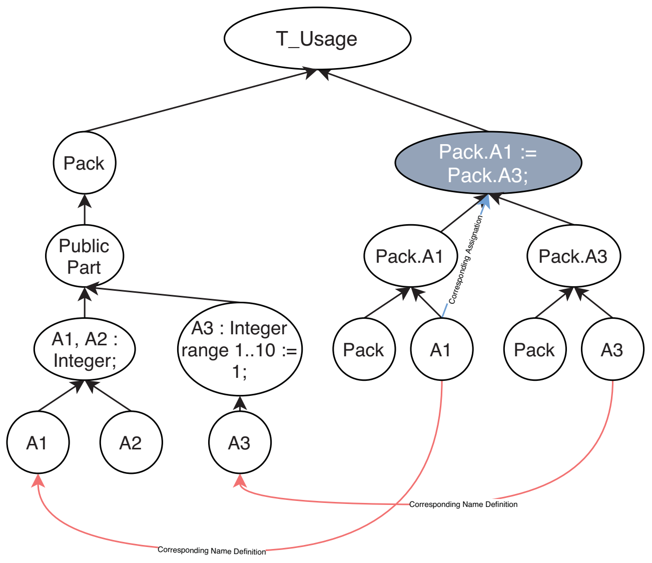

T_Usage
Pack
Pack.A1 := Pack.A3;
Public Part
Pack.A1
Pack.A3
Corresponding Assignation
A1, A2 : Integer;
A3 : Integer range 1..10 := 1;
Pack
A1
Pack
A3
A1
A2
A3
Corresponding Name Definition
Corresponding Name Definition

Figure 2: Example of program stored into Neo4J

Note that in the figure, only some nodes are represented. Black edges represent the "IS_ENCLOSED_IN" relation, which is the relation from one node to its parent. This is the only relation which is present in the AST, all other (colored) edges are relations we have created. The gray node is the starting node, where we start the query.

With this tiny example, we could formulate the following request: Given an assignment statement, I want to access to the declaration of the assigned variable. Using a GDB, we start by retrieving all assignment statement (thanks to indexes), and we follow the added relation **CORRESPONDING_ASSIGNATION**, then **CORRESPONDING_NAME_DEFINITION** to finally go to the parent node (using the dark arrow) to obtain the result. Using an AST, we would have been forced to only use black arrows. Furthermore, we have to start from the root node, and exploring child in prefix order. When we find an assignment statement, we have to explore the previously visited sub-tree to retrieve the corresponding declaration. As a reminder, not all the nodes of the graph are represented on the previous figure, so the process of retrieving the node would be costly.

The next figure present the result of the following Cypher query, which obtains the declaration of the variable from the assignment of a variable:

```
MATCH (a:AN_ASSIGNMENT_STATEMENT)
<-[:CORRESPONDING_ASSIGNATION]-
(corrAssi:AN_IDENTIFIER)
-[:CORRESPONDING_NAME_DEFINITION]->
(Id:A_DEFINING_IDENTIFIER)
-[:IS_ENCLOSED_IN]->
(varDecl:A_VARIABLE_DECLARATION) RETURN *
```

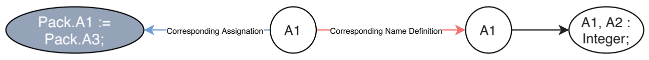


Figure 3: Result of the Neo4J query

Where we would have been forced to explore the whole AST to find the usage of all variables, with a graph we can start from a specific kind of node (the variable declaration for example, or the variable assignment in the previous example) to retrieve all information about the variables, without exploring the whole graph. This is possible thanks to indexes, and because there is no notion of "root" for starting point in a graph, and thanks to the extra edges in the graph.

4.2 Our experiment

To make our experiments measurable, we are trying to implement some AdaControl's rules, from simple rule, to more complex, in terms of analysis. Currently, we try to answer the following query: "For each variable of a program, is the variable is **READ** and/or **WRITE**?"

The first step will be to populate the database with an Ada program. We started from the source code of AdaControl where we have changed the analysis of rules by populating the database. The goal of starting with the source code of AdaControl is to reduce the time on prototyping, since AdaControl uses the Ada Semantic Interface Specification (ASIS) in the core to query the source code. Thereafter, we wrote the query using Cypher (the query language of Neo4j) and compare the results with what AdaControl obtains.

The first experiment has revealed that our approach found more results than AdaControl (AdaCtl 1.22r15). We note that using Cypher and thus "pattern matching" to query the code, we found some cases that were forgotten in the traditional (programmatic approach). This is due to a missing case in AdaControl. The current query can find usage of variables in any context (normal, in generic packages and in instantiation), but it does not currently support usage within a *renames*, since this case is tricky because of pointers, arrays, etc. that can be hidden inside the *renames*.

4.3 Current limitations of our approach

Even if the use of the graph database is convenient for the scalability and so, the response time, there are some drawbacks.

The first one is the extra time required for populating the database compared to simply building the AST. Therefore, we accumulate the time to create the AST by ASIS, plus the time to traverse the whole AST again by adding relations and sending it to Neo4j. Though we made no precise timing comparison, our database initialization was visibly slower than the computation of the AST. So, on the one hand, we can have a quick answer to the query, but, on the other hand, the time to populate the database is slow. We are currently focusing on reducing code analysis time, and we do not take into account the time required to populating the database. An improvement will be to populate the database during the creation of the AST, but this will require to create our own parser or to use a different library than ASIS, like libadalang to process the source code. This is currently out of our research.

Secondly, though Cypher allows to express complex queries, it is quite a verbose query language. Indeed, the query made to answer our ”simple” question about usage of variable took more than 200 lines, without considering the special case of **renames**, that would probably double the number of lines. An improvement could be to split the query into several elementary queries, or to generate the query using an intermediate (programming) language.

Moreover, graph databases use pattern matching approach to query the graph, compared to the tree traversal of traditional approach. Event if this approach allows finding some missed case of traditional approach, we are not immune to false negatives.

Finally, we have to keep the number of relationship types and indexes as small as possible in order to reduce as much as possible the impact in size of the database, to be more manageable, and not to sky rocket the time required to populating the database.

5 Related work

During the last decade, more and more research focused on the representation of the code in order to perform code analysis, but mainly for the search of vulnerabilities [6, 11]. Even if [6] focuses on vulnerabilities, it introduces Code Property Graph, for which it is referenced in numerous research (more than 220). This research ([6]) led to the creation of a company specialized in the detection of vulnerabilities in code. There tool is fully based on the CPG introduced into their research.

In his thesis [5], Fan introduces three hard-to challenges: hard-to-employ, hard-to-scale, and hard-to-be-recognized. Our study is focused on the hard-to-scale challenge. He explored different ways to improve the scalability, but the use of graph databases has not been discussed. In [4], authors take advantage of their experience and their industrial vision at Facebook to provide an overview of current challenges and opportunities of static and dynamic code analysis.

Ramler et al. [8] have studied the use of graph databases for various kinds of code analysis, for Java, C, C++ and C#. However, the tools used in the study sound proprietary or non-disclosed.

Furthermore, some studies are focused on the query language. Alves et al. [12] establish a comparison of different query languages, but it looks like that the literature on this subject is quite poor, and it is not our main problematic here.

6 Further Development

In this paper we demonstrate that the use of Graph DBMS could be interesting for code analysis. We have formulated a query to demonstrate the efficiency of this approach, but we have not yet benchmarked this approach on a large volume of code. Currently, some construct into the standard library (GNAT version), cause some trouble when creating the graph. We have to fix these problems before making a benchmark on large volume of code.

The future work will be to provide a benchmark on large code to validate the efficiency of our approach. We suppose that this can be beneficial for large volumes of the code, but irrelevant (slower) on small volumes compared to current approaches. The first step will be to manage some construct that we can see into the Ada standard library implementation provided by AdaCore. Next, we will be able to perform a benchmark, by comparing the execution time between a query made using a graph versus the same rule with AdaControl.

We have to select a subset of AdaControl’s rules to perform our benchmark, in order to have enough use cases to have a relevant benchmark. We have to define a ”frontier” between the programatic querying versus the GDBMS query language. We have some insights regarding this problem. We can cut the query into several ”elementary” queries (like ASIS), or, we can generate the request through another language [13]. We could also add a way to make incremental updates, to only update compilation units that have been modified. Another development will be to switch from ASIS to libadalang for the parsing. This will provide a compiler independent Ada code analysis tool.

References

[1] Y. Xie and A. Aiken, “Saturn: A scalable framework for error detection using Boolean satisfiability,” *ACM Transactions on Programming Languages and Systems*, vol. 29, p. 16, may 2007.

[2] T. Ball and S. K. Rajamani, “The SLAM project: Debugging system software via static analysis,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '02*, ACM Press, 2002.

[3] “Coverity Scan.” `https://scan.coverity.com/projects/`, 2018.

[4] M. Harman and P. O'Hearn, “From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis,” in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, sep 2018.

[5] G. Fan, *Practical static code analysis : challenges, methods, and solutions*. PhD thesis.

[6] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” may 2014.

[7] R.-G. Urma and A. Mycroft, “Source-code queries with graph databases—with application to programming language usage and evolution,” *Science of Computer Programming*, vol. 97, pp. 127–134, jan 2015.

[8] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, “Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases,” pp. 125–148, dec 2018.

[9] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens, F. Murlak, J. Perryman, O. Savković, M. Schmidt, J. Sequeda, S. Staworko, and D. Tomaszuk, “PG-keys: Keys for property graphs,” in *Proceedings of the 2021 International Conference on Management of Data*, ACM, jun 2021.

[10] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, H. Voigt, O. van Rest, D. Vrgoč, M. Wu, and F. Zemke, “Graph pattern matching in gql and sql/pgq,” Dec. 2021.

[11] A. Ponomarev, H. S. Nalamwar, and R. Jaiswal, “Source code analysis: Current and future trends challenges,” vol. 685, pp. 877–880, feb 2016.

[12] T. L. Alves, J. Hage, and P. Rademaker, “A comparative study of code query technologies,” sep 2011.

[13] T. Zhang, M. Pan, J. Zhao, Y. Yu, and X. Li, “An open framework for semantic code queries on heterogeneous repositories,” in *2015 International Symposium on Theoretical Aspects of Software Engineering*, IEEE, sep 2015.

Tracing and Measuring GPU Execution in Automotive Software Systems

Tiago Carvalho, Luís Miguel Pinho
Instituto Superior de Engenharia do Porto, Porto, Portugal; email: {tdc, lmp}@isep.ipp.pt

Abstract

The advance of technology in the automotive industry brought several new functionalities providing more efficiency and safety. This, however, has one important concern: the development has become more complex.

AMALTHEA is a framework for automotive system design and development in a model-based development fashion. It includes several features, including testing, software design, simulation and traceability.

This paper presents ongoing work to integrate GPU tracing in the AMALTHEA standard format for tracing execution events, thus enabling platform heterogeneity to be supported in the tracing model.

Keywords: Traceability, BTF, GPU, Automotive Software.

1 Introduction

Modern vehicles are expected to present more advanced and automated functions, relying on numerous electronic devices, and connected to several entities [1]. Several of those functions are expected to be performed within a deadline, specially for safety reasons. The integration of such functions, and system components, has resulted in an increased complexity in system design and development. The approach most adopted by the automotive industry for system design is the Model-based development (MBD). It promotes efficiency, among other benefits such as early testing [2], standardisation, software design support, and maintenance [3].

AMALTHEA is an open-source framework [4] that delivers automotive software development.

This framework supports the entire life cycle of the project, is compatible with automotive standards (like AUTOSAR [5], a software reference architecture, widely adopted by automotive suppliers), and it promotes automation and code generation [6]. AMALTHEA has been widely adopted for automotive software development, and it is the basis of both AMPERE [7] and the PANORAMA [8] projects.

The AMPERE project aims to provide a bridge to the existing gap between MBD approaches used to develop cyber-physical systems and the parallel programming models supported by embedded platforms. By taking advantage of the model defined in AMALTHEA, and together with non-functional properties, the system provides efficient key parallel constructs, while ensuring compliance to non-function requirements [7].

The ITEA3 project PANORAMA aims for improving design efficiency for heterogeneous automotive and aerospace systems. It provides an environment for collaboration amongst diverse hardware and software technologies and teams, especially at the early stages of design. PANORAMA provides system design and system analysis (both static and dynamic) at the AMALTHEA model level, also used in the feedback process. The analysis phase is intended to find safety and timing requirements for the top-level system functions [8].

AMALTHEA supports the traceability of the system, mainly focusing on timing properties [9]. AMALTHEA adopted BTF (Best Tracing Format) as its standard format for providing traceability. BTF is an open-source CSV-based trace fully compatible with automotive software development (e.g. AUTOSAR and OSEK) [10]. Current versions of BTF are essentially focused on tracing CPU and operating system events [10], with timestamps as the only measurable feature.

With the evolution of computational systems and the increased demand of performance, it is common to include accelerators, and other performing devices, in the system, such as graphic processing units (GPUs) and field-programming gate arrays (FPGAs). These devices are expected to perform better - e.g. in terms of execution time - than a CPU, specially when dealing with data-intensive and concurrent applications.

The design of a system has to take into account these type of devices, as they impact the architecture of the hardware and, primarily, the performance of the developed features. The performance impact involves more events than just the efficiency of the offloaded computation. The overheard of synchronization, allocating memory, copying memory between CPU host and device, all of these - and other - events are in fact important to have an accurate analysis of the overall system [11]. Therefore, it is important to include the design and analysis of these devices in the development phase. The need of tracing both activity and performance measurements for both CPU and accelerators is the gap that this work intends to fill.

In the context of both AMPERE and PANORAMA projects, we present in this paper adequate extensions to the BTF format in order to include traces (and measurements) from devices other than the CPU, mainly accelerators, specially for the communication with AMALTHEA models. As the extraction of such traces is also a missing feature in the AMALTHEA framework, we also present a tool, a work in progress, that takes advantage of CUPTI API to profile CUDA-based applications [12] and generate BTF traces with the new extensions. The final version of the tool will be integrated in the analysis flow of the projects. Despite the extensions being essentially aimed to GPUs, they were designed to be generic enough to comprise other types of accelerators or other means for offloading computation outside the CPU.

2 Traceability in Automotive Systems with AMALTHEA

In the context of AMPERE and PANORAMA, the AMALTHEA framework is used for the system design and analysis. The main flow of the analysis process is as follow:

```
model design → code generation → BTF file
  → ATDB → model annotation
```

The system is designed as an AMALTHEA model, which in turn is used for code generation. The code generation process can be whether a real application using code parcels of designed functionalities (ergo Runnables) or via synthetic code generation (e.g. with App4mc.sim [13]). The code is generated with tracing mechanisms that output the execution traces in a CSV-like format, named BTF [10].

Traces are stored in the AMALTHEA Trace Database (ATDB) [1], which provides straightforward access to the several metrics collected from simulations or executions. These metrics can then be directly annotate in the AMALTHEA models.

The Best-Trace Format (BTF) [10] is an open-source CSV-based trace fully compatible with automotive software development (e.g. AUTOSAR and OSEK). BTF was defined to record traces of embedded real-time multi-core systems in a CSV structure to log timestamped system events (e.g. tasks execution states, memory accesses), as illustrated as follows.

```
time, source,id, type, target,id, event, note
100, Task_1, 0, R, Runnable_1, 0, start
3210, Task_1, 0, R, Runnable_1, 0, terminate
```

This format is based on the specification of system events and entities state transitions, supporting the specification of timing and performance-related metrics. The information provided by BTF traces supports the traceability of software systems at Tasks and Runnables levels, allowing the metrics assessments at this granularity. With this format, it is possible to indicate event-based traces that contains information about: timestamp, executing task (source), executing runnable (target), type of event (e.g. a runnable starts), and a textual note.

BTF format identifies timestamps of events and it is limited to operating system, tasks and runnables events.

There are at least two potential extensions to this format able to provide unexplored tracing data currently not present neither in the BTF format nor the tools that generate traces in this format.

At one level, several measurements and metrics can be obtained from the system, besides timing, such as computed instructions, cache misses occurred or branch mispredictions [14]. Several tools (e.g. perf [15] and PAPI [16]) ease the programmatic access to this information.

[1]ATDB: https://www.eclipse.org/app4mc/help/latest, section 2.3.12

Another level is the inclusion of traceability when offloading functionalities to accelerators. For instance, to know when a code parcel is offloaded and executed in a device or when synchronization happens and how long it takes. This information can be retrieved with tools specially dedicated to these devices, depending on the programming model used. Some examples are CUPTI API [12] for CUDA-based applications and CodeXL[2] for OpenCL.

This work extends the analysis flow of the projects to take advantage of such tools to feedback more information to AMALTHEA models besides timestamps. We propose extensions to the BTF format and provide a working prototype able to generate new type of traces and measurements. We use PAPI [16] to retrieve runtime information from executions in the CPU, and CUPTI [12] to retrieve traces and measurements from executions using the GPU.

3 Extensions to the BTF Format

3.1 Adding Components Measurements to a Trace

Nowadays processing units such as CPUs provide access to data about their current (runtime) performance, known as performance monitoring counters (PMCs) [17]. PMCs can be accessed via low-level features, such as device-specific registries or system-level read-only files, or with a high-level API. In the case of CPU, PMCs can provide measurements about the number of computed cycles, total of instructions executed, accesses to different data cache levels, branch prediction, etc.

A very known tool for PMC access is PAPI [16], Performance Application Programming Interface, which provides an interface and methodology for accessing PMCs and relates them to processor events. The tool adapts to different target systems and the programming API is independent of the platform.

Our approach retrieves PMCs data for each executing `Runnable`, the "fine-grain" specification of a functionality in the AMALTHEA model, and a `Task` can be profiled based on the performance of each `Runnable`. By instrumenting the code around a given `Runnable` we obtain performance metrics specific to the execution of that `Runnable`.

For the BTF extension, in terms of syntactical structure, the CSV-like format and the number of columns used was maintained as the additional information does not require more features besides listing the results in a similar matter of existing trace types. The main focus was in the semantics of the BTF processing phase. Since these events represent the access to memory addresses of micro-controllers, one way of representing it consists of the specification of Signal Events [10] including the access to the PMC. This would completely maintain the structure of BTF, and only additional lines of "memory accesses" appear in the BTF file. The structure used to add performance traces is depicted in Figure 1.

Line 1 represents the structure. `Runnable` identifies the runnable for which the PMCs where read, the trace event is always specified as a Signal event (SIG), and the cell to the right of SIG represent the name of the performance counter.

[2]CodeXL: https://gpuopen.com/compute-product/codexl/

```
  Time, runnable, id, SIG, PMC_<name, 0, read,
  value
1 120, Runnable_1, 0, SIG, PMC_TOT_CYC, 0, read, 121501

2 120, Runnable_1, 0, SIG, PMC_L1_DCH, 0, read, 4301
3 120, Task_1, 0, R, Runnable_1, 0, start
4 321, Runnable_1, 0, SIG, PMC_TOT_CYC, 0, read, 126505

5 321, Runnable_1, 0, SIG, PMC_L1_DCH, 0, read, 6051
6 321, Task_1, 0, R, Runnable_1, 0, terminate
```

Figure 1: BTF example with PMCs measurements.

We use `"PMC_"` as prefix to differ PMCs readings from program labels. The two following cells (0 and read) are also fixed values, as there is not need to differ instances of PMCs and they are read-only registers. The value parameter will contain the content of the PMC in that specific timestamp. The following excerpt of the BTF trace shows the relation between the runnable execution and the PMCs reading. PMC readings are added prior to the runnable event as a way of associating PMC data directly to the Runnable. Lines 2 and 3 show PMCs reading before the runnable starts and lines 5 and 6 are the PMCs reading when the runnable finishes.

3.2 Tracing Execution of an Accelerator

This extension adds traces of the device regarding their events, similar to CPU and SO events, and PMC measurements. Once again we try to limit the extensions to BTF to what is actually essential for the new type of traces.

BTF uses a $source \rightarrow target$ format in every trace, as in the example below. The most common to appear are $Core \rightarrow Task$ for task instantiation, $Task \rightarrow Runnable$ for any runnable-related event, and $Task \rightarrow Signal$ when labels (i.e. variables/memory) are accessed.

```
time, source,id, type, target,id, event,
note
100, Core_1, 0, T, Task_A, 0, start
100, Task_A, 0, R, Runnable_A_1, 0, start
800, Task_A, 0, R, Runnable_A_1, 0, terminate
```

This relationship format allows us to easily map the $host \leftrightarrow device$ communication for accelerators, where it is only necessary to indicate the type of the target of the trace. In BTF, the type of the target is defined in the fourth position of the trace. The previous example shows the use of T for targeted tasks and R for targeted runnables.

To distinguish between code execution in a CPU and an accelerator device, we propose the identification of a new type target specifically for devices used for offloading computation. The following structure specifies the proposal for adding device-related activity traces, similar to CPU and SO events:

```
time, runnable, id, D, device, id, event, note
```

The type of device-related events are specified with `'D'` type (fourth position). The source (second and third position) identifies the runnable using the device, while the device name and instance (fifth and sixth position) identify the device and its current instance. The seventh (and optionally eighth) position will define the event (and extra information) occurred at that timestamp. The events will depend on what it is possible to trace in the device during execution.

```
1  void Runn_1(cudaStream_t stream)
2  {
3   int *h_A, *h_B, *h_C; //host variables
4   int *d_A, *d_B, *d_C; //device variables
5   ...
6   cudaMemcpyAsync(d_A, h_A, size,
7     cudaMemcpyHostToDevice, stream); //host to device

8   cudaMemcpyAsync(d_B, h_B, size,
9     cudaMemcpyHostToDevice, stream); //host to device

10  ...
11  VecAdd<<<..., stream>>>(d_A, d_B, d_C, COMPUTE_N);

12
13  cudaMemcpyAsync(h_C, d_C, size,
14    cudaMemcpyDeviceToHost, stream); //device to host

15  cudaStreamSynchronize(stream); //sync with device
16 }
```

Figure 2: A Runnable with CUDA code with offloading computations to a device.

4 Tracing CUDA-based Applications

GPUs are devices commonly included in systems that can take advantage parallelizable work (e.g. for image or vectoral processing). In this work we consider the use of a GPU device, specifically NVIDIA GPUs with CUDA capabilities [18]. CUDA is one of the most used interfaces for parallel programming with GPU devices.

Figure 2 shows an example of a Runnable that offloads computation to a GPU (using a `cudaStream_t` for the communication). The most common structure of offloading is to first allocate and copy memory from the host to the device (lines 5 to 9), then request the execution of one or more kernels and finish the process by copying memory back to the host. These and other types of events can be easily traced with CUPTI. CUPTI [12] is the CUDA Profiling Tools Interface that allows the profiling and tracing of applications programmed with CUDA, being the ideal tool to monitor CUDA applications. CUPTI provides access to activity traces, from which we highlight: memory copy from host to device and vice-versa, kernel execution deploy and host-device synchronization.

In the same way as PAPI, CUPTI can access PMCs from the device through the CUPTI event API. PMCs from devices can be integrated as SIG traces, using a prefix PMC to consider as a performance counter. The available events are device-dependent. Two examples of CUPTI events are the elapsed clock cycles and the number of transactions for shared store/load accesses. We are developing a tool that generates code with tracing mechanisms using the CUPTI API to build a complete BTF trace, considering the communication between an AMALTHEA Runnable and the device.

Currently, the tool is able to generate code for the tracing mechanims, while the integration of the generated code is done manually by essentially adding two function calls: `btf_start` and `btf_stop`. For the compilation it is necessary to add the CUPTI libraries: `-lcupti -lnvToolsExt`.

Then, at runtime, the generated tracing mechanism processes the events reported by CUPTI and builds the BTF trace, including common BTF traces, such as Task instantiation and Runnable execution. Figure 3 shows an example of a BTF

```
    time, source,id, type, target,id, event, note
1   21674, Task_1, 0, R, Runn_1, 0, start
2   58584, Runn_1, 0, D, GPU, 7, memcopy_start, HtoD
3   58616, Runn_1, 0, D, GPU, 7, memcopy_terminate, HtoD

4   58709, Runn_1, 0, D, GPU, 7, memcopy_start, HtoD
5   58735, Runn_1, 0, D, GPU, 7, memcopy_terminate, HtoD

6   58794, Runn_1, 0, D, GPU, 7, kernel_start, VecAdd

7   58822, Runn_1, 0, D, GPU, 7, kernel_terminate, VecAdd

8   58903, Runn_1, 0, D, GPU, 7, memcopy_start, DtoH
9   58930, Runn_1, 0, D, GPU, 7, memcopy_terminate, DtoH

10  59250, Task_1, 0, R, Runn_1, 0, suspend, cuda context

11  59258, Task_1, 0, R, Runn_1, 0, resume, cuda context

12  59530, Task_1, 0, R, Runn_1, 0, terminate
```

Figure 3: Extended BTF generated directly from executing the tracing tool over the example in Figure 2.

trace generated from an execution of the code in Figure 2, specifically the execution of runnable `Runn_1` (between lines 1-12). Lines 2-5, 9-10 are traces of memory events, lines 6-7 show the kernel execution in the device, and lines 10-11 relate to $host \leftrightarrow device$ synchronization. The traces are defined in a format of $(start, terminate)$ pairs. This way it is possible to observe the time spent during that CUDA-related activity. For instance, the execution of `VecAdd` took 28 µs.

5 Conclusions

The paper presented a set of extensions to the BTF format to comprise performance data and GPU activity traces. The proposal had in mind minimal extensions to the format to maintain full compatibility with existing tools. It does not change the structure (lexically and syntactically) of the BTF format, but includes new (semantic) concepts. Considering the extensions to the format, a code generator is being developed that provides tracing mechanisms for CUDA applications.

The next step for this process is the improvement of the tracing tool with a more automated approach for instrumenting the target code.

The final version of the tool will be integrated in the system analysis flow of AMPERE and PANORAMA projects, taking advantage of the code generation processes. Furthermore, we intend to develop extensions to current BTF processing tools for them to be able to interpret the new features. These additions to the process are crucial to complete the tracing process and analysis of GPU execution.

6 Acknowledgments

This research has been co-funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669, in the context of the AMPERE project, and by the Portugal COMPETE 2020 Program, through the European Regional Development Fund (ERDF), within project POCI-01-0247-FEDER-040194, in the context of the ITEA3 project Panorama.

References

[1] S. Ray, W. Chen, J. Bhadra, and M. A. Al Faruque, "Extensibility in automotive security: Current practice and challenges," in *Proc. of the 54th ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pp. 1–6, 2017.

[2] H. Guissouma, H. Klare, E. Sax, and E. Burger, "An empirical study on the current and future challenges of automotive software release and configuration management," in *Proc. of the 44th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pp. 298–305, 2018.

[3] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Benefits from modelling and MDD adoption: Expectations and achievements," in *Proc. of the 2nd Edition of the Intl. Workshop on Experiences and Empirical Studies in Software Modelling*, 2012.

[4] C. Wolff, C. Brink, R. Höttger, B. Igel, E. Kamsties, L. Krawczyk, and U. Lauschner, "Automotive software development with amalthea," *Practice and Perspectives*, vol. 432, 2015.

[5] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks Workshop*, pp. 215–217, IEEE, 2016.

[6] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel, "AMALTHEA — tailoring tools to projects in automotive software development," in *Proc. of the IEEE 8th Intl. Conf. on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, pp. 515–520, 2015.

[7] E. Quiñones, S. Royuela, C. Scordino, P. Gai, L. M. Pinho, and L. Nogueira et al., "The ampere project: : A model-driven development framework for highly parallel and energy-efficient computation supporting multi-criteria optimization," in *2020 IEEE 23rd Intl. Symposium on Real-Time Distributed Computing (ISORC)*, pp. 201–206, 2020.

[8] L. Krawczyk, J. Tessmer, and H. Mackamul, "Panorama - boosting design efficiency for heterogeneous systems," *ECLIPSE Newsletter*, July 2019.

[9] R. Höttger, H. Mackamul, A. Sailer, J.-P. Steghöfer, and J. Tessmer, "App4mc: application platform project for multi-and many-core systems," *it-Information Technology*, vol. 59, no. 5, pp. 243–251, 2017.

[10] Vector Informatik GmbH, "Best trace format (btf) – technical specification, version 2.2.1," tech. rep., Vector Informatik GmbH, 2021.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proc. of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[12] NVIDIA, "Api reference guide for cupti, the cuda profiling tools interface." URL: https://docs.nvidia.com/cuda/cupti/index.html. Accessed: 26-02-2022.

[13] Bosch GmbH, “App4mc.sim - timing simulation of embedded systems.” Available at: https:// gitlab.idial.institute/panorama.systemc.group/app4mc.sim.

[14] L. Uhsadel, A. Georges, and I. Verbauwhede, “Exploiting hardware performance counters,” in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 59–67, IEEE, 2008.

[15] A. C. De Melo, “The new linux’perf’tools,” in *Slides from Linux Kongress*, vol. 18, pp. 1–42, 2010.

[16] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “Papi: A portable interface to hardware performance counters,” in *Proc. of the department of defense HPCMP users group Conf.*, vol. 710, Citeseer, 1999.

[17] K. Singh, M. Bhadauria, and S. A. McKee, “Real time power estimation and thread scheduling via performance counters,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.

[18] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014.

Renaissance-Ada: Tools for Analysis and Transformation of Ada code*

Piërre van de Laar, Arjan Mooij

ESI (TNO), High Tech Campus 25, 5656 AE Eindhoven, The Netherlands; email: {pierre.vandelaar, arjan.mooij}@tno.nl

Abstract

In a constantly changing world, developers need to analyze and transform their code to keep it up-to-date and valuable. Currently, analysis and transformation is a time-consuming and largely manual activity.

The Renaissance approach aims to enhance insight by analysis and to reduce complexity by transformation. The Renaissance approach is supported by programming-language-specific tools. The Renaissance-Ada tools can be configured, integrated, and extended by developers to perform their program and task-specialized analysis and transformation of Ada code. Furthermore, the Renaissance-Ada tools shields developers from parsing details.

Nexperia ITEC has used Renaissance-Ada to their full satisfaction. Recently Renaissance-Ada has become open source at `https://github.com/TNO/Renaissance-Ada`.

Keywords: software analysis, software transformation.

1 Introduction

Developers need to change software to realize new requirements, to deal with obsolescence of components, and to incorporate advances in business and technology. To correctly change the software, developers need to understand the software. Previous studies [1, 2, 3] indicate that developers spend a large portion of their time on understanding software.

Software "*understanding involves dealing with specific problems that require program and task-specialized solutions*" [4]. A possible solution is a toolkit that supports a developer to create a specialized analysis. The tools in such a toolkit should be easy to configure, integrate, and extend, and should shield users from parsing details.

How can Ada developers currently understand and change their code? Besides manual analysis and transformation, the following options for automation exist:

- Regular expressions. These enable program and task-specialized analysis and transformation and provide an interface for extension and integration. However, regular expressions are text based and thus expose a lot of parsing details, including the handling of comments and white spaces. Furthermore, regular expressions cannot handle programming language structures like recursively nested parentheses.
- Ada-specific stand-alone or integrated analysis tools. GNATCheck and CodePeer are examples of the former, and the "Find All References" functionality, present in Integrated Development Environments such as GNAT-Studio and Visual Studio Code, is an example of the latter. These analysis tools shield users from parsing details, but only perform generic analysis and provide no interface to developers for integration and extension.
- Ada-specific analysis libraries, such as Libadalang[1] and ASIS[2]. Such analysis libraries enable program and task-specialized analysis and provide an interface for extension and integration, yet expose all parsing details related to the Abstract Syntax Tree (AST) of the Ada programming language.

In Section 2, we describe the Renaissance approach that is supported by programming-language-specific tools. The Renaissance-Ada tools target Ada code and are available under the BSD3 license at `https://github.com/TNO/Renaissance-Ada`. In Sections 3, 4, and 5, we describe the main Renaissance-Ada tools: the dependency graph extractor, the rejuvenation library, and the rewriters library, respectively. We end with conclusions in Section 6.

2 Renaissance approach

The Renaissance approach [5, 6, 7, 8, 9] aims to enhance insight by analysis and to reduce complexity by transformation. These aims reinforce each other as is visualized in Figure 1.

The Renaissance approach exploits the strengths of developers and computers for semi-automated analysis and transformation. In this approach, developers orchestrate and specialize the analysis and transformation, while computers execute the administrative and repetitive parts fast and reliably.

*The research is carried out as part of the Bright project under the responsibility of ESI (TNO) with Nexperia ITEC as the carrying industrial partner. The Bright research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

[1]`https://adaco.re/libadalang`
[2]`https://www.adacore.com/asis`

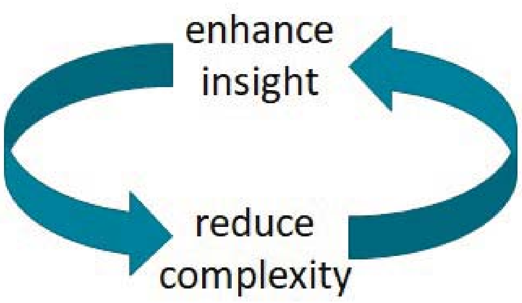
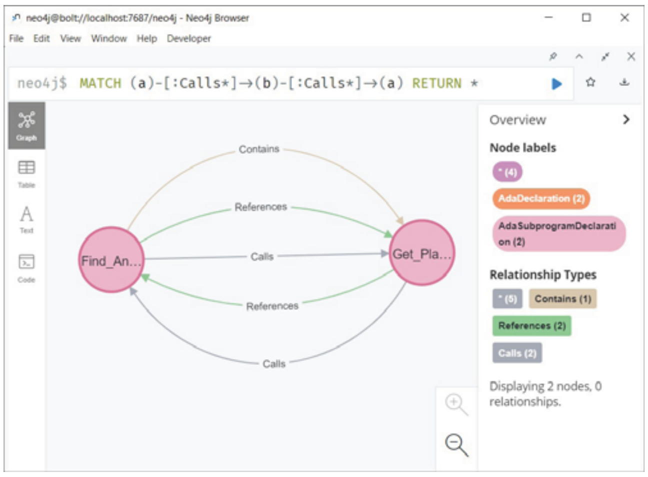


Figure 1: Reinforcing Renaissance aims.



```
neo4j$ MATCH (a)-[:Calls*]→(b)-[:Calls*]→(a) RETURN *
```

Figure 2: Result of query to find indirect recursion in Neo4j.

The Renaissance approach values human judgment to make the right decisions, since by making a transformation automatically, one loses the ability to spot the real error, and make the right change. For example, the programmer error InRange(X) **and then** InRange(X) can be simplified into InRange(X), yet should be changed into InRange(X) **and then** InRange(Y) to correct the error. In general Renaissance code transformations do not need to preserve the code's semantics, and hence code changes must be reviewed by humans.

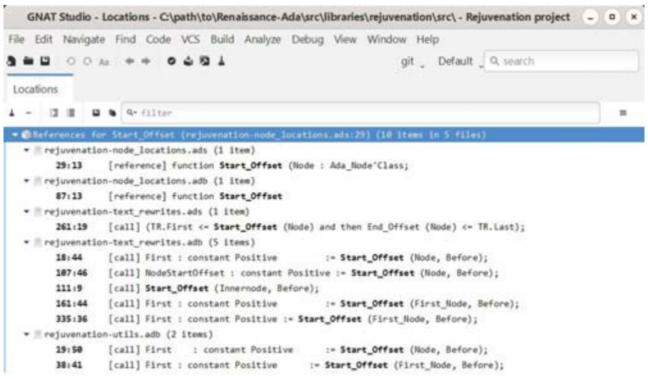
The Renaissance approach was developed by ESI (TNO) in public-private research projects together with Thermo Fisher Scientific and Philips [9]. The Renaissance approach was validated and further developed by ESI (TNO) in public-private research projects with Nexperia ITEC. Nexperia ITEC [10] reported at the AdaCore Tech Days 2021 EU Event that they already saved 300k$ and estimated to save another 900k$ using the Renaissance approach while the investment was below 100k$.

The Renaissance approach depends on tools that capture commonalities across analyses and transformations, yet are highly configurable or even programmable to support the unique aspects of a particular analysis or transformation. The Renaissance tools developed together with Thermo Fisher Scientific and Philips target the programming languages C and C++. In the cooperation with Nexperia ITEC, tools targeting the Ada programming language were developed. These tools are build on top of Libadalang and were made open source in the Renaissance-Ada project at `https://github.com/TNO/Renaissance-Ada` under the BSD3 license in the beginning of 2022. Furthermore, these tools are also made available as Alire[3] crates.

3 Interactively visualize and query code

Renaissance-Ada contains the dependency graph extractor: a tool that extracts structural elements and relations [7, 8] from Ada code for further analysis. The elements range from Ada declaration to project file. The relations include compile, import, call, override, and instantiate. The dependency graph extractor considers the static semantics of the Ada programming language at an abstraction level that balances between detail and usability. For example, details like position of components in records, order of execution, and used kind of notation, i.e., positional or named associations are hidden.

The dependency graph extractor outputs graph data in the GraphML[4] file format. Files in the GraphML file format can be imported in graph data platforms, like Neo4j[5]. Graph data platforms enable developers to perform their program and task-specialized analysis by interactively visualizing and querying for relevant code elements and relations in the Ada code. As an example, Figure 2 shows a Cypher[6] query in Neo4j to find indirect recursion: recursion involving at least two functions.


GNAT Studio - Locations - C:\path\to\Renaissance-Ada\src\libraries\rejuvenation\src\ - Rejuvenation project
File Edit Navigate Find Code VCS Build Analyze Debug View Window Help
git
Default
search
Locations
filter

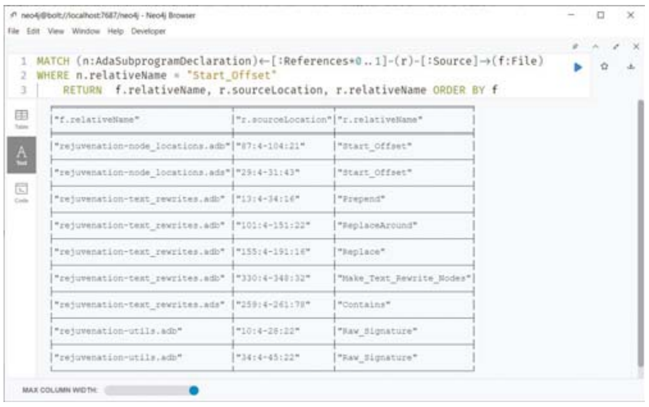
```
▾ References for Start_Offset (rejuvenation-node_locations.ads:29) (10 items in 5 files)
  ▾ rejuvenation-node_locations.ads (1 item)
      29:13     [reference] function Start_Offset (Node : Ada_Node'Class;
  ▾ rejuvenation-node_locations.adb (1 item)
      87:13     [reference] function Start_Offset
  ▾ rejuvenation-text_rewrites.ads (1 item)
      261:19    [call] (TR.First <= Start_Offset (Node) and then End_Offset (Node) <= TR.Last);
  ▾ rejuvenation-text_rewrites.adb (5 items)
      18:44     [call] First : constant Positive          := Start_Offset (Node, Before);
      107:46    [call] NodeStartOffset : constant Positive := Start_Offset (Node, Before);
      111:9     [call] Start_Offset (Innernode, Before);
      161:44    [call] First : constant Positive          := Start_Offset (First_Node, Before);
      335:36    [call] First : constant Positive := Start_Offset (First_Node, Before);
  ▾ rejuvenation-utils.adb (2 items)
      19:50     [call] First   : constant Positive       := Start_Offset (Node, Before);
      38:41     [call] First : constant Positive      := Start_Offset (First_Node, Before);
```

Figure 3: Result of "Find All References" for the function Start_Offset in GNATStudio.

A lot of functionality already provided by stand-alone and integrated analysis tools can be easily realized using the graph data obtained by the dependency graph extractor from the Ada code. Figure 3 shows the result of the "Find All References" analysis of GNATStudio for the function Start_Offset. Figure 4 shows the tabular result of a query to find all references in Neo4j for the same function. The tabular result has a row for each function that references the function Start_Offset. Each row has three columns for the referencing function's containing file name, location range within the containing

[3]`https://alire.ada.dev`

[4]`http://graphml.graphdrawing.org`
[5]`https://neo4j.com`
[6]`https://neo4j.com/developer/cypher`



```
MATCH (n:AdaSubprogramDeclaration)←[:References*0..1]-(r)-[:Source]→(f:File)
WHERE n.relativeName = "Start_Offset"
   RETURN  f.relativeName, r.sourceLocation, r.relativeName ORDER BY f
```

```
|"f.relativeName"                  |"r.sourceLocation"|"r.relativeName"          |
|"rejuvenation-node_locations.adb"|"87:4-104:21"     |"Start_Offset"            |
|"rejuvenation-node_locations.ads"|"29:4-31:43"      |"Start_Offset"            |
|"rejuvenation-text_rewrites.adb" |"13:4-34:16"      |"Prepend"                 |
|"rejuvenation-text_rewrites.adb" |"101:4-151:22"    |"ReplaceAround"           |
|"rejuvenation-text_rewrites.adb" |"155:4-191:16"    |"Replace"                 |
|"rejuvenation-text_rewrites.adb" |"330:4-348:32"    |"Make_Text_Rewrite_Nodes"|
|"rejuvenation-text_rewrites.ads" |"259:4-261:78"    |"Contains"                |
|"rejuvenation-utils.adb"         |"10:4-28:22"      |"Raw_Signature"           |
|"rejuvenation-utils.adb"         |"34:4-45:22"      |"Raw_Signature"           |
```

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
Table
Text
Code
MAX COLUMN WIDTH:

Figure 4: Result of query to find all references for the function Start_Offset in Neo4j.

file, and name, respectively. The results only differ because of the different aggregation levels: where GNATStudio uses the references with their code location, the dependency graph extractor uses the functions containing the references with their code location ranges.

Unlike stand-alone and integrated analysis tools, graph data platforms offer an interface for extension and integration by developers. For example, Neo4j enables that the information obtained by the dependency graph extractor can be combined with other information, such as code ownership, i.e., the e-mail address of the maintainer of each file, and the allowed architectural relations. Furthermore, Neo4j provides a programmatic interface that enables developers to create analyses in which queries can be composed. So, when a developer wants to change the signature of a function, the developer can easily combine the query, shown in Figure 4, with the code ownership information to obtain the e-mail addresses of all stakeholders that must be invited for the meeting on changing the function's signature. Similarly, when an architect wants to communicate the adherence to the architecture, a relatively simple program can generate graphs highlighting differences between the actual and desired relations. An anonymized example visualizing such a graph using a graph editor, like yEd[8], is shown in Figure 5.

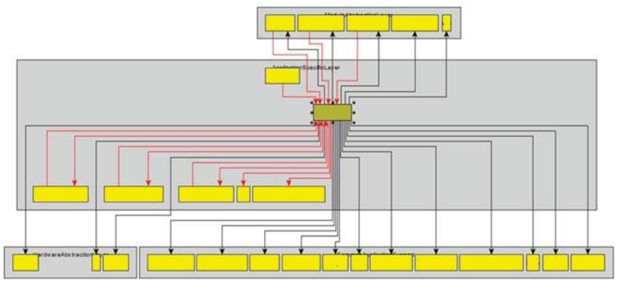


Figure 5: Dependencies of a package. The color of the dependencies reflect the adherence to the architecture: red-colored dependencies violate the architecture.

[8]`https://www.yworks.com/products/yed`

4 Specify code patterns in Ada

Renaissance-Ada contains a rejuvenation library that provides basic analysis and transformation functionality. The rejuvenation library enables the specification of code patterns using concrete Ada syntax [5,6,11,12]. The concrete syntax shields the developers from parsing details related to the Abstract Syntax Tree (AST) representation that is used for comparing pieces of Ada code and matching of patterns with Ada code. Two pieces of Ada code are considered identical when they have the same tree structure and semantically identical leaf AST nodes. In other words, when comparing two pieces of Ada code we ignore comments, white spaces, and syntactic variations that map onto a single semantic entity, e.g., due to case insensitive identifiers and underscores used in numeric literals.

We use the concrete Ada syntax with a naming convention to denote placeholders that capture arbitrary AST nodes. Two types of placeholders exist: single and multiple. Single placeholders are matched with a single AST node, be that a single expression, a single statement, a single argument, or anything else. Multiple placeholders are matched with a sequence of AST nodes, i.e., zero or more nodes. Single and multiple placeholders start with the prefix `$S_` and `$M_`, respectively. After those prefixes, any alphanumeric string is allowed.

Depending on the purpose of the pattern, we refer to either a find or a replace pattern. For example, the find pattern `Square ($S_PA)` matches all calls to the `Square` function with a single parameter association, both named and positional. Hence, the rejuvenation library will match this find pattern with the code fragments `Square(x => 3)`, `Square(a + b)`, and `Square (f (x, y) + g (z))`, but not with the code fragment `Square(a, b)`.

A pattern may contain several placeholders that each may even occur multiple times. For find patterns it holds that when the same placeholder occurs multiple times, a match will only be found when all occurrences of that placeholder are identical. For replace patterns it holds that a placeholder always refers to the same placeholder in the associated find pattern. The placeholder will be replaced by the value of that placeholder[9] in the match of the find pattern. Since the replace pattern `Power ($S_PA, n => 2)` has the same placeholders as the find pattern `Square ($S_PA)`, they can be applied together to replace function calls to the `Square` function by calls to the `Power` function, with the exponent value of two. Given this find and replace pattern, the rejuvenation library will change the code pattern `Square (x => 3)` into `Power (x => 3, n => 2)`, `Square (a + b)` into `Power (a + b, n => 2)`, and `Square (f (x, y) + g (z))` into `Power (f (x, y) + g (z), n => 2)`.

Although the previous example seems simple, it based on a real industrial case in which Nexperia ITEC wanted to remove a proprietary function that became obsolete when an equivalent function was added to Ada's standard libraries. Using regular expressions, it was not possible to automate the

[9]The rejuvenation library allows developers to specify whether trivia, i.e., comments and white spaces, before and after the placeholder should be included in its value. Note that trivia within the placeholder are always included in its value.

removal of this proprietary function, due to function calls like `Square (f (x, y) + g (z))` that contain multiple sets of matching parentheses and separating comma's within the argument. Using the rejuvenation library, Nexperia ITEC was able to automate the transformation of all function calls and thus to remove the proprietary function.

4.1 From code fragments to patterns

Suppose we would like to find all if-statements of which both branches call the same subprogram and only differ in the first argument, such as

```
if Is_Male (X) then
   Display ("Mr." & Name (X), Red);
else
   Display ("Mrs." & Name (X), Red);
end if;
```

and replace them with a call statement with an if-expression as first argument, such as

```
Display ((if Is_Male (X) then "Mr." & Name (X)
                          else "Mrs." & Name (X)), Red);
```

We can realize this particular find and replacement using the rejuvenation library. We take these code fragments and replace pieces of code by placeholders where variation is wanted. For example, we replace all instances of `Display` by `$S_F`, since we want to allow any subprogram name, but also keep the constraint of having the same subprogram name in both branches of the if-statement in the find pattern and the back reference in the replace pattern. The final transformation exists of the following find pattern:

```
if $S_Cond then
  $S_F ($S_True, $M_Tail);
else
  $S_F ($S_False, $M_Tail);
end if;
```

and the following replace pattern:

```
$S_F ((if $S_Cond then $S_True else $S_False), $M_Tail);
```

4.2 Shielding parsing details

The concrete syntax shields the developers from parsing details related to the Abstract Syntax Tree (AST) representation, unlike analysis libraries like Libadalang and ASIS. For example, we believe it is easier to understand and maintain the following find pattern specified in concrete Ada syntax:

```
if $S_Cond then
  $S_Dest := True;
else
  $S_Dest := False;
end if;
```

than the implementation of the equivalent functionality using Libadalang:

```
function Is_Match_If_Stmt_Identifiers
         (ThenIdentifier, ElseIdentifier : Identifier)
         return Boolean is
  (Ada.Strings.Equal_Case_Insensitive
     (Image (ThenIdentifier.Text), "True")
   and then Ada.Strings.Equal_Case_Insensitive
     (Image (ElseIdentifier.Text), "False"));

function Is_Match_If_Stmt_Assign_Stmts
         (ThenAssignStmt, ElseAssignStmt : Assign_Stmt)
         return Boolean is
  (ThenAssignStmt.F_Dest.Text = ElseAssignStmt.F_Dest.Text
   and then ThenAssignStmt.F_Expr.Kind = Ada_Identifier
   and then ElseAssignStmt.F_Expr.Kind = Ada_Identifier
   and then Is_Match_If_Stmt_Identifiers
     (ThenAssignStmt.F_Expr.As_Identifier,
      ElseAssignStmt.F_Expr.As_Identifier));

function Is_Match_If_Stmt_Branches
         (ThenNode, ElseNode : Ada_Node) return Boolean is
  (ThenNode.Kind = Ada_Assign_Stmt
   and then ElseNode.Kind = Ada_Assign_Stmt
   and then Is_Match_If_Stmt_Assign_Stmts
     (ThenNode.As_Assign_Stmt, ElseNode.As_Assign_Stmt));

function Is_Match_If_Stmt (IfStmt : If_Stmt) return Boolean is
  (IfStmt.F_Then_Stmts.Children_Count = 1
   and then IfStmt.F_Else_Stmts.Children_Count = 1
   and then IfStmt.F_Alternatives.Children_Count = 0
   and then Is_Match_If_Stmt_Branches
    (IfStmt.F_Then_Stmts.First_Child,
     IfStmt.F_Else_Stmts.First_Child));

function Process_Node
         (Node : Ada_Node'Class) return Visit_Status is
begin
   if Node.Kind = Ada_If_Stmt then
      declare
         IfStmt : constant If_Stmt := Node.As_If_Stmt;
      begin
         if Is_Match_If_Stmt (IfStmt) then
            Put_Line (Image (IfStmt.Full_Sloc_Image) & "␣Found");
         end if;
      end;
   end if;
   return Into;
end Process_Node;
```

where the function `Process_Node` is called on all nodes of the AST tree. For more details, see Libadalang's Ada Api tutorial[10].

5 Composing code transformations

Transformations have more acceptance criteria than just making the change. In this section, we introduce the rewriters library of Renaissance-Ada and describe how the rewriters library makes changes more acceptable for compilers and the development team, containing developers, testers, and reviewers, by composing transformations.

First, one transformation often triggers another transformation. For example, the development team indicated that the example of Subsection 4.1 could be simplified and should be transformed into

```
Display ((if Is_Male (X) then "Mr." else "Mrs.") & Name (X), Red);
```

The desired simplification can be realized with a transformation that uses the find pattern

```
if $S_Cond then $S_True & $S_Expr else $S_False & $S_Expr
```

to detect that the same element is concatenated in both branches of an if-expression, and the replace pattern

```
(if $S_Cond then $S_True else $S_False) & $S_Expr
```

to factor out the concatenation of the same element. However, applying this transformation on the whole code base might modify pieces of code not affected by the original transformation. All modifications that are not related to the original transformation are undesired, since they only distract the development team in achieving their goal. The rewriters library both supports a large set of transformations that simplify Ada code as expected by the development team, and enables that transformations can be limited to the earlier transformed code.

[10] https://docs.adacore.com/live/wave/libadalang/html/libadalang_ug/ada_api_tutorial.html#browse-the-tree

Second, compilation can fail when code styles are violated, e.g., when a line is longer than the maximum allowed number of characters. Hence, transformations should not lead to style violations. However, pretty printing a file might modify pieces of code not affected by the transformation. The rewriters library ensures that pretty printing is as much as possible limited to the transformed code while all requirements of the pretty printer are satisfied. For example, GnatPP requires that the commands to turn pretty printing on and off appear on a line by themselves, making a line the smallest piece of code that can be pretty printed.

Third, one transformation may not be sufficient according to the development team. For example, after applying the find pattern to detect adjacent declarations with the same type and initial value, i.e.,

```
$M_X : $S_Type := $M_Expr;
$M_Y : $S_Type := $M_Expr;
```

and the replace pattern to combine these declarations into a single declaration, i.e.,

```
$M_X, $M_Y : $S_Type := $M_Expr;
```

on the code fragment

```
A : Integer := 0;
B : Integer := 0;
C : Integer := 0;
```

one gets

```
A, B : Integer := 0;
C : Integer := 0;
```

On this code fragment, one can apply the same find and replace pattern once more, resulting in

```
A, B, C : Integer := 0;
```

The development team expects that the transformation is applied multiple times. Similarly, the development team expects that when code is transformed not one but both *De Morgan's laws* are applied at the same time. The rewriters library supports repeating and combining transformations to ensure acceptance by the development team.

6 Conclusion

The Renaissance-Ada toolkit enables developer to perform program and task-specialized analysis and transformation of Ada code by configuring, combining, and extending its tools. Furthermore, the Renaissance-Ada tools shield developers from parsing details.

The Renaissance-Ada toolkit includes (1) the dependency graph extractor, which enables interactively visualizing and querying of Ada elements and relations; (2) the rejuvenation library, which enables AST-based find and replace using patterns specified in the Ada language extended with placeholders; and (3) the rewriters library, which makes changes more acceptable for compilers and the development team by repeating and combining transformations, and by limiting pretty printing and additional transformations, such as simplifications, to earlier transformed code.

The Renaissance-Ada toolkit has recently been made open source. Like Nexperia ITEC, every organization can now benefit by using its tools on their own Ada code. Furthermore, everyone can contribute to Renaissance-Ada!

Acknowledgements

We would like to thank Jeroen Ketema for the development of the Renaissance-Ada version of the dependency graph extractor.

We would like to thank Jacques Verriet for his valuable feedback on earlier versions of this paper.

References

[1] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada* (J. H. Johnson, ed.), p. 21, IBM, 1997.

[2] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017* (G. Scanniello, D. Lo, and A. Serebrenik, eds.), pp. 308–311, IEEE Computer Society, 2017.

[3] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Trans. Software Eng.*, vol. 44, no. 10, pp. 951–976, 2018.

[4] S. P. Reiss, "The paradox of software visualization," in *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, September 25, 2005* (S. Ducasse, M. Lanza, A. Marcus, J. I. Maletic, and M. D. Storey, eds.), pp. 59–63, IEEE Computer Society, 2005.

[5] A. J. Mooij, M. M. Joy, G. Eggen, P. Janson, and A. Radulescu, "Industrial software rejuvenation using open-source parsers," in *Theory and Practice of Model Transformations - 9th International Conference, ICMT@STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings* (P. V. Gorp and G. Engels, eds.), vol. 9765 of *Lecture Notes in Computer Science*, pp. 157–172, Springer, 2016.

[6] S. Klusener, A. J. Mooij, J. Ketema, and H. van Wezep, "Reducing code duplication by identifying fresh domain abstractions," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pp. 569–578, IEEE Computer Society, 2018.

[7] D. Dams, A. J. Mooij, P. Kramer, A. Radulescu, and J. Vanhara, "Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018* (R. Oliveto, M. D. Penta, and D. C. Shepherd, eds.), pp. 552–556, IEEE Computer Society, 2018.

[8] D. Dams, J. Ketema, P. Kramer, A. J. Mooij, and A. Radulescu, “Developing and applying custom static analysis tools for industrial multi-language code bases,” in *Proceedings of the 20th Belgium-Netherlands Software Evolution Workshop, Virtual Event / ’s-Hertogenbosch, The Netherlands, December 7-8, 2021* (G. Catolino, D. D. Nucci, and D. A. Tamburri, eds.), vol. 3071 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.

[9] N. Roos, “ESI helps Thermo Fisher and Philips grease their software machines,” *Bits & Chips*, no. 6, pp. 38–41, 2019. https://bits-chips.nl/artikel/esi-helps-thermo-fisher-and-philips-grease-their-software-machines.

[10] F. Patschkowski, “Refactoring with confidence: A pattern-based approach built on top of libadalang to save money.” https://www.youtube.com/watch?v=EHrd-9wgALM, November 2021. AdaCore Tech Days 2021 EU Event.

[11] M. P. A. Sellink and C. Verhoef, “Native patterns,” in *5th Working Conference on Reverse Engineering, WCRE ’98, Honolulu, Hawai, USA, October 12-14, 1998*, pp. 89–103, IEEE Computer Society, 1998.

[12] E. Visser, “Meta-programming with concrete object syntax,” in *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings* (D. S. Batory, C. Consel, and W. Taha, eds.), vol. 2487 of *Lecture Notes in Computer Science*, pp. 299–315, Springer, 2002.

Boosting Productivity and Resiliency through Automated Software Replication

Adrian Munera, Eduardo Quiñones, Sara Royuela

Barcelona Supercomputing Center, Pl. Eusebi Güell, 1-3, 08034, Barcelona, Spain;
email: {adrian.munera, eduardo.quinones, sara.royuela}@bsc.es

Michael Pressler, Harald Mackamul, Dirk Ziegenbein

Robert Bosch GmbH, 71272 Renningen, Germany;
email: {michael.pressler, harald.mackamul, dirk.ziegenbein}@de.bosch.com

Abstract

The high performance demands of complex cyber-physical systems (CPS) pushes the use of parallel and heterogeneous processor architectures in systems where dependability is crucial. One widespread technique to accomplish dependability and, in particular, resilience, is replication. Replicating hardware or software components changes the overall system, hence must be taken into consideration at design-time. Domain-specific modeling languages (DSML) provide a level of abstraction that alleviates the design process of CPS while ensuring a correct-by-construction *paradigm. Developed in the frame of the AMPERE H2020 EU project, this work increases the expressiveness of the AMALTHEA DSML to describe software redundancy. Based on these extensions, the AMPERE tool-chain is capable of automatically generating parallel replicas during system deployment to boost the reliability of the system while keeping the productivity. The effectiveness of this approach is demonstrated by extending the synthetic load generator included in the APP4MC platform.*

Keywords: Software replication, productivity, DSML.

1 Introduction

Model-driven engineering (MDE) is a software engineering paradigm that typically leans on domain specific modeling languages (DSML) to describe the system from the domain point of view while hiding the complexity of the technical solution. DSMLs are further used to tailor specific modelling languages to focus certain aspects of the system important to the domain. E.g., safety standards can differ in each domain and must be handled appropriately.

Strong model semantics relevant to the target domain and systems allow to use pure model-based approaches for formal validation and verification analysis. Additionally, strong semantics enables the automatic generation of target code generation, to facilitate the verification of the system for aspects that cannot be verified at the formal model level.

Consequently, MDE is widely used for the development of complex cyber-physical systems (CPS) where dependability is crucial. Advanced functionalities, like *steering-by-wire* and *adaptive cruise control* from the automotive domain, require high-performance capabilities while still following strong safety requirements. In this context, parallel heterogeneous architectures are of particular interest, as they can provide (1) dedicated processors to real-time and safety-critical functionalities, and (2) increased number of resources to boost throughput.

The AMPERE H2020 EU project [1] addresses the development of *correct-by-construction* CPS by providing a new generation of software programming environments for low-energy and highly parallel and heterogeneous computing architectures that fulfill the non-functional requirements of the system, including real-time and reliability. Bosch, as industrial partner of the project and use-case provider, contributes with the APP4MC platform [2], tailored for engineering embedded multi- and many-core software systems. Originally, the DSML was designed to represent the dynamic system architecture of real-time CPS. This allows timing analysis as well as safety checks for data-consistency mechanisms. The checks are needed in concurrent CPS systems, especially during the transition from single to multi- and many-core systems.

With the shift from single function electronic control units (ECU) towards domain, zone and vehicle architectures [3], the burden of validation and verification processes shifts more towards the ECU. The use of high-performance embedded platforms and the need of hosting heterogeneous applications from different suppliers on the same platform, pose new challenges on validation and verification. It increases the need to adapt system design to higher abstraction levels. An earlier integration on DSML level as well as on code level is needed. APP4MC allows the integration on ECU system level and provides the synthetic load generator (SLG) to mimic the timing behavior of the system directly on the target.

This work takes advantage of the possibilities offered by DSML and parallel processor architectures to boost the productivity of dependable systems that (partially) rely on replication to ensure safety. To that end, the APP4MC framework has been extended at two levels: (1) the AMALTHEA DSML has been augmented with a new redundancy property to increase *fault tolerance*; and (2) the APP4MC SLG has been augmented with a transformation method that generates parallel replicas and uses a *voting-based consensus* algorithm to decide the solution and actions to take. The proposed extensions will push the DSML further into utilizing modern parallel processor architectures beyond classical CPS and allow to evaluate the overall impact of redundancy safety techniques starting early in the design flow. The performance of the system is evaluated on the WATERS 2016 challenge [4] proposed by Bosch.

2 The APP4MC platform

APP4MC [2] is an open source Eclipse platform that provides AUTOSAR [5] compliant data models, namely AMALTHEA, for multi-/many-core systems. The model is generated combining information from various sources including code analysis for production code, runtime traces, AUTOSAR specifications and also information added by the designer.

The platform has been proven in the automotive sector by Bosch and their partners. Different analysis tools have been build around it varying from commercial tools for system performance analysis to Bosch internal tools for data consistency checks for multi-core systems and memory layout optimization [6]. Several output artifacts can be generated based on the enhanced system model, e.g., linker files, system configuration files or source code.

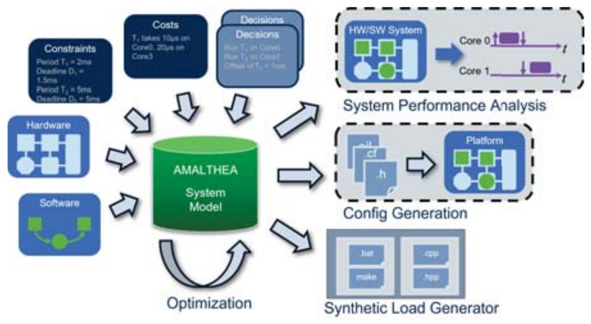


Figure 1: APP4MC

Recently a synthetic load generator was added to the tool portfolio. It transforms AMALTHEA models into sequential C code and is described in 2.2.

2.1 The AMALTHEA DSML

In order to investigate the (timing) behavior of systems, a suitably abstracted description of the factors must be available. The Eclipse APP4MC open-source project provides the AMALTHEA DSML that combines various partial models, which together contain the information necessary to conduct a performance simulation, analysis and optimization. An overview is depicted in 1. It includes *software*, *hardware* and *stimuli* models, among others. The software model allows to define *runnables*, i.e., smallest functional execution units, and *tasks*, i.e., smallest units of concurrency at the operating system (OS) level. The hardware platform is described in terms of devices, ports and connections. The data exchange between the devices (i.e. processing units, memories) happens via connections that are characterized by attributes like latency and bandwidth. The stimuli model contains the activation of tasks via stimuli. Figure 2 shows the top level elements, described above, of an AMALTHEA model.

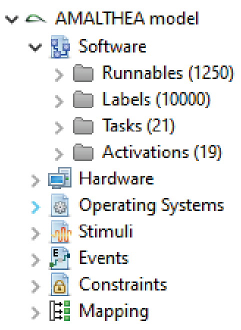

AMALTHEA model
- Software
  - Runnables (1250)
  - Labels (10000)
  - Tasks (21)
  - Activations (19)
- Hardware
- Operating Systems
- Stimuli
- Events
- Constraints
- Mapping

Figure 2: Top level elements of the WATERS 2016 AMALTHEA model.

For most model elements it is possible to attach custom properties that provide additional information for specific tools or use cases. These properties are organized as key-value pairs in a map and can be nested to represent more complex structures.

The description can vary in degree of abstraction and detail, depending on what stage of development or for what purpose the system should be examined.

2.2 The Synthetic Load Generator (SLG)

The synthetic load generator (SLG) [7] is provided as part of the APP4MC platform. The intend of the SLG is to mimic the performance of the modeled system on the target platform. In modern automotive systems, software from different suppliers will be integrated on the same target platforms. These new high-performance platforms are designed for high performance but not for timing predictability [8] and therefore challenging for safety argumentation. Analyzing and understanding the overall system early in the design process is crucial. Integration at model level and understanding the side effects on e.g., shared hardware resources or middleware overheads, is mandatory.

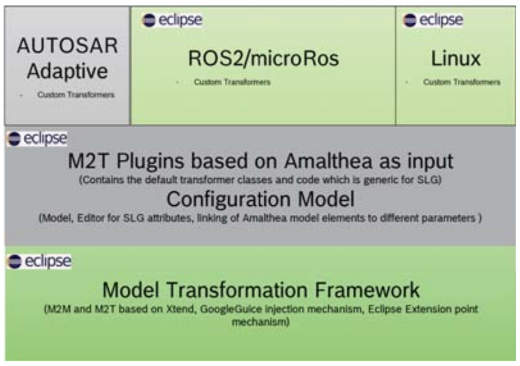

AUTOSAR Adaptive
Custom Transformers
eclipse
ROS2/microRos
Custom Transformers
eclipse
Linux
Custom Transformers
eclipse
M2T Plugins based on Amalthea as input
(Contains the default transformer classes and code which is generic for SLG)
Configuration Model
(Model, Editor for SLG attributes, linking of Amalthea model elements to different parameters)
eclipse
Model Transformation Framework
(M2M and M2T based on Xtend, GoogleGuice injection mechanism, Eclipse Extension point mechanism)

Figure 3: Synthetic Load Generator

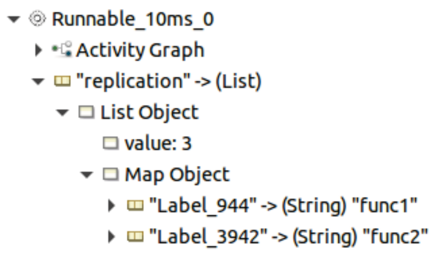
The SLG generates representative code that mimics the modeled performance behavior of AMALTHEA models to support this analysis. Relevant performance metrics include the executed cycles on the computation unit (e.g., CPU core) and the accesses to the memory subsystem. The SLG transforms every task in an AMALTHEA model into sequential C code, and each runnable into a separate C function. AMALTHEA tasks include an activity graph consisting of a sequence of calls to the different runnables.

The SLG generates task activation patterns that are either triggered periodically or by an event. The SLG covers the automatic generation of micro-ROS code. Micro-ROS [9] was implemented to bridge the gap between resource constraint μControllers with microRos components and performant μProcessors where ROS2 applications are hosted. The structure of the SLG repository is depicted in Figure 3. All boxes with the Eclipse logo are open source and available in the tools repository of APP4MC. The model transformation framework provides a general infrastructure to implement model-to-model and model-to-text transformation with mechanisms to e.g., provide dynamic code injections. On top of this framework the basic SLG is implemented. The M2T plug-in provides the general implementation how to generate executable code from AMALTHEA models. The Linux, ROS2/micro-ROS, and AUTOSAR Adaptive plug-ins extend the basic SLG to provide further customization to support Posix threads, ROS2/micro-ROS nodes and communication primitives as well as specialized code for the ETAS AUTOSAR Adaptive implementation RTE-VRTE [10]. The latter is not publicly available.

The implementation of the SLG is based on a general model transformation framework that provides an infrastructure to implement model-to-model (M2M) and model-to-text (M2T) transformations. The SLG transformations of the platform can easily be customized with dynamic code injections. These extensions generate the parallel replicas and the voting-based consensus algorithm. This automated approach will boost productivity and improve the model driven development method.

2.3 Extensions for resilience

This work addresses resilience by including features for software-based replication at the DSML level and its SLG counterpart. A new custom property, namely *replication*, can be attached to a runnable to specify that the runnable is to be replicated. This feature, illustrated in Figure 4, further specifies (1) the number of replicas to generate (3 in the example), and (2) the group of consolidation functions, combined with the labels they check, that are used after the replication process to verify the results.



```
▾ Runnable_10ms_0
  ▸ Activity Graph
  ▾ "replication" -> (List)
    ▾ List Object
        value: 3
      ▾ Map Object
        ▸ "Label_944" -> (String) "func1"
        ▸ "Label_3942" -> (String) "func2"
```

Figure 4: Snippet of an AMALTHEA model using a new custom property, namely *replication*, associated to *Runnable_10ms_0*, from WATERS 2016. The property describes the replication parameters, i.e., the number of replicas and the consolidation function(s) linked with the variables they check.

Figure 5 represents the execution flow, in the form of a direct acyclic graph (DAG), of the replication example presented in Figure 4. The extensions introduced in the SLG transform the new *redundancy* property into a number of parallel replicas of the runnable (blue nodes), insert the synchronization required after the execution of the original and the replicated functionalities, and call the consolidation functions that validate the correctness of the results (one consolidation function per variable to check). As expressed in the workflow, this work currently considers *spatial* redundancy (meaning that all replicas can be executed at the same time, ideally in different processing units) and all replicas are waited for before the consolidation functions can execute. Section 4 discusses further extensions and optimizations to the presented model. The recovery mechanisms required when the results are incorrect remain out of the scope of this paper.

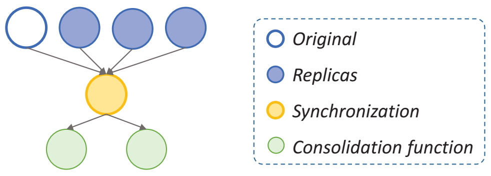


Figure 5: Execution workflow when replicating an AMALTHEA runnable as expressed in Figure 4.

A number of parallel languages can be used for the parallelization of replicas, e.g., C++ multithreading, Pthreads, OpenMP. For the purpose of this work, the SLG implements a system based on Pthreads.

3 Evaluation

This section evaluates the productivity of the proposed framework in terms of the slowdown caused by replicating different portions of a system.

3.1 Experimental setup

The application and environment used for the experiments are detailed next.

Application The experiment uses a simplified version of the *WATERS 2016 verification challenge* [4] model provided in the APP4MC platform. This model represents a complex engine management software composed of a number of cause-effect chains, and includes 21 preemptive and cooperative periodic and sporadic tasks, containing 1250 runnables that access a total of 10000 labels. The model has been simplified to execute the three more time-consuming tasks, i.e., *Task_10ms*, *Task_20ms* and *Task_50ms*, called *T10*, *T20* and *T50* henceforward. The name of each task corresponds to the period used to trigger the task, meaning that at time 0 all three tasks are triggered, at time $10ms$ only *T10* is triggered, at time $20ms$ *T10* and *T20* are triggered, etc. The tasks include sequences of runnable calls, more specifically 304 in *T10*, 307 in *T20* and 46 in *T50*. As a part of this work, the sequences of runnables have been parallelized based on the accesses to the labels, preserving sequential consistency. Figure 6 shows the DAG of the three tasks (dependencies are displayed left to right). The mechanism used to generate the graph minimizes crosses between edges to enhance readability. As a consequence, the graphs show more parallelism than actually exposed. Originally running in an infinite loop, the code generated with the APP4MC SLG for the simplified model of WATERS 2016 has been adapted to run during the hyper period of the selected tasks, i.e., 100ms. Finally, the Weibull distribution used to determine the number of ticks expended by the runnables of the model has been fixed to be upper bound in order to maximize the use of the system.

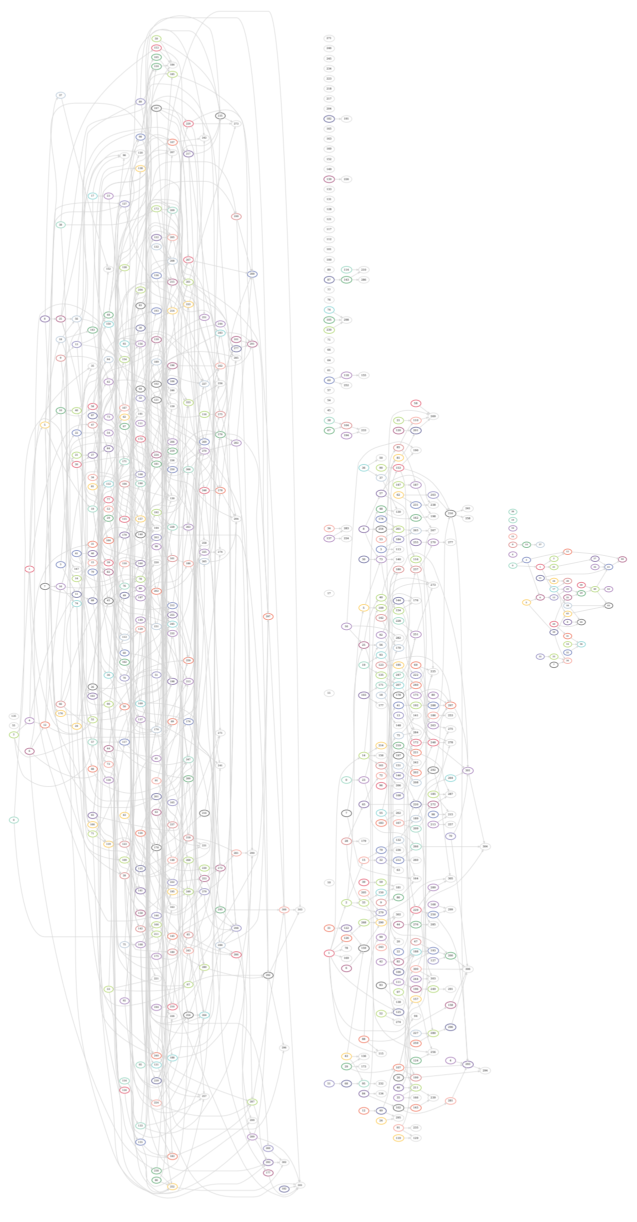

(a) T10 (b) T20 (c) T50

Figure 6: Direct acyclic graphs representing the simplified WATERS 2016 model, where nodes represent runnable instances (each color is a different runnable), and edges (left to right) represent data dependencies.

Compilation and tools The LLVM 15.0.0 compilation framework with the $-O2$ optimization flag is used to generate the executable application.

Hardware The generated code is executed on an NVIDIA Jetson AGX Xavier System-on-Module featuring 8-core NVIDIA Carmel 64-bit ARMv8.2 @2265MHz and 16GB 256-bit LPDDR4x @2133MHz, and running an Ubuntu 18.04.5 LTS operating system. Extrae [11] tracing tool and Paraver [12] visualization tool have been used during the evaluation of the application.

3.2 Preliminary analysis

In order to evaluate the feasibility of replicating computation in the aforementioned scenario of the WATERS 2016 model, a preliminary evaluation of resource occupation has been performed. Figure 7 shows the execution trace of the 100ms hyper period. The columns formed in the trace show that in all cases (i.e., executing (a) *T10*, *T20* and *T50*, (b) *T10*, and *T20*, and (c) *T10* and *T50*) most of the time between periods is idle.

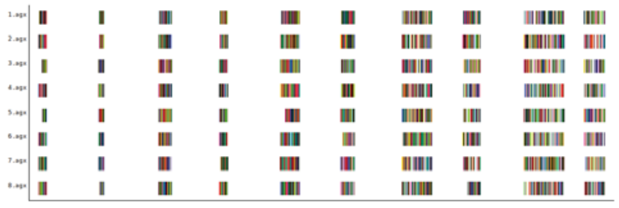


Figure 7: Execution trace of the simplified WATERS 2016 code generated with the APP4MC SLG when running *Task_10ms*, *Task_20ms* and *Task_50ms* for a hyper period of 100ms.

Runnables in the WATERS 2016 model have a very fine granularity, from $0.7\mu s$ to 22.6μ, with an average execution time of $3,47\mu s$. Extrae and Paraver are very useful to understand the overall behavior of a system. In this case however, where numerous small runnables are triggered frequently, Extrae incurs in too much overhead to be used in a performance analysis. Consequently, manual instrumentation is used at the application level to determine amount of time the machine is in use. Table 1 shows the execution time of each of the combinations of tasks that occur during the 100ms. For example, for four times *T10* is run in isolation take a total time of 0.76ms, while the only time the combination of *T10*, *T20* and *T50* is executed take 0.3ms. Overall, the system, which is running for 100ms, performs computation during a total of 2.41ms. This evinces the great potential to implement parallel replication, reducing the impact of this fault-tolerance technique, without jeopardizing the schedulability of the system.

Tasks	Execution time (ms)
T10	0.76
T10, T20	1.12
T10, T50	0.23
T10, T20, T50	0.3
TOTAL	**2.41**

Table 1: Average execution time (in ms) of all possible combination of tasks in a given period, for a hyper period of 100ms.

3.3 Impact on the use of resources

The use of the machine when running the WATERS 2016 simplified model is illustrated in Figure 8. The figure shows the amount of time computing the tasks of the system, in ms, of an overall time of 100ms (the hyper period of the considered tasks). This amount is computed as the average of 100 iterations. Since the total amount of time considered is 100ms, the time used for executing the tasks corresponds to the percentage of time the NVIDIA Jetson AGX Xavier is in use. The replication is applied to a random set of runnables, so the runnables that are replicated might be different for different replication configurations. The use of the machine shows a linear trend, indicating the contained impact of replication mainly due to the availability of resources while executing the application without replication. The variability in the granularity of the tasks and the randomization of the replication process explains the deviations in the trend. In all the iterations performed, the execution of all tasks including replication finished within the corresponding period.

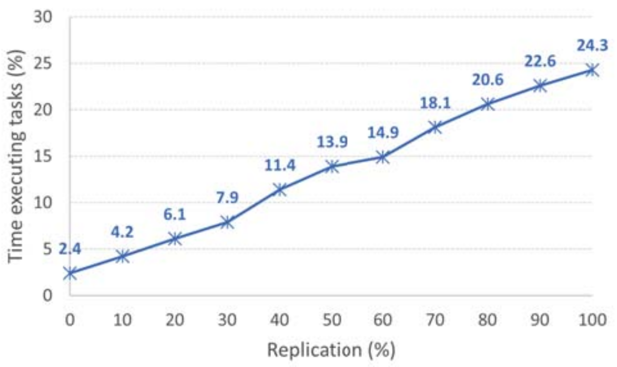


Figure 8: Computation time (in ms) of tasks *Task_10ms*, *Task_20ms* and *Task_50ms* from WATERS 2016, for a hyper period of 100ms.

4 Conclusions and Future Work

The use of parallel heterogeneous architectures in complex CPS with dependability constraints allows the introduction of replication mechanisms that enhance the reliability of the system while minimizing the overhead. As shown in the evaluation of a simplified version of the WATERS 2016 verification challenge, the exploitation of the parallel opportunities of the system allows replicating a number of critical functionalities restraining the impact on the performance and so providing better opportunities for maintaining the schedulability.

Exposing replication at the AMALTHEA DSML level further allows maintaining the *correct-by-construction* principle behind model-based design provided by (1) a number of mechanisms for verification and validation at the model level, and (2) a code generation tool that automatically transforms the model into C code.

The current work relies on a custom property that, attached to a runnable (functionality) defines its level of replication and the consolidation functions that implement the consensus-and-voting mechanism. For future versions of this work, the Automotive Safety Integrity Level (ASIL), already supported in AMALTHEA models as an attribute of the runnables, can be considered in order to determine the number of replicas and type of replication (spatial or temporal) required. Additionally, the *MooN safety architecture* used to design Safety Instrumented Functions (SIF) in order to achieve the Risk Reduction Factor (RRF) required for each ASIL can also be considered. In this architecture, *M-out-of-N* components (e.g., sensors) must act to perform the corresponding function correctly. This information can be used in software replication to determine the number of replicas that must finish before the consensus-and-voting mechanism is executed.

5 Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669.

References

[1] E. Quiñones, S. Royuela, C. Scordino, P. Gai, L. M. Pinho, L. Nogueira, J. Rollo, T. Cucinotta, A. Biondi, A. Hamann, *et al.*, "The AMPERE Project: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization," in *International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 201–206, IEEE, 2020.

[2] Eclipse, "APP4MC," 2022. `https://www.eclipse.org/app4mc/`.

[3] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf, "Special Session: Future Automotive Systems Design: Research Challenges and Opportunities," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–7, 2018.

[4] Arne Hamann, Simon Kramer, Martin Lukasiewycz and Dirk Ziegenbein, "International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) verification challenge," 2016. `https://waters2016.inria.fr`.

[5] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "AUTOSAR–A Worldwide Standard is on the Road," in *International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, p. 5, 2009.

[6] Syed Aoun Raza, "PLAT4MC: Multicore Performance Optimization with Open Source," 2017. `https://www.microconsult.de/1712-0-PLAT4MC-Multicore-Performance-Optimization-with-Open-Source-ESE-2017.html`.

[7] Eclipse, "APP4MC SLG," 2022. `https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.addon.transformation.git`.

[8] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 133–145, 2022.

[9] "micro-ROS." https://micro.ros.org/.

[10] "ETAS RTA-VRTE." https://www.etas.com/de/portfolio/rta-vrte.php.

[11] Barcelona Supercomputing Center, "Extrae," 2022. https://tools.bsc.es/extrae.

[12] Barcelona Supercomputing Center, "Paraver," 2022. https://tools.bsc.es/paraver.

Software Tool for Evaluation of Multi-Sensor Object Tracking in ADAS Systems

A.Medaglini, S. Bartolini

Department of Information Engineering and Mathematics, University of Siena, Via Roma 56, Siena, Italy; email: {alessio.medaglini, sandro.bartolini}@unisi.it

V. Di Massa

Thales Italy, Via Lucchese 33, Sesto Fiorentino, Italy; email: vincenzo.dimassa@thalesgroup.com

F. Dini

Magenta srl, Via B. Pasquini 6, Florence, Italy; email: fabrizio.dini@magentalab.it

Abstract

Nowadays, the innovations of AI and other automated decision-making software are spreading to many different areas. The automotive field in particular is rapidly shifting towards the concepts of Advanced Driver Assistance Systems (ADAS), which could bring huge benefits in the future. However, before being able to use these tools, many assurances are required regarding their functioning and safety. To this end, several control techniques exist to evaluate the performance of this software, but a reliable and repeatable method for evaluating complex scenarios and corner cases is still lacking. In this paper, we propose a suite of tools for the generation and analysis of synthetic tests, aimed at evaluating and analyzing the functioning of autonomous driving systems in order to measure their effectiveness and drive their development.

Keywords: synthetic test, autonomous driving, software tool.

1 Introduction

In the past ten years there was an exponential growth in the use of electronic components and software in automotive systems. Driven by the revolutions in AI and machine vision, the automotive field has been profoundly renewed by inserting an ever-increasing number of driving aid tools inside cars, from lane keeping to complete autonomous driving systems. In particular, thanks to the huge amount of data that can be collected, taking advantage of inertial platforms, sensors, and so on, automated decision-making systems are becoming increasingly popular. This huge amount of innovations is shifting the market towards Advanced Driver Assistance Systems (ADAS), with the aim of developing fully autonomous vehicles in the future. In such scenario, to verify the operation and evaluate the performance of this new kind of vehicle, a very broad and thorough analysis is required. In fact, as stated in [1], "Autonomous vehicles would have to be driven hundreds of millions of miles and sometimes hundreds of billions of miles to demonstrate their reliability in terms of fatalities and injuries". As one can imagine, it is not possible to verify such a requirement with tests done with real vehicles only, but it must also be accomplished by exploiting simulation tools to fully evaluate the reliability of these kinds of vehicles. With regard to tests with real vehicles, there are also some drawbacks that make their use not recommended. The main of them are listed below:

- there is a too wide variety of scenarios to be explored and it would require an unimaginable amount of time, and resources.
- it could be difficult to exactly replicate a specific test scenario to verify if an improvement can help in facing it correctly.
- it is difficult to obtain quantitative measurement of performance using real collected data since it does not gives us any reference about the desired behavior.
- danger for the personnel involved during the testing procedure is too high to take a chance, especially in edge cases [2].

Nowadays, for all these reasons, simulated tests constitute an essential way for the automotive industry to provide the safety requirements of autonomous vehicles, reducing the mentioned costs and giving a speed up to the testing procedure. Nevertheless, performing tests that produce quantitative, repeatable, and comparable results remains challenging for autonomous vehicles, since the reliability of the results is strongly dependent on the accuracy of the simulated information used as input for the software. In particular, one of the crucial parts of the systems for autonomous driving is managing the problem of Object Tracking and Obstacle Detection (OTOD) [3], which is the focus of this paper. Situation awareness is indeed a crucial aspect to be able to develop properly working decision-making systems that are fully reliable in urban traffic scenarios. For this reason, we focus on the study of generating scenarios for such critical activities. In fact, our work aims at evaluating and measuring OTOD features and performance through a modular and efficient simulated approach.

In this area, there are some proposals that have been made to manage such issues, which can be split into two categories. The first is more oriented to the generation of test scenarios starting from real data, while the other is based on the use of mathematical models. The approaches in the first category gather data from real-world runs, to obtain a database of scenarios that can be used for building the simulated scenarios. On the other hand, approaches in the second category rely on abstract scenario generation, defined by mathematical or semantic languages which are then translated into test scenarios. Unfortunately, these methods can be expensive in some respects or require a huge base of data to combine. Furthermore, the problem of evaluating the final results of the elaboration of these product scenarios is not always completely addressed, and many evaluation criteria are often based on different performance indicators whose interpretation and composition are not uniquely determined [4].

Within the second category, we propose a methodology that models the salient aspects of tracking and sensing objects, to effectively abstract the necessary facets to test OTOD behavior in ADAS systems. Our tool generates synthetic scenarios that replicate the sensors' perception of the objects around a vehicle, so that they are representative and effective without burdening the model with unnecessary information, promoting high modularity and flexibility. The main contribution of this paper can be summarized as follows:

- a method for the evaluation of object detection and tracking submodules of an ADAS system through a simulated approach.
- a synthetic scenario generation tool configurable through a simple scripting language that allows describing scenarios quickly and concisely.
- a performance evaluation based on the obtained results, through automatic calculation of aggregated Key Performance Indexes (KPIs), to produce comparable and easy-to-understand reports from test execution.

2 Related works

In the category of methods based on real data, [5] proposes an approach for scenario generation based on a scenario database. In particular, their schema builds upon the generated scenario database structure that clearly identifies the key components of a given autonomous paradigm. This abstraction enables the creation of parameterized test cases to test the autonomous functions under various adaptive conditions. Their scenario database consists of data collected from multiple sources, and stores information about real-world sensor data, parameters required to create the setup for testing, and scenario definitions related to the functions supported. Similarly, de Gelder and Paardekooper [6] propose a method for evaluating the performance of the functions in an ADAS system based on real-life scenarios, taken from the Streetwise database, combined with Monte-Carlo simulations. Another possible approach, proposed in [7], is based on the use of advanced perception systems for obtaining reference data used for the automated generation of simulated driving scenarios. In this case, the data provided by their referenced sensor system can be transferred into a simulation tool, to obtain virtual scenarios from real-world scenarios. The second category instead relies on abstract scenario generation. For instance, in [8] the scenes and the related assertions are defined by a matrix-based semantic language and translated into test scenarios in simulation. They developed a semantic language for breaking down the factors that define a scenario, taking the input from the command line, and parsing the formal grammar to generate tokens. Then, using a matrix-based system they generalize the scenario characteristics. The numerical matrix is read as input where each row is a different assertion describing a single road piece or actor that can then be parsed to generate the scenario. In [9] they generate both static, i.e. scenarios where objects just follow a predefined trajectory, and hybrid scenarios where the vehicles need to deviate due to the influence of other vehicles in order to avoid a crash. In order to do that they use a combinatorial interaction-testing algorithm together with a backtracking algorithm and a motion planner.
Our approach is mainly related to the category based on mathematical models, but we differ from the reported works for the following reasons:

- our scenario definition language is simple and human-readable, it does not involve matrix definition or other complex mathematical formalization of the items present on the scene.
- in our case complex algorithms are not required to create scenario data starting from its representation, requiring a great computational power, but the output is automatically generated starting from the objects involved and based on the sensors used, lightly and quickly.
- our tool also models how different sensors perceive the environment, replicating their transducer characteristics and specific format of data and meta-data, along with the different kinds of noises that can affect them.

3 The ADAS system: a real-world use-case

Thales Italy (TH-ITA) industry has developed a multi-sensor ADAS system for city trams for obstacle detection and collision avoidance. The system assists and supports the driver in avoiding collisions by detecting and tracking obstacles in real-time, thereby compensating for driver errors. The aim of TH-ITA is to enhance the safety of trams and light rail vehicles, to the benefit of passengers, service operators, and other traffic participants. Although technology cannot replace human drivers, it can complement human perception and decision making – often deciding between life and death. Indeed, the system will be able to significantly reduce the number of rear-end collisions involving tram vehicles and, as a result, will help to avoid high follow-up costs.
The complexity of city traffic requires cognitive capabilities to improve vehicle reactivity and perception of near- and long-range obstacles. The development of this technology and its impact on light rail transit will result in improved safety of daily operations. In fact, tramways can be considered a challenging application for autonomous driving systems for many reasons. First of all, compared to mainline railways, tram rails are not always segregated from road traffic and pedestrians.

For this reason, while in mainlines any detected obstacle on the track has to be considered a threat to safety [10], for a tram driving system it is not so straightforward to discriminate whether an object on the track rails can constitute a safety threat or not, depending on the specific situation. For example, a car could drive on the rail, in front of the tram, preceding it while driving in the same direction. In the same way, it is common to find people crossing the rails or in close proximity of them, for instance in all those cases where the tram is approaching a platform with people standing and walking in the surroundings. These tramway scenarios are normal and should not cause an alert. In contrast, when the ADAS system detects a car or pedestrian whose future trajectory can be predicted to intersect the tram's one, it shall generate an alert so that the driver can stop or slow down the tram to avoid the collision. In fact, in a typical use case, there are many vehicles that can move around the tram, which is also moving. This situation generates a variety of possible scenarios, according to the different obstacles, weather, and lighting conditions around the tram.
In addition, the same driving situation is certainly much more dangerous for a tram vehicle than an automobile. This is mainly caused by the high braking distance required by a tram, which is very different respect to the one of a car, due to the great difference in weight and coefficient of friction with the respective transit surfaces between the two types of vehicles. For instance, as highlighted in Figure 1 [11], a tram traveling at $37km/h$ needs about $20m$ to stop, the same distance required by a car moving at more than $60km/h$. For this reason, tram driving requires more caution than car driving, due to the absolutely relevant momentum of the former even at relatively low speeds, which can create safety hazards for other road users during its operation.

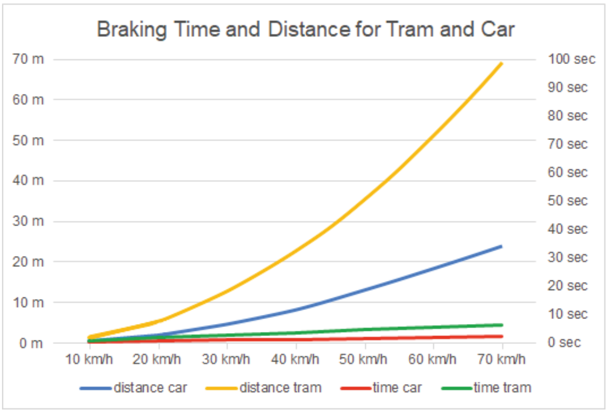
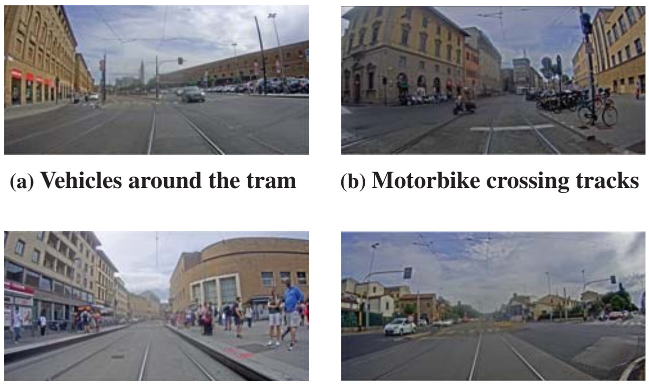


Figure 1: Braking time and distance for tram and car.

In Figure 2 some possible everyday scenarios are reported, with different objects moving around the tram. Each image within the figure shows different types of objects and road topologies that the tram encounters along its path. In fact, there are crossroads, crosswalks, platforms with pedestrians, and cars moving parallel to the rail tracks or intersecting them. Depending on the evolution of their behavior all these objects can become obstacles for the tram. We consider an obstacle any possible object (including cars, bicycles, animals, pedestrians, and other objects) that can collide with the tram because it stands between the rails or because it stands nearby and its shape and trajectory are suitable for a collision. The fact that both the tram and the surrounding objects can move poses critical issues from the point of view of the correct identification of the objects and the nature of their movement. In particular, there are situations in which the tram is stationary and has objects moving around it and others where the tram is moving and this affects the relative speed of the other objects (both moving and stationary). This latter case can produce critical effects for the object tracking algorithm. For instance when the tram curves, especially if the turn has a narrow radius, all the objects rapidly shift on the scene, and their speed and position change abruptly.



(a) Vehicles around the tram **(b) Motorbike crossing tracks**

(c) People at the tram stop **(d) Crossroads**

Figure 2: Typical urban scenarios taken from real tramway

The images in Figure 2 are taken from the camera mounted on the real tram. In fact, to be able to perform object detection and tracking algorithms, the vehicle must be equipped with a heterogeneous set of sensors. In particular, in the TH-ITA case, each tram has been equipped with two cameras, a radar, and a lidar. The way these sensors perceive the environment is critical to obtain good performance from ADAS systems [12].

The overall architecture of the TH-ITA ADAS system can be represented as depicted in Figure 3, where the entire data pipeline is reported. The system is composed of three main subsystems: a set of sensors that are installed on the tram vehicle, a data association and tracking (DAT) module, and a collision checker module (CCM).

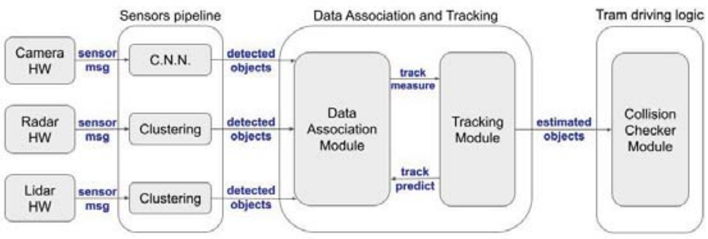


Figure 3: Architecture of an ADAS system

First of all, the sensors collect raw data from the real world, which are then analyzed by the respective pipelines to provide the system with bounding boxes representing detected objects. These data are then propagated to the DAT module, which is in charge of elaborating the raw data collected by the sensors and associating them with the objects tracked by the system. This submodule tracks the targets' evolution and produces estimates on the future positions of the objects, which are sent to the CCM of the system. This last module deals with the driving logic of the tram, deciding whether the future position of an object will be critical to the system, causing a collision, and if so providing an alert to the tram driver. As can be easily understood, the DAT module is the core of the ADAS system and, within it, we mainly focus on the detection and tracking aspects. In fact, the reliability of the system is based on the correctness of the object detection and tracking phase.

4 Proposed approach

Developing a multi-sensor ADAS requires evaluating the behavior of association and tracking algorithms, which are fed with data from sensors. To track real-world objects it is crucial to model how they are perceived by sensors. Therefore, we addressed both the simulation of reality, in a faithful but synthetic way, and also how this reality is perceived by the different sensors. Our proposal hence focuses on a method to generate scenarios that replicate as precisely as possible the trajectories of the objects and the sensor's perception of reality. In this way, it is possible to feed the data association and tracking algorithms with realistic and accurate data, for producing meaningful results that can then be analyzed. Indeed, the generation of test scenarios and the evaluation of the results obtained are two critical aspects that must be addressed jointly to be able to properly steer the development of an autonomous driving system. For this reason, we propose a software tool to guide the development of an autonomous driving system and to measure its effectiveness, starting from the description of a synthetic scenario to its implementation and evaluation. The first part of this proposal is dedicated to identifying a novel procedure to generate synthetic scenarios, while in the second part we deal with the evaluation of the obtained results using an automatic report generator based on reliable KPIs previously defined.

To define the different scenarios and cases of study we follow an incremental approach. Initially, we classify all the possible behavior of a single object moving around the vehicle, distinguishing them according to the direction, trajectory, and position of the object. Then, we model the behavior of each possible sensor used for sensing the environment (camera, radar, and lidar in our case), specifying the typical characteristic of each one and modeling the noise that can affect them. In this way, by combining all the possible trajectories of objects and the way they are perceived by the different sensors, thousands of randomized variations of each specific scenario can be easily and effectively run. Lastly, the tool aggregates the results from multiple runs and can automatically produce reports summarizing the setup parameters, for experiment repeatability, and the achieved results through easily specifiable KPIs.

4.1 Objects movements

Regarding the first aspect, simulation of real objects, we identified some macro scenarios of typical configurations for the tramway system and the objects around, based on data collected by Thales Italy. A classification of these basic macro-scenarios is presented below:

- Static obstacle on the track rails: such obstacle could be a car blocked on the rails, a fallen bicycle or motorbike, a tree branch, or other unexpected objects.
- Obstacle moving along the rail: a car or other vehicle moving along or beside the rail, from a side or the other one. This obstacle should have a size that is sufficient for impacting the tram or a trajectory too close and dangerous.
- Obstacle moving at a distance from the tram, but with trajectory and speed that are compatible with a future collision. This can happen when approaching a crossroad or a roundabout where vehicles cross the railway track.
- Obstacle moving in the nearby of the tram, but with trajectory and speed that are compatible with a future collision. This can happen for instance when the tram is stopped near a traffic light or a tramway station and other vehicles intersect the train tracks, passing close to the tram.

All these scenarios are very frequent during the travel of the tram since it is moving in a city area crowded with people and different typologies of vehicles in the surrounding of it. In particular, for the trajectories that move in front of the vehicle, it is important to make a distinction based on the distance from it, as it has an impact on the response time required to the vehicle. Furthermore, the direction in which the different trajectories are traveled is also relevant as it affects the vehicle's field of view. The above scenarios can be combined with each other to generate more complex situations, building new scenarios starting from multiple specific behaviors of different objects on the basis of the superposition principle or generating interference between them. The taxonomy of basic behaviors we found for the objects, as reported above, is summarized in Figure 4. This figure depicts the trajectories as plan views, with the trajectories as they would appear when viewed from above the tram.

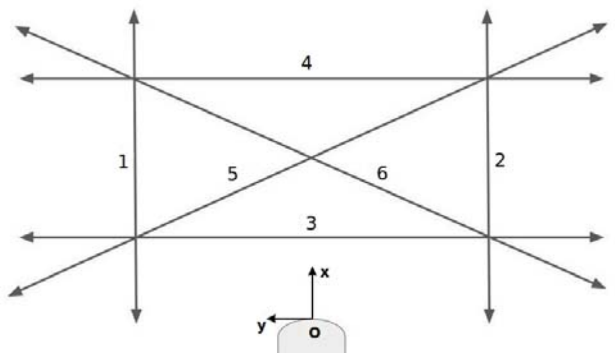


Figure 4: Basic trajectories of objects around the tram.

At the bottom of the figure the reference system is reported, which is centered in the front part of the tram with the x-axis directed along the direction of travel of the tram, the y-axis directed on the left, and the z-axis upwards. In particular, each of the lines represents one of the main categories of trajectories that we have identified as basic trajectories that an object can follow in the urban environment around a tram. In fact, from the analysis we carried out, each object around the tram can have different behaviors and therefore follow trajectories that can make it an obstacle for the train. These trajectories cover lateral movements with respect to the vehicle (num. 1,2), which does not intersect its trajectory, and movements that instead intersect its trajectory, both perpendicularly (num. 3,4) or with different angles (num. 5,6). Some examples of these trajectories can be seen in Figure 2: the cars in Figure 2a are moving following the trajectories of type 1 and 2; type 3 and 4 trajectories can happen in scenarios like the one reported in Figure 2d; and in Figure 2b an instance of the trajectory 5 is reported, but you can easily imagine similar scenarios for the type 6 trajectory. All the trajectories in Figure 4 can be traveled in both directions, as underlined by the arrows drawn on both ends.

4.2 Modeling objects and movements

To model the behavior of objects on the scene, as explained above, it is necessary to estimate the temporal progression of their kinematic characteristics, just as sensors would capture them in the real case. Each sensor collects different information from the environment, and we identified that the movement of an object can effectively be described by a limited set of basic motion patterns like uniformly accelerated rectilinear motion or circular motion. In fact, each sensor allows to collect different information, but it is always possible to describe an object through its position, speed, and acceleration at different instants of time. We, therefore, decided to model the behavior of objects based on the description of these three kinematic characteristics, evolving them over time according to appropriate physical laws. In our model, each object is defined by specifying its initial position (assumed in the origin of the Cartesian system if not specified) and an initial speed and/or acceleration (assumed null if not specified). From that moment on the motion of the object is modeled using uniform or uniformly accelerated rectilinear motion laws using the set parameters. Providing different inputs for each of them at different instants of time it is therefore possible to accurately describe arbitrary motions of the different objects. Motion parameters (e.g. direction, acceleration) can also be changed, even randomly, at each time instant within the object's lifetime, so that even more complex motion laws can be easily modeled. Furthermore, such scenarios can be combined with each other to generate more articulated situations encompassing multiple objects and evaluating the consequent interaction effects in the object detection and tracking algorithm.
The object description is based on a formal language used to define each case. It is implemented in a human-readable form so that it is easy to write and immediately understood at first glance, as can be seen in the following snippet:

```
GLOBAL_END_TIME,40000
1,0,POS,2,−20,0
1,0,VEL,1,1,0
2,0,POS,2,20,0
2,0,VEL,1,−1,0
1,300,ACC,1,0.5,0
2,450,VEL,1,−2,0
```

The language exploits a comma-separated value format, where the first element in each line is the object id, the second one is the time instant (in milliseconds) at which the following property is applied, and the third is a kinematic keyword (*POS* for position, *VEL* for velocity and *ACC* for acceleration) representing the property we want to specify and, finally, there are the values of the previously specified property for each Cartesian coordinate *(x,y,z)*. This triplet is expressed as meter (m) for position property, meter per second (m/s) for velocity, and meter per square second (m/s^2) where the property is equal to acceleration. We also create two other reserved words which are *GLOBAL_END_TIME* to define the last time instant in our scenario (on a reserved line) and *END* which is used instead of the kinematic property of an object to remove such object from the scenario at the specified time instant. Through this language, it is possible to define the position, speed, and acceleration along every Cartesian direction *(x, y, z)* for each involved object. These properties can be specified at each time instant within the lifetime of the object so that we can create even complex varying motion laws in a few steps. A specific part of our software is then in charge to verify that the described scenario is consistent, in order to avoid unexpected or not feasible behaviors. Using this procedure it is possible to describe every feasible scenario, and then obtain automatically an output file for each simulated sensor. The output file is created again in a CSV format, easily readable by both humans and machines, containing the kinematic properties of every object at each time instant, with a frequency corresponding to the specific simulated sensor's characteristic period. The output file can then be used to test each scenario within the specific application under development.

4.3 Modeling of sensors

During the scenario generation procedure we allow modeling of each sensor, accounting for its output data format, transducer characteristics, as well as the noise that can affect it. In particular, the output data format comprises the type and variety of positional data and meta-data (radar cross-section, probability of existence, object class, and so on) provided by the sensor. In fact, our system takes care of modeling each different kind of sensor in use, because sensors are one of the most relevant parts of ADAS systems. They are in charge of capturing information from the environment, and therefore the final results strictly depend on their functioning. Moreover, since sensors also capture noise in their measurements, coming from poor external conditions or electromagnetic interference depending on the kind of sensors, it must be considered in the model. Consequently, our tool allows easy modeling of different possible measurement noises like additive zero-mean Gaussian noise, the type of noise that most commonly affects sensors, with a range of variances estimated from real data, and also some random zero/saturation/missing values can be simulated according to arbitrary statistical distributions. The features described above allow you to have high repeatability and controllability of the test scenarios, to make a huge number of experiments in many different conditions with the opportunity of easily steering the software development. Through this procedure, however, some approximations are introduced. These approximations mainly concern the dynamics of objects and the model used for the sensors, which cannot be as accurate as the real one. A fundamental aspect is in fact the trade-off between the accuracy of the scenario and the computational power required to realize it, which must be optimized to make the designed scenario computable, as well as reliable, and significant.

4.4 Report generation

Another aspect covered by our tool is system evaluation. In fact, to evaluate the performance of the system it is of paramount importance to be able to compare the obtained results with a ground truth measurement of the specific scenario used. If the scenario comes from real-world data, a ground truth reference can be obtained only if we perform manual data inspection and labeling. This manual inspection and labeling procedure is prohibitively expensive to be performed extensively and accurately. Instead, using synthetic data, we describe the scenario we want to test and then simulate equivalent sensors to produce the information used to feed the data association and tracking components. Finally, we can compare the results with the known ground truth. Moreover, when we produce simulated sensor data we are able to inject known noise into the simulated data to model real-world noise. This way we can analyze and measure the performance of the implemented algorithms, and also infer properties about sensors' noise and test the resiliency of the system against them.

Once the scenarios have been tested, the final stage is to collect the produced results and aggregate them into meaningful KPIs. This phase is extremely important and comparative reports play an important role in each industry, for evaluating the performance level of a software application and for driving its evolution. Typically, those kinds of reports are manually composed from large amounts of information provided by various heterogeneous sources. Processing this information is tedious, time-consuming, and error-prone. For all these reasons, we have developed an automatic tool that aggregates the results from multiple runs of the tested software, executed on different scenarios or with different functioning parameters, and produce a comprehensive report which summarizes the environmental setup of the specific execution and the obtained results evaluated using some previously defined KPIs.

5 Performed experiments and evaluation

5.1 Methodology

Our synthetic model is based on the analysis of data collected on real trams. We carried out qualitative tests by analyzing videos of cars, pedestrians, and other objects to empirically estimate their motions and the parameters to be used. Using our generation language it is possible to specify the initial conditions of each object and modify them during its movement. Object positions evolve according to discrete-time cinematic equations of uniform or constant accelerated rectilinear or circular motions, possibly with varying parameters over time. In time periods between two changes to the dynamic of a specific object its evolution is based on the physical laws of uniformly accelerated rectilinear motion. Obviously, if the acceleration is not present in the input parameters it is assumed equal to zero and the two equations reduce to a uniform rectilinear motion.

An important aspect to evaluate the behavior of the system is represented by the performance metrics to be used. In this work, we decide to focus on evaluating the performance of the multi-object tracker inside the ADAS system. In order to do so, we need to understand what qualities we expect from it. In an ideal world, such a tracker should, at all points in time, be able to identify the correct number of objects on the scene and estimate the position of each of them as accurately as possible. Additionally, we expect the tracker to be able to consistently track each object over time. This means that, if each object has been assigned to a unique track ID, it remains constant throughout the entire sequence (even after a temporary occlusion). This leads to the following evaluation criteria for performance metrics:

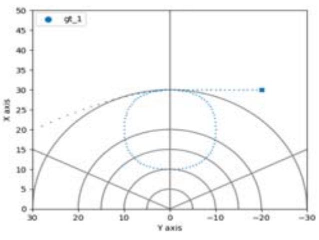
1. They should allow to judge a tracker's precision in determining exact object locations.
2. They should reflect its ability to consistently track object configurations through time, that is, to correctly trace object trajectories, producing exactly one trajectory per object.
3. They should be clear, easily understandable, and behave according to human intuition.
4. They should be few in number and yet expressive, so that they can be used, for example, in evaluating complex systems.

Based on the above criteria, we adopt the MOT Metrics [13] performance evaluation of the multi-object tracker inside the ADAS. The two most used metrics in this area are the Multiple Object Tracking Precision (MOTP) and the Multiple Object Tracking Accuracy (MOTA). In particular, we have chosen to calculate the MOTP also for each sensor used. This allows us to understand how each individual sensor contributes to the overall performance of the system, by varying the noise level modeled for each sensor. Then, we decided to add another metric to our KPIs: since we are mainly interested in evaluating the behavior of our tracker in the immediate surroundings of the tram, i.e. the areas with the greatest risk of collision, we have decided to also use the Root Mean Square Error (RMSE) as a performance index. To this aim, we have divided the tram's field of view (FOV) into sectors, based on the distance and angle of objects relative to the tram, using a plan view of the scenario, and we compute the RMSE value for each of them. This is useful for understanding how the position of an object, its distance and angle with respect to the tram, affect the ability of the system to track it correctly. Indeed, performance evaluation has different levels of interest-based on the location of sectors, with those close to the vehicle being the most critical for collision detection. For this reason, the FOV of the tram has been divided in such a way to focus the attention on the bands closest to it, with circular sections of increasing width as the distance increases. Furthermore, the FOV has been divided into four lateral slices according to the angle of the objects with respect to the tram, to separate the objects that are located in front of the tram from peripheral ones, which may be more difficult to follow. Lastly, it is worth noticing that thanks to the modularity and flexibility of our tool, it is easy to add and tune new KPIs to manage specific situations if needed.

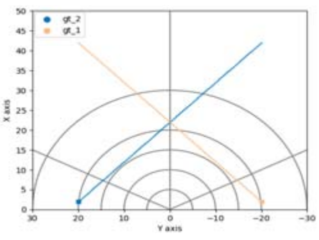
5.2 Experiments and results discussion

The test framework we developed allows us, through scenario design and test reports generation, to have deep insight into the system operations. There are a lot of parameters that an ADAS system depends on, and in this way we can evaluate the effects of each of them. Among the various analyzed trajectories and scenarios, we focus here on two of them, reported in Figure 5:

1. In the first case an object is simulated moving in front of the tram with a curvilinear trajectory, modeling its movement around a roundabout (taken from the real case of Batoni square in Florence).
2. The second case deals with a multi-object scenario where two objects move from each side of the tram with intersecting trajectories, modeling typical road intersections with vehicles coming from different directions.



(a) Roundabout motion



(b) Crossing trajectory

Figure 5: Example of test scenarios

In the first test we performed, reported in Figure 6, the sensors are modeled without noise, i.e. the ADAS system was fed with "exact" synthetic data, in order to evaluate their performance and behavior characteristics regardless of other influences. This is not realistic, but it is useful to tune the covariances of errors in the model we adopted so that it can cope with the dynamics of the objects. In fact, different sensors sense the environment differently, and this brings some difference in the measures they provide, even if they are referred to the same object. These differences in the measured position of the same object become noise for the system, and this needs to be accounted for in the measurement models. Thanks to our scenario generator we can simulate the behavior of the real sensors, and this allows us to visualize the differences between, for example, the lidar and the radar view of the same object. In Figure 6, the image above shows the trajectory estimated with lidar measures only, while in the image below the trajectory is estimated with radar measurements only. As the figure shows, the two trajectories are slightly different, because measurements from the two sensors are different in many ways: different timing, different estimated positions, and different measurement models.

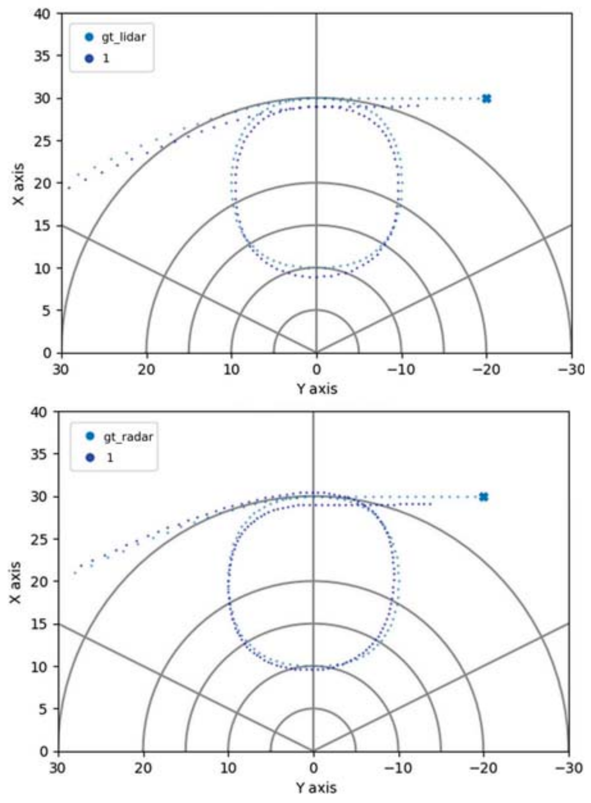


Figure 6: Single object track against GT, without noise. Above: lidar only. Below: radar only.

In this case, the track based on radar measures is more precise, as confirmed by the MOTP metrics value, which in the radar and lidar cases is equal to 84,3% and 90,8% respectively. This is due to the fact that the higher radar sensing ratio helps the system to correctly track the object. Therefore the different behavior in different conditions of the various sensors gives us the possibility of fusing their output, in order to enhance the overall performance of the system.

A second important feature we tested through our scenario generator regards the noise in the data, which we can control by varying the covariance of an additive zero-mean Gaussian noise that we inject into the input data. For different levels of noise estimated position of the targets appears farther from the ground truth and few measurements are left unassociated. This can cause the tracker to lose its target and, if the unassociated measurements are enough, they could be used to instantiate a new tracker following the same target. By carrying out various tests we are able to investigate the sensitivity of MOTP values to noise variation. In Figure 7 are reported the plots from two different simulations, with increasing noise covariance in sensor positions going from 0.5 to 0.9. A high level of noise in the measures degrades tracking performance, increasing the MOTP metric value from 53,7% to 69,3%. It is worth noting that these injected noise levels are larger than those usually recorded on sensor data, and we decided to apply them as a stress test for the system.
As a result, not only estimated position of the targets appear

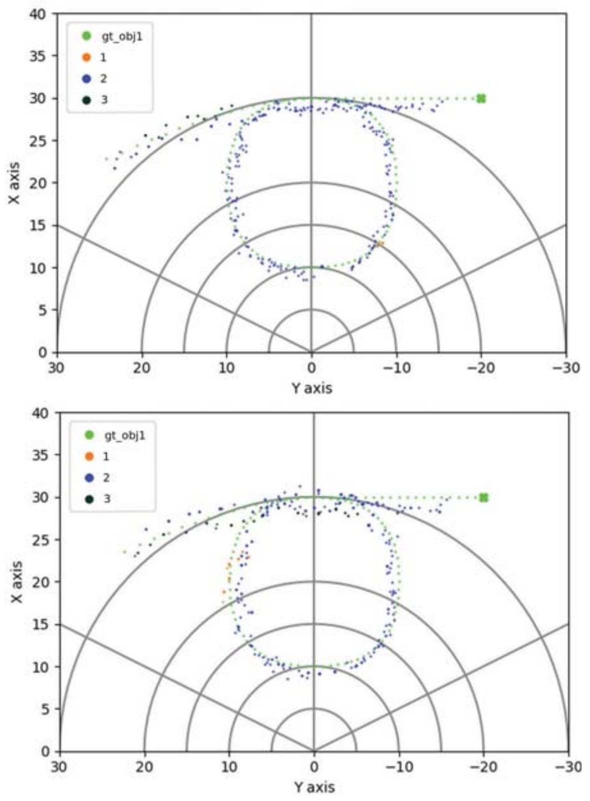


Figure 7: Effects of increasing noise in measurement data. Above: 0.5 covariance noise. Below: 0.9 covariance noise.

noisier and farther from the GT, but some measurements are also left unassociated. As expected, this causes the creation of incorrect trackers. In fact, in both graphs, the orange and dark green dots represent different trackers from the first one (blue dots) that are used to track the same object. By carrying out this kind of experiment, we can investigate the sensitivity of the system to sensors' noise variation, measuring how much the accuracy of a single sensor affects the overall tracking results of the ADAS system.

Another key aspect addressed by our test framework is the sensitivity of the system to misdetections, i.e. cases when an object is present in the scene but sensors do not detect it. In fact, we define a "misdetection" as the event in which an object present in the scene does not cause a corresponding signal in a sensor's scan. When this happens the object is non-existent for that sensor at some points in time. Actually, there are quite a few reasons why an object can be misdetected: the sensor may suffer from a temporary or local failure that prevents it from functioning correctly, there may be interference from external sources or, more likely, the object may be occluded by other objects or by the environment. Moreover, an object can also be mistakenly categorized as clutter and removed by decluttering filters. To test the system's sensitivity to objects' misdetections, our tool can simulate the random loss of measurements by defining the desired rate of misdetections for each sensor and evaluating the tracking robustness of the system. In this way, each time a measurement is supposed to be given as input by the system, there is a certain probability that the measurement will be simply dropped and not used by the system. In the plots in Figure 8 we simulated an increasing misdetection probability of 20% and 40% on all sensors.

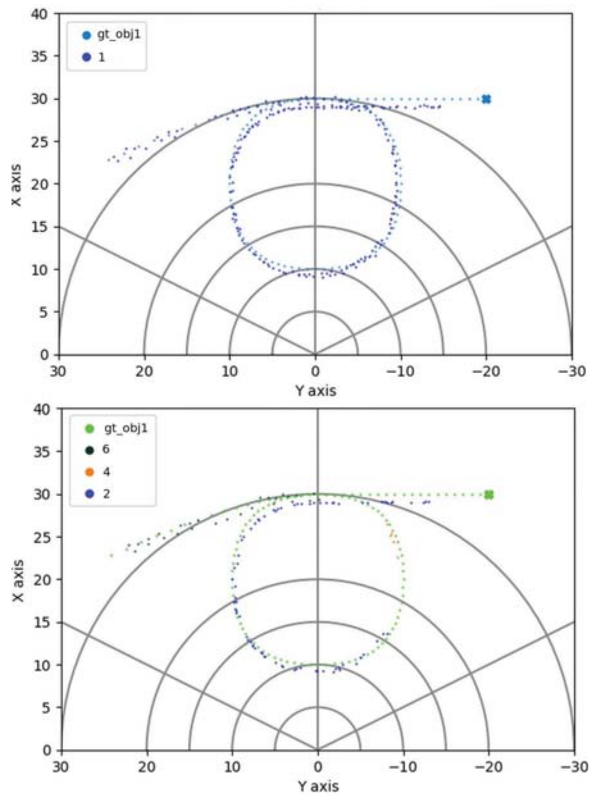


Figure 8: Effects of increasing probability of misdetection. Above: 20% misdetection probability. Below: 40% misdetection probability.

As shown from the figures, the system tolerates a high rate of misdetections, up to 20%, and this is probably due to the sequential approach we use to process the sensors' scans. In fact, although there is not a single sensor with a very high sampling rate, processing them when they arrive, one after the other, allows us to perform as if the system was fed by a single sensor having a sampling rate equal to the sum of each sensor's sampling rate. In the end, when the percentage of misdetections increases up to 40% the system cannot track the objects correctly anymore.

Single-target scenarios are useful to isolate specific issues that could affect the core of the ADAS system, and can give important information about the performance of the system in terms of intra-track performance, such as those measured by the MOTP metric. They may also give some information about simple data association errors, as we have seen by simulating misdetection. However, they are not sufficient to investigate more complex scenarios where different association errors may occur, such as identity mismatch. In order to do this, we need to define multi-object scenarios where two or more objects may interfere with each other. A very simple example of this can be seen in the second scenario we want to discuss, reported in Figure 5b. There, two objects start moving from each side of the tram, with intersecting trajectories. The objects' velocity is about 2.8 m/s, which is compatible for example with bikes or scooters. In this multi-object scenario, repeating the experiments about misdetection probability, we can observe an interesting behavior. In Figure 9 we can see that increasing the misdetection probability has no effect on the two predicted trajectories until 30% value is reached. Then, with this percentage of measurement loss, when the two objects intersect each other's trajectory, an identity mismatch occurs, as the bottom plot of Figure 9 shows.

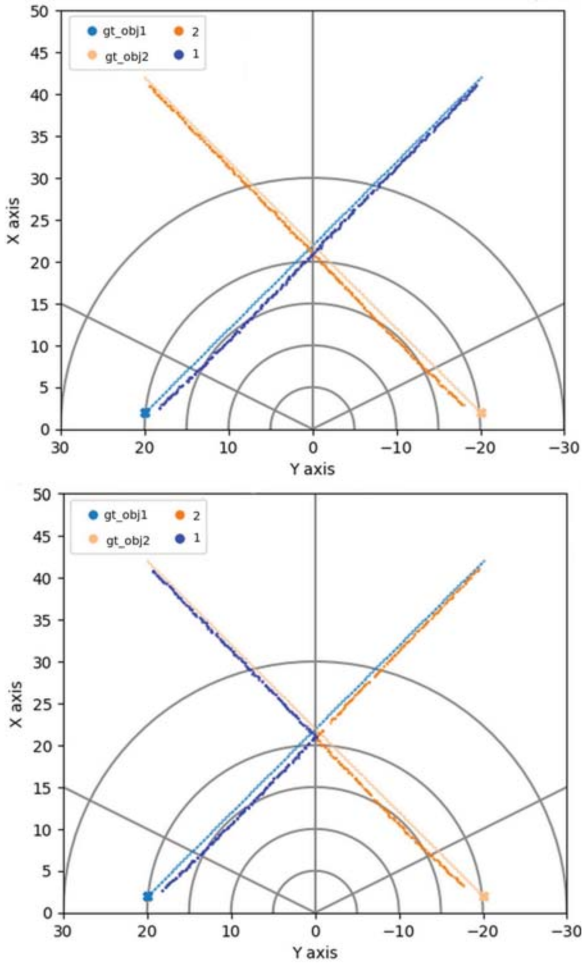


Figure 9: Effects of increasing probability of misdetection on a multi-object scenario. Above: 20% misdetection probability. Below: 30% misdetection probability.

What happened here is that one of the two trackers suffered from a lack of measures close to the intersection, and at some point the association algorithm has associated it with measures originating from the other object. These measurements were stolen by their "legal owner", who was later forced to associate with the measurements originated from the other object. In the last part of the trajectories, although other misdetections surely occurred, the two objects are too far away to interfere with each other, and the tracking proceeds smoothly up to the end.

Lastly, when the misdetection probability reaches 40%, something different happens as reported in Figure 10. Only the object moving from left to right suffered an association problem, while the other presented a continuous track. In this case, the association error is simply a misdetection error: one object suffered from a lack of measurements that elapsed long enough for the system to decide that the tracker had to be deleted, while the other continued to be fed with the correct measurements. This caused measurements from the first object to be left unassociated, and these were used to start another tracker.

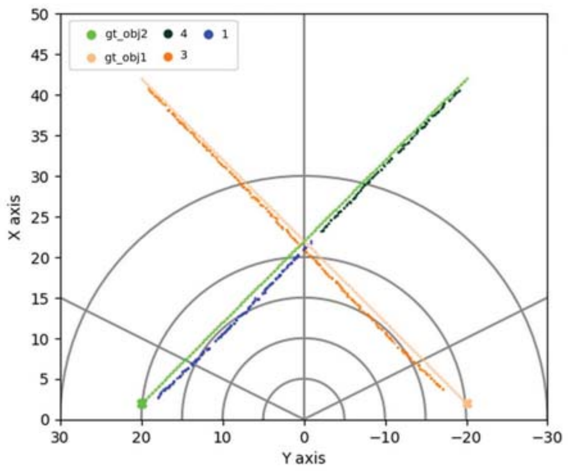


Figure 10: Objects on intersecting trajectories with 40% misdetection probability.

The overall result is two-fold. On the one hand, the object has changed its "identity" (from 1 to 4, see the legend) and therefore we could not reconstruct the object's history, should we need to. On the other hand, the actual physical object has been undetected for a while, and this is surely a more severe problem for an ADAS system. In other words, this kind of error leads to identity changes between objects and apparently strange paths, posing challenges in the obstacle-evaluation algorithms of the system.
As a final note, it has to be pointed out that the previous effect does not only depend on the increase of the misdetection probability. The phenomena we have described above could also occur with a lower level of misdetections since it all depends on the particular realization of the stochastic processes we are simulating in the test framework. For sure, higher levels of misdetections make that event more likely to occur, and this explains why we observed it in that particular experiment, but there is a non-zero probability that such an event could occur in any of the synthetic tests.

6 Conclusions and future work

Nowadays, the verification of the behavior of autonomous driving systems is critical for their deployment in everyday life and the importance of synthetic test environments is increasing. In this paper, we tackled the problem of testing the behavior of ADAS systems by means of a flexible tool for generating synthetic scenarios and evaluating KPIs. We have proposed a methodology for identifying test scenarios and presented a simulation framework for generating and running such scenarios and evaluating the system performance. Our scenario generation procedure uses a simulation model based on a simple scripting to describe the different scenarios, which are then run modeling sensors' behavior. Then, our tool can evaluate the overall system performance by aggregating execution results from multiple runs. Our methodology brings complementary features compared to existing ones for the evaluation of object detection and tracking systems in ADAS systems, as it allows modular, effective, and repeatable simulation of representative scenarios with limited effort.

As we explained in a few examples in the previous section, our performance measurement tool allows very detailed investigation about the problems that may relate to the Thales Italy ADAS system and to autonomous driving systems in general. Furthermore, the proposed tool allows evaluating results both qualitatively and quantitatively, using easily adoptable KPIs and graphical representations of the objects' trajectories. However, even if our framework works as expected, it needs to be tested more extensively, for assessing completely its functionality and fine-tuning it. In this respect, thanks to a new feature we are developing, we will include also the explicit modeling of the train motion in the scenarios. This will allow an easier and more precise modeling of setups where complex trajectories are performed by objects and by the train. Moreover, we want to enrich the level of automation inside our scenario generation tool, exploiting a statistical approach to automatically derive the object motion parameters from real-world use-cases, generating then many small variations of them to test the system behavior extensively. Lastly, we also want to compare our work with other strategies based on real-world data, trying to apply them jointly to exploit the strengths points of each one.

References

[1] N. Kalra and S. M. Paddock, *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, 2016.

[2] C.-H. Yu, Y.-Z. Chen, and I.-C. Kuo, "The benefit of simulation test application on the development of autonomous driving system," in *2020 International Automatic Control Conference (CACS)*, 2020, pp. 1–5.

[3] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, "Computing systems for autonomous driving: State of the art and challenges," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469–6486, 2021.

[4] L. Li, W.-L. Huang, Y. Liu, N.-N. Zheng, and F.-Y. Wang, "Intelligence testing for autonomous vehicles: A new approach," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 2, pp. 158–166, 2016.

[5] A. Erdogan, E. Kaplan, A. Leitner, and M. Nager, "Parametrized end-to-end scenario generation architecture for autonomous vehicles," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, 2018, pp. 1–6.

[6] E. de Gelder and J.-P. Paardekooper, "Assessment of automated driving systems using real-life scenarios," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 589–594.

[7] U. Lages, M. Spencer, and R. Katz, "Automatic scenario generation based on laserscanner reference data and advanced offline processing," in *2013 IEEE Intelligent Vehicles Symposium Workshops (IV Workshops)*, 2013, pp. 146–148.

[8] C. Medrano-Berumen and M. I. Akbaş, "Abstract simulation scenario generation for autonomous vehicle verification," in *2019 SoutheastCon*, 2019, pp. 1–6.

[9] E. Rocklage, H. Kraft, A. Karatas, and J. Seewig, "Automated scenario generation for regression testing of autonomous vehicles," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017, pp. 476–483.

[10] P. Hyde, C. Ulianov, J. Liu, M. Banic, M. Simonovic, and D. Ristic-Durrant, "Use cases for obstacle detection and track intrusion detection systems in the context of new generation of railway traffic management systems," *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, vol. 236, no. 2, pp. 149–158, 2022.

[11] W. Leutzbach, *Introduction to the Theory of Traffic Flow*. Springer Berlin, Heidelberg, 1988.

[12] S. Sivaraman and M. M. Trivedi, "Looking at vehicles on the road: A survey of vision-based vehicle detection, tracking, and behavior analysis," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 4, pp. 1773–1795, 2013.

[13] R. Stiefelhagen, K. Bernardin, R. Bowers, J. Garofolo, D. Mostefa, and P. Soundararajan, "The clear 2006 evaluation," vol. 4122, 04 2006, pp. 1–44.

The Work of Proof in SPARK

Claire Dross
AdaCore, 75009 Paris, France; email: dross@adacore.com

Abstract

Since Ada targets safety-critical programs, many features of the language introduce safety nets in the form of language-mandated checks. Even if compile-time verification is preferred to runtime verification whenever possible, many of these checks are still done dynamically, an exception being raised in case of violation. The addition of contracts in Ada 2012 follows a similar trend, as a violation causes an exception to be raised when the code is compiled with assertions enabled. The SPARK tool aims at statically verifying that language-mandated checks and the user-written contracts can never fail at runtime. In this article, we give insights on how the tool works in practice and what are the most important challenges as of today.

Keywords: SPARK, proof of program, deductive verification.

1 Introduction

Since Ada targets safety-critical programs, many features of the language introduce safety nets in the form of language-mandated checks. For example, a check is made on every signed integer computation to make sure that it will not exceed the bounds of the underlying machine type. Even if compile-time verification is preferred to runtime verification whenever possible, many of these checks are still done dynamically. If they fail, an exception is raised at runtime. For example, calling the procedure `Increment` below on `Integer'Last` will result in an exception as the increment of `X` will overflow.

```
procedure Increment (X : in out Integer) is
begin
   X := X + 1;
end Increment;
```

The addition of contracts in Ada 2012 follows a similar trend. If they are enabled at compilation, the boolean expressions supplied as pre and postconditions of subprograms or as type invariants are evaluated at runtime, and an exception is raised if this evaluation returns False. As an example, consider the following annotation for the procedure `Increment`:

```
procedure Increment (X : in out Integer) with
   Pre  => X ≠ Integer'Last,
   Post => X > X'Old;
```

If `Increment` is called on `Integer'Last` and the precondition is enabled at compilation, an exception will be raised as its boolean expression evaluates to False. Similarly, if a bug is introduced in the implementation of `Increment` so that its parameter `X` is no longer increased by the call, an exception will be raised when checking the postcondition.

SPARK is a static analysis tool for Ada. It allows users to verify that the language-mandated checks and the user-written contracts in their program will never fail at runtime without running the program. It is open-source[1] and available through the Alire package manager. In this article, we give insights on how the tool works in practice and what are the most important challenges in its implementation as of today.

2 Formal Proof of Programs

SPARK verifies programs at the source code level on all possible inputs at once using *deductive verification* [1, 2]. Figure 1 schematizes the different steps of the verification process. First of all, the user is responsible for annotating their program with contracts. These contracts can express properties the user wants to verify on their code, but we will see later that some contracts can be necessary even to verify language mandated checks. Then, the tool transforms the annotated program into a set of logical formulas called *verification conditions*. There can be one verification condition or more for each property that is verified on the Ada program, be it a check or a user-written contract. Finally, the verification conditions are given to automated solvers. If all the formulas are verified, the program is correct.

Deductive verification is modular on a per subprogram basis. This is necessary for the verification process to be scalable in practice. Contracts are used to summarize what the guarantees are for each subprogram, both from the caller's and from the callee's point of view. When analysing the subprogram itself, SPARK verifies that, for all inputs that fulfill the precondition, the subprogram executes safely (there are no runtime errors) and the postcondition holds on subprogram exit. As an example, let us consider the procedure `Increment` presented before. To be able to verify its body, it is necessary for the user to supply a precondition preventing it from being called with `Integer'Last`. With the contract proposed before, the SPARK tool is able to verify both that no overflows can occur during the increment, and that the postcondition necessarily holds at the end of the call.

As the verification is modular, the SPARK tool does not look at the body of called subprograms, it only considers the contract: the precondition is checked and the postcondition is used to get information about the values of the objects modified by the subprogram after the call. As an example, let us consider the procedure `Foo` defined below. It calls

[1] `https://github.com/AdaCore/spark2014`

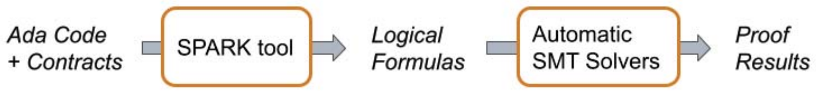


Figure 1: Deductive verification in SPARK

`Increment` twice in a row on a variable initially initialized to 0:

```
procedure Foo is
   X : Integer := 0;
begin
   Increment (X);
   Increment (X);
end Foo;
```

For each call to `Increment`, the analysis tool needs to verify that the precondition holds. For the first one, the verification succeeds. Indeed, `X` is known to be 0 before the call, which is not `Integer'Last`. The verification fails however for the second call. Looking at the body of `Increment`, we know that `X` should be 1 at this point, so the precondition would evaluate to True if we were to execute `Foo`. The SPARK tool however, will fail to verify it. As it works modularly on a per-subprogram basis, it cannot look at the body of `Increment` when analysing `Foo`. Instead it looks at its postcondition, which is not precise enough to rule out the possibility of `X` being `Integer'Last` after the first call.

In general, to verify a program using SPARK, it is necessary to annotate all subprograms with contracts precise enough to entail together the correction of the complete program. The preconditions should be strong enough to verify the subprograms themselves, and the postconditions to verify their callers. In our example, if we change the postcondition of `Increment` as below, then both procedures can be automatically verified:

```
procedure Increment (X : in out Integer) with
   Pre  ⇒ X ≠ Integer'Last,
   Post ⇒ X = X'Old + 1;
```

Unfortunately, the underlying logic used for deductive program verification is undecidable. As a result, it is not possible for such a tool to be able to decide on every program whether it is correct or not. Some tools, commonly called *bug finders*, focus on reducing the number of false alarms - they try to only report a bug when they are sure that there is one. SPARK on the other hand is a *sound* verification tool. If it can verify a program, then the program is correct[2]. However, it is not *complete* - if the verification fails, that does not necessarily mean that the program is incorrect. It is also possible that some contracts are missing, or that they are not sufficiently precise to verify the program. As we have seen before, it is the case of the postcondition of `Increment` presented in Section 1 which is not strong enough to verify the procedure `Foo`. Finally, the background solvers might also be unable to verify a valid formula in the allocated time. This sometimes makes it difficult for users to understand why the verification of a program is failing. The SPARK tool tries to help the investigation by providing counterexamples whenever possible. Unfortunately, the incompleteness of the background solvers means that this is not always possible, in particular when the program becomes more complex. Efficient and reliable counterexample generation for deductive verification is still being researched [3, 4].

[2]The soundness of the tool relies on assumptions. For example, since the verification is done on the source code, it assumes that the program is compiled correctly.

3 Why3 as an Intermediate Language

The SPARK tool works by using in the background bleeding edge technology developed by academic researchers in the formal verification domain. The Why3 platform [5], developed at Inria in France, performs deductive verification on a ML like semi-executable language called WhyML. It generates the verification conditions and translates them into the input language of various automated or manual solvers. As schematized in Figure 2, Why3 is used as a backend by several tools targeted at mainstream programming languages such as C [6] or Java and more recently Rust [7]. SPARK is such a frontend for Ada.

Compared to other frontends of Why3, SPARK can take advantage of the constraints imposed by the Ada language to reduce the need for user-written annotations and to make the verification process more efficient. In particular, the constraints imposed by the expressive type system of Ada are reflected on the Why3 side through assumptions. As an example, let us consider the record type `My_Rec` defined below. It has a discriminant `Length` which is used to bound the length of its array component `F`. The elements of the array are natural numbers bounded by a constant `Max`.

```
subtype My_Nat is Natural range 0 .. Max;
type Nat_Array is array (Positive range <>)
   of My_Nat;
type My_Rec (Length : Natural) is record
   F : Nat_Array (1 .. Length);
end record;
```

In the generated WhyML code, values of type `My_Rec` are assumed to fulfill its *dynamic invariant*: a predicate which gives both the bounds of the array component `F` and the range of its elements. It is defined as follows:

```
predicate dynamic_invariant (x : my_rec) =
     first x.rec__f = 1
  ∧ last  x.rec__f = x.rec__length
  ∧ ∀ i : int.
       first x.rec__f ≤ i ≤ last x.rec__f →
         0 ≤ get x.rec__f i ≤ max
```

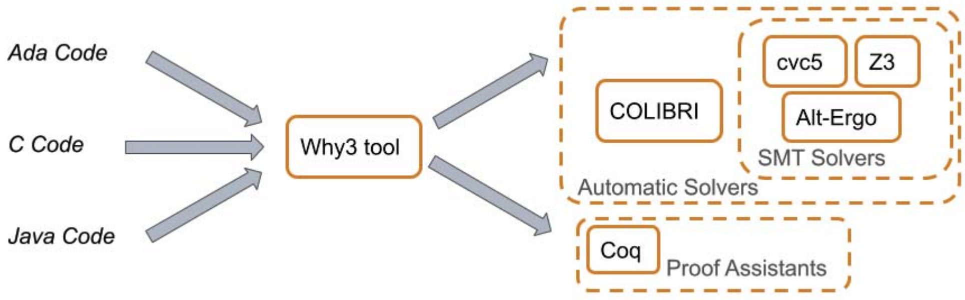


Figure 2: The Why3 platform for deductive verification

In general, the translation from Ada to WhyML does not attempt to preserve the executable semantics of the program, but rather to produce the simplest verification conditions possible while retaining soundness. In fact, the generated WhyML program is not even executable. For example, all subprograms are translated twice: once to generate the verification conditions for their body, and once to verify their callers. This has several advantages. In particular, it makes it possible to translate subprograms without worrying about the order of declarations and possible mutual recursivity. Indeed, following the order in which the Ada subprograms are declared would not be enough to avoid forward references in WhyML, as Ada allows calling functions which have not been defined yet inside contracts. It also allows the SPARK tool to use different contracts for the subprogram depending on the calling context. This is important to handle dispatching calls on tagged primitives in particular.

As an example, here is the declaration of `Increment` used for the verification of `Foo`. It is an abstract function without a body. Its `writes` contract states that it modifies its input `x`. The `requires` and `ensures` annotations are direct translations of the Ada contract. The fact that the parameter `x` fulfills the constraints of its Ada type after the call is made explicit in the postcondition.

```
val increment (x: ref int) : unit
  writes   {x}
  requires {!x ≠ 2147483647}
  ensures
    { !x > old !x ∧ dynamic_invariant !x }
```

When verifying increment itself, the translation is quite different. First of all, the parameters are declared as global variables instead of actual parameters of the subprogram. It makes it easier to verify entities nested inside the subprogram (a nested package or a nested subprogram for example). Then, while the Why3 translation of `Increment` has a postcondition, it does not have a precondition. This allows the SPARK tool to verify that the Ada precondition is *self-guarded*, that is, it can be evaluated in any context without raising an exception. Afterward, the Ada precondition is assumed before verifying the rest of the subprogram:

```
(* The parameters of Increment are global
   variables *)
val x : ref int;
let increment__def (_ : unit)
  ensures { !x.int__content > old !x }
=
  (* Assume that x follows the Ada typing
     rules *)
  assume { dynamic_invariant ! x };
  (* Check that no runtime error can occur
     while evaluating the precondition *)
  (begin
     ensures { true }
     let _ = !x ≠ 2147483647 in  ()
   end);
  (* Assume the precondition of Increment *)
  assume { !x ≠ 2147483647 };
  ...
```

4 Different Solvers in the Background

To discharge the verification conditions, SPARK relies mostly on *Satisfiability Modulo Theory (SMT)* solvers. These automated solvers are well-suited for program verification because they support natively theory symbols, that is, symbols which have a generally understood meaning outside of the context of the verification condition. These symbols include in particular integer or floating point literals and arithmetic operators, which occur often in programs. In general, a verification condition coming from an Ada program references symbols from several theories. As an example, the dynamic invariant of type `My_Rec` defined in Section 3 uses symbols from:

- the theory of linear arithmetic on mathematical integer types for the integer literals and the comparison operators,
- the theory of abstract data-types for record components, and
- the theory of infinite immutable arrays for the access to the array component.

It also uses the universal logical equality symbol, first-order quantification, and uninterpreted function symbols like `max`.

As stated before, the resulting logic is undecidable. It means that it is not possible to design an algorithm which would be able to determine on all such logical formulas whether they are valid or not without errors. The solvers used in the backend of SPARK are all sound - they never verify an invalid formula. However they are incomplete, so they might not be able to verify valid formulas. In particular, certain constructs are not efficiently supported by any solvers and are the subject of active research. Here are some of these topics:

- first-order quantification, in particular with alternating quantifiers,
- non-linear integer arithmetic, which is undecidable even on quantifier free formulas, and
- conditions involving symbols from different theories - floating point numbers and integers for example.

To alleviate these concerns, SPARK takes advantage of the capability of the Why3 platform to target different provers. By default, it uses three SMT solvers as a backend, Alt-Ergo [8], cvc5 [9], and Z3 [10]. All are independent open-source tools developed, at least initially, as part of a research endeavor. The three solvers are run on every verification condition, one after the other, until the condition is proved. This allows users to take advantage of the different strengths of these solvers.

In addition, as Why3 makes it possible to tune the translation specifically for each solver, the SPARK tool increases the diversity between the solvers by purposefully encoding the problem differently for each of them. Indeed, the best way to encode a feature of the language might depend on the use-case, and having several increases the chance of finding a proof. As an example, consider modular integer types. They can be encoded either as a mathematical integer, or as a bitvector of fixed size. The first encoding is often preferable when doing standard integer manipulation as well as when converting toward other numeric types such as signed integers and floating-point numbers. The second makes the support of bitwise manipulation more effective. To get the advantages of both worlds, the SPARK tool uses bitvectors for cvc5 and Z3, and mathematical integers for Alt-Ergo.

Even if SMT solvers are better suited for program verification, the use of other kinds of solvers, which might not suffer from the same caveats, is also investigated. In particular, constraint solvers could improve the provability on quantifier-free verification conditions involving conversions between values of different theories. The solver COLIBRI [11] is already available from the SPARK verification tool, though it is not used by default yet.

In general, choosing which provers to run and with which options and encoding is a question of trade-off. Running one more prover is time consuming, as it will be launched on all the verification conditions, and possibly run up-to the provided time-limit. This is why it is important to limit the number of solvers used by the tool, and to assess the efficiency and interest of each new addition thoroughly.

5 Expressivity versus Efficiency

Taking a step back, we have seen that verifying Ada programs using deductive verification is complex, and requires user input, in particular in the form of contracts. To make the tool usable in practice, it is important to find a balance in the accepted language between expressivity and ease of annotation and verification. For that, the SPARK tool introduces *simplifying assumptions*. At the simplest, these assumptions are features of Ada which are not supported by the tool, like side-effects in functions, or handling of exceptions. Others are more complex, for example all values are assumed to be valid, and no two objects can be aliases of each others - modifying an object cannot affect another object in a visible way.

For the SPARK tool to remain sound, it is important that it verifies that these assumptions are valid on the program. This is generally ensured by a separate analysis done by the SPARK tool, prior to running deductive verification. As part of this effort, a program flow analysis is run on the code to determine the effect of all subprograms and the global data that they access. Based on the results of this analysis, it is possible for the SPARK tool to ensure that functions do not have side-effects, and that no uninitialized variables can be read, which helps to rule out invalid values. Absence of aliases in programs using pointers is enforced through an ownership policy with its own specialized analysis.

Simplifying assumptions need to be chosen carefully so they are both effective - they reduce significantly the complexity of either the verification itself or the manual annotation process - and not overly restrictive. The right balance is generally hard to find, and can be refined in subsequent releases of the tool. For example, by default, SPARK enforces correct initialization of variables by a strict initialization policy: all inputs of a subprogram shall be entirely initialized at the point of call and all its outputs shall be entirely initialized when the subprogram returns. This restriction is useful, as it saves the user from having to annotate all their subprograms with contracts about initialization of values. However, it makes it impossible to annotate and verify a program which initializes a record component by component in separate procedures as the program below:

```
type Two_Fields is record
   F, G : Integer;
end record;

procedure Init_F (X : in out Two_Fields) is
begin
   X.F := 0;
end Init_F;

procedure Init_G (X : in out Two_Fields) is
begin
   X.G := 0;
end Init_G;

procedure Process (X : in out Two_Fields) is
  ...

procedure Main is
   X : Two_Fields;
begin
   Init_F (X);
   Init_G (X);
   Process (X)
end Main;
```

Since the parameter `X` of `Init_F` has mode `in out`, it is an input of the subprogram and SPARK will try to verify that it

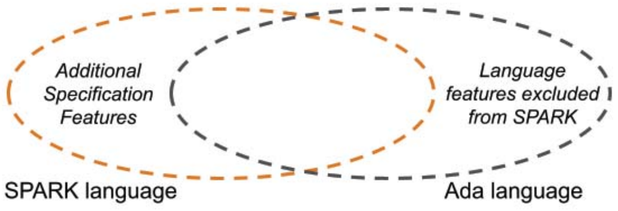

Additional Specification Features
Language features excluded from SPARK
SPARK language
Ada language

Figure 3: The SPARK language

is entirely initialized before the call to `Init_F` in `Main`. Obviously, it is not the case. Unfortunately, changing the mode of the parameter `X` of `Init_F` to `out` will not solve the issue as the tool will then try to verify that `X` is entirely initialized by `Init_F`, which is not the case either. To alleviate this issue, more recent versions of SPARK allow users to annotate their objects to exempt them from the initialization policy. However, it then becomes necessary for the user to manually add information about initialization in subprogram contracts:

```
type Two_Fields is record
  F, G : Integer;
end record;

procedure Init_F (X : in out Two_Fields) with
  Relaxed_Initialization => X,
  --  X is no longer subjected to the
  --  initialization policy of SPARK.
  Post => X.F'Initialized
  --  X.F is initialized by Init_F
  and (X.G'Initialized = X.G'Initialized'Old)
  --  X.G is left as it was
is
begin
  X.F := 0;
end Init_F;

procedure Init_G (X : in out Two_Fields) with
  Relaxed_Initialization => X,
  Post => X.G'Initialized
  and (X.F'Initialized = X.F'Initialized'Old)
is
begin
  X.G := 0;
end Init_G;

procedure Process (X : in out Two_Fields) is
   ...
--  The parameter X is not exempted from
--  initialization checks.

procedure Main is
  X : Two_Fields with Relaxed_Initialization;
begin
  Init_F (X);
  Init_G (X);
  --  Complete initialization of X is
  --  checked at this point.
  Process (X);
end Main;
```

The parameter `X` of `Init_F` is exempted from the initialization policy, so SPARK no longer tries to verify that it is initialized before the call in `Main`. Without additional annotations, it does not assume that it is initialized afterward either. It makes it necessary for the user to add a postcondition to `Init_F` to say that it initializes the component `F` of its parameter. The postcondition also says that the component `G` is not uninitialized by the call. It is not necessary to verify `Main`, as `Init_F` is called before `Init_G`, but it preserves the symmetry between `Init_F` and `Init_G`. The parameter `X` of `Process` does not need to be exempted from the initialization policy, as `Process` is necessarily called on entirely initialized data. As a result, we do not need to supply a contract about initialization for it. With these annotations, the code is entirely verified by the SPARK tool.

All these restrictions imposed by SPARK on top of Ada define a language subset. The language supported as input of the SPARK tool is called the *SPARK language*. It is not a strict subset of Ada, as can be seen on Figure 3, as it also introduces additional annotation features. As an example, *contract cases* are used to specify a contract as several smaller contracts with their own precondition and postcondition, and code which is only used for the specification process can be marked as *ghost* so that it is removed by the compiler when the assertions are not enabled at runtime. This language evolves continuously as more features are added to the tool. Major additions to the SPARK language are discussed online[3] and community input is always valuable.

6 Conclusion

SPARK is a static analysis tool for Ada. It verifies that no language mandated checks can fail at runtime, and that user written contracts always hold. The analysis is modular on a per subprogram basis: when analyzing a subprogram, the tool only uses the contract of called subprograms and not their bodies. As a consequence, the tool requires users to annotate all their subprograms with contracts precise enough to ensure together the correction of the whole program.

The SPARK tool is based on the Why3 plateform for program verification. The program and the contracts are transformed

[3] `https://github.com/AdaCore/ada-spark-rfcs`

into a set of logical formulas which are then verified by automatic solvers. The solvers used as the backend of SPARK are mostly SMT solvers. They work on first-order formulas with interpreted symbols coming from various theories - integer and floating-point arithmetic, bitvectors... This logic is undecidable. To work around prover's limitations, SPARK uses several automatic solvers for the verification.

To make both the verification and the manual annotation process tractable, SPARK introduces simplifying assumptions. These assumptions are associated to language restrictions which are verified by the tool. Both the SPARK proof tool and the related language restrictions are evolving continuously. For example, support for access types and its ownership policy have been added and extended in the last couple of years. Precise support for handling of exceptions is being discussed currently.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[3] S. Dailler, D. Hauzar, C. Marché, and Y. Moy, "Instrumenting a weakest precondition calculus for counterexample generation," *Journal of logical and algebraic methods in programming*, vol. 99, pp. 97–113, 2018.

[4] B. Becker, C. Lourenço, and C. Marché, "Explaining counterexamples with giant-step assertion checking," in *F-IDE 2021-6th Workshop on Formal Integrated Development Environments*, Electronic Proceedings in Theoretical Computer Science, 2021.

[5] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *European symposium on programming*, pp. 125–128, Springer, 2013.

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*, pp. 233–247, Springer, 2012.

[7] X. Denis, J.-H. Jourdan, and C. Marché, *The Creusot Environment for the Deductive Verification of Rust Programs*. PhD thesis, Inria Saclay-Île de France, 2021.

[8] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-ergo 2.2," in *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[9] 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.

[10] 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.

[11] B. Blanc, C. Junke, B. Marre, P. Le Gall, and O. Andrieu, "Handling state-machines specifications with gatel," *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 3, pp. 3–17, 2010.

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. Julio Medina
Facultad de Ciencias
Universidad de Cantabria
Avda. de los Castros s/n
39005 Santander
Spain
Phone: +34-942-201477
Email: julio.medina@unican.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

Ada Edge

27 Rue Rasson
B-1030 Brussels
Belgium
Contact:Ludovic Brenta
ludovic@ludovic-brenta.org

AdaCore

46 Rue d'Amsterdam
F-75009 Paris
France
Contact: Jamie Ayre
sales@adacore.com
www.adacore.com

ADA LOG

2 Rue Docteur Lombard
92441 Issy-les-Moulineaux Cedex
France
Contact: Jean-Pierre Rosen
rosen@adalog.fr
www.adalog.fr/en/

Deep Blue Capital

Jacob Bontiusplaats 9
1018 LL Amsterdam
The Netherlands
Contact: Wido te Brake
wido.tebrake@deepbluecap.com
www.deepbluecap.com

Ellidiss Technologies

24 Quai de la Douane
29200 Brest, Brittany
France
Contact: Pierre Dissaux
pierre.dissaux@ellidiss.com
www.ellidiss.com

KONAD
Software for Control and Administration

In der Reiss 5
D-79232 March-Buchheim
Germany
Contact: Frank Piron
info@konad.de
www.konad.de

PTC® Developer Tools

3271 Valley Centre Drive,Suite 300
San Diego, CA 92069
USA
Contact: Shawn Fanning
sfanning@ptc.com
www.ptc.com/developer-tools

SYSADA

Enterprise House
Baloo Avenue, Bangor
North Down BT19 7QT
Northern Ireland, UK
enquiries@sysada.co.uk
sysada.co.uk

systerel
Safe real-time solutions

1090 Rue René Descartes
13100 Aix en Provence
France
Contact: Patricia Langle
patricia.langle@systerel.fr
www.systerel.fr/en/

Tid rum

Tiirasaarentie 32
FI 00200 Helsinki
Finland
Contact: Niklas Holsti
niklas.holsti@tidorum.fi
www.tidorum.fi

VECTOR

Corso Sempione 68
20154 Milano
Italy
Contact: Massimo Bombino
massimo.bombino@vector.com
www.vector.com

WhiteElephant

Beckengässchen 1
8200 Schaffhausen
Switzerland
Contact: Ahlan Marriott
admin@white-elephant.ch
www.white-elephant.ch

http://www.ada-europe.org/info/sponsors

Ada
europe