24th International Conference on
RELIABLE SOFTWARE TECHNOLOGIES
Ada-Europe 2019

11-14 June 2019, Warsaw, Poland

CONFERENCE BOOKLET

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

In cooperation with

acm In-Cooperation

SIGAda

acm SIGBED

acm SIGPLAN

Ada Resource Association ARA

About this booklet

This booklet contains short summaries of all the presentations included in the conference core program. It groups presentations by session, prefixing their title with their type: 'IP' for industrial presentations, 'RP' for research presentations. The proceedings of the research papers will appear in a dedicated Special Issue of Elsevier's Journal on Systems Architecture. The proceedings of the industrial papers will appear in successive issues of Ada-Europe's Ada User Journal. We invite you to use this booklet as "navigational tool" throughout the conference program and its blank spaces provided in it, as a pad for your notes and commentaries.
Enjoy the conference!

Table of Contents

SESSION 1: ASSURANCE ISSUES IN CRITICAL SYSTEMS

IP: CONTRACT-BASED DESIGN AND VERIFICATION USING SPARK 2014

Simon Buist, Stuart Matthews, Thomas Wilson
simon.buist@altran.com, stuart.matthews@altran.com, thomas.wilson@altran.com

Summary

In this talk we will present our practical experience of using SPARK 2014 contracts in the implementation of a critical system. The system in question is an embedded protection system which monitors a set of sensor inputs. If the inputs indicate that the system is operating outside a safe envelope then the system is responsible for putting the system into a safe state. The system has to be engineered to the highest level of integrity under UK DEF STAN 00-56. We will consider how the use of contracts affects both the design and verification approach to engineering the system.

In the design process, low-level requirements for the system were captured as a set of control laws. As a first step we translated these control laws into a declarative specification in the form of SPARK contracts (Ada 2012 Pre and Post aspects) before writing the implementation in the subprogram bodies. This was initially a manual process performed against a well-defined set of rules but latterly has been automated through development of an in-house tool.

Verification of the system takes a hybrid approach, using both proof and test to establish functional correctness of the implementation. The SPARK contracts play a role in both these verification activities. First, they act as a formal specification of the required behaviour against which the SPARK toolset can prove the correctness of the implementation. Second, the run-time checking of the contracts ensures that they are always met during independent system testing. They provide a secondary level of expected outcome in these system tests, supplementing an independently produced reference model.

The Pre & Post contracts are also used in a third way, by leaving them built-in to the delivered executable. We designed the system so that any failure of such a run-time check will have the effect of putting the system into its safe state.

In an earlier stage of the project, we used TDD (Test-driven development) in the development of a proof of concept implementation. We needed to keep costs low, validate the early requirements as well as verify the implementation against them, and could tolerate some defects. However, a higher assurance level in the next stage of the project required us to increase the level of static analysis, so we opted to use our contract-based approach. In doing so, we dropped the use of TDD because the level of defects found in module testing was so low; we now had formally specified and validated requirements that we used static analysis to rigorously verify our implementation against. Even though statically proved, we still enabled the run-time checks of our contracts during independently- produced system tests and operation, primarily to detect things other than defects relating to the logical correctness of our module implementations against their contracts. For example, to detect: failure of assumptions on which the static analysis was based, and, indirectly, some hardware faults, low-level software faults, compiler faults and SPARK tool faults. We were able to take some credit for these run-time checks in our safety argument, and they even found an error in a low-level interrupt handler.

RP: JUSTIFYING THE SERVICE TO LOW-CRITICALITY TASKS IN A MIXED-CRITICALITY SYSTEM

Stephen Law, Iain Bate
stephen.law@rolls-royce.com, iain.bate@york.ac.uk

Summary

Significant work has been presented over the last decade looking at the application of Mixed Criticality Scheduling. The premise being that if a failure occurs the scheduler performs a mode change from normal mode to high-criticality mode. In high-criticality mode, some lower-criticality tasks are given a reduced service (e.g. not executed or executed at a different period). Recently work has been performed to bound the number of low-criticality jobs that might be skipped when the scheduler changes to high-criticality mode. However, a significant gap in the analysis has appeared with respect to identifying for how long the service provided to lower-criticality tasks may be reduced. This is essential as part of supporting software certification. In this paper, we consider a process designed to allow a system integrator to address this gap. The result is a safety argument with supporting evidence based on a real life case study, taken from a DAL-A certified aircraft engine control system.

SESSION 2: TOOLING AID FOR VERIFICATION

RP: INTEGRATING AN EVENT-BASED SIMULATION TOOL INTO THE ART2KITEKT FRAMEWORK

Joan Valls, Miguel García, Sergio Sáez
jvalls@iti.es, miguelgarcia@iti.es, ssaez@disca.upv.es

Summary

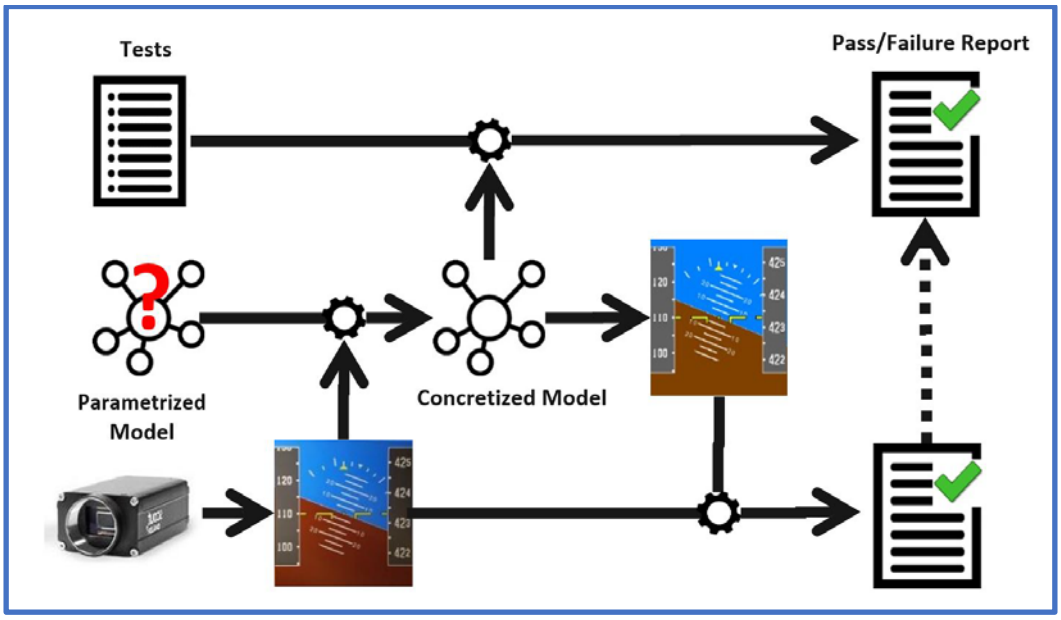
Simulation tools are a widely utilized method in the process of analysing and evaluating real- time systems. In this paper, we propose a new event-based simulation tool that is integrated into the art2kitekt framework, a web-based toolchain focused on the design and analysis of critical systems.
The main goal of this extension is to provide the capability of performing model-based simulation verification of autonomous reconfigurable systems to this framework. These systems are able to detect and correct hardware faults produced by hard environmental conditions e.g. in space navigation systems. However, verifying the correct behaviour of such systems is too complex from the analytical point of view, since the correct behaviour of the system not only depends on the temporal attributes of the different tasks but also on the functional behaviour of these tasks in presence of different system conditions.
The modelling capabilities of the art2kitekt framework have been extended to deal with the conditional activation of each executable block within a task depending on the current system state. This conditional execution has also been used to model the operational and recovery mode changes required in such systems.
Particular attention is paid to the specification of non-periodic events, such as the activation of sporadic tasks and external events or functional mode change requests, in order to perform a more detailed and accurate evaluation of the system behaviour.

IP: AUTOMATED DISPLAY TESTING IN TESTPASS

Tomasz Stanislawski, GE Aviation (PL)
tomasz.stanislawski@ge.com

Summary

Automated display testing is an important problem due to abundance of mission-critical devices enhanced with displays. Moreover, qualification of automated tests is proven to be a difficult problem. Machine vision algorithms tend to be complicated, costly to develop, and often cannot be fully analyzed and tested (i.e. neural networks).
TestPASS helps to simplify display testing. A concept of a parametrized model is a key component of the methodology. A model contains an abstract description of an image expressed in XML-based structured form. Models admit input parameters and an embedded script to improve flexibility and expressiveness.
Verification is split into two phases. First, the model is checked whether it correctly describes an observed image. Next, requirements and constraints are checked on an abstract model rather than on captured image.
The display content is captured using a camera or a frame grabber depending on an application. Next, the model is rendered using a qualified renderer producing a reference image. The image is compared with captured one with standard metrics like SSIMi [Structural SIMilarity Index (SSIM) – a measure of perceived similarity between two images based on local intensity, contrast and shape]. The successful check provides a certificate that a model is a valid interpretation of captured data.



TestPASS allows to express image tests in "pixel-agnostic" way. A tester expresses checks using only abstract attributes. The scheme allows to code tests using problem-specific attributes rather than results of image processing algorithms. Tester developer is potentially released from creation of any image processing code. Moreover, cost of updating tests is minimized because most modifications would be bound to a model only.

The process of estimation of input parameters can be delegated to a module named "MAGIC Algorithms for Graphical Interface Concretization" (abbrev. MAGIC). MAGIC is a combination of machine learning (ML) and discrete optimization algorithms. The module is extendable allowing developer to add dedicated estimators for cases not handled by generic primitives. The MAGIC module accepts a parametrized model and a captured image on input and outputs candidates for model parameters. The candidates become true parameters after verification by a simple qualifiable algorithm for image comparison.

This differs from a popular approach to ML algorithm, where the model is trained and qualified against a large dataset. The main drawback is an expensive acquisition of a large amount of labeled data that must be proven to be correct and representative for all relevant scenarios. On the contrary, the MAGIC models are cheaply trained with arbitrary number of samples generated directly from parametrized models. In TestPASS, the ML algorithm itself is not qualified and moved outside of qualification rigor. Only the outputs are verified individually for each execution of a test.

Overall, the following advantages are earned:

- Simplify development of test for display enhanced devices
- Allow development and qualification of tests before tested device is available
- Release tester from development of qualified machine vision algorithms
- Improve robustness of tests by expressing checks in abstract problem-related domain
- Improve tests re-usage by delegating image processing and interpretation to parametrized models and MAGIC

SESSION 3: BEST PRACTICES FOR CRITICAL APPLICATIONS

IP: CO-ENGINEERING OF SECURITY AND SAFETY LIFE CYCLES FOR SECURITY-INFORMED SAFETY-CRITICAL AUTOMOTIVE SYSTEMS

Barbara Gallina, Muhammad Atif Javed, Helmut Martin, Robert Bramberger
helmut.martin@v2c2.at

Summary

Automotive systems are becoming more and more connected. As a consequence, (cyber)security is paramount for safety and co-engineering of (cyber)security and safety life-cycles becomes fundamental to be ready for the engineering of security-informed safety-critical systems. Currently, no standard provides a co-engineering process. ISO 26262 [1] provides a standardized safety life-cycle, which needs to be complemented by requirements stemming from security standards (e.g., the upcoming security standard ISO-SAE 21434 [14]) and/or guidelines (e.g., SAE J3061 [2]). SAE J3061 is the only published guidebook that provides suggestions for considering both concerns.

The co-engineering of safety and security life-cycles and more broadly the co-engineering of multi mono-concern life-cycles, separately proposed in standards targeting mono-concerns, can be facilitated by the explicit systematization and management of commonalities and variabilities, implicitly stated in the requirements of the different standards.

Security-informed Safety-oriented Process Line Engineering (SiSoPLE) [3] represents and extension of SoPLE, Safety-oriented Process Line Engineering [3]. Similar to SoPLE, SiSoPLE consists of a two-phase method for engineering families of safety life-cycles/processes. The first phase is aimed at engineering the domain from a process perspective i.e., identifying and systematizing process-related commonalities and variabilities, focusing on SiS (Security-informed Safety)-related commonalities and variabilities, in order to concurrently engineer a set of processes. The second phase is aimed at deriving single processes via selection and composition of commonalities and variabilities. From a tooling perspective, SiSoPLE as well as SoPLE can be supported by the integration between Eclipse Process Framework (EPF) Composer [8], recently re-brought to life [11], and Base Variability Resolution (BVR) Tool [15]. This integration was qualitatively evaluated as promising in [6]. To make the abstract self-contained, we recall that EPF Composer permits users to engineer processes in compliance with a SPEM (Software & Systems Process Engineering Metamodel) 2.0-like language [7], while BVR Tool permits users to orthogonally manage variability in compliance with the BVR [13] language. The integration between EPF Composer and BVR Tool for enabling the variability management at process level is hosted by OpenCert [16].

In this presentation, we point out our process for co-engineering security and safety life-cycles aligned with the SAE J3061 guidelines. This process exploits cross-concern commonalities and variabilities, which are systematized and managed via BVR Tool, integrated with EPF Composer.

Figure 1 shows a high-level view of the overall workflow of the proposed co-engineering process.

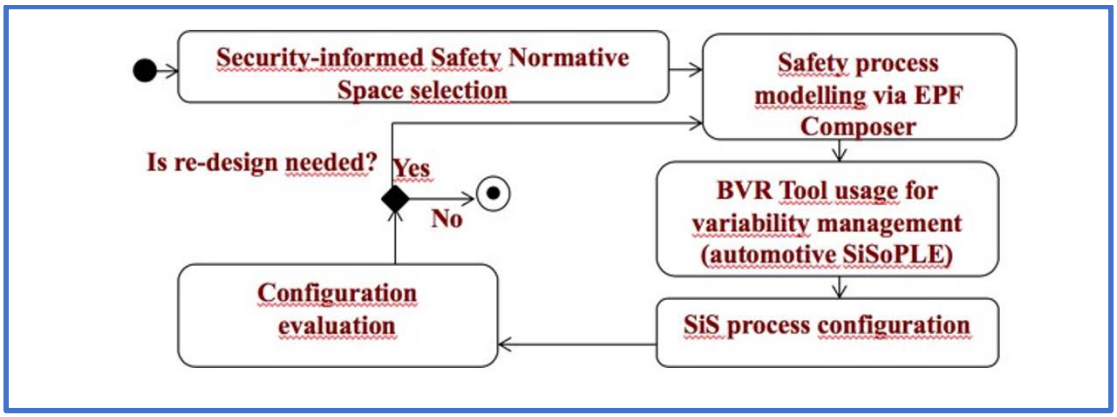


Figure 1 Co-engineering of Security and Safety Life-cycles via Exploitation of Cross-concern Commonalities and Variabilities

Further, the presentation shows the usage of the proposed co-engineering process in the context of the automotive regulations comprising ISO 26262 and SAE J3061, focusing on the risk analysis phase, as initially done in [17], where the automotive Security-informed Safety terminological framework for retrieving the implicit commonalities was proposed. Both ISO 26262 and SAE J3061 recommend the usage of specific methods for conducting the risk analysis phase. More specifically, ISO 26262 recommends HARA (Hazards Analyses and Risk Assessment) for safety; while SAE J3061 recommends TARA (Threat Analysis and Risk Assessment) for security-related analysis.

Process variability is managed based on parameters for safety, i.e. ASIL, and security, i.e. Security risk level (SecRL), which influence selection/inclusion of Safety and Security Activities. For example, an ASIL B process only requires deductive analysis e.g. FMEA (Failure Mode and Effect Analysis) for the verification of the safety concept, where ASIL D requires deductive and inductive analysis e.g. FTA (Fault Tree Analysis). With the help of the BVR tool the process can be tailored based on specified parameters. The tailoring activity deals with safety and security aspects and includes the specification of their dependencies.

The presentation will illustrate the co-engineering workflow (see Figure 1) by using the AMASS toolchain in a use case, which deals with verification of the system design of a car2x communication management unit. Safety and security concerns will be properly tailored via BVR Tool. The presentation aims at illustrating the importance of explicitly systematizing commonalities and variabilities for co-engineering the life-cycles needed for engineering multi-concern-critical systems.

This work is partially supported by the AMASS project [9], whose main objectives were presented in [10] and the SECREDAS project [18].

References

[1] ISO26262. Road vehicles – Functional safety. International Standard, November 2011.

[2] SAE J3061- Cybersecurity Guidebook for Cyber-Physical Automotive Systems. SAE - Society of Automotive Engineers.

[3] B. Gallina, L. Fabre. (2015, September). Benefits of security-informed safety-oriented process line engineering. In Digital Avionics Systems Conference (DASC), IEEE/AIAA 34th (pp. 8C1-1), IEEE, 2015.

[4] B. Gallina, I. Sljivo, O. Jaradat. Towards a Safety-oriented Process Line for Enabling Reuse in Safety Critical Systems Development and Certification. Post-proceedings of the 35th IEEE Software Engineering Workshop (SEW-35), 2012.

[5] B. Gallina, S. Kashiyarandi, H. Martin, R. Bramberger. (2014, July). Modeling a safety-and automotive-oriented process line to enable reuse and flexible process derivation. In Computer Software and Applications Conference Workshops (COMPSACW), IEEE 38th International (pp. 504-509), 2014.

[6] I. Ayala, B. Gallina. Towards Tool-based Security-informed Safety Oriented Process Line Engineering. 1st ACM International workshop on Interplay of Security, Safety and System/Software Architecture (ISSA), Copenhagen, Denmark, November 28th, 2016.

[7] OMG. Software & systems Process Engineering Meta-model (SPEM), v 2.0. Full Specification formal/08-04-01, Object Management Group, 2008.

[8] Eclipse Process Framework http://www.eclipse.org/epf/.

[9] AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems), http://www.amass- ecsel.eu.

[10] Ruiz A., Gallina B., de la Vara J. L., Mazzini S. and Espinoza H., “Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems”, 5th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR), Trondheim, September 2016.

[11] Javed, M. A. and Gallina, B. (2018a). Get epf composer back to the future: A trip from Galileo to Photon after 11 years. In EclipseCon, Toulouse, France, JUNE 13-14, 2018.

[12] M. A. Javed and B. Gallina. Safety-oriented Process Line Engineering via Seamless Integration between EPF Composer and BVR Tool. In 22nd International Systems and Software Product Line Conference (SPLC), Sept 10-14, Gothenburg, Sweden, in press. ACM Digital Library, 2018.

[13] VARIES D4.2- BVR - The language. http://bvr.modelbased.net/docs/VARIES_D4.2_v01_PP_FINAL.pdf

[14] ISO-SAE 21434 Road vehicles –Cybersecurity Engineering- General Overview. https://www.iso.org/standard/70918.html

[15] BVR Tool. https://github.com/SINTEF-9012/bvr

[16] OpenCert- hosting the AMASS platform. https://www.polarsys.org/opencert/about/

[17] Castellanos Ardila, J., Gallina, B.: Towards Efficiently Checking Compliance Against Automotive Security and Safety Standards. In: The 7th IEEE International Workshop on Software Certification., Toulouse, France, 2017.

[18] SECREDAS (Product Security for Cross Domain Reliable Dependable Automated Systems), http://secredas.eu/

IP: VERIFICATION & VALIDATION OF LAUNCHER FLIGHT SOFTWARE

David Lesens. Julien Grand
david.lesens@ariane.group, julien.grand@ariane.group

Summary

ArianeGroup is the prime contractor of the Ariane 6 launcher. As such, it is responsible for the flight software development. After a short introduction on the techniques used to develop this flight software, this paper will present the main principles of verification and validation (V&V).

The Ariane 6 flight software

The Ariane 6 flight software is developed in Ada 2012. It is a real time software of criticality B strongly constrained by the limited available hardware resources (CPU and memory). In order to facilitate the V&V, its real-time design relies on two main principles: synchronous and LINO. The synchronous approach means in this context that (1) all the tasks of the software are cyclic and cannot be stopped (they contains an infinite loop) and (2) all the communications between the software and the avionics or between two tasks are performed through time triggered rendezvous. In a LINO approach ("last in, next out"), the measurements used by the software at cycle n are acquire during cycle n-1 and the commands computed during cycle n are sent during cycle n+1.

Its code is composed of three main parts:

- A reusable Ada library implementing a generic middleware, a generic mission management, and a generic mathematical library. This library uses intensively generics and tagged types.
- An automatically generated part. The "sequential" of the launcher (the scheduling of all the events occurring during a flight depending on the mission and on the failures cases: engine ignition and shut-down, stage release, "fault, detection, isolation and recovery", etc…) as well as the static architecture of the software code (definition of "processing" and of data- flows between them) is modelled in the SysML modelling language. The dynamic architecture (organisation of tasks, communication between tasks, etc…) is modelled by a specific ArianeGroup Domain Specific Language. The corresponding Ada code is then automatically generated by an in-house tool.
- A manually coded part: Numerical algorithms (flight control, engine control), monitoring, telemetry, acyclic behaviours.

Verification and validation

The objective of cost's and delay's decrease gets the software team to deploy some advanced techniques aiming at maximising the automation of the V&V activities.

Starting from the SysML modelling, a set of automatic quality checks have been implemented: use of architectural patterns, verification of ranges (i.e. minimal and maximal allowed values), of units (meters, velocity…), of state reachability, etc…

A strategy has been defined for the validation of each type of code:

- The generic Ada code is first validated on a simplified instantiation independently from the Ariane 6 project. Its instantiation specific to the Ariane 6 project is used on all validations steps considering it is correct.
- To decrease the costs, it has been decided to not certify the automatic in-house code generator. In order to ensure anyway the quality of the generated code, several implementations of the model transformations needed to generate code have first been defined. Second, some tests are automatically generated by an independent tool and then automatically executed to validate the main features of the generated code. Finally, this in- house tool has been developed with the objective to generate code with the same readability than a manual code; this property allows performing manual code review when needed (which is seldom possible with a COTS code generator).
- Automatically generated tests allow validating the correct integration between the communication bus and a software process (hard / soft integration) and between two software processes (soft / soft integration).
- The "sequential" is validated with stubs of the "algorithmic" code. One of the main difficulties for the validation of such code is indeed to master the interactions between the "algorithmic" code and the "sequential" one. The stubs compute automatically the events to be sent to the "sequential" depending on the test objective defined by the validation responsible.

- The manually coded part is validated either:
- In classic open loop for “non-algorithmic / non sequential code”: inputs are defined by the validation responsible, the software is executed and the results are checked to be equal to the expected ones
- In close loop in front of a simulator of the environment for “algorithmic code”. For such code (navigation, guidance, flight control, engine regulation control, etc...), it is indeed not possible to manually deduce the expected outputs from the inputs provided to the software. The simulator of the environment models the avionics, the launcher behaviour (e.g. propulsion) and the external environment (wind, gravity, etc...).

Once the code is generated, it takes benefit from Ada features, such as the capability to define contracts (through the definition of ranges for each data and with the definition of pre and post- conditions) and to generate automatically checks. The ranges are automatically generated from the SysML model. All the tests are then played twice: (1) on an emulator of the target with the checks “on” in order to verify the correctness of these ranges and (2) on the real target with the checks “off” in order to be compatible with the real-time requirements. SPARK (a deductive prover provided by AdaCore) as well as CodePeer (an abstract interpretation tool also provided by AdaCore) have also been assessed. Unfortunately and despite their undeniable capacity of early bug detection, they have not been deployed because of the use of access types in the code; the use of such access types is made mandatory to optimise the code in regards with the limited available resources but is often incompatible with automated analysis tool.

Finally, the synchronous approach (communication at predefined time) implies the property that the software behaviour (functional but also real-time) is fully independent from the WCET of each individual function, providing that each task respects its own WCET. This property facilitates greatly the execution of some tests. The numerical algorithms and the launcher sequential can thereby be independently validated (see above): The numerical algorithms are validated on an emulator of the target (GNATEmulator provided by AdaCore) with some stubs of the sequential; in the same manner, the sequential is validated on host some stubs of the numerical algorithms.

The global validation tests with both the numerical algorithms and the sequential are thus limited to a minimum.

RP: GUIDING ASSURANCE OF ARCHITECTURAL DESIGN PATTERNS FOR CRITICAL APPLICATIONS

Irfan Sljivo, Garazi Juez Uriagereka, Stefano Puri and Barbara Gallina
irfan.sljivo@mdh.se, barbara.gallina@mdh.se

Summary

Development of critical systems nowadays is hardly achievable without reuse of previous knowledge. Design patterns have an important role in the design of such systems as they define and document (via appropriate templates) a common solution to recurring design problems. However, critical systems such as those that are safety or security related, often require specific assurances that the system is adequate to operate in a given environment. Just as with any other reused knowledge in such systems, the reuse via application of design patterns needs to be assured every time to ensure that they are adequate to address the targeted design problem.

In this paper, we present a methodology for assuring the application of design patterns in critical domains. The proposed methodology is part of the solutions developed within the AMASS project for architecture- driven design and assurance. In particular, we enrich the template for specifying design patterns to support their further assurance. We define the corresponding assurance argument pattern to detail the aspects that should be tackled during the assurance of an application of a design pat- tern. We use the information specified in the design pattern template to guide the automated instantiation of the argumentation for each de- sign pattern application in the system. We evaluate our tool-supported AMASS methodology in an automotive case study.

This work is supported by the EU and VINNOVA via the ECSEL Joint Undertaking project AMASS (No 692474).

THE SPEAKER'S CORNER

IP: EXPERIENCE FROM 40 YEARS OF TEACHING ADA

Jean-Pierre Rosen
rosen@adalog.fr

Summary

Back in 1979, shortly after Green was selected by the DoD, Ichbiah gave the first presentation of this brand new language – preliminary Ada. I attended this presenta- tion, and since at that time I was a teacher in a French engineering school, I immedi- ately returned that presentation to the students. I never ceased teaching Ada ever since then. This paper presents my experience after 40 years of teaching Ada.

The language: what's easy

The syntax is easy and readable, statements are the same as of other languages (except for low level syntax). Although most attendees have experience only with C-syntax languages, there are few difficulties with the basic statements. However, the peculiarities of Ada (completeness of case statement, safety of for loop) are an opportunity to start showing the intrinsically safe approach of the language.
Exceptions are no more a problem, since they are present in Java and C++. The principles of tasking are also easily received, since most of the attendees did at least some pthreads programming.
The low-level features, and especially representation clauses are also easily under- stood.

The language: what's difficult

Those who have learned only Java have problems understanding that a variable can exist just by declaring it, without having to use "new"! In general, ideas or mechanisms imported from other languages may lead to misunderstanding of Ada mechanisms.
It is important to insist on packaging, or more precisely on using packages for in- formation hiding and separation of specification from implementation. Students tend to jump to implementation even before the specification has been checked.
For those who come from languages where the class is the only structuring con- cept, it is sometimes difficult to understand the difference between a package and a type.
Generics are inherently difficult to grasp, but a good hands-on exercise is generally sufficient to understand the concept.
Many have difficulties in understanding the concept of rendezvous; they have a mental picture that is more of a kind of call-back from a task into another task, rather than the server deciding when to serve a client. Here again, a proper exercise clarifies things.
The difficulties encountered vary greatly with the previous experience of the stu- dents. Since the way software (including languages) is taught in schools evolves over time, an Ada course has to adjust its content to follow this evolution.

Beyond the language

But the real issue is to promote a different way of thinking. Explaining that the type Integer is just a regular predefined type and that there is no reason to use it; that the art of Ada programming is in defining types reflecting abstractions of the real world, in separating specification from implementation, in designing for reusability, safety, and reliability. Ada maturity comes when switching from a mentality of "damn compiler, why don't you accept what I wrote" to "thank you, gentle compiler, for showing me my mistakes".
Ada looks a lot like a conventional language; the benefit of a live course is in stressing what makes it different from other languages. Good examples and proper exercises are paramount to passing the message; a good Ada course does not teach how to program with Ada, but how to program in Ada.

SESSION 4: USES OF ADA IN CHALLENGING ENVIRONMENTS

RP: ENABLING ADA AND OPENMP RUNTIMES INTEROPERABILITY THROUGH TEMPLATE-BASED EXECUTION

S. Royuela, E. Quiñones, L.M. Pinho
sara.royuela@bsc.es, eduardo.quinones@bsc.es, lmp@isep.ipp.pt

Summary

Safety-critical systems have evolved to such a degree that the use of parallel paradigms is crucial to deliver the levels of performance necessary to implement the most advanced functionalities (e.g., autonomous driving). This trend has arrived to Ada, a language designed to keep safeness that is widely used in safety-critical domains such as avionics. In this regard, two complementary research lines are tackling the extension of Ada to support parallelism: a) the simple yet powerful tasklet model that, based on a fully strict fork-join model, is able to exploit structured parallelism on shared memory architectures, and b) the incorporation of OpenMP into Ada, to efficiently exploit structured and unstructured parallelism. This work focuses in the latter approach.

OpenMP offers a tasking model very suitable to cope with unstructured and highly dynamic parallelism. It defines tasks as units of scheduling composed of the task's executable code and its data environment, as well as different synchronization mechanisms (e.g., point-to-point synchronizations via data dependencies, and full synchronizations via memory fences). This, coupled with the accelerator model, allows targeting from simple SMP machines, to complex and heterogeneous architectures, all using the same programming model.

The objective of our work is to safely integrate OpenMP into Ada to exploit all the benefits of OpenMP, i.e., portability, programmability and performance. We divide our work in three main pillars: (1) the programming model, i.e., how OpenMP is integrated in Ada at a language level, (2) the compiler, i.e., the static analysis and transformations needed to ensure robustness, and (3) the runtime, i.e., the interoperability between the Ada and the OpenMP runtimes. Regarding the programming model, we propose a new syntax for OpenMP and Ada (OpenMP is built on top of C, C++ and Fortran) that maintains the clarity and unambiguity that characterizes the Ada language. Regarding the compiler, we propose a series of compiler analyses that seek data races in Ada and Ada+OpenMP programs, and provide the user with feedback to solve the errors. Finally, regarding the runtime, we provide prove that OpenMP fully supports the Ada tasklet model (and hence can be used to implement it), as well as analyze the information that must be interchanged between the two runtimes (Ada and OpenMP) in order to fulfill safety requirements (such as a priority driven scheduling). This journal includes a new compiler transformation that enables the use of OpenMP to parallelize Ada tasks while providing the scheduler with the tools to honor the priorities defined in the Ada tasks.

RP: SHARED-MEMORY MULTICORE SYNCHRONIZATION: PROGRAMMABILITY, SCALABILITY AND PERFORMANCE

Bernd Burgstaller, Johann Blieberger
bernd.burgstaller@gmail.com, johann.blieberger@tuwien.ac.at

Summary

The mutual-exclusion property of locks stands in the way to scalability of parallel programs on many-core architectures. Locks do not allow progress guarantees, because a task may fail inside a critical section and thereby prevent other tasks from accessing shared data. Because of the disadvantages of mutual exclusion locks, it is desirable to give up on method-level locking and allow method calls to overlap in time. Synchronization is then performed on a finer granularity within a method's code, via atomic read-modify-write operations. It thus becomes possible to provide progress guarantees, which are unattainable with locks. In particular, a method is non-blocking, if a task's pending invocation is never required to wait for another task's pending invocation to complete.

Non-blocking synchronization is nevertheless conceptually difficult when exposed to the programmer. We investigate three possible abstraction-levels to provide programmers with non- blocking synchronization in the context of the Ada programming language: (1) Our lock elision of protected objects with the help of hardware transactional memory shields the programmer from the non-blocking synchronization problem. (2) Concurrent objects are our novel programming primitive to encapsulate the complexity of non-blocking synchronization in a language-level construct. (3) By exposing atomic read-modify-write operations on language-level, programmers gain fine-grained control over the synchronization-problem, including the memory consistency model.

We investigate the trade-offs between programmability, scalability and performance of non- blocking synchronization on these three abstraction-levels. We contrast the scalability of non- blocking synchronization with state-of-the-art lock-based queues and mutual-exclusion locks. This paper thereby makes the following contributions:

1. We introduce lock-elision for monitors in the context of Ada protected objects. Our lock- elision technique is based on the Intel transactional synchronization extensions (TSX).
2. We introduce concurrent objects as a high-level programming primitive for non-blocking monitor constructs in Ada.
3. We relax several recent state-of-the-art lock and queue synchronization primitives from sequential consistency to acquire-release consistency, to explore the performance advantage of low-level atomic operations and relaxed memory consistency in Ada and C++.
4. We provide extensive experiments on scalability and performance of the proposed techniques on the x86 64 and ARM v8 hardware platforms.
5. For reproducibility, we have open-sourced all benchmark code in the form of a shared- memory synchronization benchmark suite.

IP: RCLADA, OR BRINGING ADA TO THE ROBOTIC OPERATING SYSTEM

Alejandro R. Mosteo, Centro Universitario de la Defensa de Zaragoza, Spain
amosteo@unizar.es

Summary

The Robot Operating System (ROS) is a de-facto standard in many fields of robotics research, with increasing presence in the industry. The next iteration of this framework, ROS2, aims to improve critical shortcomings of its predecessor like deterministic memory allocation and real-time characteristics. The main topic of this talk, RCLAda, is a binding to the ROS2 framework that enables the programming of ROS2 nodes in pure Ada with seamless integration into the ROS2 workflow.

The ROS project [1] was born from the experiences with the Player/Stage [2] robotic programming and simulation tools, lead in development by Willow Garage [3]. Today, many research institutions contribute and maintain components of ROS. The scope of ROS goes from build tools and communication libraries to reference-frame

composition facilities among robotic parts. These core elements enable the bringing together of heterogeneous code bases, with which the community has built a myriad of almost plug-and-play components that provide hardware drivers and software algorithms. Today, most research-oriented robotic hardware comes with a ROS node that enables its integration out of the box.

ROS was not designed with hard real-time considerations in mind, which has hindered its penetration in industrial domains, despite efforts like the ROSin [4] and ROS-industrial [5] projects. ROS2 [6] design squarely addresses these concerns, documenting dynamic memory allocation and threading characteristics of its API calls, and adopting the Data Distribution Service (DDS) standard for data exchange [7].

In essence, the RCLAda project [8] is an Ada binding (with support integration facilities) to the ROS2 Client Library C API (RCL), a common denominator designed to enable the integration of third-party languages on an equal footing. The RCLAda project has reached a level of completeness where it can be used to write ROS2 packages using only Ada code, with some bits of CMake for the building integration. In this talk, I will try to interest the audience in the vast landscape of ROS/ROS2, from the perspective of development with Ada, and its integration with other ROS2 tools.

Technical Overview

1. There are three main aspects involved in development for ROS2:
2. Using a client library in a supported (C++ or Python) or third-party language to develop nodes (stand-alone ROS2 processes that encapsulate functionality, usually while also communicating with other nodes to obtain/publish data). Client libraries expose the underlying ROS2 Client Library API (in C) in a language-specific way that is familiar to the language practitioners.
3. Exchanging information with other nodes, possibly programmed in other languages, via either topics (a publish-subscribe messaging paradigm) or services (remote procedure calls). Both rely on the exchange of record-like data structures hierarchically built from a basic set of standard data types.
4. Building the nodes inside the build system of ROS2. This is performed by a ROS2 tool called colcon (from collective construction) that manages dependencies between packages, determining a safe compilation order. Each package is built in isolation from the rest, enabling the characteristic heterogenous ecosystem of ROS.

RCLAda includes facilities to ease the creation of Ada nodes in all these aspects, with minimum impact on the working environment of organizations already using ROS2. Conversely, Ada expertise can be brought to ROS2 with ease:

1. The RCLAda binding component enables the writing of Ada-only nodes, without having to resort to custom binding code to the C or C++ libraries. Both Ubuntu and Windows ROS2 distributions rely on the gcc toolchain, which includes the GNAT Ada compiler, hence not bringing in any new complex dependencies. The binding is two-layered, with an automatically generated lower layer that ensures synchronicity with the C API, and a high-level layer that provides a thick Ada API.
2. Messages are accessible through Ada functions that ensure type safety and valid array indexing. These functions return references to the underlying DDS types without data conversions, enabling efficient read and writing of message fields.
3. The build system, colcon, expects either CMake or Python setup scripts. At present, RCLAda offers a set of CMake functions that allow the integration of GNAT project files into the build process, and the exporting of build products to final users or other nodes. Since the build method of each package is isolated, Ada nodes do not impact the rest of the build process.

In closing, RCLAda takes the brunt of the integration process into ROS2, allowing the use of Ada in a straightforward fashion. The thick binding favors programming in a natural Ada way, while the build functions minimize the impact of integrating GNAT projects into the ROS2 build process.

References

1. ROS project: http://www.ros.org/
2. Player/Stage binding: https://github.com/mosteo/player-ada
3. Willow Garage: http://www.willowgarage.com/
4. ROSin H2020 project: http://rosin-project.eu/
5. ROS-industrial project: https://rosindustrial.org/
6. ROS2 project: https://index.ros.org/doc/ros2/
7. Object Management Group DDS: https://www.omg.org/omg-dds-portal/
8. RCLAda project: https://github.com/ada-ros

SESSION 5: VERIFICATION CHALLENGES

IP: FAST, FLEXIBLE DO-178C TOOL QUALIFICATION USING A MODULAR APPROACH

Daniel Wright, Ian Broster, Zoë Stephenson, David Allsopp and Sevane Fourmigue
dwright@rapitasystems.com

Summary

This presentation describes a new modular approach to generating DO-178C qualification materials for critical code that enables their efficient reuse and reproduction to provide better support for DO-178C certification. We also present a case study of using the approach to produce qualification kits for Rapita Systems' RapiCover verification software.
The presentation will address the following points:

- The role of tool qualification kits in DO-178C certification.
- How a modular approach to document generation, requirements and traceability management supports an agile development process.
- How to qualify critical software verification tools such as structural coverage, unit testing and execution time measurement tools.

According to RTCA DO-330, when software verification tools are used in a project certified at DO-178C, those tools must often be qualified to ensure that they do not introduce or fail to detect errors in the software being verified.
Qualification is often supported by commercial-off-the-shelf qualification kits supplied by tool vendors. Qualification kits are tied to a specific vendor tool version or integration. The high effort required to generate new qualification kits using a traditional approach limits flexibility of the tool vendor to respond to customer requests and apply incremental improvements.
We present a new, modular approach to generation of qualification kit material, which reduces the effort needed to reproduce qualification kits by doing the following:

- Breaking qualification artefacts down into a fine-grained hierarchy and storing related requirements, definitions and notes in individually tracked files. This significantly minimises change impact and lets engineers complete tasks quickly, helping to keep effort proportional to the size of the change being made.
- Tracking requirements, test plans and test scripts using the same review system, helping to reduce programme risk and simplify external audits.
- Integrating product line facilities into the traceability management system, letting us produce qualification kits including only the material needed to fit a customer's claim for credit, thereby reducing unnecessary review burden.

We have implemented the approach and built new infrastructure in our development pipeline, including: (1) Custom tools to build documentation automatically from requirements, minimizing review effort and simplifying document generation. (2) Tool automation for regression testing, detection of inconsistencies, software bugs and traceability errors in the tool being qualified. (3) Tool automation for progress tracking and guidance for management of team prioritization and effort allocation.
We present a case study of using this modular approach to generate DO-178C qualification kits for the TQL-5 structural coverage analysis tool, RapiCover, which is written in Ada, widely used for structural coverage analysis of critical aerospace and automotive software and part of the RVS tool-suite. We will cover:

- What we learned while transitioning to a more agile technology.
- Benefits to Rapita including significantly increased engineering efficiency.
- Benefits to our customers including improved quality and increased ability to make incremental qualification kit changes per customer needs, minimising project schedule risks.

This technology can also be used to produce ISO 26262 qualification kits and was developed as part of the EU-funded ECSEL-JU AMASS project, which aims to reduce certification costs for cyber-physical systems.

RP: ECTM: A NEW COMMUNICATION MODEL FOR NETWORK-ON-CHIP SCHEDULABILITY ANALYSIS

Mourad Dridi, Frank Singhoff, Stephane Rubini and Jean-Pierre Diguet
mourad.dridi@univ-brest.fr, stephane.rubini@univ-brest.fr, frank.singhoff@univ-brest.fr, jean-philippe.diguet@univ-ubs.fr

Summary

Network-On-chips are widely used in industrial applications since they provide communication parallelism and reduce energy consumption. The use of NoC has been recently extended to real-time systems, whose execution has to meet temporal constraints. Communication delays introduced by the network make the scheduling analysis challenging. In this paper, we propose a new NoC communication model called ECTM. The main goal of this model is to assess the schedulability of dependent periodic tasks exchanging messages on NoC. ECTM transforms NoC messages to tasks in order to take into account communication delays during scheduling analysis. It supports Store-And-Forward and Wormhole NoC. We have implemented ECTM in a real-time scheduling analysis tool called Cheddar and performed experiments to assess its efficiency. ECTM is more efficient than existing solutions with an improvement of 30% for Store-And-Forward NoCs and of 100% for Wormhole NoCs, while the proposed model requires a larger computation time about 17% for Store-And-Forward NoCs and 54% for Wormhole NoCs.

IP: A "NEW" C STATIC ANALYZER, THE COMPILER

Maurizio Martignano
maurizio.martignano@spazioit.com

Summary

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

(A historical) Introduction

The size of software systems in spacecrafts has increased dramatically over time: e.g. in 1981 NASA Space Shuttle Primary Flight Software was about 400 K lines of code (LOC) while in 2012 Curiosity had 2.5 MLOC; in 2008 ESA Automatic Transfer Vehicle (ATV) had about 1 MLOC. A similar trend has occurred in avionics: e.g.: in 1974 an F16A plane had 135 KLOC while in 2012 an F35 had about 10 MLOC. An even more dramatic increase has occurred in the automotive sector where a nowadays car software ranges from 10 to 150 MLOC. This increase in the software size, and consequently its importance and criticality, calls for new and more efficient methodologies and tools able to properly support Independent Software Verfication and Validation (ISVV) activities, so that they are feasible and economic even when applied to very large codebases.
In the C language, from the very beginning, around the end of the seventies, a decision was taken to somehow separate the compiler from the static analyzer ("Lint") where the compiler gives the programmer as much freedom (and responsibility) as possible and the "Lint" analyzer looks for programming and stylistic errors, bugs, questionable constructs and so on... This initial separation has caused over time the independent evolution of compilers and static

analyzers as well as the communities of people working on them. While the compiler communities concentrated on the actual software trends, i.e. its continuous increase in size (by definition, a build system must be able to build the system), the static analyzer communities have concentrated on developing deeper and more detailed analysis techniques without really considering their actual applicability to large codebases. Nowadays, especially thanks to Clang (C language family front-end) and LLVM (compiler infrastructure), two open source software systems mostly developed by Apple, Microsoft, Google, ARM, Sony, Intel and Advanced Micro Device, the compiler and the static analyzer are getting back together, in the attempt of combining the compiler efficiency with analytical power of the static analyzer. In Clang and LLVM [1] systems all main functionalities are available as libraries: e.g. the lexer is a library (clangLex") used by the both the compiler (Clang) and its static analyzers (Clang-Check and Clang-Tidy).
Section two of this paper will present various types of static analyzers, differentiated based on the technique they use, i.e. pattern matching, data flow analysis, abstract interpretation and model checking. The paper will show the actual applicability of these techniques to large codebases.
Section three will present the Clang/LLVM based static analyzers, will show their main characteristics and how they perform on large codebases. The section will present also "libclang", a C programming interface (API) to the compilation system, accessible via C and Python. The section will end presenting Clang "JSON Compilation Database Format Specification" [2], which is a data "format for specifying how to replay single compilations independently of the build system". This format is becoming a "de-facto" standard among tools vendors and simplifies the task of properly configuring static analyzers.
Section four will present the current status of the art of the integration of the Clang/LLVM static analyzers with SonarQube [3], an open source code quality platform. Section five will present the author's current on code analysis, the "SAFe Toolset", a toolset based on open source tools, able to facilitate and simplify the execution of static analysis on large codebases.

Static Analyzers

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

- pattern matching – the analyzer looks for particular constructs in the code at both lexical and syntactical levels – e.g. PC-Lint [4], Cppcheck [5];
- abstract interpretation – the analyzer verifies the code by "executing" it on some kind of abstract machine that approximates the original system – e.g. Polyspace [6] and Frama-C Eva [7] and Clang-Check [8];
- data-flow analysis – the analyzer keeps track of a set of values and how they might change during (virtual) execution – e.g. Cppcheck and Clang-Check;
- Hoare logic – the analyzer uses a set of logical rules for reasoning rigorously about the code – e.g. Frama-C WP on C code extended with ACSL [9] (Ansi C Specification Language);
- (bounded) model checking – the analyzer works on a finite state machine representation of the code under analysis – e.g. CBMC [10].

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

Clang/LLVM based Static Analyzers. "libclang" and JSON Compilation Database

In the Clang/LLVM website it is possible to read "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. (...) The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many CPUs. (...) Clang is an LLVM native C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles." In few words Clang/LLVM is a compilation toolchain where absolutely everything is built in a modular fashion as collection of libraries. In this toolchain the two static analyzers are Clang-Check (a.k.a. Clang-SA) and Clang-Tidy. Clang-Check relies on a set of Clang modules to perform things like lexical analysis, parsing, semantic analysis, AST manipulation and the like. Clang-Tidy relies on the very same Clang modules plus some additional modules of Clang-Check itself (this is why Clang-Tidy can be considered a sort of superset of Clang-Check). So, for instance, the lexer and parser of the static analyzers are the very same lexer and parser of the compiler: when the compiler evolves to keep up with the changes and improvements in the language, the very same evolution occurs also in the static analyzers (i.e. no disconnect between the compiler and the static analyzer worlds). Clang-Check implements path-sensitive, inter-procedural analysis based on symbolic execution and data flow analysis techniques. Even if the analyzer attempts to perform a sort of abstract interpretation it is still very efficient because it does not perform this interpretation on the entire codebase but only on "suspected" areas, identified via fast techniques like pattern matching; the performed checks are listed in document [11]. Clang-Tidy adds up on top of Clang-Check and the list of its checks is available in document [12].
What really makes Clang-Check stand apart from other static analyzers is its ability to show graphically from within the code the causes of a potential issue/bug, as shown here below.

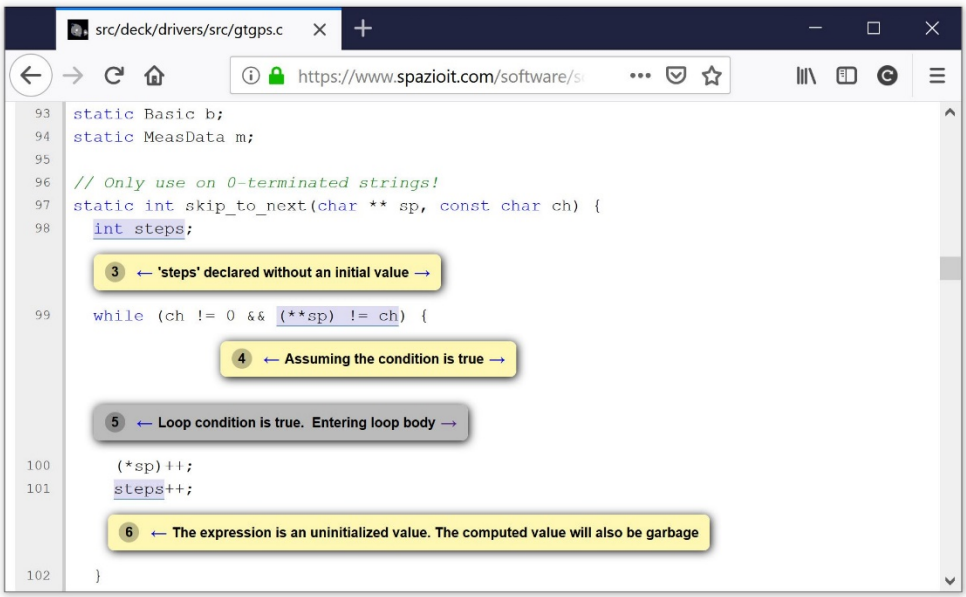


Figure 1 - Clang-Check Graphic Output

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

separated from the actual Ada Compilers (e.g. the GNAT compiler); and this is why, for example, ASIS does not yet support Ada 2012.
When performing code analysis static analyzers are used to examine the source code; the more static analyzers the better, the more issues are found. The problem is that, at least up to now, each static analyzer uses its own format for the configuration files controlling their behavior (e.g. which source files to analyze, how to analyze them, and so on…). In the attempt of trying to solve this problem, the Clang/LLVM community has developed a standard format specification, called "JSON Compilation Database Format Specification", defining "a format for specifying how to replay single compilations independently of the build system". This "de facto" standard is rather young, immature: several build tools (e.g. cmake [15], compiledb [16], bear [17], ninja [18],) support it but in (slightly) different ways; the same applies to static analyzers - e.g. among Clang-Check, Clang-Tidy, Cppcheck and SonarQube C/C++ Community Plugin [19] only the first two (obviously) properly support it.

Integration with SonarQube

Static analyzers produce their results in the form of tabular data (e.g. a *.csv or a *.xml file) where per each found issue, the provided information is the issue type, where it was found (file name, line number) and some additional explanation. It is very difficult to assess the found issues in this format. An already mentioned exception is Clang-Check, which can produce graphic html pages with the flow of events causing a given issue to occur. SonarQube, an open source quality platform, is a web application able to gather the analyses results produced by the various analyzers and display them from within the source code itself. SonarQube is language agnostic and the interface between SonarQube and a particular language is provided by "Plugins". Inside the "Plugin" per each supported static analyzer there is a "Sensor" that actually converts the analyzer results into a format processable by SonarQube. For the C and C++ languages there are two Plugins, one commercial developed by SonarSource [20] and one open source [19]. The open source plugin supports Clang-Check and Clang-Tidy (as well as PC-Lint, Cppcheck and some other analyzers) and the author has always been modifying, improving it according to the various project needs; in particular the author has been working on the Clang-Check and PC-Lint "Sensors". Notably, the Clang-Check in the community plugin has gone through a dramatic improvement towards the end of April 2019 (version 1.3.0-SNAPSHOT); in particular now it is able to show multilocation issues as well as if not better than the static analyzer itself, as shown below.

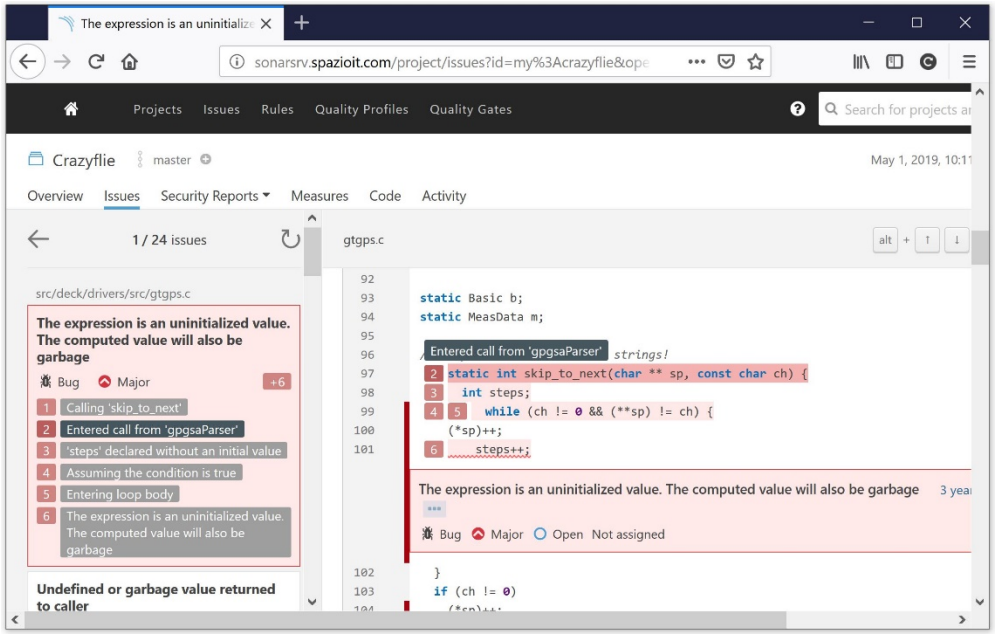


Figure 2 - Clang-Check Sensor Output displayed in SonarQube

The Safe Toolset

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

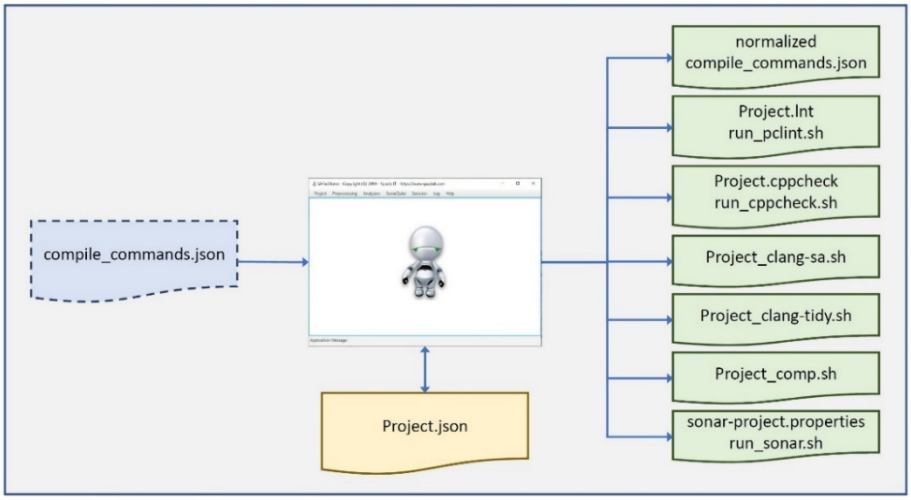


Figure 3 – the SAFacilitator (Static Analysis Facilitator)

The SAFacilitator, for a given codebase, allows the creation and editing of a single project file containing all the information required to properly drive and control the execution of the selected static analyzers on that codebase. This information may initially derive from an externally provided compilation database file. Once the information contained in the project file is correct, it is used to automatically generate all the configuration files and execution scripts required by the static analyzers (and SonarQube). The development of the "SAFe Toolset" has been funded by the European Space Agency Contract number RFP/3-15558/18/NL/FE/as.

Conclusions

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

References

[1] http://clang.llvm.org/
[2] https://clang.llvm.org/docs/JSONCompilationDatabase.html
[3] https://www.sonarqube.org/
[4] https://www.gimpel.com/
[5] http://cppcheck.net/
[6] https://www.mathworks.com/products/polyspace.html
[7] https://frama-c.com/value.html
[8] https://clang-analyzer.llvm.org/
[9] http://frama-c.com/wp.html
[10] http://www.cprover.org/cbmc/
[11] https://clang-analyzer.llvm.org/available_checks.html
[12] https://clang.llvm.org/extra/clang-tidy/checks/list.html
[13] http://gnat-asis.sourceforge.net/
[14] https://github.com/AdaCore/libadalang
[15] https://cmake.org/
[16] https://github.com/nickdiego/compiledb
[17] https://github.com/rizsotto/Bear
[18] https://ninja-build.org/
[19] https://github.com/SonarOpenCommunity/sonar-cxx
[20] https://www.sonarsource.com/
[21] https://www.spazioit.com/pages_en/sol_inf_en/code_quality_en/safe-toolset-en/

IP: VERIFICATION OF ADA PROGRAMS WITH ADAHORN

Tewodros A. Beyene, Christian Herrera, Vivek Nigam
beyene@fortiss.org, herrera@fortiss.org, nigam@fortiss.org

Summary

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

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

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

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

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

AdaHorn Architecture

The model checking procedure implemented in AdaHorn consists of two steps: constraints generation and constraints solving.

- *Constraints generation*: In this step, AdaHorn takes a set of Ada programs as input, and generates a set of CHCs as output. The constraints generation step first makes use of the gnat2xml utility in the GNAT Compiler Tool [11] to obtain an XML serialization of an Abstract Syntax Tree of each input Ada program. Then, the Ada AST written as XML is translated construct by construct into one or more CHCs. The interested reader finds in [6] a reference for translating program constructs into CHCs. Moreover, this step needs to also make special considerations for constructs that are not directly supported by the Horn constraints solvers used in this work. For example, as arrays may result in Horn clauses with nonlinear structure or higher orders, further processing needs to be done to translate resulting constraints into array-free Horn constraints [12].
- *Constraints solving*: In this step, the generated CHCs are passed to a Horn clause solver to get a result, which will be the final result of the model checking procedure. We have used the Horn clause solvers Eldarica and Z3 in this work, however, any CHC solver can in principle be used. Possible results of the solving step are: SAT (the CHCs are satisfiable), UNSAT (the CHCs are unsatisfiable), and UNKNOWN (the solver is not able to output any conclusive result).

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

Experimental Evaluation

We evaluate AdaHorn on a set of Ada benchmarks that consists of four categories of programs: arrays, floats, loops, and simple (https://bitbucket.org/umaya/adabenchmarksfromsvcomp17). The first three categories are inspired by C programs from the software verification competition SV-COMP 2017 [13]. For the C programs that can exclusively be translated to the subset of Ada handled in this work, we have manually created equivalent Ada programs. Programs in the simple category are written by the authors of this paper.

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

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

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

Table 1: Comparison of results between GNATProve and AdaHorn

Coming to the results, GNATProve takes less than 3 seconds for verifying each benchmark, whereas AdaHorn timed out for one example and took less than 60 seconds to verify the remaining examples. GNATProve concludes false positives in 48 occasions (out of 68 benchmarks) and 2 false negatives!

After reporting those false negatives to developers of GNATProve, we were told that GNATProve generally does not output correct results if intermediate checks have failed, which is the case for the programs related to those false negatives. This is mainly due to GNATprove's assumption that any prior check must be correct in order to prove the next ones. While AdaHorn does not result in any false negatives, it results in 4 false positives. The reason behind the false positives was the difference in precisions for floating- point numbers between the Ada compiler and the Horn constraint solvers used by AdaHorn (Eldarica and Z3). AdaHorn, however, is not able to verify 4 benchmarks due to failure of the Horn constraints solvers to conclude whether input CHCs are satisfiable or unsatisfiable.

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

References

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

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

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

[4] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. volume 9206 of LNCS, pages 343–361. Springer, 2015.

[5] T. Kahsai, P. Ru¨mmer, H. Sanchez, and M. Scha¨f. JayHorn: A Framework for Verifying Java Programs. volume 9779 of LNCS, pages 352–358. Springer, 2016.

[6] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn Clause Solvers for Program Verifica- tion. volume 9300 of LNCS, pages 24–51. Springer, 2015.

[7] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn Clauses as an Intermediate Representation for Program Analysis and Transformation. TPLP, 15(4-5):526–542, 2015.

[8] N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program Verification as Satisfiability Modulo Theories. In SMT 2012, volume 20 of EPiC Series in Computing, pages 3–11. EasyChair, 2012.

[9] H. Hojjat, F. Konecny´, F. Garnier, R. Iosif, V. Kuncak, and P. Ru¨mmer. A Verification Toolkit for Numerical Transition Systems - Tool Paper. volume 7436 of LNCS, pages 247–251. Springer, 2012.

[10] L. Mendonc¸a de Moura and N. Bjørner. Z3: An Efficient SMT Solver. volume 4963 of LNCS, pages 337–340. Springer, 2008.

[11] Project Hi-Lite / GNATprove, 2014.

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

[13] D. Beyer. Software Verification with Validation of Results - (Report on SV-COMP 2017). In TACAS 2017, volume 10206 of LNCS, pages 331–349, 2017.

SESSION 6: REAL-TIME SYSTEMS

RP: PERIOD ADAPTATION OF REAL-TIME CONTROL TASKS WITH FIXED PRIORITY SCHEDULING

Xiaotian Dai and Alan Burns
xd656@york.ac.uk, alan.burns@york.ac.uk

Summary

Long-lived, non-stop cyber-physical systems (CPS) are subject to evolutionary changes that can undermine the guarantees of schedulability that were verified at the time of deployment. At the same time, knowledge gleamed from extended periods of execution will reduce the uncertainties that were inevitably presented in the system models that is used to define the temporal behaviours of the control tasks. In this paper, we present an adaptation method that actively extends the period of control tasks at run-time. This can lead to lower power consumption or to the accommodation of increased computation resource demands from other components of the CPS. This method relies on online monitoring and model-based prediction to degrade control performance while having a minimal and acceptable impact on ongoing operations. Cloud-based computing is used to facilitate decision-making and offload the computation. We evaluate the effectiveness of the proposed method through experiments of control-scheduling co-simulation.

RP: ON ADA PROTECTED OBJECTS AND MULTIPROCESSOR SPIN-LOCKING PROTOCOLS

Jorge Garrido, Juan Zamorano, Alejandro Alonso, Juan Antonio de la Puente
jorge.garrido@upm.es, juanrafael.zamorano@upm.es, alejandro.alonso@upm.es, juan.de.la.puente@upm.es

Summary

Real-time multiprocessor systems are being used extensively in industrial applications. Ada provides ample support for such systems, including a complete tasking model providing time predictability, especially when restricted by the Ravenscar profile. A fundamental element of this tasking model is inter-task communication by means of protected objects. The definition of resource locking policies with bounded priority inversion is a fundamental aspect of protected objects, which has received considerable attention, with some interesting results that can be used in multiprocessor real-time systems. However, there is another important subject, the service policy for protected entries, that has received less attention in the research community and is also important in order to guarantee a predictable time behaviour. The impact of the service model on the response time analysis of multiprocessor real-time systems is evaluated in the paper for the self-service model and the proxy model, and their relation to the locking policies of relevant spin-locking protocols is discussed. Extensions to response time analysis for the proxy model with the studied locking policies are also contributed.

RP: A HIERARCHICAL ARCHITECTURE FOR TIME- AND EVENT-TRIGGERED REAL-TIME SYSTEMS

Jorge Real, Sergio Sáez., Alfons Crespo
jorge@disca.upv.es, ssaez@disca.upv.es

Summary

Real-time computer systems can be broadly described in terms of activities, or tasks, that must be executed along time, together with restrictions such as the time by which they must be completed. In their design and implementation, these systems tend to use either a time-triggered or an event-triggered approach to schedule their tasks. In time-triggered systems, a pre-elaborated static plan dictates the exact points in time when the tasks must be executed. In event-triggered systems, on the other hand, all tasks are started by event triggers, whose times of occurrence are not necessarily known in advance. Those events may be periodic, hence somehow related with time, but they can also be non-periodic, such as entering a particular system state, processing sensor or user inputs, reacting to alarms, etc.
Both time- and event-triggered designs exhibit advantages and disadvantages with respect to each other, and none can be deemed superior in general. Time-triggered systems excel in terms of run-time determinism, at the cost of higher design complexity and lack of flexibility. Event- triggered systems allow designers to conveniently separate logical and timing concerns and to naturally incorporate non-periodic tasks, at the cost of indeterminism, which affects run-time control over important timing aspects, such as jitter in real-time control applications.
This paper proposes an architecture for the combined execution of time- and event-triggered task sets. A Ravenscar implementation is also pro- posed, together with a library of helping patterns. This architecture makes it possible to choose which release mechanism to apply to which application tasks, hence allowing the designer to use the most appropriate depending on the role and nature of each task in the system. The proposed architecture is hierarchical, because it allows one to choose the priority levels at which time- and event-triggered tasks are executed, and these levels can be intermixed. This gives the designer an additional degree of freedom to make compromise decisions upon contradicting timing requirements, such as granting reduced jitter and at the same time providing prompt service to non-periodic events, for example.

Ada Europe 2019

ORGANIZERS

Conference & Program Chair

Tullio Vardanega
University of Padova, Italy

Industrial Chair

Maurizio Martignano
Spazio-IT, Italy

Publicity Chair

Dirk Craeynest
Ada-Belgium & KU Leuven, Belgium

Educational Tutorial & Workshop Chair

Dene Brown
SysAda Ltd, UK

Exhibition & Sponsorship Chair

Ahlan Marriott
White Elephant GmbH, Switzerland

Local Chair

Maciej Sobczak
GE Aviation – EDC Warsaw, Poland

Program Committee

Mario Aldea (Universidad de Cantabria, ES), Johann Blieberger (Vienna University of Technology, AT), Bernd Burgstaller (Yonsei University, KR), António Casimiro (LASIGE/U. Lisboa, PT), Juan A. de la Puente (Universidad Politécnica de Madrid, ES) , Barbara Gallina (Mälardalen University, Sweden), Michael González Harbour (Universidad de Cantabria, ES), J. Javier Gutiérrez (Universidad de Cantabria, ES), Jérôme Hugues (ISAE, FR), Hubert Keller (Karlsruhe Institute of Technology, DE), Franco Mazzanti (ISTI-CNR, IT), Laurent Pautet (Telecom ParisTech, FR), Luís Miguel Pinho (CISTER/ISEP, PT), Erhard Plödereder (Universität Stuttgart, DE), Jorge Real (Universitat Politècnica de València, ES), Sergio Sáez (Universitat Politècnica de València, ES), Frank Singhoff (Université de Bretagne Occidentale, FR), Tullio Vardanega (Università di Padova, IT).

Industrial Committee

Ian Broster (Rapita Systems), Dirk Craeynest (Ada-Belgium & KU Leuven), Ismael Lafoz Pastor (Airbus Defence and Space), Ahlan Marriott (White Elephant GmbH, CH), Maurizio Martignano (Spazio-IT, IT), Silvia Mazzini (Intecs, IT), Marco Panunzio (Thales Alenia Space, FR), Patricia Lopez Cueva (Thales Alenia Space, FR), Jean-Pierre Rosen (Adalog, FR), Emilio Salazar (GMV, ES).

CONFERENCE SPONSORS

AdaCore · GE · PTC Developer Tools · RAPITA SYSTEMS A DANLAW COMPANY · VECTOR

PROCEEDINGS

The proceedings of the peer-reviewed papers presented at the conference will appear in a dedicated, Open Access, Special Issue of Elsevier's Journal of Systems Architecture, due by December 2019-January 2020.

The proceedings of the industrial papers and of the DeCPS co-located workshop will appear in the Ada User Journal.

ELSEVIER