António Casimiro
Pedro M. Ferreira (Eds.)

LNCS 10873

Reliable Software Technologies – Ada-Europe 2018

23rd Ada-Europe International Conference
on Reliable Software Technologies
Lisbon, Portugal, June 18–22, 2018, Proceedings

Lecture Notes in Computer Science 10873

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, Lancaster, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Zurich, Switzerland

John C. Mitchell
Stanford University, Stanford, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan
Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen
TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at http://www.springer.com/series/7408

António Casimiro · Pedro M. Ferreira (Eds.)

Reliable Software Technologies – Ada-Europe 2018

23rd Ada-Europe International Conference on Reliable Software Technologies
Lisbon, Portugal, June 18–22, 2018
Proceedings

Editors
António Casimiro
University of Lisbon
Lisbon
Portugal

Pedro M. Ferreira
University of Lisbon
Lisbon
Portugal

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92431-1 ISBN 978-3-319-92432-8 (eBook)
https://doi.org/10.1007/978-3-319-92432-8

Library of Congress Control Number: 2018944394

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG, part of Springer Nature 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The 23rd edition of the International Conference on Reliable Software Technologies (Ada-Europe 2018) took place in Lisbon, returning to Portugal 12 years after Porto in 2006. The previous editions of the conference were held in Spain (Santander, 1999, Palma de Mallorca, 2004, Valencia, 2010, Madrid, 2015), France (Toulouse, 2003, Brest, 2009, Paris, 2014), the UK (London, 1997, York, 2005, Edinburgh, 2011), Austria (Vienna, 2017 and 2002), Switzerland (Montreux, 1996, Geneva, 2007), Sweden (Uppsala, 1998, Stockholm, 2012), Germany (Potsdam, 2000, Berlin, 2013), Italy (Venice, 2008, Pisa, 2016), and Belgium (Leuven, 2001).

The Faculty of Sciences of the University of Lisbon was the lead organizer for this edition, with aid from an international core team that included members of Ada-Europe, the organization that oversees and sponsors the conference series.

The conference took place in the week of June 18–22, 2018, with a rich program for both technical content and social opportunities. The scientific program featured 10 papers selected among 27 peer-reviewed submissions, grouped into five presentation sessions scheduled in the central days of the conference week, to address the following topics: safety and security, Ada 202X, handling implicit overhead, real-time scheduling, and new application domains. The proceedings contained in this volume reflect these contributions (see the Table of Contents for details).

The conference program also included 12 industrial contributions arranged in four industrial presentation sessions. The regular sessions were complemented by four more presentations selected among the regular submitted papers, as well as by vendor presentations. Vendor exhibitions completed the core program.

The first and the last day of the conference were dedicated to tutorials and workshops. A total of ten tutorials took place, eight of which were half-day tutorials and 2 full-day ones. On Monday the program included the workshop on Runtime Verification and Monitoring Technologies for Embedded Systems (RUME), and on Friday the fifth edition of the Workshop on Challenges and New Approaches for Dependable and Cyber-Physical Systems Engineering (DeCPS) took place. The proceedings from this part of the conference program will be published, in successive instalments, in the *Ada User Journal*, the quarterly magazine of Ada-Europe.

The scientific and industrial submissions originated from 19 countries from Europe, Asia, North and South America and Africa. Thanks to that wealth, the final program was an international digest of contributions from Austria, France, Germany, Italy, Norway, Poland, Portugal, South Korea, Spain, Sweden, Switzerland, the UK, and the USA.

Each of the three days of the technical program opened with a keynote talk focusing on topics of interest to the conference scope. The three keynote talks were:

– "Security and Dependability Challenges of IT/OT Integration" by Paulo Esteves-Veríssimo, from the University of Luxembourg, Luxembourg, who argued about the need for paradigms and techniques to endow systems with the capacity of

defeating incremental adversary power and sustaining perpetual and unattended operation, in a systematic and automatic way.
- "From Physicist to Rocket Scientist, and How to Make a CubeSat That Works" by Carl Brandon, from the Vermont Technical College, USA, who explained how to have a successful CubeSat, where many others have failed, in which the reliability of SPARK/Ada software plays a big part.
- "Vulnerabilities in Safety, Security, and Privacy" by Erhard Plödereder, from the University of Stuttgart, Germany, who discussed the differences and commonalities in threats that affect safety, security, or privacy in today's systems, also arguing that vulnerabilities made possible by programming language features form a common base for violating safety, security, or privacy.

The tutorial program covered the following topics:

- "Recent Developments in SPARK 2014," Peter Chapin, Vermont Technical College, USA
- "Access Types and Memory Management in Ada 2012," Jean-Pierre Rosen, Adalog, France
- "Design and Architecture Guidelines for Trustworthy Systems," William Bail, The MITRE Corporation, USA
- "Numerics for the Non-Numerical Analyst," Jean-Pierre Rosen, Adalog, France
- "Requirements Development for Safety- and Security-Critical Systems," William Bail, The MITRE Corporation, USA
- "Scheduling Analysis of AADL Architecture Models," Frank Singhoff, Lab-STICC/UBO, France and Pierre Dissaux, Ellidiss Technologies, France
- "Writing Contracts in Ada," Jacob Sparre Andersen, JSA Research & Innovation, Denmark
- "Introduction to Libadalang," Raphaël Amiard and Pierre-Marie de Rodat, AdaCore, France
- "Unit-Testing with Ahven," Jacob Sparre Andersen, JSA Research & Innovation, Denmark
- "Frama-C, a Framework for Analyzing C Code," Julien Signoles, France

The industrial program featured the following presentations:

- "Managing the Endianness of Software Building Blocks with GNAT Ada Pragmas: A Case Study," Patricia Lopez Cueva and Marco Panunzio
- "Using Ada in Non-Ada Systems," Ahlan Marriott
- "Easy Ada Tooling with Libadalang," Pierre-Marie de Rodat and Raphaël Amiard
- "Ariane 6 Flight Software Designed for a Simpler Validation," Philippe Gast and Cyrille Pierre
- "I3DS A Modular Sensor Suite for Space Robotics," Kristoffer Nyborg Gregertsen
- "Multi-Concern Dependability-Centered Assurance for Space Systems via ConcertoFLA," Barbara Gallina, Zulqarnain Haider, Anna Carlsson, Silvia Mazzini, and Stefano Puri
- "Applying Formal Timing Analysis to Satellite Software," Andreas Wortmann

- “Multicore Timing Analysis for Safety-Critical Software,” Ian Broster, Guillem Bernat, Francisco Cazorla, Christos Evripidou, and Suzana Milutinovic
- “KhronoSim: Simulation and Testing of Real-Time Critical Cyber-Physical Systems,” Gonçalo Gouveia, João Esteves, Cláudio Maia, and Luis Miguel Pinho
- “C Guidelines Compliance and Deviations (the MISRA and CERT Cases),” Maurizio Martignano
- “Agile in Safety Critical Projects,” Pawel Zakrzewski
- “AGILE-R: Agile Software Development for Railways,” Silvia Mazzini, John Favaro, Guido Ioele, Paolo Panaroni, Giancarlo Gennaro, and Umile Paone

Complementing the regular sessions, the program of the conference included the following technical presentations:

- “The IRONSIDES Project: Final Report,” Barry Fagin and Martin Carlisle
- “Concurrent Reactive Objects in Rust—Secure by Construction,” Marcus Lindner, Jorge Aparicio, and Per Lindgren
- “Alire: A Library Repository Manager for the Open Source Ada Ecosystem,” Alejandro R. Mosteo
- “Real-Time Ada Applications on Android,” Alejandro Pérez Ruiz, Mario Aldea Rivas, and Michael González Harbour

We would like to acknowledge the work of all the people who contributed, with various responsibilities and official functions, to the making of the conference program overall. The success of the conference depends in large part on the quality of the program contents. The authors of the selected contributions are to be thanked first and foremost for that. The members of the Program and Industrial Committees had the difficult task of screening the submissions and selecting the contributions to include in this proceedings volume and in the *Ada User Journal*.

The Organizing Committee put it all together: Nuno Neves (Conference Chair); Marcus Völp (Special Session Chair); José Rufino and Marco Panunzio (Industrial Co-chairs); David Pereira (Tutorial and Workshop Chair); Dirk Craeynest (Publicity Chair); Ahlan Marriott and José Neves (Exhibition Co-chairs). All of them deserve our gratitude for their effort.

We hope that the attendees enjoyed every element of the conference program as much as we did in organizing it.

June 2018

António Casimiro
Pedro Ferreira

Organization

General Chair

Nuno Neves	LASIGE/University of Lisbon, Portugal

Program Chair

António Casimiro	LASIGE/University of Lisbon, Portugal

Special Session Chair

Marcus Völp	University of Luxembourg, Luxembourg

Tutorial and Workshop Chair

David Pereira	CISTER/ISEP, Portugal

Industrial Co-chairs

Marco Panunzio	Thales A.S., France
José Rufino	LASIGE/University of Lisbon, Portugal

Publication Chair

Pedro Ferreira	LASIGE/University of Lisbon, Portugal

Exhibition Co-chairs

José Neves	GMV Skysoft, Portugal
Ahlan Marriott	White Elephant GmbH, Switzerland

Publicity Chair

Dirk Craeynest	Ada-Belgium and KU Leuven, Belgium

Local Secretariat

Madalena Almeida	Viagens Abreu, Portugal

Sponsoring Institutions

AdaCore
PTC
RAPITA Systems
Ellidiss Software
CRITICAL Software
LASIGE/FCT

Program Committee

Mario Aldea	Universidad de Cantabria, Spain
Ezio Bartocci	Vienna University of Technology, Austria
Johann Blieberger	Vienna University of Technology, Austria
Rakesh Bobba	Oregon State University, USA
Bernd Burgstaller	Yonsei University, South Korea
António Casimiro	LASIGE/University of Lisbon, Portugal
Juan A. de la Puente	Universidad Politécnica de Madrid, Spain
Virgil Gligor	Carnegie Mellon University, USA
Michael González Harbour	Universidad de Cantabria, Spain
J. Javier Gutiérrez	Universidad de Cantabria, Spain
Jérôme Hugues	ISAE, France
Ruediger Kapitza	Technische Universität Braunschweig, Germany
Hubert Keller	Karlsruhe Institute of Technology, Germany
Raimund Kirner	University of Hertfordshire, UK
Adam Lackorzynski	TU Dresden and Kernkonzept GmbH, Germany
Kristina Lundkvist	Mälardalen University, Sweden
Franco Mazzanti	ISTI-CNR, Italy
Laurent Pautet	Telecom ParisTech, France
Luís Miguel Pinho	CISTER/ISEP, Portugal
Erhard Plödereder	Universität Stuttgart, Germany
Jorge Real	Universitat Politècnica de València, Spain
José Ruiz	AdaCore, France
Sergio Sáez	Universitat Politècnica de València, Spain
Elad Schiller	Chalmers University of Technology, Sweden
Frank Singhoff	Université de Bretagne Occidentale, France
Jorge Sousa Pinto	University of Minho, Portugal
Tucker Taft	AdaCore, USA
Elena Troubitsyna	Åbo Akademi University, Finland
Santiago Urueña	GMV, Spain
Tullio Vardanega	Università di Padova, Italy
Marcus Völp	University of Luxembourg, Luxembourg

Industrial Committee

Ian Broster	Rapita Systems, UK
Luís Correia	EMPORDEF-TI, Portugal
Dirk Craeynest	Ada-Belgium and KU Leuven, Belgium
Thomas Gruber	Austrian Institute of Technology (AIT), Austria
Andreas Jung	European Space Agency, The Netherlands
Ismael Lafoz	Airbus Defence and Space, Spain
Ahlan Marriott	White Elephant, Switzerland
Maurizio Martignano	Spazio IT, Italy
Marco Panunzio	Thales Alenia Space, France
Paul Parkinson	Wind River, UK
Jean-Pierre Rosen	Adalog, France
José Rufino	LASIGE/University of Lisbon, Portugal
Emilio Salazar	GMV, Spain
Helder Silva	EDISOFT, Portugal
Jacob Sparre Andersen	JSA Consulting, Denmark
Andreas Wortmann	OHB System, Germany

Additional Reviewers

Akshith Gunasekaran
Rahma Bouaziz
Hai Nam Tran
Hector Perez
Wenbo Xu

Contents

Safety and Security

Using Safety Contracts to Verify Design Assumptions During Runtime 3
Omar Jaradat and Sasikumar Punnekkat

Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation 19
Irfan Sljivo, Barbara Gallina, Jan Carlson, Hans Hansson, and Stefano Puri

Ada 202X

Safe Dynamic Memory Management in Ada and SPARK 37
Maroua Maalej, Tucker Taft, and Yannick Moy

Safe Non-blocking Synchronization in Ada2x 53
Johann Blieberger and Bernd Burgstaller

Handling Implicit Overhead

On the Effect of Protected Entry Servicing Policies on the Response Time of Ada Tasks 73
Jorge Garrido, Juan Zamorano, Alejandro Alonso, and Juan A. de la Puente

Improved Cache-Related Preemption Delay Estimation for Fixed Preemption Point Scheduling 87
Filip Marković, Jan Carlson, and Radu Dobrin

Real-Time Scheduling

Combined Scheduling of Time-Triggered and Priority-Based Task Sets in Ravenscar 105
Jorge Real, Sergio Sáez, and Alfons Crespo

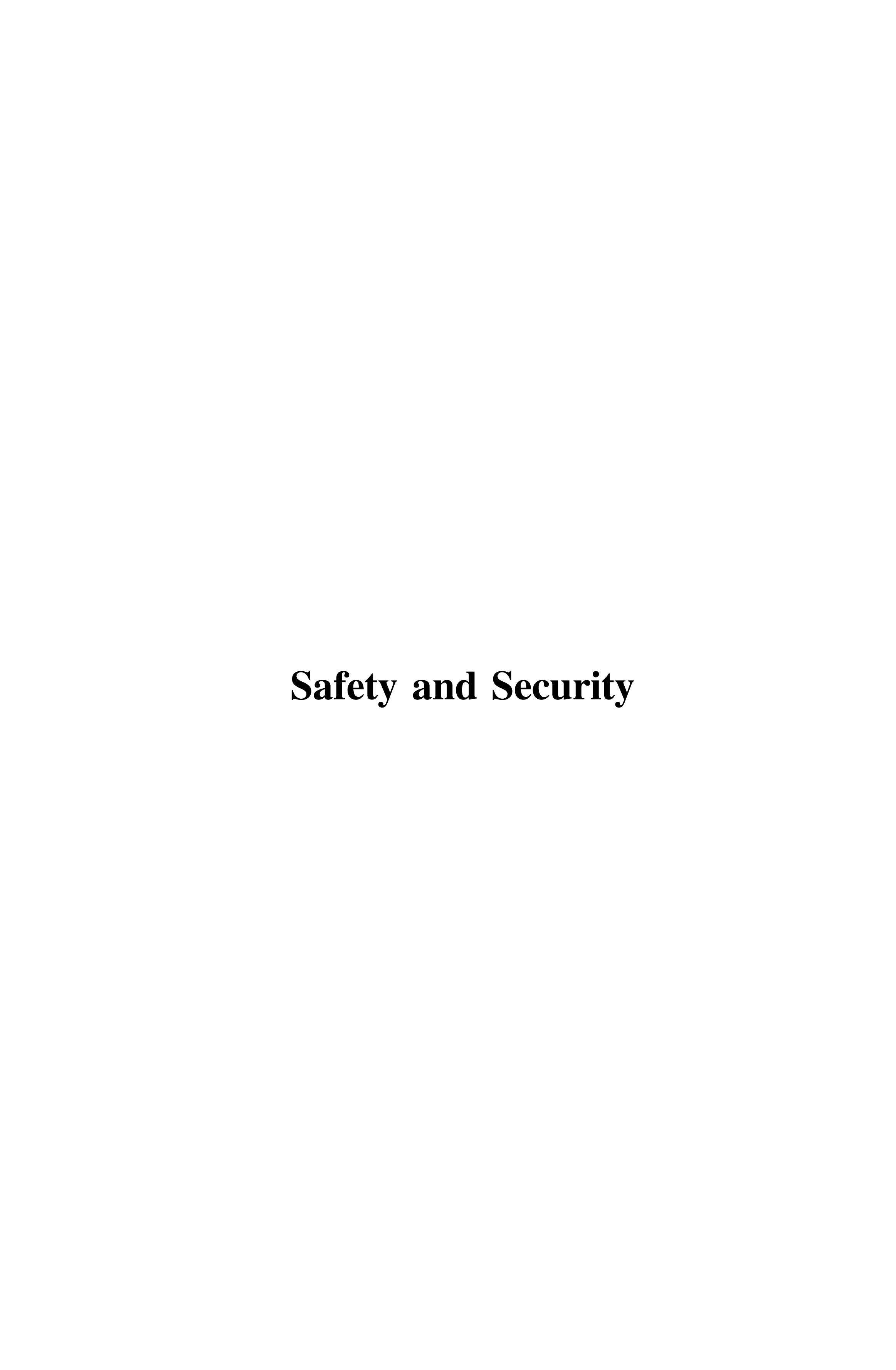
Theory and Practice of EDF Scheduling in Distributed Real-Time Systems 123
J. Javier Gutiérrez and Héctor Pérez

New Application Domains

Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP 141
Sara Royuela, Xavier Martorell, Eduardo Quiñones, and Luis Miguel Pinho

Microservice-Based Agile Architectures: An Opportunity for Specialized Niche Technologies 158
Stefano Munari, Sebastiano Valle, and Tullio Vardanega

Author Index 175



Safety and Security

Using Safety Contracts to Verify Design Assumptions During Runtime

Omar Jaradat(✉) and Sasikumar Punnekkat

School of Innovation, Design and Engineering,
Mälardalen University, Västerås, Sweden
{omar.jaradat,sasikumar.punnekkat}@mdh.se

Abstract. A safety case comprises evidence and argument justifying how each item of evidence supports claims about safety assurance. Supporting claims by untrustworthy or inappropriate evidence can lead to a false assurance regarding the safe performance of a system. Having sufficient confidence in safety evidence is essential to avoid any unanticipated surprise during operational phase. Sometimes, however, it is impractical to wait for high quality evidence from a system's operational life, where developers have no choice but to rely on evidence with some uncertainty (e.g., using a generic failure rate measure from a handbook to support a claim about the reliability of a component). Runtime monitoring can reveal insightful information, which can help to verify whether the preliminary confidence was over- or underestimated. In this paper, we propose a technique which uses runtime monitoring in a novel way to detect the divergence between the failure rates (which were used in the safety analyses) and the observed failure rates in the operational life. The technique utilises safety contracts to provide prescriptive data for what should be monitored, and what parts of the safety argument should be revisited to maintain system safety when a divergence is detected. We demonstrate the technique in the context of Automated Guided Vehicles (AGVs).

Keywords: Confidence · Safety contracts · Safety case
Safety argument · Monitoring · Runtime · Failure rate
Probability of failure · Through-life safety assurance

1 Introduction

Safety critical systems are those systems whose failure could result in loss of life, significant property damage or damage to the environment [1]. Factories are often categorised as safety critical systems since failures of these systems, under certain conditions, can lead to severe consequences [2]. Assuring safety for such systems should provide justified confidence that all potential risks due to system failures are either eliminated or acceptably mitigated. Hence, all failures which might expose the manufacturing processes to hazards shall be analysed

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 3–18, 2018.
https://doi.org/10.1007/978-3-319-92432-8_1

and controlled as part of pre-deployment safety assurance and monitored and controlled as part of operational phase.

Developers of some safety critical systems build a safety case to demonstrate the safety aspect of their system by identifying all unreasonable risks and describing, in the light of the available evidence, how these risks have been eliminated or adequately mitigated. Typically, a safety case comprises both safety evidence (e.g. safety analyses, software and hardware inspection reports, or functional test results) and a safety argument (i.e., reasoning) explaining that evidence. The safety argument shows which claims the developer uses each item of evidence to support and how those claims, in turn, support broader claims about system behaviour, hazards addressed, and, ultimately, acceptable safety [3].

An organisation building a safety case should be accountable for the ownership of the risks to be controlled by adopting an appropriate safety management system, performing a hazard assessment, selecting appropriate controls, and implementing them [4]. In order to help building a sufficient and credible (i.e., on a scientific basis) confidence in the safe performance of a system, its safety case shall always communicate the actual safe performance of the system, and shall always contain only acceptable items of evidence that this system meets its safety requirements. However, an item of evidence is valid only in the operational and environmental context in which it is obtained or to which it applies. More clearly, as the system evolves after deployment, there could be a mismatch between our communicated understanding of the system safety by the safety case and the safety performance of the system in actual operation, which might invalidate many of the prior assumptions made, undermine the collected items of evidence and thus defeat safety claims [5]. Despite the improvements in operational safety monitoring, there is insufficient clarity on how to utilise the analysis results of the monitored data on the documented confidence in safety cases.

In safety critical systems, failure rates are sometimes used as quantitative criteria while performing safety assessment (i.e., Probabilistic Safety Assessment (PSA)). Failure Rate ($FR = \lambda$) is defined as the probability per unit time that a component experiences a failure at time "t", given that the component was operating at time "0" and has survived to time "t" [6]. Failure rates can be deemed as a reliability prediction that together with the consequences (*Risk = probability of failure * consequence of failure*) determine the Safety Integrity Level (SIL), which in turn specifies a target level of risk reduction that should be considered by a safety function or instrument. The quality of the failure rate measure determines the quality of the PSA. Hardware components are usually provided by generic failure rates which are derived by the statistical analyses of the failure frequency [7]. Failure frequency is usually obtained by the test results and the historical data of the components. Although the calculation of a generic failure rate is based on complex models which include factors using specific component data such as temperature, environment, and stress [6], it is, at its best, just a probability that is still subject to a percentage error even if it is used in the same context as in specifications. Assuming the perfection of the failure

rate calculations is not judicious and can be misleading. Hence, a minimum level of fault tolerance in the architectural design of the safety functions should be considered. For example, the functional safety standards IEC 61508 [8] and IEC 61511 [9] recognise that there is always some degree of uncertainty in the assumptions made in calculation of failure rate and probability [10].

In this paper, we propose a novel technique to detect the discrepancies between the failure rates of system's components during their operational life and their generic failure rates used for analysis and assurance during the design time. Since it is infeasible to monitor the failure rates of all components of a system, the technique utilises probabilistic Fault Tree Analysis (FTA) to evaluate the criticality of the system components, and selects the most critical ones for monitoring. The technique derives safety contracts for the selected components and associate them with the relevant events in the FTA and the relevant parts in the safety case. If a discrepancy is detected between an observed failure rate (λ_O) and a generic failure rate (λ_G) of the same component, where $\lambda_O > \lambda_G$, then the relevant contract should be flagged and the referred parts of both the FTA and the safety case should be revisited.

Our hypothesis is that using safety contracts for monitoring the failure rates during the operational life of a system can help to provide essential feedback on the overall confidence in safety. More clearly, getting more precise measure of failure rates than the predicted ones will (1) improve the efficacy of the system design to reduce the risk (mitigate by design), (2) define stronger evidence (e.g., refine or rectify the test results) and (3) highlight the required preventive, corrective, perfective or adaptive maintenance for safer operation

In this paper, we specifically make the following four contributions:

1. A novel technique to continuously reassess the failure rates and use the results to suggest system changes or maintenance
2. A new way to derive safety contracts to facilitate the traceability between the system design, safety analysis and the safety case
3. An example of how to argue more compelling over the failure rate in the light of the derived evidence from the operational phase
4. An example of how to carry out a through-life safety assurance

The rest of the paper is organised as follows: In Sect. 2, we present our approach to verify the design assumptions during runtime by safety contracts. In Sect. 3, we apply our technique to an AGV system to illustrate the main steps. In Sect. 4, we discuss how the suggested approach enables a through-life safety assurance. Finally, we conclude and describe the future directions in Sect. 5.

2 Using Safety Contracts to Verify Design Assumptions During Runtime

Failures of components in safety critical systems are typically divided into four modes, namely, Safe Detected (SD), Safe Undetected (SU), Dangerous Detected (DD), and Dangerous Undetected (DU). DD and DU failures can cause loss of

a safety function while we believe that we are protected and this might happen in fraction of diagnostic interval in case of DD failures or during the unknown downtime in case of DU failures [11]. DU failures are typically due to either random or systematic failures. In this paper, we specifically focus on dangerous failures (DD and DU). Whenever FTAs are constructed to evaluate hazards, the basic event failure data must describe only failures that contribute to that hazard and thus only dangerous failure rates (λ_D) should be included for the basic events, where $\lambda_D = \lambda_{DD} + \lambda_{DU}$.

In this section, we propose a technique that aims to determine the λ_D of particular HW components in their operational life (observed $\lambda_D = \lambda_{D_O}$) and compare the results with the design assumptions of these components (generic $\lambda_D = \lambda_{D_G}$) to ultimately highlight any discrepancies between λ_{D_O} and λ_{D_G}. The technique uses criticality importance measure to rank the components from the most to the less critical so that safety engineers can select particular components for monitoring when it is infeasible to monitor all of them. The technique also uses sensitivity analysis to determine whether a highlighted discrepancy is acceptable or not. The technique heavily depends on probabilistic FTAs, and it comprises 8 steps as follows:

2.1 Determine the PFD or the PFH in the FTA

In this step, we calculate the PFD (Probability of Failure on Demand) or the PFH (Probability of Failure per Hour) using a probabilistic FTA where each component is specified by its λ_{D_G}. The selection between PFD and PFH is based on the demand of a safety function. More clearly, if the safety function will be working in a continuous mode, then we have to select PFH [8]. However, if the safety function is expected to work once per year (at most), then PFD should be selected [8]. To calculate the PFD or PFH of an FTA, four sub-steps should be performed as follows:

A. Calculate the Failure Probability of the Basic Events: There are different formulas used to calculate PFD depending on different factors, such as system's structure (K-out-of-N structures), Common Cause Factor (CCF), operational maintenance, safety standards obligations, etc. For example, Exida (a leading product certification and knowledge company) provides a realistic formula to calculate the PFD [12]. However, the difference between PFD formulas will not be influential in our technique. For the sake of simplicity, we adopt the PFD formula given in [13]. Formula 1 shows how we calculate the PFD for the basic events:

$$PFD(i) = \lambda_{D,i} * \tau \tag{1}$$

where i denotes the basic event and τ is the proof test interval. The component reparation or replacement time is assumed to be short and thus it is negligible.

The main difference between calculating PFD and PFH is in the logic of determining the probability of failures for the basic events. To calculate the PFH for the FTA's events, Formula 1 should be replaced with Formula 2, which

is basically the famous unreliability exponential equation where only λ_D is considered. Unreliability in the context of functional safety is interpreted as the probability of a function to fail during a given time interval.

$$PFH(i) = 1 - e^{-\lambda_D t} \tag{2}$$

For calculating the PFD or PFH, we assume the failure rates of all components are constants, independent and have the same τ. We also assume that all potential CCFs are explicitly modelled as basic events in the FTAs. The rest of the sub-steps (B, C and D) are the same irrespective of we use PFD or PFH.

B. Determine Minimal Cut Set (MCS) in the FTA: The MCS is defined as: *"A cut set in a fault tree is a set of basic events whose (simultaneous) occurrence ensures that the top event occurs. A cut set is said to be minimal if the set cannot be reduced without losing its status as a cut set"* [14]. There are several algorithms to find the MC. We apply Mocus cut set algorithm [14].

C. Calculate the Failure Probability of the Determined MCS: Calculating the probability of occurrence for the top event in a FTA with many MCS requires calculating the probability of those MCS. The failure probability of each determined MCS in the previous sub-step should be calculated according to formula 3 [11], as follows:

$$\check{Q}_j(t) = \prod_{i \in C_j} q_i(t) \tag{3}$$

where $q_i(t)$ denotes the probability of basic event i at time t, $\check{Q}_j(t)$ is the probability that minimal cut set j is in failed state at time t, $i \in C_j$ denotes the minimal cut set j that contains the basic event i.

D. Calculate the PFD or PFH of the Top Event: We calculate the actual PFD or PFH by the *upper bound approximation formula* 4 [11] using the determined MCS, as follows:

$$PFD_{Act}(Top), PFH_{Act}(Top) = \sum_{j=1}^{k} \check{Q}_j(t) \tag{4}$$

So far, all PFD or PFH calculations are based on λ_{D_G}. We refer to the result of the probability calculation based on λ_{D_G} as *Actual or Act*. The PFD$_{Act}$(Top) or PFH$_{Act}$(Top) are design assumptions which will be compared with the observed λ to check the correctness/validity of the design assumptions.

2.2 Identify the Most Critical Components

Monitoring every single component in safety critical systems is infeasible especially since such systems become bigger and more sophisticated over time. However, some components in a system are more critical for the system safety than

other components. The objective of this step is to identify the most critical components in a system w.r.t the FTA. There are different measures through which FTA's events can be ranked based on their importance (e.g., Birnbaum, Criticality Importance, Fussel-Vesely Importance, Risk Achievement Worth (RAW)). In our technique, however, we are interested to rank the components based on their contributions to system safety. More specifically, we are interested in the components whose failures have the maximum impact on system safety. RAW is a measure that focuses on the 'worth' of the basic event in 'achieving' the present level of risk and indicates the importance of maintaining the current level of reliability for the basic event [14]. RAW is often used as an importance measure to rank components in terms of safety significance [15] and hence we will adopt it for our work.

The failure probability of the component i at time t may be described as:

$$P(i) = \begin{cases} 0 & \text{if the component is functioning at time}\, t \\ 1 & \text{if the component is in a failed state at time}\, t \end{cases}$$

The RAW, I^{RAW}(i|t) is the ratio of the (conditional) system unreliability if component i is $P(1)$, and it is calculated as follows [14]:

$$I^{RAW}(i|t) = \frac{1 - h(0_i, p(t))}{1 - h(p(t))} \text{ for } i = 1,\, 2,\, ...,\, n \tag{5}$$

where $h(0_i, p(t))$ is the probability of top event with component $i = P(1)$, and $h(p(t))$ is probability of top event. All basic events should be ranked from the most important to the less important. The most important event is the event for which Formula 5 has the maximum value.

2.3 Refine the Identified Critical Parts

The idea of this step is to discuss with system developers (e.g., safety engineers) and refine the ranked list of the critical components. This step is important, since it embeds the system level knowledge and experience of engineers regarding the uncertainty in a generic λ as well as helps as a validation step in the decision making process. For example, it could be the case that a high ranked critical component in the list has a stable λ_G and systems engineers decide not to monitor it. That is, it is envisaged that some events may be removed from the list or the rank of some of them change. Moreover, the list can be extended to add any additional events by the developers.

2.4 Perform Sensitivity Analysis

The idea of this step is to determine the maximum allowable λ_D (λ_{D_Max}) of the system components which are selected for monitoring. More specifically, we need to define the upper- and lower bounds of the acceptable λ_D of each event in the MCS, where PFD_{Act}(Top) or PFH_{Act}(Top) is less than or equal to the

required probabilities PFD$_{Req}$(Top) or PFH$_{Req}$(Top), respectively. The required probability is described as safety requirements by the safety standards (e.g., SIL, ASIL and DAL). It is important for our technique to determine to which extent PFD$_{Act}$(i) or PFH$_{Act}$(i) can be deviated while PFD$_{Act}$(Top) or PFH$_{Act}$(Top) still satisfies PFD$_{Req}$(Top) or PFH$_{Req}$(Top), respectively. To this end, two main activities should be performed, as follows:

Determine the Maximum Allowable $q_{i,\ Max(t)}$ for Each Component. The $q_{i,Max}$(t) for each component should be determined with respect to PFD$_{Req}$(Top) or PFH$_{Req}$(Top). Formula 6 should be used to determine $q_{i,Max}$(t) for each component at a time.

$$\frac{PFD_{Req}(Top), PFH_{Req}(Top) - (\sum \check{Q}_{i \notin C_j}(t))}{\sum \check{Q}_{i \in C_j}(t) \neg q_i(t)} = \frac{\sum \check{Q}_{i \in C_j}(t)}{\sum \check{Q}_{i \in C_j}(t) \neg q_i(t)} \tag{6}$$

where $i \notin C_j$ denotes the minimal cut set j that does not contain basic event i.
Determine λ_{D_Max} for Each Component. Once we have $q_{i,Max}$(t) for a component it is easy to determine its $\lambda_{D,Max}$. Formula 7 determines $\lambda_{D,Max}$ in case of PFD, as follows:

$$\lambda_{D,Max} = \frac{q_{i,Max}(t)}{\tau_i} \tag{7}$$

Formula 8 determines $\lambda_{D,Max}$ in case of PFH, as follows:

$$\lambda_{D_Max} = \frac{-\ln(q_{i,Max}(t))}{\tau_i} \tag{8}$$

After calculating $\lambda_{D,Max}$ for all events, the latter should be ranked from the most sensitive to the less sensitive to change. The most sensitive event is the event for which Formula 9 is the minimum:

$$Sensitivity(\lambda_{D_i,G}) = \frac{\lambda_{D_i,Max} - \lambda_{D_i,G}}{\lambda_{D_i,G}} \tag{9}$$

2.5 Derive Safety Contracts

In this step, safety contracts should be derived from FTAs. The main objectives of deriving safety contracts are: (1) highlight the most important components to make them visible up front for developers attention [16], and (2) record the thresholds of $\lambda_D(i)$ to continuously compare them with the monitoring results (λ_{D_O}). Hence, if λ_{D_O} of component i exceeds the guaranteed $\lambda_{D_Max}(i)$ in the contract of that component, then we can infer that the contract in question is broken and the related FTA should be re-assessed in the light of the λ_{D_O}. Another objective to derive safety contracts is to associate these contracts with safety arguments as reference points so that developers know the related part of the argument when they review a FTA and vice versa. To this end, we introduce two templates to derive contracts. The first contract template is for deriving a contract for the top event only. The *top event safety contract* is annotated with

the abbreviation "**TE**" in the upper-right corner of the contract to denote that this contract is derived for a **T**op **E**vent as shown in Fig. 1-A.

The second contract template is for deriving a safety contract for each event in the MCS (i.e., events related to important components). This type of contracts is referred to as "*monitoring safety contracts*" and it is is annotated with the abbreviation "**BE**" in the upper-right corner to denote that this contract is derived for a **B**asic **E**vent as shown in Fig. 1-B.

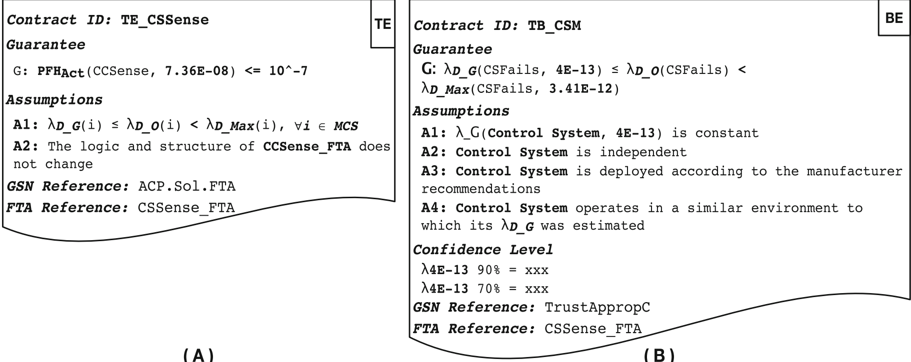


Fig. 1. A. Contract template: Top Event. **B.** Contract template: Basic Event

2.6 Associate Safety Contracts with Safety Arguments

In this step, all safety contracts which were derived in Step 4 should be associated with safety arguments. This step assumes that the safety argument should come down to a claim that the "probability of failure of hazard H due to component failure is acceptable", in turn supported by a context element about what that probability is in the context of an applicable definition of acceptable, in turn supported by the FTA as evidence. An Assurance Claim Points (ACP) [17] should be created between the claim about the acceptable probability and the evidence, where a separate confidence argument should extend this ACP to argue over the quality of the used failure rates to calculate $\mathrm{PFD}_{Act}(\mathrm{Top})$ or $\mathrm{PFH}_{Act}(\mathrm{Top})$.

It is necessary that the argument should be clearly structured and the items of evidence to be clearly asserted to support the argument [18]. There are several ways to represent safety arguments (e.g., textual, tabular, graphical, etc.). In this paper, we use the Goal Structuring Notation (GSN) [18], which provides a graphical means of communicating (1) safety argument elements, claims (goals), argument logic (strategies), assumptions, context, evidence (solutions), and (2) the relationships between these elements. The basic notations of GSN are shown in Fig. 2 (in the upper left side corner). A goal structure shows how goals are successively broken down into ('solved by') sub-goals until eventually supported by direct reference to evidence. GSN can clarify the argument strategies adopted

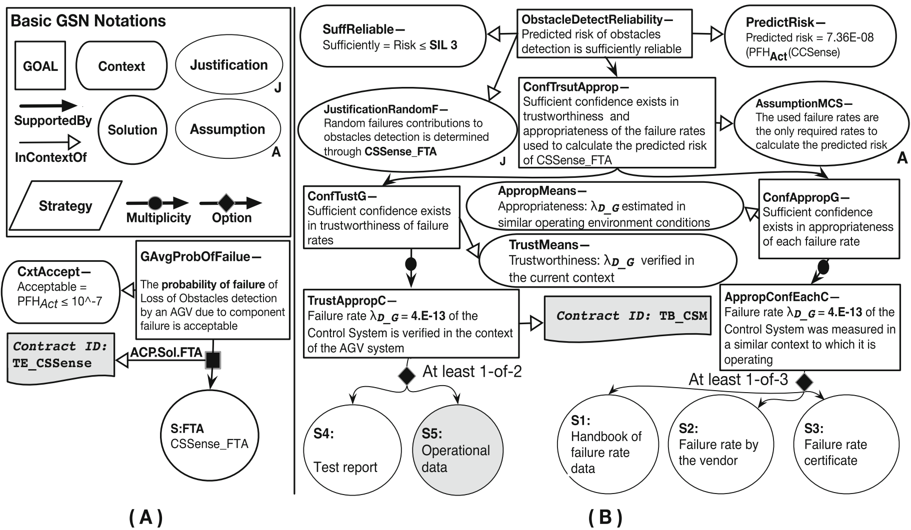


Fig. 2. A. A probability of failure argument with an association of a top event safety contract. **B.** Confidence argument with an association of a monitoring safety contract

(i.e., how the premises imply the conclusion), the rationale for the approach (assumptions, justifications) and the context in which goals are stated.

Assertions in a safety argument relate to the sufficiency and appropriateness of the inferences declared in the argument, the context and assumptions used and the evidence cited [17]. For example, when an item of evidence is used to support a claim, it is asserted that this evidence is sufficient to support the claim. However, a simple 'SolvedBy' relation between the evidence and the claim will not satisfy a reviewer's concerns to reach a certain level of confidence, such as, 'why the reviewer should believe that the evidence is appropriate for the claim?' or 'whether it is trustworthy'.

Hawkins et al. [17] introduced "An assured safety argument" as a new structure for arguing safety in which the safety argument is accompanied by a confidence argument that documents the confidence in the structure and bases of the safety argument. Hawkins suggests that instead of decomposing the arguments further to argue over the appropriateness and trustworthiness of the supporting evidence, an ACP can be created to indicate an assertion in the safety argument. An ACP is indicated in GSN with a named black rectangle on the relevant link and a confidence argument should be developed for each ACP [17]. Three types of assertions were defined as ACPs as follow:

1. Asserted inference: the ACP for an asserted inference is the link between the parent claim and its strategy or sub-claims
2. Asserted context: the ACP for asserted context is the link to the contextual element
3. Asserted solution: the ACP for asserted solutions is the link to the solution element

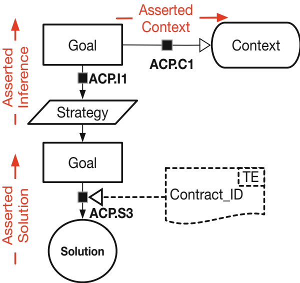


Fig. 3. Types of ACPs with an example of each usage [17]

In this step, we suggest to use the principle of the ACP. Hence, the *top event safety contract* should be associated with the ACP (i.e., asserted solution) between the GSN goal which claims the acceptability of the hazard probability due to a component failure and the GSN solution which refers to the relevant FTA. Whereas, each *monitoring safety contract* should be associated with a GSN goal about the relevant component in the confidence argument. Figure 2-A shows a pattern of PFD or PFH argument and an example of top event safety contract association. Figure 2-B shows a confidence argument pattern with an association of a monitoring safety contract. Figure 3 instantiates an example of each ACP type and it also represents our suggested traceability means which associates the derived contracts from FTAs with safety arguments (the dotted part in the figure).

2.7 Determine λ_{D_O} Using the Data from Operation and Compare it to the Guaranteed λ_{D_Max} in Safety Contracts

In this step, λ_{D_O} of specified components should be obtained during the components' runtime. Using runtime monitors is one way to obtain data from operation. There are many proposed architectures to detect or test a system (or parts of it) for bad behaviour [19]. We provide a monitoring logic which requires two parameters (inputs) from any monitoring framework, namely, the number of recorded failures (i.e., DD and DU) as well as τ in time unit (e.g., hours). Algorithm 1 should be used to determine λ_{D_O} using the data from operation and compare it to the guaranteed λ_{D_Max}. The more we monitor a component and record its failures the more confident we will be in its actual λ_D in a specific context. The calculated level of confidence can reveal how long we still need to monitor a component to reach a certain level of confidence. Hence, our algorithm also calculates the confidence level of λ_{D_O}(i)70% and λ_{D_O}(i)90% continuously and cumulatively using the *Chi-Squared distribution*. The calculated levels of confidence of a monitored component are automatically inserted into its "*monitoring safety contract*" and get updated continuously so that developers and assessors can review them in the FTA and the safety argument.

2.8 Update the Safety Contracts and Re-visit the Safety Argument

If a *monitoring safety contract* is broken it means that there is at least one broken *top event safety contract* as well. In this case, the broken safety contracts should be used to trace the FTA events and elements of safety arguments (for which the contracts were derived). As a result of doing this, developers can specify the entry point of the impact of failure in the safety analysis and the safety argument. It is worth mentioning that we assume the existence of a redundant component of the failing component. Hence, a broken safety contract does not necessarily lead to a total system failure.

Algorithm 1. The monitoring logic to determine λ_{D_O} and compare it to λ_{D_Max}

Data: MissionTime, τ, λ_{D_Max}, λ_{DU_O}, DUfailures = 0, λ_{DD_O}, DDfailures = 0, λ_{D_O}, Num_Comp, CL90, CL70;

Result: Determine λ_{D_O} and compare it to λ_{D_Max}

1 TotMonTime = clock(); *Comment*: start monitoring the mission time
2 **while** *TotMonTime* $\leq$ *MissionTime* **do**
3 Test_Interval_Monitor = clock(); *Comment*: start the monitoring time of the test interval time
4 **while** *Test_Interval_Monitor* $\leq \tau$ **do**
5 **if** *a DD failure is found* **then**
6 DDfailures++; *Comment*: add an observed failure from a diagnosis log file
7 **end**
8 **if** *a DU failure is recorded* **then**
9 DUfailures++; *Comment*: add an observed failure which was inserted manually
10 **end**
11 λ_{DU_O} = 1/((TotMonTime * Num_Comp) / DUfailures); *Comment*: calculate λ_{DU_O}
12 λ_{DD_O} = 1/((TotMonTime * Num_Comp) / DDfailures); *Comment*: calculate λ_{DD_O}
13 $\lambda_{D_O} = \lambda_{DU_O} + \lambda_{DD_O}$; *Comment*: calculate λ_{D_O}
14 CL70 = Chi-Squared($X^2_{70\%,2(DUfailures+DUfailures+1)}$)/(2*Num_Comp*TotMonTime); *Comment*: λ_{D_O}70%
15 CL90 = Chi-Squared($X^2_{90\%,2(DUfailures+DUfailures+1)}$)/(2*Num_Comp*TotMonTime); *Comment*: λ_{D_O}90%
16 **if** $\lambda_{D_O} \geq \lambda_{D_Max}$ **then**
17 Contract [**C**] is broken; *Comment*: highlight the broken contract whenever $\lambda_{D_O} \geq \lambda_{D_Max}$
18 **end**
19 **end**
20 Test_Interval_Monitor = 0; *Comment*: reset the τ timer to start a new one
21 **end**

3 Motivating Example: Automated Guided Vehicles (AGVs)

AGVs are being extensively used for more than 40 years now. They are used for intelligent transportation and distribution of materials in warehouses and auto-production lines. There are different setups and operational assumptions for each application of AGVs in industry. In our example, however, the AGVs are a number of battery-powered vehicles whose movements are autonomous. The AGVs are interfaced to automated warehouse and holding area, and to the machine tools, so that stock movement requirements can be fulfilled. The plant, in our example, is not fully automated so that people cannot be fully excluded from the areas where the AGVs work. Clearly, one of the most important safety features of the AGV vehicles is their ability to detect obstacles and stop quickly in order to avoid a collision with humans, hazardous objects (e.g., flammable materials, electrical resources, other AGVs, etc.). After performing safety analysis, a number of safety hazards were identified. In this paper, we will focus on one hazard, which is: *Loss of obstacle detection while the vehicle is in motion.* A redundant 2-D LiDAR sensor with all-round (360°) visibility is used for detecting obstacles within up to 30 m range. Information about detected obstacles are sent to the control system to determine the manoeuvring strategy to ultimately avoid any potential collision.

According to the likelihood of occurrence, potential consequences and other safety countermeasures in the AGVs, the obstacle detection function is assigned **SIL 3** (Safety Integrity Level) according to IEC 61508. Moreover, since the function under discussion operates in a high demand (i.e., in a continuous mode), the allowable frequency of dangerous failure according to the same standard is PFH $< 10^{-7}$. The proof test interval τ is assumed as 1 year (i.e., 8760 h) for all components. Figure 4 shows an overview of the AGV design (on upper left-hand corner). The figure also shows the FTA of the system where the top event together with the basic events are specified by λ_{D_G}.

Applying the first 5 steps in Sect. 2 is straightforward. Table 1 provides the results of the steps 1–5. The *Refine* column reflects the experts judgment that is supported by the RAW and Sensitivity ranking. For the sake of giving a clear example of what should be done next, we assume that *Control system* got the highest priority for monitoring (the grey row in Table 1). Hence, two contracts should be derived in the case: (1) TE contract TB_CSM and, (2) BE contract (i.e., monitoring contract) $TE_CSSense$.

Step 6 requires associating the derived contracts with the safety argument. For AGV system example, we use our suggested GSN patterns in Sect. 2.6 to create the confidence argument first and then associate the contracts with it through an ACP. Figure 2 presents our safety argument and the role of the proposed monitoring technique to provide supportive evidence for the articulated claims about the failure rates in the argument. Figure 1 shows the derived TE and BE for the top event *CSSense* and the basic event *CSFails*. The figure also shows the GSN and FTA references which reveal the associations (or traceability) of the contracts with the safety argument and the FTA, respectively.

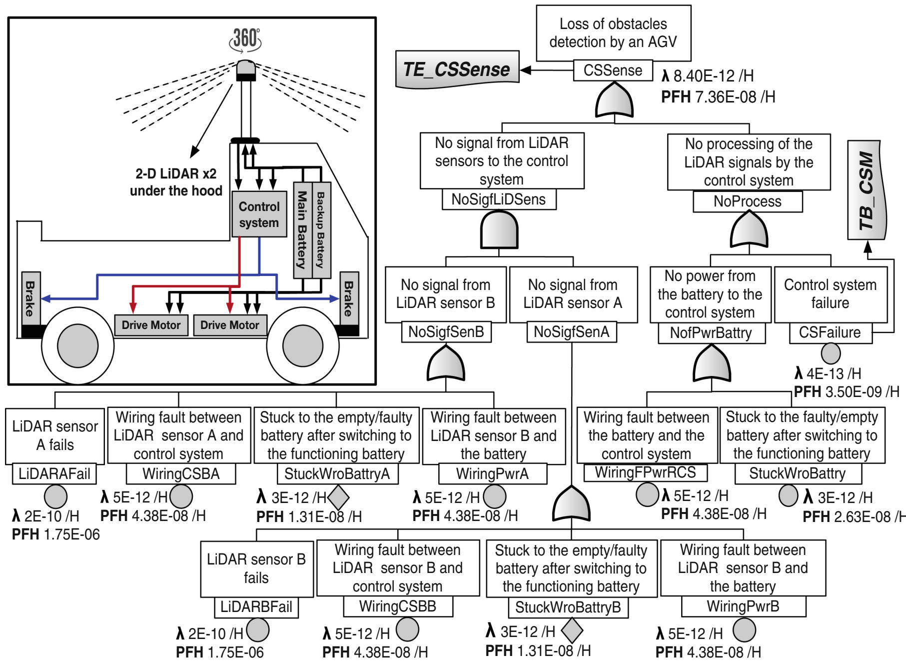


Fig. 4. An overview of AGV's and its probabilistic FTA (CSSense_FTA)

Table 1. A summary of the results of applying the steps 1–5

			STEP 1	STEP 2	STEP 3			STEP 4	STEP 5
No.	**Events**	$\lambda_{D,G}$	**PFH**	**RAW**	**Max PFH**	λ_{D_Max}	**Sensitivity**	**Refine**	**Contract**
1	CSSense (Top)	8.4E-12	7.36E-08		10^{-7}				TE_CSSense
2	CSFails	4E-13	3.50E-09	13589269.0946	2.99E-08	3.41E-12	7.5380	1	TB_CSM
3	WiringFPwrRCS	5E-12	4.38E-08	13589268.5470	7.02E-08	8.02E-12	0.6030		
4	StuckWroBattry	3E-12	2.63E-08	13589268.7851	5.27E-08	6.02E-12	1.0051	3	
5	LiDARAFail	2E-10	1.75E-06	26.3559	1.42E-02	1.63E-06	8137.5	2	
6	WiringCSBA	5E-12	4.38E-08	26.3559	1.42E-02	1.63E-06	325499		
7	StuckWroBattryA	3E-12	2.63E-08	26.3559	1.42E-02	1.63E-06	542499	3	
8	WiringPwrA	5E-12	4.38E-08	26.3559	1.42E-02	1.63E-06	325499		
9	LiDARBFail	2E-10	1.75E-06	26.3559	1.42E-02	1.63E-06	8137.5	2	
10	WiringCSBB	5E-12	4.38E-08	26.3559	1.42E-02	1.63E-06	325499		
11	StuckWroBattryB	3E-12	2.63E-08	26.3559	1.42E-02	1.63E-06	542499	3	
12	WiringPwrB	5E-12	4.38E-08	26.3559	1.42E-02	1.63E-06	325499		

4 A Through-Life Safety Assurance Technique

Denney et al. [5] introduced the term "Dynamic Safety Cases (DSCs)" as a novel operationalisation of the concept of through-life safety assurance. The main motivation for introducing DSCs is that the appreciable degree of certainty about the expected runtime behaviour of a system might not be precise or it perhaps over- or underestimate the actual behaviour, which can create deficiencies in the

reasoning about the safety performance of that system. Hence, there is a need for a new class of safety assurance techniques that exploit the runtime related data (operational data) to continuously assess and evolve the safety reasoning to, ultimately, provide through-life safety assurance [5]. The suggested lifecycle of DSCs comprises four main activities as follows [5]:

1. **Identify** the sources of uncertainty in a safety case.
2. **Monitor** the runtime operation of the related system to collect data about system and environment variables, events, and assurance deficits in the safety argument(s).
3. **Analyse** the collected operational data from the former activity to examine whether the defined thresholds are met, and to update the confidence in the associated claims.
4. **Respond** to operational events that affect safety assurance. Deciding on the appropriate response depends on a combination of factors including the impact of confidence in new data, the available response options already planned, the level of automation provided, and the urgency with which certain stakeholders have to be alerted.

In this section, we explain how using the described technique in Sect. 2 enables a through-life safety assurance, where we (1) identify a source of uncertainty, (2) provide a runtime monitoring mechanism, (3) analyse the collected operational data, and (4) suggest a response to the operational events.

1. Identify a source of uncertainty: Evidence supporting a claim about a prediction of a hardware failure rate may be obtained from different sources. Handbooks produced by commercial, military or government sources can support a claimed prediction of a hardware failure rate. A hardware vendor or an expert might also support such claims. The explicit logic of a claim about a failure rate prediction and its supported evidence is that the predicted likelihood of component C to fail during time T of operation is λ because a handbook, a vendor or an expert "says so". The implicit assumption of such claims is that the actual λ will conform to the predicted λ during the operational life. This assumption is an obvious source of *uncertainty* (i.e., lack of confidence) which can influence the level of confidence in the safety argument. Hence, it is particularly important to know whether or not the actual failure rate of a component during the operational life will be similar to the predicted (i.e., generic) rate as the evidence suggests.
2. Monitor the actual failure rate: Algorithm 1 provides the runtime monitoring logic through which the number of failures of a hardware component is continuously calculated during runtime.
3. Analyse the collected operational data: Algorithm 1 also analyses the calculated number of failures by comparing it with a predefined threshold.

4. Respond to operational events: If an observed λ exceeds the generic λ and it is not tolerated by the maximum allowed λ, then a safety contract is broken. The monitoring algorithm highlights broken contracts indicating that an additional safety countermeasure should be considered, such as replacing a hardware component with an ultra reliable component or add a redundant component. Since the contracts under monitoring by the algorithm is associated with ACPs in the safety argument, a broken contract indicates the affected GSN elements in the argument.

5 Discussion and Conclusion

Numerous studies and data analysis have shown either a decreasing or increasing failure rate with time. Runtime monitoring enables a new source of data which improves our perception of some functions, components, and behaviours within safety critical systems. Monitoring a property of interest of a system component and analysing the collected data enable us to know more about this component (e.g., the way it behaves, fails, etc.). As a result, we can improve our confidence in safety based upon more conscious reasoning that replaces the intuitive evidence by more cognitive one. Some safety standards require monitoring and re-assessing the reliability parameters which were used during the design time. For example, IEC 61511-1 [9] requires operators to monitor and assess whether reliability parameters of the Safety Instrumented Systems (SIS) are in accordance with those assumed during the design time [10]. Although runtime monitoring is not a new technique, there is no single way to specify what to monitor, why and how. Safety contracts, on the other hand, are useful for building, reusing or maintaining safety critical systems. The cost of maintaining system components can be drastically reduced by using contracts as system developers may rework the components with knowledge of the constraints placed upon them [20].

In this paper, we proposed a novel technique to monitor the runtime of a system and detect the divergence between the failure rates (which were used in the safety analyses) and the observed failure rates in the operational life. The technique enables through-life safety assurance by utilising safety contracts to provide prescriptive data for what should be monitored, and what parts of the safety argument should be revisited to maintain system safety when a divergence is detected. Future work will focus on creating a more in-depth case study to validate both the feasibility and efficacy of the technique for software and hardware applications. We also plan to formally define safety contracts and to fully automate the application of the technique.

Acknowledgment. This work has been partially supported by the Swedish Foundation for Strategic Research (SSF) (through SYNOPSIS and FiC Projects) and the EU-ECSEL (through SafeCOP project).

References

1. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th International Conference on Software Engineering (ICSE), pp. 547–550, May 2002
2. Jaradat, O., Sljivo, I., Habli, I., Hawkins, R.: Challenges of safety assurance for industry 4.0. In: European Dependable Computing Conference (EDCC). IEEE Computer Society, September 2017
3. Jaradat, O., Graydon, P., Bate, I.: An approach to maintaining safety case evidence after a system change. In: Proceedings of the 10th European Dependable Computing Conference (EDCC), UK (2014)
4. Graydon, P.J., Holloway, C.M.: An investigation of proposed techniques for quantifying confidence in assurance arguments. Saf. Sci. **92**(Supplement C), 53–65 (2017)
5. Denney, E., Pai, G., Habli, I.: Dynamic safety cases for through-life safety assurance. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 587–590, May 2015
6. Reliability prediction basics. Technical report, ITEM Software Inc. (2007)
7. Pittiglio, P., Bragatto, P., Delle Site, C.: Updated failure rates and risk management in process industries. Energy Procedia **45**(Supplement C), 1364–1371 (2014). ATI 2013 - 68th Conference of the Italian Thermal Machines Engineering Association
8. Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508-4 (2010)
9. Functional safety - Safety instrumented systems for the process industry sector. IEC 61511-1 (2016)
10. Generowicz, M., Hertel, A.: Reassessing failure rates. Technical report, I&E Systems Pty Ltd. (2017)
11. Rausand, M.: Reliability of Safety-critical Systems: Theory and Applications. Wiley, Hoboken (2014)
12. van Beurden, I., Goble, W.M.: The Key Variables Needed for PFDavg Calculation. White paper, Exida, Sellersville, PA 18960, USA, July 2015
13. Goble, W.M.: Control System Safety Evaluation and Reliability, 2nd edn. (1998)
14. Rausand, M., Høyland, A.: System Reliability Theory: Models and Statistical Methods and Applications. Wiley, Hoboken (2004)
15. van der Borst, M., Schoonakker, H.: An overview of PSA importance measures. Reliab. Eng. Syst. Saf. **72**(3), 241–245 (2001)
16. Jaradat, O., Bate, I., Punnekkat, S.: Using sensitivity analysis to facilitate the maintenance of safety cases. In: Proceedings of the 20th International Conference on Reliable Software Technologies (Ada-Europe), pp. 162–176, June 2015
17. Hawkins, R., Kelly, T., Knight, J., Graydon, P.: A new approach to creating clear safety arguments. In: Dale, C., Anderson, T. (eds.) Advances in Systems Safety, pp. 3–23. Springer, London (2011). https://doi.org/10.1007/978-0-85729-133-2_1
18. GSN Community Standard Version 1. Technical report, Origin Consulting (York) Limited, November 2011
19. Kane, A.: Runtime monitoring for safety-critical embedded systems. PhD thesis, Carnegie Mellon University, September 2015
20. Bates, S., Bate, I., Hawkins, R., Kelly, T., McDermid, J., Fletcher, R.: Safety case architectures to complement a contract-based approach to designing safe systems. In: Proceedings of the 21st International System Safety Conference (ISSC) (2003)

Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation

Irfan Sljivo[1(✉)], Barbara Gallina[1], Jan Carlson[1], Hans Hansson[1], and Stefano Puri[2]

[1] Mälardalen University, Västerås, Sweden
{irfan.sljivo,barbara.gallina,jan.carlson,hans.hansson}@mdh.se
[2] Intecs, SpA, Pisa, Italy
stefano.puri@intecs.it

Abstract. Contracts are envisaged to support compositional verification of a system as well as reuse and independent development of their implementations. But reuse of safety-relevant components in safety-critical systems needs to cover more than just the implementations. As many safety-relevant artefacts related to the component as possible should be reused together with the implementation to assist the integrator in assuring that the system they are developing is acceptably safe. Furthermore, the reused assurance information related to the contracts should be structured clearly to communicate the confidence in the component. In this work we present a tool-supported methodology for contract-driven assurance and reuse. We define the variability on the contract level in the scope of a trace-based approach to contract-based design. With awareness of the hierarchical nature of systems subject to compositional verification, we propose assurance patterns for arguing confidence in satisfaction of requirements and contracts. We present an implementation extending the AMASS platform to support automated instantiation of the proposed patterns, and evaluate its adequacy for assurance and reuse in a real-world case study.

1 Introduction

Software-intensive systems are rarely developed from scratch. Instead, components developed previously are reused for building new systems [1]. The same trend is visible in safety-critical systems, which usually need to be assured that they are acceptably safe to be deployed. The assurance entails gathering a body of evidence in form of a safety assurance case to communicate that any unreasonable risk in the system has been mitigated. Due to this, reuse of components in such systems is not complete without the reuse of assurance information associated with the component. While reuse of safety-related components is very much present in safety-critical systems development, the lack of systematic approaches to managing reuse of both components and their accompanying assurance information has shown to be dangerous in the past [2].

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 19–33, 2018.
https://doi.org/10.1007/978-3-319-92432-8_2

To address the issue of reuse in safety-critical systems, some reuse principles have been promoted through the safety standards. For example, the automotive functional safety standard ISO 26262 [3] with its concept of Safety Element out-of-Context (SEooC) for reuse of components together with the related safety assurance information. It promotes principles that should be followed to begin the assurance process on the level of the SEooC, which is being developed independently from the system in which it will be used. The purpose of the early start of the assurance process is to support the integrator of the SEooC in assuring their system according to the standard. Ideally, if all suppliers would provide their components as SEooC, the integrator should have an easier job of assuring that the integrated system is acceptably safe. The core aspect of SEooC development are assumptions on the context in which the SEooC component could be reused, such that their validation upon reuse establishes whether the component and the related assurance information is reusable in the particular context.

To support SEooC development and reuse, we have proposed to use *assumption/guarantee component contracts* in our previous work [4]. A contract is a pair of assertions called assumptions and guarantees, where the component guarantees a certain behaviour, given that the environment in which it is deployed fulfils the assumptions [5]. Such contracts provide a systematic way to capture the context assumptions and relate them with the properties that the SEooC component implements. We have proposed to relate contracts with the assurance information [4] and support contract-driven assurance by automating the generation of assurance argument-fragments on satisfaction of both such contracts and the system requirements that can be validated via those contracts.

Reusable components such as SEooC are often characterised with parameters that are used to tailor the behaviour of the component in the different settings in which the component is reused. To address such need for variability at the contract level, we have made a distinction between strong and weak contracts [6]. On the one hand, the strong contracts are those whose assumptions should be met by every context in which the component is reused, hence its guarantees are always offered by the component. On the other hand, the weak contract assumptions do not need to be satisfied by every context in which the component is reused, but when they are met, only then the component offers the corresponding weak guarantees. This variability on the contract level can be used to identify which assumed safety requirements offered by the SEooC component are relevant in the system in which the SEooC is reused. Hence, the safety case information related to those requirements and contracts can also be identified for reuse. To set the ground for tool support, we have proposed a generic SEooC MetaModel (SEooCMM) that defines relationships between SEooC components, contracts, requirements and assurance assets [4]. The basic elements needed for the tool support are a system modelling tool compliant with the SEooCMM, a contract checking engine, and a safety case modelling tool.

In this paper we present our efforts to provide tool-support for contract-based design that incorporates strong and weak contracts as well as the automated

generation of assurance arguments. We turn to the AMASS[1] platform for our implementation as it includes the needed tools for system modelling (CHESS[2]), contract checking (OCRA[3]) and safety assurance case modelling (OpenCert[4]). Two challenges arise when using the AMASS platform for contract-driven reuse and assurance: (1) the contract-based design framework [7] implemented in OCRA does not distinguish between the strong and weak contracts; (2) the connection between the system and assurance modelling domains is not clearly defined. To address the first challenge, we define the strong and weak contracts in the scope of the contract-based framework implemented in OCRA. Moreover, we present how refinement checking can be adapted to support strong and weak contracts through the interaction of CHESS and OCRA. To address the second challenge, we first identify the information needed to perform contract-driven assurance and extend CHESS to allow for its modelling. We structure that information by extending the argument pattern for assurance of contract satisfaction to account for the hierarchical component decomposition defined through the notion of refinement. Then, we develop a transformation from the system model to the assurance model that automatically instantiates the defined argument-fragment for each component in the system. Finally, we validate the tool-supported contract-based assurance and reuse methodology in a real-world case study.

As assurance cases are gaining popularity, there is an increasing number of tools supporting their development with particular focus on automation capabilities. For example, *Safety.Lab* [8] focuses on model-based safety analysis and generates an argument structure from rich models of various safety-relevant artefacts. The Eclipse-based *Resolute* tool [9] facilitates generating assurance arguments from architectural models. The *Evidence Confidence Assessor (EviCA)* [10] is a diagramming tool that supports automated generation of confidence arguments related to manually created arguments. The *AdvoCATE* [11] toolset includes a variety of automated features for assurance case creation and analysis. AdvoCATE automates instantiation of pre-developed argumentation pattern from a hazard and safety requirement analysis. While we also automatically instantiate a pre-developed pattern, we do so from architectural models enriched with assumption/guarantee contracts coupled with safety-relevant artefacts. This allows us to filter the relevant artefacts and provide additional support for reuse and tailoring of context-specific automated argument generation.

The rest of the paper is organised as follows: In Sect. 2, we present some background information. We present the tool-supported methodology for contract-driven assurance and reuse in Sect. 3. In Sect. 4, we present our case study. Finally, we bring conclusions and indicate future work in Sect. 5.

[1] AMASS - Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems, https://amass-ecsel.eu/.
[2] https://www.polarsys.org/chess.
[3] https://ocra.fbk.eu/.
[4] https://www.polarsys.org/projects/polarsys.opencert.

2 Background

In this section we first present the tools and concepts we build upon, and then we present the system description of the considered case study.

2.1 AMASS Platform

The AMASS platform encompasses different tools, but we focus on the three tools that facilitate system modelling (CHESS), formal verification of assumption guarantee contracts (OCRA), and assurance case modelling (OpenCert). An overview of the three tools is shown in Fig. 1. In the reminder of the section, we present the tools together with their underlying theoretical concepts.

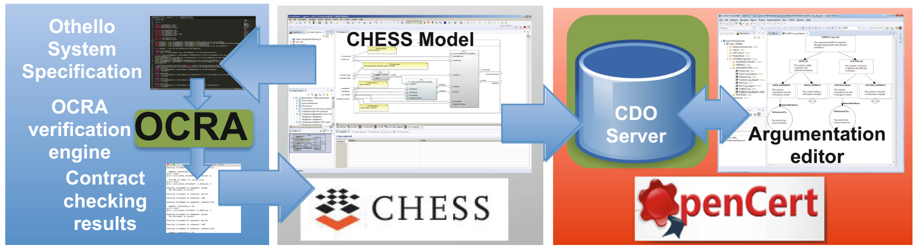


Fig. 1. The overview of the tool information flow

System Modelling: CHESS provides an editor to model all phases of system development: from requirements definition, architecture modelling to software design and its deployment to hardware. In the CHESS toolset, components can be modelled as component types or component instances. Component types can be seen as elements out of context, and component instances as the in-context representation of the corresponding component types. Component instances inherit the attributes of the corresponding component type. System modelling in CHESS includes support for contract-based design, which relies on describing behaviours of components in terms of contracts. CHESS supports modelling of both strong and weak contracts and their association with components and system requirements. Moreover, *delegationConstraint* modelling element can be used to instantiate a component parameter in the given system model. Furthermore, CHESS facilitates interfacing with OCRA, such that the CHESS model together with the contracts is exported in the Othello System Specification (OSS) format used by OCRA. The contract checking is done by OCRA and the result is back-propagated to the CHESS model, as shown in Fig. 1.

Contract-Based Design: OCRA [7] is a tool for compositional verification of logic-based contract refinement built upon the OSS language, supporting a

trace-based approach to contract based design. The semantics of both components and contracts is built around the notion of a *trace*, i.e., the observable part of an execution of a component. Following the trace-based semantics, a *component* S is described with a set V_S of variables that are visible outside of the component, and a set of all traces over V_S is denoted as $Tr(V_S)$. Then, an environment of S is a subset of $Tr(V_S)$. Assuming an assertion language, an assertion A can be described by an associated set of ports V_A and a semantics $[\![A]\!]$ defined as a subset of $Tr(V_A)$. Building on top of the assertion language, a contract $\mathrm{C} = (A, G)$ of the component S is a pair of assertions namely assumptions (A) and guarantees (G) over V_S. An environment E is said to be a correct environment of C iff $E \subseteq [\![A]\!]$. Contract *refinement* represents the backbone of checking the component decomposition [7]. Informally, a set of contracts of the sub-components *refines* a contract of the composite component if: (i) the assumptions of all subcomponent contracts are met by the other sub-components and the environment defined by the assumptions of the composite component contract; and (ii) the sub-component contracts deployed in the environment defined by the composite contract assumptions imply the composite contract guarantees. For a formal definition of the refinement refer to [7].

Safety Case Modelling: A safety assurance case is often defined as an explained and well-founded (supported by evidence) structured argument to show that the system is acceptably safe to operate in a given context [12]. It is often required (explicitly or implicitly) by safety standards such as ISO 26262. Safety case is composed of all the work products gathered during the development of a safety-critical system. The spine of a safety case is a safety argument which connects the safety requirements and the evidence supporting and justifying those requirements. Goal Structuring Notation (GSN) [12] is a graphical argumentation notation used for safety case modelling. Since similar rationales exist behind specific arguments in different contexts, argument patterns of reusable reasoning are defined by generalising the specific details of an argument. The basic elements of GSN are shown in Fig. 3, for more details we refer the reader to the GSN Standard document [12]. To provide a better portability and exchange of the safety arguments, a Structured Assurance Case Meta-model (SACM) [13] standard is developed by Object Management Group. Since SACM captures the basic argumentation elements and their relationships, it can be used to instantiate different compliant meta-models for different argumentation notations such as GSN and Claims-Arguments-Evidence (CAE).

OpenCert is an assurance and certification tool environment with a safety argumentation modelling editor compliant with the standardised SACM. It further includes a Connected Data Objects[5] (CDO) server that supports collaborative modelling. In particular, it stores the safety case models in a database on a CDO server such that different distributed clients can access the models and work on the same safety case concurrently.

[5] https://www.eclipse.org/cdo/.

2.2 The Motivating Case

In this paper we will use a wheel-loader use case [4] to validate our approach. Wheel-loaders are usually equipped with a loading arm, which can perform up and down movements. The Loading Arm Control Unit (LACU) is the software control unit that coordinates the arm movement. The LACU architecture modelled in CHESS is shown in Fig. 2. It consist of a component providing the current arm position, and an arm controller which sends the arm movement command. Moreover, it includes the Loading Arm Automatic Positioning (LAAP) component which can automatically move the arm to a pre-defined position. In this particular LACU the position is fixed (whereas it in other cases can be modified by the operator), while the maximum ground speed of the vehicle is 70 km/h and the speed limit for moving the arm is 20 km/h, as shown in Fig. 2. The LAAP component is developed independently of this system as a SEooC.

The LACU safety analysis revealed the following system hazards: (1) unintended arm movement, and (2) arm movement during high speed (i.e., when the maximum speed of is greater than the ground speed limit). Some of the safety requirements defined to minimise the risks of those hazards from occurring are SR1: "*The stop position of the loading arm shall not deviate more than* ± 0.04 *rad*" and SR2: "*The loading arm shall be disabled during high speed*".

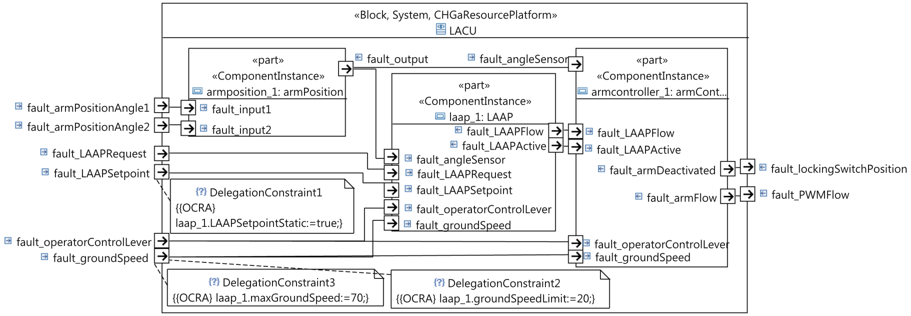


Fig. 2. The CHESS diagram of the LACU architecture

3 Contract-Driven Assurance and Reuse

In this section, we present the methodology for supporting contract-driven assurance and reuse of safety relevant components. We first describe how to assure safety requirements validated through contract-based design. Then, we focus on the support for the contract-driven reuse of the components and their assurance information in the context of a trace-based approach to contract-based design.

3.1 Contract-Driven Assurance

To assure that a system such as LACU satisfies a given safety requirement based on the related contract, we need to provide evidence that the contract correctly

represents the requirement (often said that its guarantees formalise the requirement) and evidence that the contract is satisfied with sufficient confidence in the given system context. We refer to this argument strategy as the contract-based requirements satisfaction pattern, shown in Fig. 3.

While compositional verification of a system using contracts establishes validity of a particular requirement on the system model in terms of contracts, confidence that the system implementation actually behaves according to the contracts should also be assured. Hence, to drive the system assurance using contracts we have associated assurance assets with each contract. Those assets can be different kinds of evidence that increase confidence that the component (i.e., the implementation of the contracts) behaves according to the contract, i.e., that the component deployed in any environment that satisfies the contract assumptions exhibits the behaviours specified in the corresponding contract guarantees. To argue that a contract is satisfied with sufficient confidence we need to assure that the component actually behaves according to the contract, and that the environment in which the component is deployed satisfies the contract assumptions [14]. But when we deal with hierarchical systems where contracts are defined on each hierarchical level with well defined decomposition conditions, then to argue that the composite component behaves according to the contract, we should explicitly argue over the component decomposition. The argument-pattern in Fig. 4 presents an extended contract-satisfaction argument pattern [14] with contract decomposition.

The extension assures that for each of the contracts on the composite component level (e.g., LACU) related to the requirement we are assuring, we should ensure that we have confidence in the component decomposition described by the refinement relationship (the *contractDecomp* goal). The goal is decomposed such that we argue over confidence in all subcomponent contracts specified through the refinement relationship. While the *contractDecomp* goal assures that what the component offers is supported by the confidence in the internal subcomponent specification, the *contractAssume* goal assures that the environment of the component/system meets the relevant assumptions.

3.2 Contract-Driven Reuse of Safety-Relevant Components

Reuse is intrinsic to contract-based design. It enables checking if a component can be reused in a particular system, i.e., whether the system meets its demands

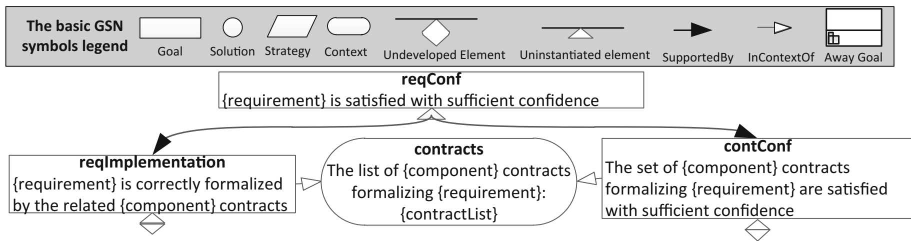


Fig. 3. Contract-driven requirement satisfaction assurance argument pattern

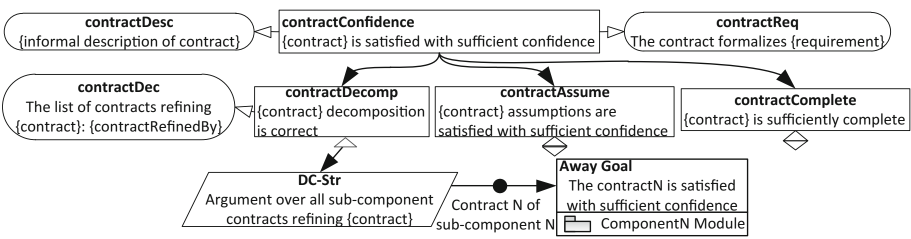


Fig. 4. Contract satisfaction assurance argument pattern

and whether the component meets the demands of the system. The support for reuse in contract-based design has been mainly focused on components (i.e., implementations of contracts) and not reusable components as implementations of a set of contracts for different environments that may or may not be satisfiable together. As mentioned in Sect. 1, we refer to contracts that are not required to be satisfied by all correct environments as weak contracts. Conversely, the strong contracts define all the correct environments, i.e., all correct environments need to comply with all the strong contracts, while typically only some correct environments need to comply with a particular weak contract.

We formally describe the strong and week contracts in terms of environments in the context of the trace-based contract framework [7]: for a component S described with a set of strong contracts $\xi_S(S)$ and a set of weak contracts $\xi_W(S)$, we say that an environment E is a *correct environment* of S if: $\forall (A, G) \in \xi_S(S), E \subseteq [\![A]\!]$, i.e., for an environment of S to be correct, it must satisfy the assumptions of all the strong contract of S. We denote with $\mathcal{E}(S)$ all the correct environments of S. Such correct environments may or may not satisfy the assumptions of the weak contracts of S. While this provides some flexibility in specification of contracts, it may also mean that some weak contracts may never be validated in any of the correct environments e.g., if a weak contract is contradicting a strong contract. For S not to contain such unnecessary weak contracts we require that each weak contract of S has at least one correct environment that satisfies its assumptions, i.e.,: $\forall (A, G) \in \xi_W(S), \exists E \in \mathcal{E}(S), E \subseteq [\![A]\!]$.

The problem with specifying such contracts is that if we try to check refinement by considering all the specified weak contracts, the check will fail since a single environment might not be able to meet the assumptions of all the weak contracts. To overcome this problem without redefining the notion of contract refinement, we can either (i) filter the weak contracts before checking the refinement, such that only weak contracts whose assumptions are met by the current environment are included in the refinement check; or (ii) transform the weak contracts in a different format such that refinement can be performed:

Weak Contract Filtering: While a SEooC is described with sets of both strong and weak contracts, when instantiated to a particular correct environment E then, for the purpose of refinement check, it is enough to describe the

SEooC instantiation with a subset of contracts that are applicable in the environment E. Given a SEooC component S and its instantiation S' in a correct environment $E \in \mathcal{E}(S)$, the set of contracts of S' denoted with $\xi(S')$, which contains the contracts considered during refinement check, is a union of all the strong contracts from $\xi_S(S)$ and only those weak contracts from $\xi_W(S)$ whose assumptions are satisfied by the environment E.

Weak Contract Transformation: Instead of filtering only some weak contract to perform the refinement check, the refinement check could be performed if the weak contracts are transformed such that they do not impose restrictions on the environment. This can be done if the weak contract assumptions are relaxed. For a weak contract $C = (A, G)$ of a component S, a relaxed counterpart of this weak contract would be $C' = (true; A \implies G)$, where *true* represents an assertion satisfied by all environments. The relaxed counterpart has relaxed assumptions, hence it differs from the corresponding weak contract in terms of environments, but they are the same from the perspective of implementations. Since the assumption of C' is satisfied by every correct environment of S, it can be regarded as a strong contract. Since any contract that is refined by C is also refined by C', either form can be used for the sake of checking refinement of a weak contract. If we have a set of weak contracts and we transform them to their relaxed form and conjunct them to a single contract by conjunction of their guarantees, then any contract that is refined by at least one of those weak contracts is also refined by the conjuncted contract. The SEooC instantiation in a particular context does not require contract filtering in this case, but the in-context component can inherit both strong and weak contracts. Since the refinement check by considering all the strong and weak contracts would fail in case of two weak contracts that do not share the same correct environments, we transform the weak contracts to the appropriate format described as follows: given a SEooC component S and its instantiation S' in a correct environment $E \in \mathcal{E}(S)$, the set of contracts of S' denoted with $\xi(S')$, which contains the contracts considered during refinement check, is a union of all the strong contracts from $\xi_S(S)$ and the conjuncted contract of all the weak contracts in their relaxed form from $\xi_W(S)$.

Although this approach allows all the contract specifications to be used for checking the refinement, it does not reveal which weak contracts are relevant in the environment E, i.e., assumptions of which weak contract are satisfied by E. Not knowing which weak contract is relevant in the current environment means that we do not know which weak contract and its assurance assets we should use in the assurance case. For the sake of reuse we still need to check which weak contracts are relevant in the environment E.

3.3 Tool Support

We build upon the synergy of the three tools presented in Sect. 2 and implement the contract-driven assurance and reuse methodology by developing new and

upgrading the existing plugins within the tools. We extend CHESS to support SEooCMM by adding the possibility to capture information about assurance assets and their relation to the corresponding contracts. With OCRA results back-propagated to the CHESS model, we perform automated weak contract filtering for the component instances. Upon updating the CHESS model, we then automatically instantiate the contract-driven assurance argumentation patterns for each component in the CHESS model. The generated argumentation is stored on a CDO server which can be accessed by any OpenCert argumentation editor connected to the CDO server. In the reminder of the section we detail the implementation (available on the CHESS [6] and OpenCert[7] repositories) of refinement checking with strong and weak contracts as an extension of CHESS and the automatic argument generation as an OpenCert plugin.

Refinement Checking with Strong and Weak Contracts: As mentioned in Sect. 3.2, to use a contract checking engine such as OCRA, which does not distinguish between strong and weak contracts, we can either support "weak contract filtering" as a part of reusable component instantiation or weak contract transformation to an appropriate format. We extend CHESS so that we can check all the weak contract validity and automatically update the component instance by indicating which weak contracts are valid in the given environment.

To fully support the presented methodology, we have also implemented the second solution that includes all weak contracts in contract refinement checking. The choice of which type of refinement with strong and weak contract to use is up to the user, as it allows for different possibilities. When the users are manually selecting which weak contracts they want in the given context, then they may have to manually check which of them are relevant for their system. Conversely, when the user selects to perform refinement check with all the weak contracts, then if any of the weak contracts meet the system demands, the refinement will be successful and the weak contracts applicable in the given context will be automatically indicated without the need to manually select them. Our CHESS extensions to support the contract-driven assurance and reuse are hosted in the following CHESS plugins:

- *org.polarsys.chess.contracts.transformations* – contains model to text [15] transformation for generating the .oss file representing the model;
- *org.polarsys.chess.contracts.integration* – contains interface for communicating with OCRA.

Automated Argument-Fragment Generation: To facilitate automated instantiation of the contract-driven assurance pattern from Sect. 3, we implement the *ArgumentGenerator* plugin[8] within OpenCert. The user is prompted to select both the source CHESS model and the target assurance case in the CDO

[6] https://git.polarsys.org/c/chess.

[7] https://git.polarsys.org/c/opencert.

[8] org.eclipse.opencert.chess.argumentGenerator.

repository. The plugin generates a set of argument-fragments from the source CHESS model and stores them in the corresponding target assurance case in the CDO repository. The ArgumentGenerator assumes that the CHESS model contains contract specifications and that the contract refinement check has been performed such that the status of both strong and weak contracts is updated to indicate if the contract is validated in the given context or not. The argument generation creates an argument-fragment for each component. The connection between different argument-fragments is done through away goals. The resulting argument-fragments can be viewed in the target assurance case by anyone with access to the CDO server from an OpenCert argumentation editor.

4 LACU Case Study

In this section, we present our case study with the objective to apply the tool-supported contract-driven assurance and reuse methodology on a real-world case and evaluate its adequacy for automated support of assurance and reuse of assurance assets. We first present the failure propagation modelling in CHESS of the LACU and its in-context components, as well as the reusable LAAP component. Then, we discuss the contract checking results, and present the automatically generated argument-fragments.

4.1 Failure Propagation Modelling

To analyse the satisfaction of the safety requirement SR1 mentioned in Sect. 2.2, we model LACU with faults as different input/output ports of the components. For example, we consider the deviation of ± 0.04 rad from the stop position to be a fault of the LACU arm positioning command represented by the *fault_PWMFlow* port. Hence, the goal of the contract corresponding to such an interpretation of SR1 would be to guarantee that *fault_PWMFlow* never occurs. To guarantee such a property in the context of the LACU defined by the parameters specified in Fig. 2, both one of the angle sensors and the ground speed sensor need to provide correct values. Furthermore, the operator inputs LAAPRequest and operatorControlLever should be fault free as well. This is captured in the *LACU_fault_propagation* contract in Fig. 5. To ensure that the component decomposition with respect to the fault propagation is done correctly, we define fault propagation contracts on the sub components as well. Figure 5 presents the contracts for LACU, and its armPositioning and armController sub-components, as these are the components modelled for the particular wheel-loader.

The contracts of LAAP as a reusable component are specified separately, as they deal with not just this particular wheel-loader, but also other wheel-loaders that may support dynamic automatic positioning or that may or may not be able to move at high speed. We define four different weak contracts for the four different environments based on the two aspects of the wheel-loaders: dynamic automatic positioning and high-speed capability. In all environments the LAAP depends on fault-free user input, hence all contracts have the same assumptions

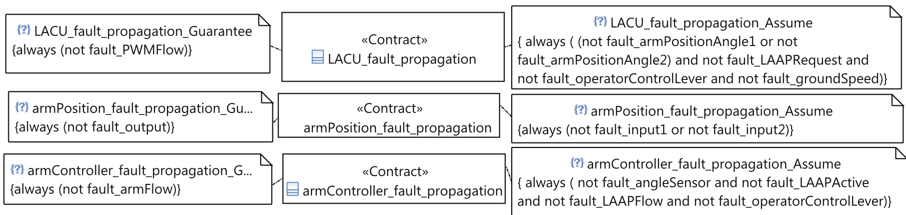


Fig. 5. The LACU strong contracts specified in CHESS

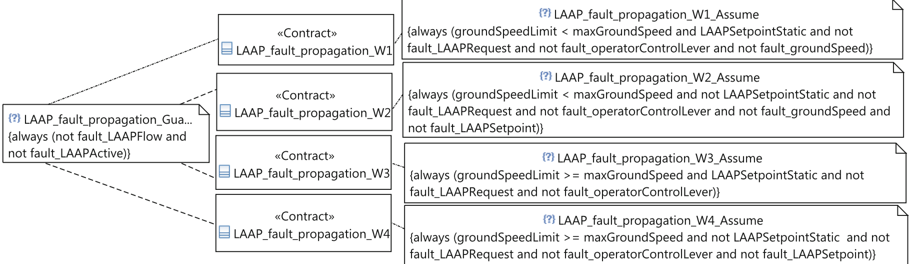


Fig. 6. LAAP fault propagation weak contracts specified in CHESS

considering fault free LAAPRequest and operatorControlLever. But on top of those conditions, for LAAP to ensure it will not issue a faulty arm positioning command in each of the four environments it needs additional conditions to be met, sometimes stronger and sometimes weaker. The LAAP weak contracts for the four contexts modelled[9] in CHESS are shown in Fig. 6. In particular, for the LAAPFlow not to be faulty when the wheel-loader is capable of high speed and has a static automatic positioning setpoint, the only additional assumption on the environment is that the ground speed sensor is not faulty, as captured by the *LAAP_fault_propagation_W1* contract. On the other hand, when in addition to the high speed capability, the setpoint is dynamic, then the LAAP component requires not only ground sensor to be fault free, but also the LAAP setpoint value to be correct (the *LAAP_fault_propagation_W2* contract). Conversely, when the vehicle is not capable of high-speed and when the setpoint is static, then LAAP has no additional constraints (the *LAAP_fault_propagation_W3* contract). Finally, when the vehicle is not capable of high speed and the setpoint is not static, then the only additional constraint is on the correctness of the LAAP setpoint value (the *LAAP_fault_propagation_W4* contract).

4.2 LACU Assurance

To assure SR1 related to the fault propagation contracts of LACU, we first validate the weak contracts and then perform a refinement check. The weak contract

[9] The contract type information is attached to the component and not shown here.

validity check identifies that only the *LAAP_fault_propagation_W1* weak contract is valid in the given LACU context. Hence, only that contract is selected in the LAAP component instance. The informal description of each of the contracts is added to the CHESS model, as well as relations to the requirements. Once all the information is saved in the CHESS model and the status of the contracts is updated, we can proceed to automatically generate argument-fragments for each component in the system. Figure 7 shows the screenshot of the Opencert interface presenting the result of the automatic instantiation of the contract-satisfaction argument pattern (Fig. 4) based on the information from the CHESS model of LACU. The list of automatically generated argument-fragment diagrams for each LACU component is in the top-left corner of the OpenCert interface.

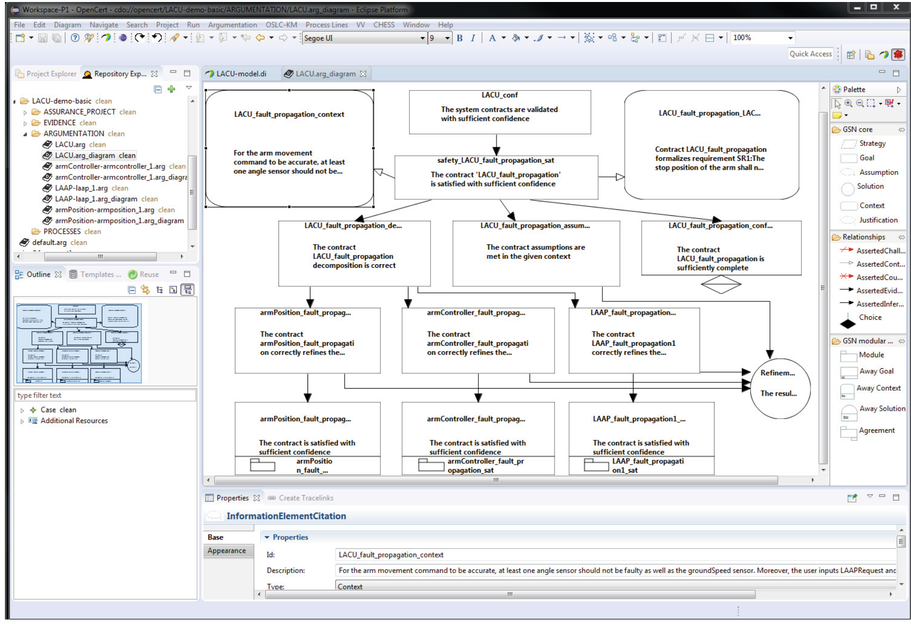


Fig. 7. Preview of the LACU automatically generated argument-fragment in OpenCert

4.3 Discussion

Contract-based design inherently supports reuse of components in form of contract implementations. But to fully understand the behaviour of a component and its safety implications, the context in which that behaviour is exhibited needs to be known. While component contracts represent a way of capturing a part of that context, additional context information is typically needed when dealing with safety-relevant components. In this case study, we have demonstrated how contract-based design can support reuse beyond implementations, to also include safety assurance artefacts related to those implementations.

Whether we perform the development of safety-relevant components in- or out-of-context, for reuse or just for a single system, different stakeholders are usually involved in the development process. For example, the expert performing the contract specification in a formal specification language such as OSS is not necessarily the same stakeholder as the one performing assurance modelling. Adoption of contracts just as any other formal specification is often hindered by the fact that not everyone can master a formal language [16]. Hence, for the stakeholder performing assurance modelling that should build upon different contract checks, we deem it is useful to accompany the contracts with additional information by the stakeholder that actually specified the contracts. Furthermore, a potential verifier assigned to verify certain behaviours of the component specified in a contract can directly associate that evidence with the contract and describe the results. While the goal of the LACU case study was not to evaluate the influence of our methodology on the quality of communication between different stakeholders in the development, during testing of the AMASS platform and collaborating on both modelling and assuring different systems, we could experience some of the communication benefits. Capturing all the safety assurance relevant information provided by different stakeholders in safety-critical system development in a traceable way has the potential of enhancing the collaboration between different stakeholders in building an assurance case. Moreover, by automatically generating parts of the argumentation, the safety engineer gets a head-start in assuring the system safety.

5 Conclusions and Future Work

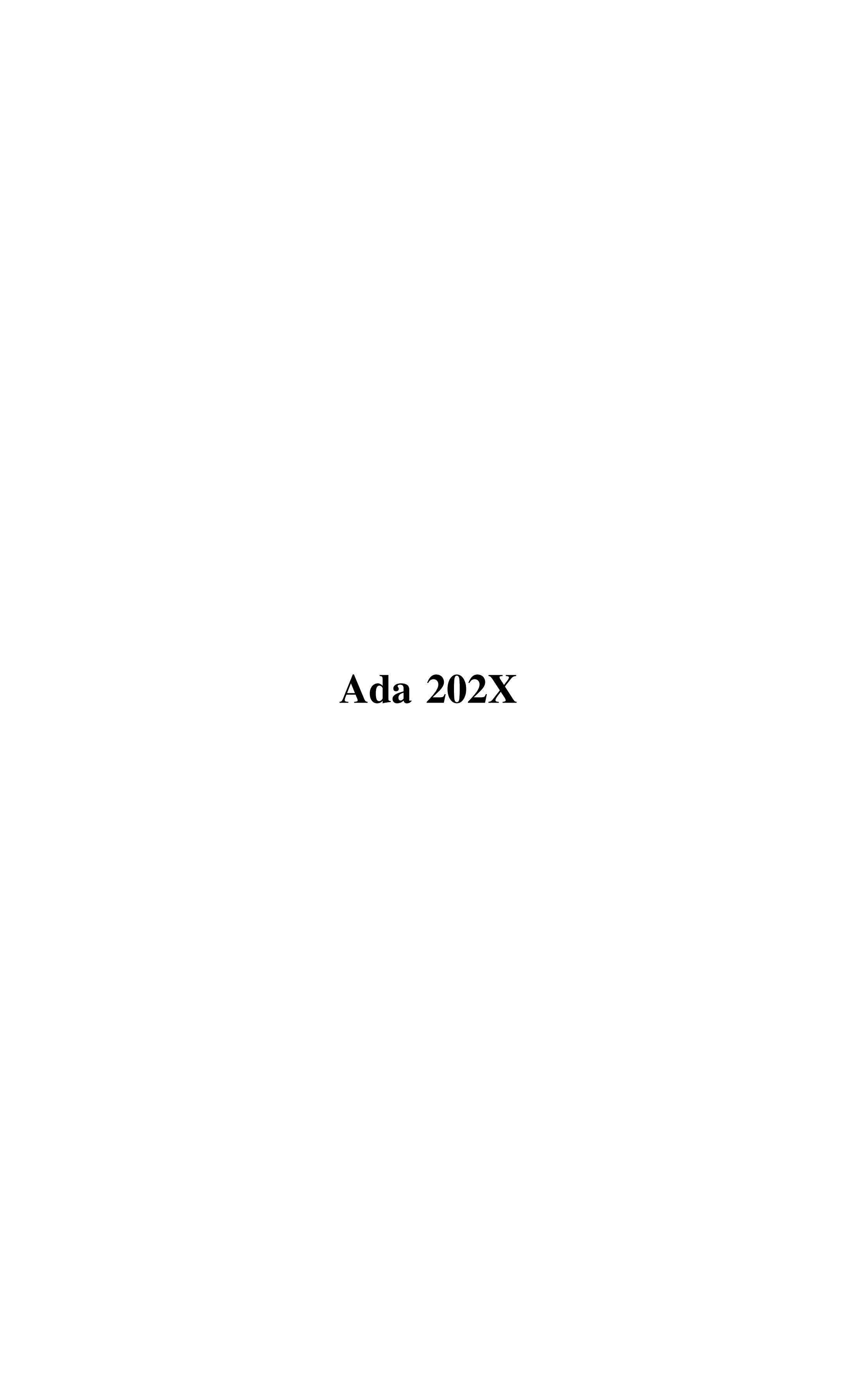
Reuse of safety-relevant components in safety-critical systems needs to cover more than just the implementation. Enriching contract-based design by associating contracts with assurance information enables us to reuse assurance artefacts together with the accompanying contract implementations. Furthermore, enabling variability modelling of the contract specifications in terms of strong and weak contracts allows us to provide greater support for reuse of components explicitly developed for reuse in different contexts. We have presented a tool support for the methodology by introducing system modelling with strong and weak contracts and their alignment with trace-based contract-based design. Furthermore, we have enabled automatic instantiation of assurance argument-fragments from the enriched system models. The presented tool support and the case study illustrate the feasibility of our contract-driven assurance and reuse methodology to assist in assuring requirements satisfaction and reusing assurance information.

To reap the full benefits of contract-driven assurance and reuse, further extensions to the AMASS platform are needed. Extending the underlying meta-model to connect the contracts with component failure behaviour could enable instantiation of many argument patterns that focus on failure behaviour. Furthermore, the traceability between the system and assurance modelling achieved through the contracts could be further enriched to support analysis and assurance of the interplay of multiple system concerns such as safety and security.

Acknowledgements. This work is supported by the EU and VINNOVA via the ECSEL Joint Undertaking projects AMASS (No 692474) and SafeCop (No 692529), as well as the Swedish Foundation for Strategic Research (SSF) via the FiC project.

References

1. Varnell-Sarjeant, J., Andrews, A.A., Stefik, A.: Comparing reuse strategies: an empirical evaluation of developer views. In: 8th International Workshop on Quality Oriented Reuse of Software, pp. 498–503. IEEE (2014)
2. Jézéquel, J.-M., Meyer, B.: Design by contract: the lessons of Ariane. IEEE Comput. **30**(1), 129–130 (1997)
3. International Organization for Standardization (ISO). ISO 26262: Road vehicles – Functional safety. ISO (2011)
4. Sljivo, I., Gallina, B., Carlson, J., Hansson, H., Puri, S.: A method to generate reusable safety case argument-fragments from compositional safety analysis. J. Syst. Softw. Spec. Issue Softw. Reuse **131**, 570–590 (2016)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for system design. Research report RR-8147, Inria, November 2012
6. Sljivo, I., Gallina, B., Carlson, J., Hansson, H.: Strong and weak contract formalism for third-party component reuse. In: 3rd International Workshop on Software Certification, pp. 359–364. IEEE, November 2013
7. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. **97**(3), 333–348 (2014)
8. Ratiu, D., Zeller, M., Killian, L.: Safety.Lab: model-based domain specific tooling for safety argumentation. In: Koornneef, F., van Gulijk, C. (eds.) SAFECOMP 2015. LNCS, vol. 9338, pp. 72–82. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24249-1_7
9. Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M.: Resolute: an assurance case language for architecture models. ACM SIGADA Ada Lett. **34**(3), 19–28 (2014)
10. Nair, S., Walkinshaw, N., Kelly, T., de la Vara, J.L.: An evidential reasoning approach for assessing confidence in safety evidence. In: 26th International Symposium on Software Reliability Engineering, pp. 541–552. IEEE (2015)
11. Denney, E., Pai, G.: Tool support for assurance case development. Autom. Softw. Eng., 1–65 (2017)
12. Goal Structuring Notation Working Group. GSN Community Standard V1.0. Origin Consulting (York) Limited (2011)
13. Object Management Group. SACM: Structured Assurance Case Metamodel. Technical report, V1.0 (2013). http://www.omg.org/spec/SACM
14. Sljivo, I., Gallina, B., Carlson, J., Hansson, H.: Generation of safety case argument-fragments from safety contracts. In: Bondavalli, A., Di Giandomenico, F. (eds.) SAFECOMP 2014. LNCS, vol. 8666, pp. 170–185. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10506-2_12
15. Object Management Group. MOFM2T: MOF Model to Text Transformation Language. Technical report, V1.0 (2008). http://www.omg.org/spec/MOFM2T
16. Filipovikj, P., Nyberg, M., Rodriguez-Navas, G.: Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In: 22nd International Requirements Engineering Conference. IEEE, August 2014



Ada 202X

Safe Dynamic Memory Management in Ada and SPARK

Maroua Maalej[1(✉)], Tucker Taft[2], and Yannick Moy[1]

[1] AdaCore, Paris, France
{maalej,moy}@adacore.com
[2] AdaCore, New York, USA
taft@adacore.com

Abstract. Handling memory in a correct and efficient way is a step toward safer, less complex, and higher performing software-intensive systems. However, languages used for critical software development such as Ada, which supports formal verification with its SPARK subset, face challenges regarding any use of pointers due to potential pointer aliasing. In this work, we introduce an extension to the Ada language, and to its SPARK subset, to provide pointer types ("access types" in Ada) that provide provably safe, automatic storage management without any asynchronous garbage collection, and without explicit deallocation by the user. Because the mechanism for these safe pointers relies on strict control of aliasing, it can be used in the SPARK subset for formal verification, including both information flow analysis and proof of safety and correctness properties. In this paper, we present this proposal (which has been submitted for inclusion in the next version of Ada), and explain how we are able to incorporate these pointers into formal analyses.

Keywords: Compilation · Safe pointers · Formal verification
Memory management

1 Introduction

Standard Ada supports safe use of pointers ("access types" in Ada) via strong type checking, but safety is guaranteed only for programs where there is no explicit deallocation of pointed-to objects – explicit deallocation is considered "unchecked" programming in Ada, meaning that the programmer is responsible for ensuring that the deallocation is not performed prematurely. Ada can provide automatic reclamation of the entire memory pool associated with a particular pointer type when the pointer type goes out of scope, but it does not automatically reclaim storage prior to that point. It is possible for a user to implement abstract data types that do some amount of automatic deallocation at the object level, but this requires additional programming, and typically has certain limitations. As part of its strong type checking, Ada also prevents dangling references to objects on the stack or the heap, by providing automatic compile-time checking of "accessibility" levels, which reflect the lifetimes of stack and heap objects.

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 37–52, 2018.
https://doi.org/10.1007/978-3-319-92432-8_3

Conversions between pointer types are restricted to ensure pointers never outlive the objects they designate. Values of a pointer type are by default initialized to null to prevent use of uninitialized pointers, and run-time checks verify that a null pointer is never dereferenced.

SPARK is a subset of the Ada programming language, targeted at the most safety- and security-critical applications. SPARK starts with the basic Ada features oriented toward building reliable and long-lived software, then adds restrictions that ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic assessment of conformance between a program specification and its implementation. The SPARK language and toolset for formal verification have been applied over many years to on-board aircraft systems, air traffic control systems, cryptographic systems, and rail systems [8,9].

As a consequence of our focus in SPARK on proof automation and usability, we have forbidden the use in SPARK of programming language features that either prevent automatic proof, or require extensive user effort in annotating the program. Pointer types are the main example of this for SPARK. SPARK supports many Ada features that can make up for the lack of pointers: by-reference parameter passing, the ability to specify the address of objects, and the support for arrays as first-class objects. On the other hand, pointers are sometimes desirable, which forces one to exclude from formal SPARK analysis the parts of a program that make use of pointers. While there are idioms that facilitate this isolation of pointers in non-SPARK parts of a program [2], it would be desirable to provide some level of support for pointers in SPARK.

In this work, we propose a restricted form of pointers for Ada that is safe enough to be included in the SPARK subset. As our main contribution, we show how to adapt the ideas underlying the safe pointers from permission-based languages like Rust [3] or ParaSail [13], to safely restrict the use of pointers in more traditional imperative languages like Ada. In Sect. 2, we provide rationale for the rules that we propose to include in the next version of Ada, which takes into account specifics of Ada such as by-copy/by-reference parameter passing and exception handling. In Sect. 3, we outline how these rules make it possible to formally verify SPARK programs using such pointers. Finally, we present related work and conclude.

2 A Proposal for Ownership Types in Ada

Pointers (access types) are essential to many complex Ada data structures, but they also have downsides, and can create various safety and security problems. When attempting to prove properties of a program, particularly those with multiple threads of control, the enemy is often the unknown "aliasing" of names introduced by access types and certain uses of (potentially) by-reference parameters. We say that two names may alias if they have the possibility to refer to overlapping memory regions. By *unknown* aliasing of names, we mean the case where two distinct names might refer to the same object, without the compiler

being aware of it. A rename introduces an alias, but not an "unknown alias," because the compiler is fully aware of such an alias. However, if a global variable is passed by reference as a parameter to a subprogram that also has direct access to the same global, the by-reference parameter and the global are now aliases within the subprogram, and the compiler generating its code has no way of knowing this, hence they are "unknown aliases." One approach is to always assume the worst, but that makes analyses much harder, and in some cases infeasible. Access types also introduce unknown aliasing, and in most cases, an analysis tool will not be sure whether the aliases exist, and will again have to make worst-case assumptions, which again may make any interesting proof infeasible.

The question that emerges in this context is: can we create a subset of access-type functionality that supports the creation of interesting data structures without bringing along the various problems associated with unknown aliasing? The notion of pointer "ownership" has emerged as one way to "tame" pointer problems, while preserving flexibility [12]. The goal is to allow a pattern of use of pointers that avoids dangling references as well as storage leaks, by providing safe, immediate, automatic reclamation of storage rather than relying on unchecked deallocation, while also not having to fall back on the time and space vagaries of garbage collection. As a side benefit, we can also get safer use of pointers in the context of parallelism. We propose the use of pointer ownership (as well as additional rules, detailed in [1], disallowing "aliasing" involving parameters) to provide safe, automatic, parallelism-friendly heap storage management while allowing the flexible construction of pointer-based data structures, such as trees, linked lists, hash tables, etc.

Although we took inspiration from Rust and ParaSail to produce this proposal, it is also different in many ways, due to the different objectives pursued in Ada and SPARK. Firstly, this proposal is designed to work with existing features of Ada such as by-copy/by-reference parameter passing and exception handling: raising an exception should not lead to memory leaks, and upon handling of a raised exception, objects should not be left in an inconsistent state. Secondly, this proposal relies on Ada's exiting mechanisms for avoiding uninitialized or dangling pointers: uninitialized and freed pointers should be set to null so that dereferencing such pointers results in a run-time error.

By relying on pointer ownership, we can ensure that pointer-based structures in SPARK can be supported while preserving SPARK's strict anti-aliasing parameter passing rules, thereby allowing the SPARK proof tools to prove the same range of safety and correctness properties, including freedom from data races, even in programs that use pointer-based structures in conjunction with concurrent and parallel programming constructs (see Sect. 3).

2.1 Ownership Types

In this section, we describe the *Ownership* aspect, a Boolean value that can be specified True for an Ada access type, with the effect that the compiler enforces an additional set of rules to ensure that there can be at most one writable access path to the data *designated* (i.e. pointed to) by an object of such an access type,

or alternatively one or more read-only access paths via such access objects, but never both concurrently. This is known as the Concurrent-Read-Exclusive-Write (CREW) access policy [11]. The CREW policy prevents multiple access via different objects to the same memory area whenever one of those access objects is modifying that area.

In addition to access types, Ownership can also be specified True for composite types, allowing for a record, an array, or even a private type with potentially multiple components that are access objects with Ownership True. Any object of a type with Ownership True is called an *ownership* object. In addition, we use the more general term *managed object* to refer to any object that is reachable by following ownership access objects. Here is the overall taxonomy:

- *Ownership* objects: objects of a type with Ownership aspect True, including:
  - *Owning access* objects: access-to-variable objects with Ownership aspect True;
  - *Observing access* objects: access-to-constant objects with Ownership aspect True;
  - *Composite ownership* objects: records and arrays with Ownership aspect True;
- Other *managed* objects: non-ownership objects that are pointed to by an owning or observing access object.

In the remainder, we presume that all objects in the code samples we present are managed objects; we refer the reader to [1] for further details on the Ownership aspect specification, and specific rules that apply to non-ownership managed objects. Note also that some of the Ownership rules will be expressed in terms of "names" rather than "objects," since objects do not really exist at compile-time, when these rules are intended to be enforced.

In the next section, we will focus on owning and observing access objects, which as parameters are passed by copy in Ada, before we consider the situation of composite ownership objects, for which we require pass-by-reference. As it turns out, when worrying about the number of ways one can reach an object, passing a parameter by copy or by reference can make a difference, particularly in the context of propagating and handling exceptions.

2.2 Ownership for Access Objects

To ensure safe memory management, the basic rule is that at most one object that gives update access may be used at one time to refer to a designated object, and while such an updater exists, access via any other object is disallowed. There might be multiple access objects that designate the same object (or some part of it) at certain times, but all of them must provide only read access.

The manipulation of ownership access objects in the program is limited to the following three kinds of operations:

- *Moving*: An assignment operation that leads to moving the value of one access object into another, leaving behind a null;

- *Borrowing*: The declaration of a short-term read-write reference by copying an existing access object, borrowing its value for the lifetime of the *borrower*.
- *Observing*: The declaration of a short-term object that gives read-only access, by copying an existing access object, observing the object(s) reachable from it for the lifetime of the *observer*.

Given an access object, it should have one of three possible states at any point in its scope:

- *Unrestricted*: the object may be dereferenced and used to read or update the designated object;
- *Observed*: the object may be used for read-only access to all or part of the designated object, or part of some object directly or indirectly reachable via a chain of owning access objects from the designated object;
- *Borrowed*: the object's ownership has been temporarily transferred to another object, and while in such a state the original access object is not usable for reading or updating the designated object (nor for any other purpose).

Moving Access Values. The *move* operation involves a complete transfer of the ownership from the right hand side to the left hand side in an assignment operation, where both left- and right-hand-side objects are owning access variables in the unrestricted state. After the assignment, the right-hand side gets set to null, and the newly assigned object becomes the (unrestricted) owner. By setting the original access object to null, any name that starts with a dereference of that original access object is effectively "destroyed"; even if an exception is raised before we explicitly assign a new value to the right-hand side, there is no danger a handler for the exception will be able to dereference the old value of the right-hand side. Being able to use the old value to reach the designated object would, at a minimum, violate our CREW policy, and could cause havoc if the designated object had been deallocated after the move, but prior to the exception being raised.

In addition to setting the right-hand side to null, a move also finalizes and deallocates the object, if any, designated by the left-hand side prior to the assignment. This automatic deallocation means memory is reclaimed as soon as it is no longer accessible, thereby preventing memory leaks, without the need for an asynchronous garbage collector. This is safe to do, because a move requires the left-hand side to be in the *unrestricted* state, meaning that it is the only access object pointing to the object about to be deallocated.

In addition to considering certain assignment statements to be *moves*, we also consider the assignments inherent in passing by copy an `out` or `in out` parameter to be moves, as well as returning a value from a function. Updating a subcomponent of a composite object is also considered a move, but we consider these in the section focused on composite types (see below).

Figure 1 illustrates an example of "move" operations. Objects `X` and `Y` are access-to-variable objects of the named type `Int_Ptr`. At a `Swap` procedure call site, the actual parameter `X`, which is required to be in the unrestricted state, is

copied in to the formal X_Param. The *ownership* of X.all (X.all is Ada's notation for dereferencing X – X.all denotes the object *designated* by X) is similarly moved from X to X_Param. This requires setting X to null until the subprogram returns (to ensure safety in the presence of an exception), at which point the final value of X_Param is *copied back* to X. Variable X then reasserts its ownership over X.all. Similar state transitions apply for Y and Y_Param. At lines ℓ_4, ℓ_6, and ℓ_7, we have additional move operations, which consist of moving, respectively, the objects X_Param, Y_Param, and Tmp. Thanks to our move-related rules, even such a straightforward implementation of the Swap procedure for access types is nevertheless guaranteed to be alias safe while Swap is executing, both from the *caller* perspective and from *inside* Swap itself, since the ownership is transferred as part of each access-to-variable object assignment.

```
1  type Int_Ptr is access integer;
2
3  procedure Swap(X_Param, Y_Param : in out Int_Ptr) is
4    Tmp : Int_Ptr := X_Param;
5  begin
6    X_Param := Y_Param;
7    Y_Param := Tmp;
8  end Swap;
9
10 X : Int_Ptr := new Integer'(7);
11 Y : Int_Ptr := new Integer'(11);
12
13 Swap(X, Y);
```

Fig. 1. Example of moving the ownership of an object.

Borrowing Access Values. We say that an access value has been "borrowed" if that value has been copied into a short-lived (owning) access-to-variable object. A borrowing operation is a *temporary* transfer of the ownership of the said borrowed object until the end of the scope of the borrower. We want the original access object to still designate the same object until the borrower goes away. As a result, while an access object is in the borrowed state, its value may not be changed; furthermore, to preserve our CREW policy, we disallow using or copying it again until the current borrower goes away; in the borrowed state, the original access object is completely "dead" – it cannot be read nor be the target of an assignment. Furthermore, borrowing applies recursively down the tree rooted at the original access object, meaning that at the point where a name is borrowed, every name with that name as a prefix, is similarly borrowed.

The assignment operations that are considered borrowing are those that *initialize* a stand-alone object of an *anonymous* access-to-variable type, or a constant or an in parameter of a (named or anonymous) access-to-variable type. We also consider as borrowing passing an object of a composite ownership type as a parameter of mode out or in out – see Sect. 2.3 below. The code snippet of Fig. 2 is a simple example of borrowing. X and Y are both access-to-variable

objects. We want to swap the objects *designated* by the two pointers (their "contents") using the `Swap_Contents` procedure. To that end, we declare `X_Param` and `Y_Param` as formal parameters of mode in. Objects `X` and `Y` become borrowed in the caller, and inside `Swap_Contents X_Param` and `Y_Param` are the borrowers, in the unrestricted state. This state allows reading and updating via these formal parameters, which enables swapping the value of their designated objects. Note that we allow an in parameter or a constant of an owning access type to provide read/write access to its designated object to accommodate existing Ada practice in the use of such "constant" access-to-variable values to nevertheless update their designated objects.

```
1  type Int_Ptr is access integer;
2
3  procedure Swap_Contents (X_Param, Y_Param : in Int_Ptr) is
4    Tmp : integer := X_Param.all;
5  begin
6    X_Param.all := Y_Param.all;
7    Y_Param.all := Tmp;
8  end Swap_Contents;
9
10 X : Int_Ptr := new Integer'(13);
11 Y : Int_Ptr := new Integer'(17);
12
13 Swap_Contents(X, Y);
```

Fig. 2. Example of borrowing via in parameters.

Observing Access Values. We say an access-to-variable object is "observed" when its value has been copied into an "observer," and both the original access object and the copy, starting at that point, can only be used for read access to the designated object. The original object remains in the observed state until the end of the scope of the observer. While being observed, neither the observed object nor the observer is allowed to be moved or borrowed. The original access object cannot be used as the target of an assignment since we need the observed object to continue to designate the same object as long as any observers exist. As with borrowing, observing applies recursively down the tree rooted at the original access object, meaning that at the point where a name is observed, every name with that name as a prefix, is similarly observed.

We consider as observing the assignment operations used to *initialize* standalone objects of an anonymous access-to-*constant* type, as well as in parameters of such a type. In the code snippet of Fig. 3, `X_Param` and `Y_Param` are access-to-constant objects of an anonymous type. Since the assignment of the value of `X` to `X_Param` as well as to `Y_Param` are part of the initialization of the target objects, this initiates the observing, and while `X_Param` and `Y_Param` exist they provide read-only access. Note that this allows us to call the function `Sum` using `X` as a first and second parameter – upon the first occurrence of `X` it enters the observed state, but we can still observe it further.

```
1 type Int_Ptr is access integer;
2
3 function Sum (X_Param, Y_Param : access constant Integer) return
4     Integer is
5 begin
6   return X_Param.all + Y_Param.all;
7 end Sum;
8
9 X : Int_Ptr := new Integer'(42);
10
11 Y : constant Integer := Sum (X, X);
```

Fig. 3. Example of observing via access-to-constant parameters.

Preventing Read-Write Aliasing. We have seen that the observing rules allow multiple access objects to observe the same designated object. In the scope of these objects, the original object is in the observed state; its designated object cannot be written, so there is no read-write aliasing problem here.

We have seen that after borrowing an object, its name allows neither reading nor updating until the borrowing ends. For example, this prevents a call to `Swap_Contents(X, X)`, as borrowing X via parameter `X_Param` makes it illegal to borrow it again via parameter `Y_Param`. The actual order of evaluation does not matter here, as any other order would also be illegal.

We have also seen that after moving an object, its value is set to null, which prevents accessing the designated object again through the original name. This rule by itself does not prevent a call to `Swap(X, X)`, but moving `X` into parameter `X_Param` makes `X` null, so that if it is then moved into parameter `Y_Param`, the value null will be passed, ensuring that a run-time check will prevent read-write aliasing. In fact, in current Ada, passing the same object twice in the same call as an `out` or `in out` parameter is illegal, so this existing Ada rule will catch simple cases such as this at compile time. Furthermore, as part of our proposed extension to Ada, an additional restriction `No_Parameter_Aliasing` is defined, which prevents at compile time the more complex cases as well. We refer the reader to [1] for further details on the `No_Parameter_Aliasing` restriction.

2.3 Extension to Composite Types

The rules presented previously for access objects are extended in natural ways to composite ownership objects (records or arrays with owning access objects as subcomponents) to enforce the Concurrent-Reads-Exclusive-Write principle.

Moving Composite Values. As with access objects, the composite move operation is a complete transfer of the ownership from the right hand side composite object to the left hand side object as part of an assignment operation. And as with access objects, a composite object to be moved must be in the unrestricted state before the assignment. The rules that apply for moving an access object are applied here to each access subcomponent of the composite type: access subcomponents of the moved objects are set to null after being copied, and to avoid

memory leaks, if the prior value of the subcomponent in the target composite object is different from the new value, the object designated by this prior value is finalized and its storage deallocated.

As before, we consider as a move each assignment operation for a composite ownership type where the target is a variable (or the "return object" of a function), but this time we do not consider passing of `out` or `in out` parameters to be moves, because for composite ownership objects, parameters are passed by reference and no true copying is occurring. Composite parameter passing is described further below. In the code snippet of Fig. 4, `Rec` is a record with components of an owning access type. The move operation occurs at line ℓ_9 where `R` is moved to `S`, which involves moving `R.X` into `S.X` and moving `R.Y` into `S.Y`. As a result, the objects originally designated by `S.X` and `S.Y` are deallocated and `R.X` and `R.Y` end up null after the assignment.

```
1 type Int_Ptr is access Integer;
2 type Rec is record
3    X, Y : Int_Ptr;
4 end record;
5
6 R : Rec := (...);
7 S : Rec := (...);
8
9 S := R;
```

Fig. 4. Example of moving a composite object.

Borrowing Composite Values. Borrowing composite ownership objects occurs when passing such an object as an `out` or `in out` parameter, consistent with these composite ownership objects being passed by reference. Note how this differs from `out` or `in out` parameters of an access type, which are passed by copy and are thus considered as being *moved* as part of parameter passing. Ada normally allows composite objects to be passed either by copy or by reference, but for ownership composite types, we specify that they must always be passed by reference, to avoid having two different sets of rules for composite objects that would depend on whether the type is passed by copy or by reference.

In the code snippet of Fig. 5, procedure `Swap_Rec` has an `in out` formal parameter `R` of a record type. At the point of call to `Swap_Rec`, the actual parameter name `R1` becomes borrowed until returning from `Swap_Rec`, with the borrower being the formal parameter name `R`. Inside `Swap_Rec`, the formal parameter `R` is initially in the unrestricted state, hence its components `R.X` and `R.Y` can be successively moved in and out through the call to `Swap`, and then borrowed through the call to `Swap_Contents`. Note that subcomponents of a composite type can be individually moved and borrowed, without impacting the state of other non-overlapping subcomponents of the same composite object. We refer the reader to [1] for further details.

```
1 procedure Swap_Rec (R : in out Rec) is -- R1 is borrowed
2 begin
3   Swap (R.X, R.Y);
4   Swap_Contents (R.X, R.Y);
5 end Swap_Rec;
6
7 R1 : Rec := (...);
8
9 Swap_Rec (R1);
```

Fig. 5. Example of borrowing via a composite in out parameter.

Observing Composite Values. Observing composite ownership objects occurs when passing such an object as an in parameter, or initializing a stand-alone constant object of such a type.

In the code snippet of Fig. 6, procedure Sum_Rec has an in formal parameter R of a record type. At the point of call to Sum_Rec, the actual parameter name R1 becomes observed, with the formal parameter R as the observer, until returning from Sum_Rec. Inside Sum_Rec, the formal parameter R is initially in an observed state, hence its components R.X and R.Y can only be read (observed) through the call to Sum.

```
1 function Sum_Rec (R : in Rec) return Integer is
2 begin
3   return Sum (R.X, R.Y);
4 end Sum_Rec;
5
6 R1 : Rec := (...);
7
8 Y : Integer := Sum_Rec (R1);
```

Fig. 6. Example of read only access to an object of a composite type.

Traversing Data Structures with Local Variables. In the rules for borrowing access values (Sect. 2.2), initializing a stand-alone object of an anonymous access-to-variable type corresponds to *borrowing* the access object being copied. Similarly, in the rules for observing access values (Sect. 2.2), initializing a stand-alone object of an anonymous access-to-constant type corresponds to *observing* the object being copied. Without these special cases, such initializations might be treated as moves, which would not allow for a non-destructive traversal of a recursive data structure, since every assignment to such a "handle" would deallocate its prior designated object and set to null the object that was moved. Hence, such an object of an anonymous access type acts as a kind of short-term "handle" on the tree of objects rooted at the original access object.

We also allow certain kinds of updates to such "handles," in order to allow traversing the data structure by changing where the handle points. In the borrowing case (for an access-to-*variable* object), we allow the borrower to be updated to point to an object within the tree rooted at the prior value of the borrower;

this is not considered a new borrowing action, as the existing borrower remains the only object providing any read or write access to the subtree rooted at its original value. By limiting the initial borrowing to the initialization of a new stand-alone object, we ensure that borrowing lasts only as long as the lifetime of the "handle." If we allowed any given assignment statement to initiate a new borrowing action, tracking when such borrowing would end might require complex data-flow analysis, potentially across conditional and iterative paths in the program. Somewhat less stringent restrictions are applied when updating an observer – the observer may be updated to point to an already observed object with a compatible scope. Again, doing otherwise might require complex data-flow analysis to determine the extent of the observing action.

In the code snippet of Fig. 7, local variable `Walker` is a stand-alone object of an anonymous access-to-*constant* type, which allows traversing the input binary search tree, so as to find the maximal value (obtained by searching for the rightmost leaf of the tree). After initializing `Walker` with the value of parameter `T`, `T` becomes observed, and `Walker` starts in the observed state (thus preventing updates to `T.all` through `Walker`). The data structure traversal is performed by the instruction of line ℓ_{16}.

```
1  type Rec;
2  type Tree is access Rec;
3  type Rec is record
4    Data : Natural;
5    Left, Right : Tree;
6  end record;
7
8  function Max (T : in Tree) return Integer is
9    Walker : access constant Rec := T; -- Walker observes T
10   Max_Value : Natural := 0;
11 begin
12   while Walker /= null loop
13     if Walker.Data > Max_Value then
14       Max_Value := Walker.Data;
15     end if;
16     Walker := Walker.Right; -- assignment to Walker
17   end loop;
18 return Max_Value;
19 end Max;
```

Fig. 7. Example of traversing a data structure with read-only access: Max on a Binary Search Tree.

In the code snippet of Fig. 8, local variable `Walker` is a stand-alone object of an anonymous access-to-*variable* type, which allows traversing the input binary search tree to insert the input value `V` at the correct leaf position (obtained by searching for the branch where this value would be stored, if it were already present). After initializing `Walker` with a copy of the value of parameter `T`, `T` becomes borrowed, and `Walker` starts its life in the unrestricted state (thus

allowing updates via `Walker` to the tree pointed to by `T`). The data structure traversal is performed by the instruction of line ℓ_8 and ℓ_{15}. Insertion in the tree is performed at lines ℓ_{10} and ℓ_{17}.

```
1  procedure Insert (T : in Tree; V : Natural) is
2    Walker : access Rec := T;
3  begin
4    loop
5      if V < Walker.Data then
6        if Walker.Left /= null then
7          Walker := Walker.Left;
8        else
9          Walker.Left := Build_Leaf(V);
10         exit;
11       end if;
12     elsif V > Walker.Data then
13       if Walker.Right /= null then
14         Walker := Walker.Right;
15       else
16         Walker.Right := Build_Leaf(V);
17         exit;
18       end if;
19     end if;
20   end loop;
21 end Insert;
```

Fig. 8. Example of traversing a data structure with read and update access: Insert into a Binary Search Tree. `Build_Leaf(V)` creates a node with `Data` = `V`, and `Left`, and `Right` components both null.

3 Formal Verification with Ownership Types in SPARK

The existing SPARK restrictions imposed on its current subset of Ada ensure that an assignment to one variable cannot change the value of some other visible variable. This property is essential to allow sound modular static analysis, where each subprogram can be analyzed independently while detecting all possible violations of the kinds targeted by the analysis.

This is currently enforced by forbidding all use of access types in SPARK, and by restricting aliasing between parameters and global variables so that only benign aliasing is permitted (i.e. aliasing that does not cause interference). The aliasing restrictions are as follows:

- Two output parameters should never be aliased.
- An input and an output parameter should not be aliased, unless the input parameter is always passed by copy.
- An output parameter should never be aliased with a global variable referenced by the subprogram.
- An input parameter should not be aliased with a global variable updated by the subprogram, unless the input parameter is always passed by copy.

To understand why aliasing matters in SPARK, consider procedure `Add_One` in Fig. 9. If `X_Param` and `Y_Param` are not aliased, then the result of calling `Add_One` on actual parameters `X` and `Y` will increase their contents by one. If `X` and `Y` are aliased, then calling `Add_One` on `X` and `Y` will increment the underlying content by two.

```
1 procedure Add_One(X_Param, Y_Param : in Int_Ptr) is
2 begin
3   X_Param.all := X_Param.all + 1;
4   Y_Param.all := Y_Param.all + 1;
5 end Add_One;
```

Fig. 9. A simple procedure where aliasing would create problems in SPARK.

If SPARK ignored aliasing, it would conclude that procedure `Add_One` always increments by exactly one the content of each of its parameters `X_Param` and `Y_Param`. In particular, it could prove the following postcondition on the procedure.

```
1 procedure Add_One(X_Param, Y_Param : in Int_Ptr) with
2   Post => X_Param.all = X_Param.all'Old + 1
3     and Y_Param.all = Y_Param.all'Old + 1;
```

Indeed, by presuming that the assignment to `X_Param.all` on line ℓ_2 does not influence the value of `Y_Param.all`, proof would be able to derive that the values `Y_Param.all` has been incremented by 1. Similarly, flow analysis could derive wrong data dependencies if possible aliasing is not taken into account.

This wrong postcondition would allow a proof that an incorrect assertion is satisfied in the code snippet of Fig. 10, while in fact it fails at run time. Thus, normal Ada pointers could not be treated like any other component in SPARK, given the possibility for aliasing. But the rules we have described for ownership objects precisely prevent aliasing when one of the objects can be written. This is analogous to the rules in SPARK for preventing aliasing between by-reference parameters, and these rules allow SPARK to treat such pointers like other components.

```
1   X : Int_Ptr := new Integer'(1);
2   (...)
3
4   Add_One (X, X);
5   pragma Assert (Y.all = 2);  --  incorrect assertion
```

Fig. 10. Example of proof of an incorrect assertion due to the presence of aliasing in SPARK.

In the case of `Add_One`, this means that SPARK analysis will be able to conclude that the postcondition above is satisfied by the implementation of `Add_One`.

But unsafe calls such as `Add_One(X, X)` will be rejected both by compilation and analysis.

The SPARK tools also provide detection of potential data races in programs that use concurrent and parallel programming constructs. This detection depends on the strict anti-aliasing conditions on parameters, and provides a sound assurance that no two threads concurrently manipulate the same data, if either has update access. This matches the CREW condition imposed on access objects through the proposed ownership rules, and means that the SPARK tools can handle pointer-based structures that obey these rules, in the same way it already handles record- and array-based structures, enabling provably safe concurrent and parallel programming in SPARK even when enhanced with this more flexible data structuring capability.

4 Related Work

C-like languages are mostly based on pointers and often sacrifice safety for performance purposes. To overcome safety shortcoming and manage the storage of a pointer, C++ introduces the notion of *unique pointers*. An object defined as a `unique_ptr` has the ability to take ownership of an object. It becomes responsible for its deletion at some point. Although these rules help provide greater language safety, the unique pointer concept is limited because it prohibits pointer arithmetic and copy assignments.

Separation logic [10] is an extension of Hoare-Floyd logic that allows reasoning about pointers. In general, it is not well integrated with deductive verification, and, in particular, is not supported by most SMT provers.

Dafny associates each object with its *dynamic frame*, the set of pointers that it owns [7]. This dynamic version of Ownership is enforced by modeling the Ownership of pointers in logic, generating verification conditions to detect violations of the single-owner model, and proving them using SMT provers. In Spec#, Ownership is similarly enforced by proof, to detect violations of the so-called Boogie methodology [4].

The inspiration for much of our work springs from the systems programming languages Cyclone [5], Rust [3], and ParaSail [13], which achieve absence of harmful aliasing by enforcing an Ownership type system on the memory pointed to by objects. Rust and ParaSail are recent programming languages providing safe systems programming, with a focus on memory safety for concurrent programs. Rust and ParaSail also deal with the lifetime of allocated memory, while preventing dangling pointer references.

The most closely related work to ours springs from Jaloyan *et al.* [6] anti-aliasing rules. In [6], access-to-variable objects and composite objects with access subcomponent objects are considered as *deep* variables and their ownership states are transferred in the same way when used to call subprograms. Actual deep parameters are considered as borrowed and the durations of borrows are only limited to the duration of procedure calls. It turns out that their rules do not allow traversing a linked data structure with read/write permission, or

even traversing with read-only permission. In our work, the distinction between stand-alone access objects and composite ones and moving or observing composite objects instead of borrowing them has allowed us to support safely traversing a data structure for read or update.

In our work, we use a permission-based mechanism for detecting potentially harmful aliasing, in order to make the presence of pointers transparent for automated provers. Our approach does not require additional user annotations required in some of the previously mentioned techniques. We instead rely on the existing distinctions in Ada between `in` and `in out` parameters, and between access-to-variables and access-to-constants. We thus achieve high automation and usability, which was one of our goals in supporting pointers in SPARK.

5 Conclusion

We have presented an extension to the Ada language to provide pointer types ("access types" in Ada) that provide provably safe, automatic storage management without any asynchronous garbage collection, and without explicit deallocation by the user. Although we took inspiration from Rust and ParaSail, the extension we propose differs so as to work well with existing features of Ada such as by-copy/by-reference parameter passing and exception handling, and because we rely on the existing mechanisms in Ada for preventing access to uninitialized pointers and freed memory.

This extension relies on the notion of ownership, where only one access object can provide update access to the designated object at any given time. Ownership of a designated object can be *moved* to another object through assignment, which deallocates the object previously designated by the target of the assignment and leaves the source of the assignment null. Ownership can also be *borrowed* giving a short-term borrower read-write access to the designated object. Finally, the value of a designated object can be *observed* by multiple read-only observers, with limited lifetimes. Collectively, these mechanisms enforce a principle of Concurrent-Reads-Exclusive-Write.

Because the mechanism for these safe pointers relies on a strict control of aliasing, they can be used in the SPARK subset for formal verification, which includes both analysis of flows and proof of properties, including in the presence of multiple threads of control.

This proposal has been formalized as Ada Issue [1] for inclusion in a future version of Ada. We have also implemented a prototype of these permission rules in the GNAT/GCC compiler for Ada developed at AdaCore. Our implementation successfully proves the safety of all programs presented in this article.

Acknowledgements. We thank the anonymous reviewers for their remarks, and Georges-Axel Jaloyan for his initial work on the design, formalization and implementation of these ownership rules for Ada and SPARK.

References

1. AdaCore: Access value ownership and parameter aliasing (2018). http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0240-1.txt
2. AdaCore, Thales: Implementation guidance for the adoption of SPARK (2017). https://www.adacore.com/books/implementation-guidance-spark
3. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamaric, Z., Ryzhyk, L.: System programming in rust: Beyond safety. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, 8–10 May 2017, pp. 156–161 (2017)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
5. Grossman, D., Morrisett, J.G., Jim, T., Hicks, M.W., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, 17–19 June 2002, pp. 282–293 (2002)
6. Jaloyan, G.A., Moy, Y., Paskevich, A.: Borrowing safe pointers from rust in spark. In: International Conference on Computer-Aided Verification - 29th International Conference, Heidelberg, Germany (2018, in submission)
7. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
8. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
9. O'Neill, I.: SPARK - a language and tool-set for high-integrity software development. In: Industrial Use of Formal Methods: Formal Verification (2012)
10. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of th 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, pp. 55–74 (2002)
11. Sant, P.M.: Concurrent read, exclusive write. In: Pieterse, V., Black, P.E. (eds.) Dictionary of Algorithms and Darta Structures (2004). https://www.nist.gov/dads/HTML/concurrentReadExcluWrt.html
12. Svoboda, D., Wrage, L.: Pointer ownership model. In: Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS 2014), 6–9 Jan 2014, Waikoloa, Hawaii, pp. 5090–5099 (2014)
13. Taft, S.T.: Multicore programming in ParaSail. In: Romanovsky, A., Vardanega, T. (eds.) Ada-Europe 2011. LNCS, vol. 6652, pp. 196–200. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21338-0_16

Safe Non-blocking Synchronization in Ada2x

Johann Blieberger[1](✉) and Bernd Burgstaller[2](✉)

[1] Institute of Computer Engineering, Automation Systems Group, TU Wien, Vienna, Austria
blieb@auto.tuwien.ac.at

[2] Department of Computer Science, Yonsei University, Seoul, Korea
bburg@yonsei.ac.kr

Abstract. 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 keep holding a lock that blocks other tasks from accessing shared data. With non-blocking synchronization, the drawbacks of locks are avoided by synchronizing access to shared data by atomic read-modify-write operations.

To incorporate non-blocking synchronization in Ada 202x, programmers must be able to reason about the behavior and performance of tasks in the absence of protected objects and rendezvous. We therefore extend Ada's memory model by synchronized types, which support the expression of memory ordering operations at a sufficient level of detail. To mitigate the complexity associated with non-blocking synchronization, we propose concurrent objects as a novel high-level language construct. Entities of a concurrent object execute in parallel, due to a fine-grained, optimistic synchronization mechanism. Synchronization is framed by the semantics of concurrent entry execution. The programmer is only required to label shared data accesses in the code of concurrent entries. Labels constitute memory-ordering operations expressed through aspects and attributes. To the best of our knowledge, this is the first approach to provide a non-blocking synchronization construct as a first-class citizen of a high-level programming language. We illustrate the use of concurrent objects by several examples.

1 Introduction

Mutual exclusion locks are the most common technique to synchronize multiple tasks to access shared data. Ada's protected objects (POs) implement the monitor-lock concept [12]. Method-level locking requires a task to acquire an exclusive lock to execute a PO's entry or procedure. (Protected functions allow concurrent read-access in the style of a readers–writers lock [11].) Entries and procedures of a PO thus effectively execute one after another, which makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO. Informally, sequential consistency ensures that method

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 53–69, 2018.
https://doi.org/10.1007/978-3-319-92432-8_4

calls act as if they occurred in a sequential, total order that is consistent with the program order of each participating task. I.e., for any concurrent execution, the method calls to POs can be ordered sequentially such that they (1) are consistent with program order, and (2) meet each PO's specification (pre-condition, side-effect, post-condition) [11].

Although the sequential consistency semantics of mutual exclusion locks facilitate reasoning about programs, they nevertheless introduce potential concurrency bugs such as dead-lock, live-lock and priority inversion. The mutual-exclusion property of (highly-contended) locks stands in the way to scalability of parallel programs on many-core architectures [16]. Locks do not allow progress guarantees, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data.

Given the disadvantages of mutual exclusion locks, it is thus 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* (RMW) operations. In the absence of mutual exclusion locks, the possibility of task-failure inside a critical section is eliminated, because critical sections are reduced to single atomic operations. These atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA). 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 [11].

```
1 -- Initial values:
2 Flag := False;
3 Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

(a)

```
1 Data : Integer with Volatile;
2 Flag : Boolean with Atomic;
```

(b)

Fig. 1. (a) Producer-consumer synchronization in pseudo-code: Task 1 writes the `Data` variable and then signals Task 2 by setting the `Flag` variable. Task 2 is spinning on the `Flag` variable (lines 2 to 5) and then reads the `Data` variable. (b) Labeling to enforce sequential consistency in Ada 2012.

Non-blocking synchronization techniques are notoriously difficult to implement and the design of non-blocking data structures is an area of active research. To enable non-blocking synchronization, a programming language must provide a strict memory model. The purpose of a memory model is to define the set of values a read operation in a program is allowed to return [1].

To motivate the need for a strict memory model, consider the producer-consumer synchronization example in Fig. 1(a) (adopted from [18] and [4]). The

programmer's intention is to communicate the value of variable `Data` from Task 1 to Task 2. Without explicitly requesting a sequentially consistent execution, a compiler or CPU may break the programmer's intended synchronization via the `Flag` variable by re-ordering memory operations that will result in reading `R2 = 0` in Line 6 of Task 2. (E.g., a store–store re-ordering of the assignments in lines 2 and 3 of Task 1 will allow this result.) In Ada 2012, such re-orderings can be ruled out by labeling variables `Data` and `Flag` by aspect `volatile`. The corresponding variable declarations are depicted in Fig. 1(b). (Note that by [8, C.6§8/3] aspect `atomic` implies aspect `volatile`, but not vice versa.)

The intention for volatile variables in Ada 2012 was to guarantee that all tasks agree on the same order of updates [8, C.6§16/3]. Updates of volatile variables are thus required to be sequentially consistent, in the sense of Lamport's definition [14]: "*With sequential consistency (SC), any execution has a total order over all memory writes and reads, with each read reading from the most recent write to the same location*".

However, the Ada 2012 aspect `volatile` has the following shortcomings:

1. Ensuring SC for multiple tasks without atomic access is impossible. Non-atomic volatile variables therefore should not be provided by the language. Otherwise, the responsibility shifts from the programming language implementation to the programmer to ensure SC by pairing an atomic (implied volatile) variable with each non-atomic volatile variable (see, e.g., Fig. 1(b) and [17] for examples). (Note that a programming language implementation may ensure atomicity by a mutual exclusion lock if no hardware primitives for atomic access to a particular type of shared data are available.)
2. Requiring SC on all shared variables is costly in terms of performance on contemporary multi-core CPUs. In Fig. 1, performance can be improved by allowing a less strict memory order for variable `Data` (to be addressed in Sect. 2).
3. Although Ada provides the highly abstract PO monitor-construct for blocking synchronization, there is currently no programming primitive available to match this abstraction level for non-blocking synchronization.

Contemporary CPU architectures relax SC for the sake of performance [2,9, 18]. It is a challenge for programming language designers to provide safe, efficient and user-friendly non-blocking synchronization features. The original memory model for Java contained problems and had to be revised [15]. It was later found to be unsound with standard compiler optimizations [19]. The C++11 standard (cf. [13,20]) has already specified a strict memory model for concurrent and parallel computing. We think that C++11 was not entirely successful both in terms of safety and in terms of being user-friendly. In contrast, we are convinced that these challenges can be met in the upcoming Ada 202x standard.

It has been felt since Ada 95 that it might be advantageous to have language support for synchronization based on atomic variables. For example, we cite [10, C.1]: *"A need to access specific machine instructions arises sometimes from other considerations as well. Examples include instructions that perform compound*

operations atomically on shared memory, such as test-and-set and compare-and-swap, and instructions that provide high-level operations, such as translate-and-test and vector arithmetic."

Ada is already well-positioned to provide a strict memory model in conjunction with support for non-blocking synchronization, because it provides tasks as first-class citizens. This rules out inconsistencies that may result from thread-functionality provided through libraries [6].

To provide safe and efficient non-blocking synchronization for Ada 202x, this paper makes the following contributions:

1. We extend Ada's memory model by introducing synchronized types, which allow the expression of memory ordering operations consistently and at a sufficient level of detail. Memory ordering operations are expressed through aspects and attributes. Language support for spin loop synchronization via synchronized variables is proposed.
2. We propose *concurrent objects* (COs) as a high-level language construct to express non-blocking synchronization. COs are meant to encapsulate the intricacies of non-blocking synchronization as POs do for blocking synchronization. Contrary to POs, the entries and procedures of COs execute in parallel, due to a fine-grained, optimistic synchronization mechanism.
3. We provide an alternative, low-level API on synchronized types, which provides programmers with full control over the implementation of non-blocking synchronization semantics. Our main purpose with the low-level API is to provoke a discussion on the trade-off between abstraction versus flexibility.
4. We illustrate the use of concurrent objects and the alternative, low-level API by several examples.

The remainder of this paper is organized as follows. We summarize the state-of-the-art on memory models and introduce synchronized variables in Sect. 2. We introduce aspects and attributes for specifying memory ordering operations in Sect. 3. We specify concurrent objects in Sect. 4 and discuss task scheduling in the presence of COs in Sect. 5. Section 6 contains two CO example implementations with varying memory consistency semantics. We discuss our low-level API in Sect. 7. Section 8 contains our conclusions.

Due to space constraints, we have issued a technical report [5], which contains additional programming examples and the rationale for the design of the proposed non-blocking synchronization mechanisms. Although the paper is intended to be comprehensive without the appendices, we felt the additional material might nevertheless be useful. For the same reason of constrained space, the description of our approach cannot be compared to the specification of Ada on RML-level.

2 The Memory Model

For reasons outlined in Sect. 1, we do not consider the Ada 2012 atomic and volatile types here. Rather, we introduce *synchronized* types and variables. Synchronized types provide atomic access. We propose aspects and attributes for

specifying a particular memory model to be employed for reading/writing synchronized variables.

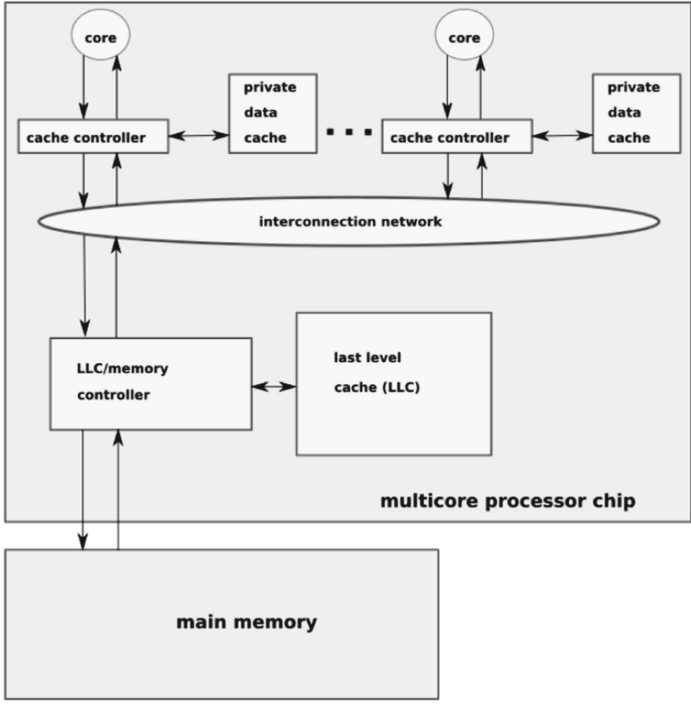


Fig. 2. System model from [18]. "Core" refers to the software's view of a core, which may be an actual core or a thread context of a multithreaded core.

Modern multi-core computer architectures are equipped with a memory hierarchy that consist of main memory, caches and registers. We assume the system model depicted in Fig. 2. It is important to distinguish between memory consistency and coherence. We cite from [18]: *'For a shared memory machine, the memory consistency model defines the architecturally visible behavior of its memory system. Consistency definitions provide rules about loads and stores (or memory reads and writes) and how they act upon memory. As part of supporting a memory consistency model, many machines also provide cache coherence protocols that ensure that multiple cached copies of data are kept up-to-date.'*

The purpose of a memory consistency model (or memory model, for short) is to define the set of values a read operation is allowed to return [1]. To facilitate programmers' intuition, it would be ideal if all read/write operations of a program's tasks are sequentially consistent. However, the hardware memory models provided by contemporary CPU architectures relax SC for the sake of performance [2,9,18]. Enforcing SC on such architectures may incur a noticeable

performance penalty. The workable middle-ground between intuition (SC) and performance (relaxed hardware memory models) has been established with SC for data race-free programs (SC-for-DRF) [3]. Informally, a program has a data race if two tasks access the same memory location, at least one of them is a write, and there are no intervening synchronization operations that would order the accesses. "SC-for-DRF" requires programmers to ensure that programs are free of data races under SC. In turn, the relaxed memory model of a SC-for-DRF CPU guarantees SC for all executions of such a program.

It has been acknowledged in the literature [1] that Ada 83 was perhaps the first widely-use high-level programming language to provide first-class support for shared-memory programming. The approach taken with Ada 83 and later language revisions was to require legal programs to be without synchronization errors, which is the approach taken with SC-for-DRF. In contrast, for the Java memory model it was perceived that even programs with synchronization errors shall have defined semantics for reasons of safety and security of Java's sandboxed execution environment. (We do not consider this approach in the remainder of this paper, because it does not align with Ada's current approach to regard the semantics of programs with synchronization errors as undefined, i.e., as an *erroneous execution*, by [8, 9.10§11].) The SC-for-DRF programming model and two relaxations were formalized for C++11 [7]. They were later adopted for C11, OpenCL 2.0, and for X10 [22] (without the relaxations).

On the programming language level to guarantee DRF, means for synchronization (ordering operations) have to be provided. Ada's POs are well-suited for this purpose. For non-blocking synchronization, atomic operations can be used to enforce an ordering between the memory accesses of two tasks. It is one goal of this paper to add language features to Ada such that atomic operations can be employed with DRF programs. To avoid ambiguity, we propose synchronized variables and types, which support the expression of memory ordering operations at a sufficient level of detail (see Sect. 3.1).

The purpose of synchronized variables is that they can be used to safely transfer information (i.e., the value of the variables) from one task to another. ISAs provide atomic load/store instructions only for a limited set of primitive types. Beyond those, atomicity can only be ensured by locks. Nevertheless, computer architectures provide memory fences (see e.g., [11]) to provide means for ordering memory operations. A memory fence requires that all memory operations before the fence (in program order) must be committed to the memory hierarchy before any operation after the fence. Then, for data to be transferred from one thread to another it is not necessary to be atomic anymore. I.e., it is sufficient that (1) the signaling variable is atomic, and that (2) all write operations are committed to the memory hierarchy before setting the signaling variable. On the receiver's side, it must be ensured that (3) the signaling variable is read atomically, and that (4) memory loads for the data occur after reading the signaling variable (Listing 1.2 provides an example.)

In addition to synchronized variables, synchronized types and aspect `Synchronized_Components` are convenient means for enhancing the usefulness of synchronized variables.

The general idea of our proposed approach is to define *non-blocking* concurrent objects similar to protected objects (cf., e.g., [11]). However, entries of concurrent objects will not block on guards; they will spin until the guard evaluates to true. In addition, functions, procedures, and entries of concurrent objects are allowed to execute and to modify the encapsulated data in parallel. Private entries for concurrent objects are supported. It is their responsibility that the data provides a consistent view to the users of the concurrent object. Concurrent objects will use synchronized types for synchronizing data access. Several memory models are provided for doing this efficiently. It is the responsibility of the programmer to ensure that the entries of a concurrent object are free from data races (DRF). For such programs, the non-blocking semantics of a concurrent object will provide SC in the same way as protected objects do for blocking synchronization.

2.1 Synchronizing Memory Operations and Enforcing Ordering

For defining ordering relations on memory operations, it is useful to introduce some other useful relations.

The *synchronizes-with* relation can be achieved only by use of atomic types in the sense of Ada 2012 or synchronized types in our notion. Even if monitors or protected objects are used for synchronization, the runtime implements them employing synchronized types. The general idea is to equip read and write operations on a synchronized variable with information that will enforce an ordering on the read and write operations. Our proposal is to use aspects and attributes for specifying this ordering information. Details can be found below.

The *happens-before* relation is the basic relation for ordering operations in programs. In a program consisting of only one single thread, happens-before is straightforward. For inter-thread happens-before relations the synchronizes-with relation becomes important. If operation `X` in one thread synchronizes-with operation `Y` in another thread, then `X` happens-before `Y`. Note that the happens-before relation is transitive, i.e., if `X` happens-before `Y` and `Y` happens-before `Z`, then `X` happens-before `Z`. This is true even if `X`, `Y`, and `Z` are part of different threads.

We define different memory models. These memory models originated from the DRF [3] and properly-labeled [9] hardware memory models. They were formalized for the memory model of C++ [7]. The "sequentially consistent" and "acquire-release" memory models provide SC for DRF. The models can have varying costs on different computer architectures. The "acquire-release" memory model is a relaxation of the "sequentially consistent" memory model. As described in Table 1, it requires concessions from the programmer to weaken SC in turn for more flexibility for the CPU to re-order memory operations.

Sequentially Consistent Ordering is the most stringent model and the easiest one for programmers to work with. In this case all threads observe the

same, total order of operations. This means, a sequentially consistent write to a synchronized variable synchronizes-with a sequentially-consistent read of the same variable.

Relaxed Ordering does not obey synchronizes-with relationships, but operations on the same synchronized variable within a single thread still obey happens-before relationships. This means that although one thread may write a synchronized variable, at a later point in time another thread may read an earlier value of this variable.

Acquire-Release Ordering when compared to relaxed ordering introduces some synchronization. In fact, a read operation on synchronized variables can then be labeled by *acquire*, a write operation can be labeled by *release*. Synchronization between release and acquire is pairwise between the thread that issues the release and that acquire operation of a thread that does the first read-acquire after the release.[1] A thread issuing a read-acquire later may read a different value than that written by the first thread.

Table 1. Memory order and constraints for compilers and CPUs

Memory order	Involved threads	Constraints for reordering memory accesses (for compilers and CPUs)
Relaxed	1	No inter-thread constraints
Release/acquire	2	(1) Ordinary stores[a] originally[b] before `release` (in program order) will happen before the release fence (after compiler optimizations and CPU reordering) (2) Ordinary loads originally after `acquire` (in program order) will take place after the acquire fence (after compiler optimizations and CPU reordering)
Sequentially consistent	All	(1) All memory accesses originally before the `sequentially_consistent` one (in program order) will happen before the fence (after compiler optimizations and CPU reordering) (2) All memory accesses originally after the `sequentially_consistent` one (in program order) will happen after the fence (after compiler optimizations and CPU reordering)

[a]Memory accesses other than accesses to synchronized variables.
[b]Before optimizations performed by the compiler and before reordering done by the CPU.

It is important to note that the semantics of the models above have to be enforced by the compiler (for programs which are DRF). I.e., the compiler

[1] In global time!

"knows" the relaxed memory model of the hardware and inserts memory fences in the machine-code such that the memory model of the high-level programming language is enforced. Compiler optimizations must ensure that reordering of operations is performed in such a way that the semantics of the memory model are not violated. The same applies to CPUs, i.e., reordering of instructions is done with respect to the CPU's relaxed hardware memory model, constrained by the ordering semantics of fences inserted by the compiler. The constraints enforced by the memory model are summarized in Table 1.

3 Synchronization Primitives

3.1 Synchronized Variables

Synchronized variables can be used as atomic variables in Ada 2012, the only exception being that they are declared inside the lexical scope (data part) of a concurrent object. In this case aspects and attributes used in the implementation of the concurrent object's operations (functions, procedures, and entries) are employed for specifying behavior according to the memory model. Variables are labeled by the boolean aspect `Synchronized`.

Read accesses to synchronized variables in the implementation of the concurrent object's operations may be labeled with the attribute `Concurrent_Read`, write accesses with the attribute `Concurrent_Write`. Both attributes have a parameter `Memory_Order` to specify the memory order of the access. (If the operations are not labeled, the default values given below apply.) In case of read accesses, `Memory_Order` can be either `Sequentially_Consistent`, `Acquire`, or `Relaxed`. The default value is `Sequentially_Consistent`. For write accesses the values allowed are `Sequentially_Consistent`, `Release`, and `Relaxed`. The default value is again `Sequentially_Consistent`.

For example, assigning the value of synchronized variable `Y` to synchronized variable `X` is given like

```
X'Concurrent_Write(Memory_Order => Release) :=
                    Y'Concurrent_Read(Memory_Order => relaxed);
```

In addition we propose aspects for specifying variable specific default values for the attributes described above. In more detail, when declaring synchronized variables the default values for read and write accesses can be specified via aspects `Memory_Order_Read` and `Memory_Order_Write`. The allowed values are the same as those given above for read and write accesses. If these memory model aspects are given when declaring a synchronized variable, the attributes `Concurrent_Read` and `Concurrent_Write` need not be given for actual read and write accesses of this variable. However, these attributes may be used to temporarily over-write the default values specified for the variable by the aspects. For example

```
   X: integer with Synchronized, Memory_Order_Write => Release;
   Y: integer with Synchronized, Memory_Order_Read => Acquire;
...
   X := Y;
```

does the same as the example above but without spoiling the assignment statement.

Aspect `Synchronized_Components` relates to aspect `Synchronized` in the same way as `Atomic_Components` relates to `Atomic` in Ada 2012.

3.2 Read-Modify-Write Variables

If a variable inside the data part of a concurrent object is labeled by the aspect `Read_Modify_Write`, this implies that the variable is synchronized. Write access to a read-modify-write variable in the implementation of the protected object's operations is a read-modify-write access. The read-modify-write access is done via the attribute `Concurrent_Exchange`. The two parameters of this attribute are `Memory_Order_Success` and `Memory_Order_Failure`. The first specifies the memory order for a successful write, the second one the memory order if the write access fails (and a new value is assigned to the variable).

`Memory_Order_Success` is one of `Sequentially_Consistent`, `Acquire`, `Release`, and `Relaxed`.

`Memory_Order_Failure` may be one of `Sequentially_Consistent`, `Acquire`, and `Relaxed`. The default value for both is `Sequentially_Consistent`. For the same read-modify-write access the memory order specified for failure must not be stricter than that specified for success. So, if `Memory_Order_Failure => Acquire` or `Memory_Order_Failure => Sequentially_Consistent` is specified, these have also be given for success.

For read access to a read-modify-write variable, attribute `Concurrent_Read` has to be used. The parameter `Memory_Order` has to be given. Its value is one of `Sequentially_Consistent`, `Acquire`, `Relaxed`. The default value is `Sequentially_Consistent`.

Again, aspects for variable specific default values for the attributes described above may be specified when declaring a read-modify-write variable. The aspects are `Memory_Order_Read`, `Memory_Order_Write_Success`, and `Memory_Order_Write_Failure` with allowed values as above.

3.3 Synchronization Loops

As presented below synchronization by synchronized variables is performed via spin loops. We call these loops *sync loops*.

4 Concurrent Objects

4.1 Non-blocking Synchronization

Besides the aspects and attributes proposed in Sect. 3 that have to be used for implementing concurrent objects, concurrent objects are different from protected objects in the following way. All operations of concurrent objects can be executed in parallel. Synchronized variables have to be used for synchronizing the

executing operations. Entries have Boolean-valued guards. The Boolean expressions for such guards may contain only synchronized variables declared in the data part of the protected object and constants. Calling an entry results either in immediate execution of the entry's body if the guard evaluates to `true`, or in spinning until the guard eventually evaluates to `true`. We call such a spin loop *sync loop*.

4.2 Read-Modify-Write Synchronization

For concurrent objects with read-modify-write variables the attributes proposed in Sect. 3 apply. All operations of concurrent objects can be executed in parallel. Read-modify-write variables have to be used for synchronizing the executing operations. The guards of entries have to be of the form `X = X'OLD` where `X` denotes a read-modify-write variable of the concurrent object. The attribute `OLD` is well-known from postconditions. An example in our context can be found in Listing 1.1.

If during the execution of an entry a read-modify-write operation is reached, that operation might succeed immediately, in which case execution proceeds after the operation in the normal way. If the operation fails, the whole execution of the entry is restarted (*implicit sync loop*). In particular, only the statements being data-dependent on the read-modify-write variable are re-executed. Statements not being data-dependent on the read-modify-write variables are executed only on the first try.[2] Precluding non-data-dependent statements from re-execution is not only a matter of efficiency, it sometimes makes sense semantically, e.g., for adding heap management to an implementation.

5 Scheduling and Dispatching

We propose a new state for Ada tasks to facilitate correct scheduling and dispatching for threads synchronizing via synchronized or read-modify-write types. If a thread is in a sync loop, the thread state changes to "in_sync_loop". Note that sync loops can only happen inside concurrent objects. Thus they can be spotted easily by the compiler and cannot be confused with "normal" loops. Note also that for the state change it makes sense not to take place during the first iteration of the sync loop, because the synchronization may succeed immediately. For read-modify-write loops, iteration from the third iteration on may be a good choice; for spin loops, an iteration from the second iteration on may be a good choice.

In this way the runtime can guarantee that not all available CPUs (cores) are occupied by threads in state "in_sync_loop". Thus we can be sure that at least one thread makes progress and finally all synchronized or read-modify-write

[2] For the case that the compiler cannot figure out which statements are data-dependent, we propose an additional Boolean aspect `only_execute_on_first_try` to tag non-data-dependent statements.

variables are released (if the program's synchronization structure is correct and the program does not deadlock).

After leaving a sync loop, the thread state changes back to "runable".

6 Examples

Non-blocking Stack. Listing 1.1 shows an implementation of a non-blocking stack using our proposed syntax for concurrent objects.

```
1  subtype Data is Integer;
2
3  type List;
4  type List_P is access List;
5  type List is
6    record
7      D: Data;
8      Next: List_P;
9    end record;
10
11 Empty: exception;
12
13 concurrent Lock_Free_Stack
14 is
15   entry Push(D: Data);
16   entry Pop(D: out Data);
17 private
18   Head: List_P with Read_Modify_Write,
19     Memory_Order_Read => Relaxed,
20     Memory_Order_Write_Success => Release,
21     Memory_Order_Write_Failure => Relaxed;
22 end Lock_Free_Stack;
23
24 concurrent body Lock_Free_Stack is
25   entry Push (D: Data)
26       until Head = Head'OLD is
27     New_Node: List_P := new List;
28   begin
29     New_Node.all := (D => D, Next => Head);
30     Head := New_Node;
31   end Push;
32
33   entry Pop(D: out Data)
34       until Head = Head'OLD is
35     Old_Head: List_P;
36   begin
37     Old_Head := Head;
38     if Old_Head /= null then
39       Head := Old_Head.Next;
40       D := Old_head.D;
41     else
42       raise Empty;
43     end if;
44   end Pop;
45 end Lock_Free_Stack;
```

Listing 1.1. Non-blocking Stack Implementation Using Proposed New Syntax

The implementation of entry `Push` (lines 25–31) works as follows. In Line 29 a new element is inserted at the head of the list. Pointer `Next` of this element is set to the current head. The next statement (Line 3.) assigns the new value to the head of the list. Since variable `Head` has aspect `Read_Modify_Write` (line 18),

this is done with RMW semantics, i.e., if the value of `Head` has not been changed (since the execution of `Push` has started) by a different thread executing `Push` or `Pop` (i.e., `Head = Head'OLD`), then the RMW operation succeeds and execution proceeds at Line 31, i.e., `Push` returns. If the value of `Head` has been changed (`Head /= Head'OLD`), then the RMW operation fails and entry `Push` is re-executed starting from Line 29. Line 27 is not re-executed as it is not data dependent on `Head`.

Several memory order aspects apply to the RMW operation (Line 3.) which are given in lines 19–21: In case of a successful execution of the RMW, the value of `Head` is released such that other threads can read its value via memory order acquire. In the failure case the new value of `Head` is assigned to the "local copy" of `Head` (i.e., `Head'OLD`) via relaxed memory order. "Relaxed" is enough because RMW semantics will detect if the value of `Head` has been changed by a different thread anyway.

The implementation of entry `Pop` (lines 33–44) follows along the same lines.

Memory management needs special consideration: In our case it is enough to use a synchronized counter that counts the number of threads inside `Pop`. If the counter equals 1, memory can be freed. Ada's storage pools are a perfect means for doing this without spoiling the code.

This example also shows how easy it is to migrate from a (working) blocking to a (working) non-blocking implementation of a program. Assume that a working implementation with a protected object exists, then one has to follow these steps:

1. Replace keyword `protected` by keyword `concurrent`.
2. Replace protected operations by DRF concurrent operations, thereby adding appropriate guards to the concurrent entries.
3. Test the non-blocking program which now has default memory order `sequentially_consistent`.
4. Carefully relax the memory ordering requirements: Add memory order aspects and/or attributes `Acquire`, `Release`, and/or `Relaxed` to improve performance but without violating memory consistency.

Generic Release-Acquire Object. Listing 1.2 shows how release-acquire semantics can be implemented for general data structures with help of one synchronized Boolean.

```
1  generic
2    type Data is private;
3  package Generic_Release_Acquire is
4
5    concurrent RA
6    is
7      procedure Write (d: Data);
8      entry Get (D: out Data);
9    private
10     Ready: Boolean := false with Synchronized,
11       Memory_Order_Read => Acquire,
12       Memory_Order_Write => Release;
13     Da: Data;
14   end RA;
```

```
15
16 end Generic_Release_Acquire;
17
18 package body Generic_Release_Acquire is
19
20   concurrent body RA is
21
22     procedure Write (D: Data) is
23     begin
24       Da := D;
25       Ready := true;
26     end Write:
27
28     entry Get (D: out Data)
29       when Ready is
30       -- spin-lock until released, i.e., Ready = true;
31       -- only sync. variables and constants allowed in guard expression
32     begin
33       D := Da;
34     end Get;
35   end RA;
36
37 end Generic_Release_Acquire;
```

Listing 1.2. Generic Release-Acquire Object

7 API

As already pointed out, we feel that providing concurrent objects as first-class citizens is the right way to enhance Ada with non-blocking synchronization on an adequate memory model. On the other hand, if the programmer needs synchronization on a lower level than concurrent objects provide, an API-based approach (generic function `Read_Modify_Write` in package `Memory_Model`) would be a viable alternative. Listing 1.3 shows such a predefined package `Memory_Model`. It contains the specification of generic function `Read_Modify_Write`, which allows to use the read-modify-write operation of the underlying computer hardware.

Exposing sync loops to the programmer makes it necessary to introduce a new aspect `sync_loop` to let the runtime perform the state change to "in_sync_loop" (cf. Sect. 5). Because the correct use of this aspect on the part of the programmer cannot be ensured, the information transferred to the runtime may be false or incomplete, giving rise to concurrency defects such as deadlocks, livelocks, and other problems.

```
1  package Memory_Model is
2
3    type Memory_Order_Type is (
4      Sequentially_Consistent ,
5      Relaxed ,
6      Acquire ,
7      Release);
8
9    subtype Memory_Order_Success_Type is Memory_Order_Type;
10
11   subtype Memory_Order_Failure_Type is Memory_Order_Type
12     range Sequentially_Consistent .. Acquire;
13
14   generic
15      type Some_Synchronized_Type is private;
16      with function Update return Some_Synchronized_Type;
17      Read_Modify_Write_Variable: in out Some_Synchronized_Type
18        with Read_Modify_Write;
```

```
19      Memory_Order_Success: Memory_Order_Success_Type :=
20        Sequentially_Consistent;
21      Memory_Order_Failure: Memory_Order_Failure_Type :=
22        Sequentially_Consistent;
23    function Read_Modify_Write return Boolean;
24
25  end Memory_Model;
```

Listing 1.3. Package Memory_Model

8 Conclusion and Future Work

We have presented an approach for providing safe non-blocking synchronization in Ada 202x. Our novel approach is based on introducing concurrent objects for encapsulating non-blocking data structures on a high abstraction level. In addition, we have presented synchronized and read-modify-write types which support the expression of memory ordering operations at a sufficient level of detail. Concurrent objects provide SC for programs without data races. This SC-for-DRF memory model is well-aligned with Ada's semantics for blocking synchronization via protected objects, which requires legal programs to be without synchronization errors [8, 9.10§11].

Although Ada 2012 provides the highly abstract protected object monitor-construct for blocking synchronization, there was previously no programming primitive available to match this abstraction level for non-blocking synchronization. The proposed memory model in conjunction with our concurrent object construct for non-blocking synchronization may bar users from having to invent ad-hoc synchronization solutions, which have been found error-prone with blocking synchronization already [21].

Until now, all previous approaches are based on APIs. We have listed a number of advantages that support our approach of making non-blocking data structures first class language citizens. In contrast, our approach for Ada 202x encapsulates non-blocking synchronization inside concurrent objects. This safe approach makes the code easy to understand. Note that concurrent objects are not orthogonal to objects in the sense of OOP (tagged types in Ada). However, this can be achieved by employing the proposed API approach (cf. Sect. 7). In addition, it is not difficult to migrate code from blocking to non-blocking synchronization. Adding memory management via storage pools integrates well with our modular approach and does not clutter the code.

A lot of work remains to be done. To name only a few issues: Non-blocking barriers (in the sense of [8, D.10.1]) would be useful; details have to be elaborated. Fully integrating concurrent objects into scheduling and dispatching models and integrating with the features for parallel programming planned for Ada 202x have to be done carefully.

Acknowledgments. This research was supported by the Austrian Science Fund (FWF) project I 1035N23, and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015-M3C4A7065522.

References

1. Adve, S.V., Boehm, H.-J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM **53**(8), 90–101 (2010)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer **29**(12), 66–76 (1996)
3. Adve, S.V., Hill, M.D.: Weak ordering–a new definition. In: Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA 1990, pp. 2–14. ACM, New York (1990)
4. Barnes, J.: Ada 2012 Rationale: The Language – The Standard Libraries. LNCS, vol. 8338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45210-9
5. Blieberger, J., Burgstaller, B.: Safe non-blocking synchronization in Ada 202x. CoRR, abs/1803.10067 (2018)
6. Boehm, H.-J.: Threads cannot be implemented as a library. SIGPLAN Not. **40**(6), 261–268 (2005)
7. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 68–78. ACM, New York (2008)
8. Brukardt, R.L. (ed.) Annotated Ada Reference Manual, ISO/IEC 8652:2012/Cor 1:2016 (2016)
9. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. SIGARCH Comput. Archit. News **18**(2SI), 15–26 (1990)
10. Barnes, J. (ed.): Ada 95 Rationale: The Language – The Standard Libraries. LNCS, vol. 1247. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0051526
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2012)
12. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
13. ISO/IEC: Working Draft N4296, Standard for Programming Language C++. ISO, Geneva, Switzerland (2014)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
15. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 378–391. ACM, New York (2005)
16. Scott, M.L.: Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Francisco (2013)
17. Simpson, H.: Four-slot fully asynchronous communication mechanism. IEE Proc. E Comput. Digit. Tech. **137**, 17–30 (1990)
18. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture, vol. 16. Morgan & Claypool, San Francisco (2011)
19. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_3
20. Williams, A.: C++ Concurrency in Action. Manning Publ. Co., Shelter Island (2012)

21. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI 2010, Berkeley, CA, USA, pp. 163–176. USENIX Association (2010)
22. Zwinkau, A.: A memory model for X10. In: Proceedings of the 6th ACM SIGPLAN Workshop on X10, X10 2016, pp. 7–12. ACM, New York (2016)

Handling Implicit Overhead

On the Effect of Protected Entry Servicing Policies on the Response Time of Ada Tasks

Jorge Garrido(✉), Juan Zamorano, Alejandro Alonso, and Juan A. de la Puente

Sistemas de Tiempo Real e Ingeniería de Servicios Telemáticos (STRAST), Information Processing and Telecommunications Centre, Universidad Politécnica de Madrid (UPM), Madrid, Spain
{jgarrido,jzamorano,aalonso,jpuente}@dit.upm.es

Abstract. 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 MSRP and the MrsP locking policies is discussed. Extensions to response time analysis for the proxy model with both locking policies are also contributed.

Keywords: Real-time systems · Multiprocessor systems
Compiler implementation · Ada Ravenscar profile
Schedulability analysis

1 Introduction

Ada support for multiprocessors allows developers to build real-time embedded applications with enhanced performance and full control over the execution of applications on the available processor cores.

This work has been partially funded by the Spanish National R&D&I plan (project M2C2, TIN2014-56158-C4-3-P).

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 73–86, 2018.
https://doi.org/10.1007/978-3-319-92432-8_5

Real-time systems require temporal properties of the tasks to be ensured by the implementation. For hard real-time systems, task deadlines must be guaranteed in all cases, including the worst possible conditions. This usually requires using scheduling methods with a predictable temporal behaviour [10], as well as analysis methods that enable developers to verify that the temporal behaviour of the system meets the requirements (see e.g. [7] for a comprehensive presentation of the topic).

The Ada Real-Time Annex [1, annex D] provides a flexible priority-based dispatching model that allows developers to use some common scheduling policies. Dynamic priorities provide additional flexibility. Dispatching domains can be assigned to the tasks in order to specify the processor or processors on which they may execute. The flexibility of the full Ada tasking model, however, makes temporal analysis difficult or even impossible. The Ravenscar profile [1, D.13] was defined in order to provide a limited tasking model that ensures that static temporal analysis techniques can be applied to real-time systems. In the following, a multiprocessor system model based on fully-partitioned fixed-priority scheduling with inter-task communication based on shared protected objects (PO) will be assumed. This model is compatible with the Ravenscar Profile, although not necessarily limited to it.

Two important aspects of the implementation of protected objects are the mechanisms used for servicing blocked PO entry queues, and the locking policies that must be used for minimising the effects of priority inversion [18]. A number of multiprocessor locking protocols have been proposed [16], among which MSRP [11] and MrsP [8] have received widest attention.

While the use of resource locking protocols, both in general Ada and the Ravenscar profile, has been analysed in detail [12], the impact of entry queue servicing policies on the temporal behaviour of multiprocessor real-time systems has not been discussed in detail. The aim of this paper is to contribute to such analysis, with focus on the so-called proxy model. The contents are organised as follows: Sect. 2 describes the details of entry servicing in Ada protected objects. Section 3 summarises the main results on real-time analysis of the MSRP and MrsP protocols. The main contribution is the impact of service models on response time analysis, which is discussed in Sect. 4. Finally, the conclusions of the study and suggestions for future work are presented in Sect. 5.

2 Protected Objects in Ada

2.1 Protected Objects and Protected Operations

Protected objects are the preferred mechanism for inter-task communication in Ada. A protected object consists of one or more private data fields, together with a set of operations that can be carried out on the data. Protected operations can be of three kinds: functions, procedures, and entries, and are executed in mutual exclusion. Only procedures and entries can change the protected data, and therefore multiple calls to protected functions may be executed concurrently if the implementation chooses to do so.

2.2 Protected Entries

Protected entries have a Boolean barrier. When a task issues a call to an entry, the barrier is evaluated and the call is accepted if the barrier is true. Otherwise, the calling task is suspended on a queue associated to the entry. Barriers are reevaluated at the end of every execution of a protected procedure or entry. If there are any pending calls on entries with open barriers, one of the calls is selected to be serviced, i.e. removed from the queue and executed.

The service order for pending entries can be specified with a pragma Queuing_Policy. The default policy is FIFO. Alternatively, priority order or some other implementation-defined policy can be defined. The default policy does not specify which entry queue is to be served first if there are more than one queue with open barriers.

Pending queued entries take precedence over new calls trying to access the protected object. In this way, a task that was waiting for change in the state of the protected object can resume its execution with the guarantee that the state has not changed again, thus avoiding possible race conditions and starvation [4]. This rule is commonly known as the "eggshell model".

The evaluation of barriers and the execution of the entry body that is selected to be serviced at the end of a protected operation can be executed in different ways, as the standard does not specify which task should serve the entry. Two possible approaches are the *self-service model* and the *proxy model* [14,17].

2.3 Self-service Model

Under this approach, when a task ends a protected operation and an entry with an open barrier is selected for execution, the task that has called the entry (the *caller task*) is resumed, and executes the entry body. This is a simple approach that allows for parallel execution of both tasks, and may thus be preferable for multiprocessor implementations. However, execution on monoprocessors may be less efficient, as it requires more context switches, and may be difficult to implement on some real-time kernels [15].

2.4 Proxy Model

An alternative approach is the *proxy model* [14]. In this model, the task ending a protected operation acts as a *server task* that reevaluates the barriers and executes the selected entry body on behalf of the caller task. The process is repeated while there are pending entries with true barriers. This approach saves context switches and simplifies the design of the run-time system, which makes it the method of choice for implementing protected entry servicing on monoprocessors. It can also be used on multiprocessors, although the implementation may become more complex in this case, depending on whether the caller and the server task execute on the same or different processors [9].

2.5 Ravenscar Restrictions

The proxy model may make real-time analysis difficult with unrestricted Ada tasking, since the number of tasks that may be waiting on entry queues may be high, thus leading to a very long execution time for the server task. However, in the Ravenscar profile protected objects may have at most one entry, and there may be at most one waiting task on a closed entry. Therefore, the server task may have to execute at most one entry body, and its execution time stays bounded, thus making response time analysis feasible from a practical point of view.

3 Resource Sharing Protocols for Multiprocessor Systems

3.1 Resource Sharing Protocols

A fundamental issue in multiprocessor real-time systems is the definition of resource access protocols that provide for bounded task blocking. Although other approaches are possible [16], most of the published work has been aimed at adapting some well-known monoprocessor methods, such as the Priority Ceiling Protocol (PCP) [18] or the Stack Resource Policy (SRP) [2,3], to multiprocessor systems.

The default policy for Ada is Ceiling_Locking, which is based on SRP, a generalisation of PCP also valid for Earliest Deadline First (EDF) scheduling. In the following we examine in detail two multiprocessor protocols derived from SRP that have received significant attention from the research community, namely MSRP and MrsP.

3.2 Multiprocessor Stack Resource Policy

The Multiprocessor Stack Resource Policy (MSRP) [11] is an extension of SRP for multiprocessors. The MSRP system model is defined by a fully partitioned scheduling with global resources that are, in turn, not bounded to a specific processor. In this policy, unsatisfied resource accesses are serviced in FIFO order, and tasks spin-wait non-preemptively until access is granted. The following properties are inherited from SRP:

- A task can only be locally blocked before it starts executing.
- A task can only be locally blocked once per activation, bounded to one critical section length.
- It can be easily implemented on a multiprocessor Ravenscar-compliant kernel by assigning all global resources a ceiling priority equal to the highest priority in the system.
- The access cost to a shared resource. i.e. the longest time a task can be blocked awaiting is bounded.

The access cost to a shared resource is bounded as requests are serviced in FIFO order and at most one request per processor can be issued at a time, because shared resource accesses are not preemptable. Therefore, the maximum

time a task τ_i running on processor P_m can be spinning waiting to access a resource r^k can be expressed as:

$$\text{spin}(r^k, P_m) = \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k \tag{1}$$

where w_x^k is the worst-case access time to resource r^k which a task τ_x executing on a remote processor may experience. The spin time calculated as above is to be added, for each access, to the task worst-case execution time when carrying out the schedulability analysis. This result can be improved, as shown by Brandenburg and Wieder [5,19], by using holistic analysis and mixed-integer linear programming techniques to safely reduce the pessimism in the number of times a remote processor can cause spin delay on each task activation.

Using MSRP with fixed-priority scheduling, as in the Ravenscar profile, has a major drawback. Since waiting for access to a shared resource is not preemptable, the blocking incurred by a high-priority task does not depend on its use of shared resources, but on that of lower-priority tasks, even though the ceiling priorities of the shared resources are lower than the priority of the high-priority task. Since high-priority tasks often have short deadlines, especially if priorities are assigned in deadline-monotonic order, they can be expected to be most affected by priority inversion.

3.3 Multiprocessor Resource Sharing Protocol

The Multiprocessor resource sharing Protocol (MrsP) [8] was devised to address the above described drawback in MSRP. This protocol also relies on the properties of PCP and SRP: resources are assigned a ceiling priority, and all access requests are performed at that priority. As in MSRP, access requests are dealt with in FIFO order, and the tasks waiting for access to a resource spin-wait until they are granted access. However, the spin-wait and access itself are done at the ceiling priority of the resource, and thus calling tasks can be preempted. Therefore, tasks with priorities higher than the ceiling priority of the resource are not blocked by lower-priority tasks accessing the resource.

Another benefit of waiting at ceiling priorities is that, as in MSRP, at most one task per processor can be trying to access the resource at a time. As a result, and like in MSRP, the length of the resource FIFO queue is bounded by the number of processors from where the resource is accessed. A desirable access cost to a resource would then be the sum of worst access times of each remote processor, plus the cost of the access to be performed [12], as in Eq. (1) above. However, since all the shared resource activity is executed at its ceiling priority, accesses are not guaranteed to be completed without being preempted by a higher-priority task.

In order to achieve the same access cost with MrsP as with MSRP, it must be made sure that a task executing an operation on a shared resource makes progress while other tasks are spin-waiting for the resource. To this end, spin-waiting tasks must be capable of undertaking the access operation of a locally

preempted task holding the resource lock. This cooperation must respect the FIFO order [8]. This can be accomplished by delegating the execution on the waiting task, or by migrating the preempted task to a processor where a task is spin-waiting for the resource.

The first method is not practical. This approach would require accesses to shared objects to be atomic, without side effects and potentially being executed in parallel by waiting task, accepting only one final commit. The second approach can, on the contrary, be easily implemented since task migration mechanisms are integrated in most run-times of multiprocessor operating systems. In particular, an Ada implementation based on a smart modification of the affinities of the involved entities which may enable this kind of migration is outlined in [6]. The main drawback of this approach is the overhead caused by such migrations. A way to account for this overhead and an evaluation of its influence is presented in [20], where MrsP is shown to provide better schedulability than MSRP, even including this overhead.

4 Impact of Service Modes in Response Time Analysis

An aspect that has not received enough attention to date is the impact of the entry queue service models on the response time of multiprocessor real-time systems. The next paragraphs discuss the main issues related to using the proxy and the self-service models with MSRP and MrsP.

4.1 Entry Servicing in MSRP

MSRP is compatible with the Ada definition of protected objects, as long as the ceiling priority of all protected objects is assigned a non-preemptable value, i.e. one which is higher than any task priority.

The current GNAT implementation of protected objects on multiprocessors follows the proxy model. As previously explained, a task calling an entry with a closed barrier is suspended. When a call to another protected procedure or entry completes, barriers are reevaluated, and pending calls to entries with newly opened barriers are executed by the calling task, which acts as a server task. Since the server task executes non-preemptively, all pending entry bodies are executed until no remaining task is enqueued on an open entry.

The eggshell model implies that the resource is busy while pending entries are executed by the server task, and thus further calls to protected operations are postponed until the enqueued requests have been served. This makes the maximum number of calls to be executed by the server task to be bounded by the maximum number of tasks that can issue entry calls on the resource.[1] Let $G_e(r^k)$ be the set of tasks calling entries in resource r^k. If $|G_e(r^k)|$ is the

[1] For this bound to be effective it must be assumed that the task set is static, or at least that there is a bound on the number of caller tasks for the resource.

size of this set, the worst-case overhead incurred in each access to a protected subprogram operation in r^k is:

$$\text{overhead}(r^k, P_m) = |G_e(r^k)| \times C_e^k + \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k + |G_e(r^k)| \times C_e^k \quad (2)$$

where C_e^k is the maximum cost of servicing an entry request in resource r^k, P is the set of all processors, and P_m is the processor on which the server task runs, as in Eq. (1).

With the Ravenscar profile, the analysis is simplified because there may be at most one task blocked on an entry, and a protected object can have at most one entry. The worst-case overhead is then:

$$\text{overhead}(r^k, P_m) = C_e^k + \sum_{p \in \{P - P_m\}} \max_{\tau_x \in T_p} w_x^k + C_e^k \quad (3)$$

4.2 Entry Servicing in MrsP

MrsP does not have a defined behaviour for the Ada entry model. The Ada implementation proposed in [6] gives no hint on how enqueued entries should be serviced and analysed under MrsP. In order to complete the protocol definition, some alternatives for such implementation are explored in the following paragraphs.

Self-service Model. This approach is widely accepted as best suited for multiprocessor systems, since it is supposed to better support parallelism. However, in practice it presents some drawbacks, particularly a loss of efficiency when implemented on top of POSIX threads, even on monoprocessors [17].

The expected benefit of using self-service in multiprocessor systems comes from the fact that the task waiting on a closed entry and the task opening the barrier can be allocated to different processors. However, this can lead to unexpected blocking in the execution of higher-priority tasks, contrary to the monoprocessor case, where a task can only be blocked once per activation by a lower-priority task, and only before the higher-priority task starts executing.

Consider the example depicted in Fig. 1 including tasks τ_1 and τ_2 with priorities $p_1 = 1$ and $p_2 = 2$, respectively.[2] Let τ_1 be blocked on an entry belonging to a protected object r with a ceiling priority $p_r = 3$, and then τ_2 is released and executes non-protected code on the same processor. If the barrier is opened after a protected operation executing on some other processor, τ_1 becomes runnable again with an active priority of 3, thus preempting τ_2.

To prevent such a case, one possibility would be to decrease the active priority of a task waiting on a closed entry barrier. A similar scheme is used in GNAT for the proxy model, where the caller task, after locking the protected object, reverts to its previous active priority, to be rescheduled at that priority once

[2] Following Ada convention, a higher value indicates a higher priority.

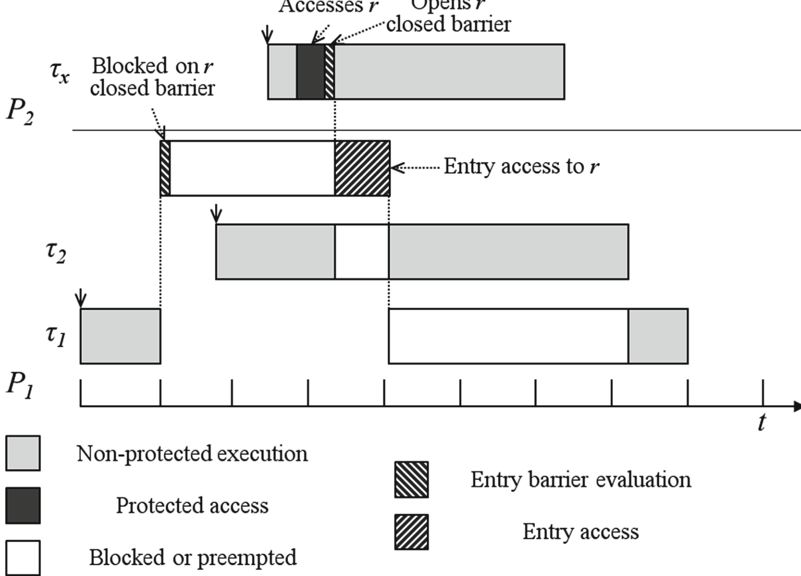


Fig. 1. Potential delayed priority inversion under self-service model.

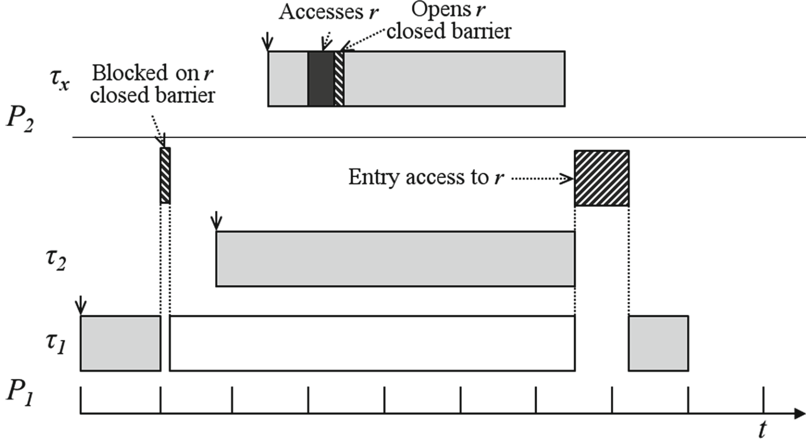


Fig. 2. Tasks blocked on closed entry barriers decrease their active priority.

its entry request has been served. This approach is illustrated in Fig. 2. In this example, τ_1 can perform its entry access to the resource (raising again its active priority to 3) when τ_2 execution is completed.

This approach, however, can lead to further issues. A task that is spin-waiting on an entry call may not be able to access the resource even after the barrier has been opened by another task, because the priority of the waiting task is not

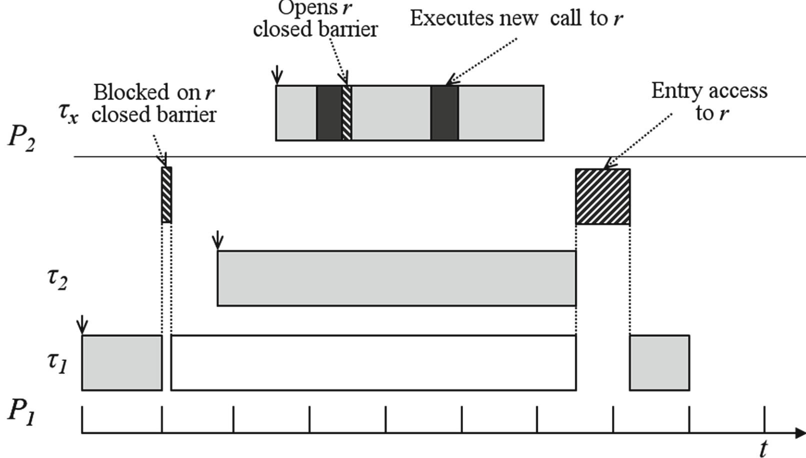


Fig. 3. Approach breaking the eggshell model.

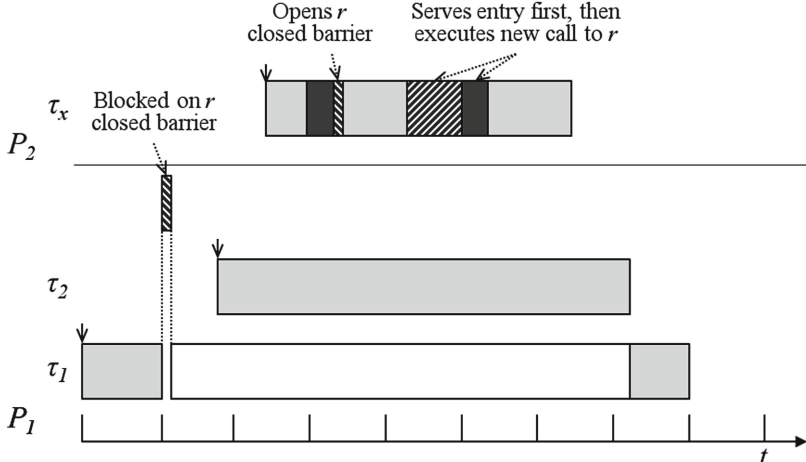


Fig. 4. Approach implementing a delayed proxy model.

high enough, as it has been reverted to its basic priority. If there is a new call to a protected operation in the same object, it should not be accepted until the pending entry call has been serviced, as per the eggshell model. The second call is thus delayed even if the entry barrier is now open, thus leading to further priority inversion. Note that this can also happen when the task that makes the new call runs on the same processor as the waiting task.

Aside from being highly inefficient, the timing behaviour in such situations cannot be analysed without adding extra pessimism. Two possible ways of tackling this issue are:

- Letting the new call proceed without serving the entries, thus breaking the eggshell model (Fig. 3).
- Serving the entries and then letting the new call proceed. This would be somewhat of a 'delayed proxy model' (Fig. 4).

The former option is not compatible with the Ada standard, and therefore will not be further discussed. The latter one, on the contrary, would not be far from the current proxy GNAT implementation. Furthermore, as the entry calls are only executed by delegation, as shown in Fig. 4, when strictly required, parallelism can be improved.

Unfortunately, this solution would only make a true benefit if both the task calling the entry and that making the new call are hosted on the same processor. In any other case this solution would complicate the implementation. Executing a subprogram call (even a function call) would require to check whether there are remote entry callers queued on open barriers, then to check their scheduling state (if they are currently running or not) and finally to undertake their access if necessary to preserve the FIFO order.

In any case, this solution is actually a variant of the proxy model, and its implications can be studied along with the discussion in the following paragraphs.

Proxy Model. As previously shown, serving outstanding entry calls at the end of the call that opens the corresponding barriers improves efficiency and is consistent with the Ada semantics.

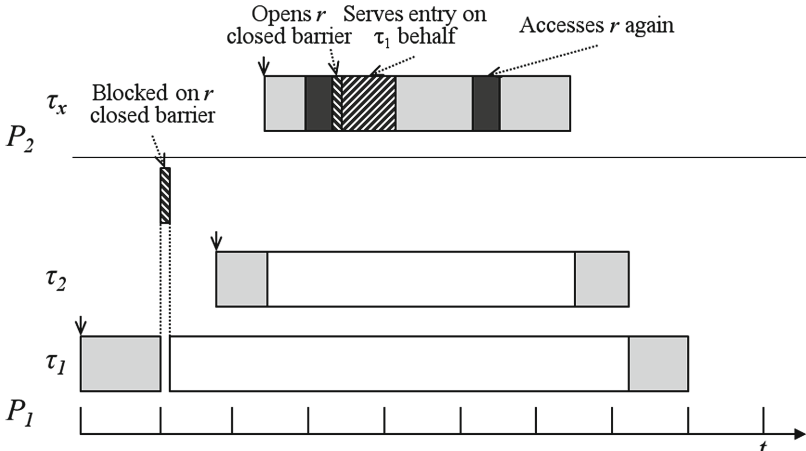


Fig. 5. Proxy model entry servicing.

When accesses to protected objects are carried out in a non-preemptive way, the entry servicing is deterministic: entries are served in FIFO order (as required

by MSRP and MrsP) by the server task on its own processor, until the barrier is closed again or all queued entry calls have been served.

However, when accesses to protected objects can be preempted, a different kind of problem arises, as the servicing of pending entry calls can be delayed by a preemption of the server task, which compromises response time analysis. In order to overcome this issue and ensure progress in the server tasks, two possible solutions can be envisioned:

– To migrate the server task to a processor where the access to the resource can be completed, according to the helping mechanism presented in Sect. 3.2.
– To perform the entry servicing non-preemptively in order to avoid such situation.

In general, under MrsP a task holding a resource lock may be preempted. As explained in Sect. 3, the preempted task is migrated to a remote processor if there is a task actively waiting for the resource. This approach further increases the potential access cost to a shared resource, since the migration cost has to be included in the analysis. The worst-case number of migrations can be obtained by calculating the number of local higher-priority tasks releases during the access [20], i.e. for each valid migration target (each processor that hosts at least one task accessing the resource), calculate the number of times it can suffer interference from higher-priority tasks. This is calculated by obtaining the ceiling value of the resource access time divided by the period of each higher-priority task on the migration targets. This value, multiplied by the measured worst-case cost of a migration is to be added to the final resource access cost. By adding the time of servicing an entry to the resource access time, i.e. extending the time in which the access can be preempted and migrated again, the previously presented analysis yields a safe upper bound for the overhead induced by migrations under the proxy model.

The other possible solution is to serve the entry in a non-preemptable way. While this would limit the overhead in the server task, it might also negatively affect the response time of higher-priority tasks, even those not accessing the involved resource. MrsP was designed to avoid or at least reduce the unnecessary blocking suffered by higher-priority tasks from resources only used by lower-priority tasks. However, short non-preemptable sections can still be beneficial with this protocol, as shown in the evaluation made in [20]. Keeping non-preemptable sections short should not be specially costly in real systems, especially those implemented with the Ravenscar profile, since entries are mainly used for task synchronization purposes.

In order to consistently use MrsP with the Ada tasking model, the impact of servicing entries first, according to the eggshell rules, must be evaluated. The following equation, which follows a similar approach as (2), can be used to extend the results in [8] and [13, Eq. 5]:

$$e^k = |map(G(r^k))| \times (c^k + |G_e(r^k)| \times C_e^k) \tag{4}$$

where e^k is the cost of a single access to a resource r^k. Function $map(G(r^k))$ returns the set of processors that host at least one task accessing resource r^k

and $|map(G(r^k))|$ returns the size of that set. This safely bounds the number of access requests to r^k that can be issued at a time, since, due to the use of ceiling priorities, only one task per processor can be trying to access the resource at any given moment. This number is multiplied by the time required for an access to the resource, that is the sum of is execution time c^k, plus the time required to serve the potentially queued entry calls with barriers now open. For Ravenscar systems, this equation can be simplified as only one entry request can be queued per access:

$$e^k = |map(G(r^k))| \times (c^k + C_e^k) \tag{5}$$

As shown in [20], this way of analysing resource contention is highly pessimistic, especially when resource access request patterns are uneven among tasks and processors. Consider a task running on a processor issuing requests to a resource every few milliseconds, while there is only another task accessing the same resource from a different processor, with a rate in the order of seconds. Then it is clearly pessimistic to assume that all accesses from the first task will be delayed by accesses from the second task. Therefore, the response time analysis must provide means to reduce that pessimism based on the periodicity of the requests issued on each processor. This same reasoning can be used to reduce the pessimism on the need to service entry queue calls present in Eqs. 4 and 5. Given the semantics of entry calls their frequency may not be comparable to that of other protected actions. This is particularly true for Ravenscar systems, where entries are only meant to be used to synchronize tasks, and thus the frequency of entry requests may be expected to be clearly lower than that of other requests. If this is the case, it is clearly pessimistic to assume that each non-entry access will suffer an entry servicing overhead (Fig. 5).

A safe upper bound for the maximum entry-servicing overhead a task τ_i can incur on a single activation due to a resource r^k can be calculated as:

$$EN_i^k = \sum_{\tau_x \neq \tau_i} \left\lceil \frac{R_i}{T_x^k} \right\rceil \tag{6}$$

where T_x^k is the minimum inter-arrival time of entry requests to resource r^k by a task τ_x, and R_i is the response time of τ_i. This result can be used to reduce the pessimism of the previous equations, and therefore, the cost of all accesses to a resource r^k by τ_i during an activation can be expressed as:

$$E_i^k = EN_i^k \times C_e^k + N_i^k \times (|map(G(r^k))| \times c^k) \tag{7}$$

where N_i^k is the maximum number of times τ_i accesses r^k. Note that, while Eqs. 6 and 7 are also valid for general Ada tasking, with the Ravenscar profile, EN_i^k can be also safely bounded by $N_i^k \times |map(G(r^k))|$, since each access to the resource can only have to serve at most one entry. In consequence, the lower of both values is to be used for schedulability analysis.

5 Conclusions

Protected objects and entries are a powerful mechanism for controlling the synchronization of concurrent tasks. Nevertheless, Ada protected entries exhibit some peculiarities that have to be taken into account when analysing the temporal behaviour of real-time systems.

Among the possible implementation of protected entry servicing in multiprocessors, self-service has a potential for taking advantage of parallel execution to improve the efficiency of the mechanism. However, it cannot be used with locking policies based on PCP or SRP, such as MSRP or MrsP, without compromising the properties of these protocols or violating the eggshell definition.

On the other hand, the proxy model is simpler to implement, and can be used with MSRP and MrsP. It has also been shown to be analysable for systems using non-preemptive spin-locking. The overhead caused by the extra execution time in protected calls, as well as the impact of giving priority to entry servicing over new calls, as required by the eggshell model, on the response time analysis, have been calculated, and new response time equations have been derived.

References

1. Ada Reference Manual, ISO/IEC 8652:2012(E) with COR.1:2016 (2016). http://www.ada-auth.org/arm.html
2. Baker, T.P.: A stack-based resource allocation policy for realtime processes. In: 1990 Proceedings of the 11th Real-Time Systems Symposium, pp. 191–200, December 1990
3. Baker, T.P.: Stack-based scheduling for realtime processes. Real-Time Syst. **3**(1), 67–99 (1991)
4. Barnes, J.: Programming in Ada 2012. Cambridge University Press, Cambridge (2014)
5. Brandenburg, B.B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2011)
6. Burns, A., Wellings, A.J.: Locking policies for multiprocessor Ada. Ada Lett. **33**(2), 59–65 (2013)
7. Burns, A., Wellings, A.: Analysable Real-Time Systems: Programmed in Ada. CreateSpace Independent Publishing Platform (2016)
8. Burns, A., Wellings, A.J.: A schedulability compatible multiprocessor resource sharing protocol-MrsP. In: 2013 25th Euromicro Conference on Real-Time Systems (ECRTS), pp. 282–291. IEEE (2013)
9. Chouteau, F., Ruiz, J.F.: Design and implementation of a Ravenscar extension for multiprocessors. In: Romanovsky, A., Vardanega, T. (eds.) Ada-Europe 2011. LNCS, vol. 6652, pp. 31–45. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21338-0_3
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). http://doi.acm.org/10.1145/1978802.1978814
11. Gai, P., Lipari, G., Natale, M.D.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the 22nd IEEE Real-Time Systems Symposium. IEEE Computer Society (2001)

12. Garrido, J., Zamorano, J., Alonso, A., de la Puente, J.A.: Evaluating MSRP and MrsP with the multiprocessor Ravenscar profile. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 3–17. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_1
13. Garrido, J., Zhao, S., Burns, A., Wellings, A.: Supporting nested resources in MrsP. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 73–86. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_5
14. Giering, E.W., Baker, T.P.: The GNU Ada Runtime Library (GNARL): design and implementation. In: WADAS 1994: Proceedings of the Eleventh Annual Washington Ada Symposium & Summer ACM SIGAda Meeting on Ada, pp. 97–107. ACM Press, New York (1994)
15. Giering, E.W., Mueller, F., Baker, T.P.: Implementing ada 9x features using posix threads: design issues. In: Proceedings of the Conference on TRI-Ada 1993, TRI-Ada 1993, pp. 214–228. ACM, New York (1993). http://doi.acm.org/10.1145/170657.170736
16. Lin, S., Wellings, A.J., Burns, A.: Ada 2012: resource sharing and multiprocessors. Ada Lett. **33**(1), 32–44 (2013)
17. Miranda, J.: A detailed description of the GNU Ada run time (2003). http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/
18. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
19. Wieder, A., Brandenburg, B.B.: On spin locks in AUTOSAR: blocking analysis of FIFO, unordered, and priority-ordered spin locks. In: Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, 3–6 December 2013, pp. 45–56 (2013). https://doi.org/10.1109/RTSS.2013.13
20. Zhao, S., Garrido, J., Burns, A., Wellings, A.: New schedulability analysis for MrsP. In: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10. IEEE (2017)

Improved Cache-Related Preemption Delay Estimation for Fixed Preemption Point Scheduling

Filip Marković(✉), Jan Carlson, and Radu Dobrin

School of Innovation Design and Technology (IDT), Mälardalen University, Västerås, Sweden
{filip.markovic,jan.carlson,radu.dobrin}@mdh.se

Abstract. Cache-Related Preemption Delays (CRPD) can significantly increase tasks' execution time in preemptive real-time scheduling, potentially jeopardising the system schedulability. In order to reduce the cumulative CRPD, Limited Preemptive Scheduling (LPS) has emerged as a scheduling approach which limits the maximum number of preemptions encountered by real-time tasks, thus decreasing CRPD compared to fully preemptive scheduling. Furthermore, an instance of LPS, called Fixed Preemption Point Scheduling (LP-FPP), defines the exact points where the preemptions are permitted within a task, which enables a more precise CRPD estimation. The majority of the research, in the domain of LP-FPP, estimates CRPD with pessimistic upper bounds, without considering the possible sources of over-approximation: (1) accounting for the infeasible preemption combinations, and (2) accounting for the infeasible cache block reloads. In this paper, we improve the analysis by accounting for those two cases towards a more precise estimation of the CRPD upper bounds. The evaluation of the approach on synthetic tasksets reveals a significant reduction of the pessimism in the calculation of the CRPD upper bounds, compared to the existing approaches.

Keywords: Real-time systems · CRPD analysis · WCET analysis
Limited Preemptive Scheduling · Fixed preemption point approach

1 Introduction

In preemptive real-time systems, each preemption causes a *preemption delay* which needs to be accounted in the timing analysis. The main part of preemption delay comes from the memory cache blocks which need to be reloaded from higher memory units before the preempted task is able to be executed again, and it is called Cache-Related Preemption Delay (CRPD). CRPD may increase the worst-case execution time of a task up to 33%, as shown by Pellizzoni et al. [1], thus making a significant impact on the system schedulability. Therefore, it is important to compute the upper bounds on this delay as tight as possible.

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 87–101, 2018.
https://doi.org/10.1007/978-3-319-92432-8_6

In order to control the CRPD, Limited Preemptive Scheduling (LPS) has emerged as a scheduling paradigm which has the ability to reduce the number of preemptions encountered by real-time tasks, thus reducing the CRPD experienced by the preempted tasks. LPS has been instantiated in several approaches, e.g., Preemption Thresholds Scheduling proposed by Wang and Saksena [2], Deferred Preemption Scheduling proposed by Baruah [3], and Fixed Preemption Points (LP-FPP), proposed by Burns [4]. LP-FPP facilitates the estimation of CRPD, compared to other LPS approaches, because in this approach each task is divided in non-preemptive regions, separated by offline predefined preemption points. Since the tasks can be preempted only at the predefined preemption points, this approach allows for a more precise CRPD estimation compared to the other LPS approaches, where preemption points are unknown before runtime. However, the existing works on LP-FPP use pessimistic CRPD estimations which may be significantly tightened through a more detailed CRPD analysis.

The computation of CRPD needs to take into account several factors, such as: (1) cache blocks belonging to a preempted task that may be evicted, called ***useful cache blocks*** (**UCB**), (2) cache blocks that are accessed by a preempting task which may evict useful cache blocks of a preempted task, called ***evicting cache blocks*** (**ECB**). Since LP-FPP considers predefined non-preemptive regions, those two types of cache blocks are static and known for each preemption point. Many papers in LP-FPP scheduling, e.g., [5–7] assume estimation of CRPD which accounts that all of the preempted points are preempted, and each point exhibits the maximum possible eviction scenario of their useful cache blocks. However, this over-provisional computation is very pessimistic compared to a tighter upper bound that can be computed by considering more realistic system behaviour.

There are two important sources of over-approximation that can be considered when estimating CRPD in the LP-FPP domain, and they are intertwined between the infeasible preemption combinations, and the infeasible reloads of the useful cache memory blocks, in more details:

– In a majority of the cases, some preemption combinations are infeasible, i.e., one task can rarely preempt another task instance at all preemption points.
– The existing CRPD analysis over-approximates the number of useful cache block (UCB) reloads in the preempted task. This number can be significantly reduced by analysing the cache block access throughout the task execution.

In this work, we address both potential sources of over-approximation jointly, accounting for the infeasible reloads and the infeasible preemption combinations. We achieve this by building on the constraint satisfaction model which we proposed in [8], thus even more significantly reducing the pessimism in the CRPD estimation.

The remainder of the paper is organised as follows: In Sect. 2 we describe the system model used in the paper. The motivating example is described in Sect. 3. The proposed method is described in Sect. 4, followed by the experimental results in Sect. 5. In Sect. 6 we enlist the related work, and we conclude the paper in Sect. 7.

2 System Model

While the proposed approach is independent of the underlying scheduler, without loss of generality, in this paper we consider a sporadic task model scheduled under the Fixed Priority paradigm on a single processor. A taskset Γ is composed of n tasks, ordered in a decreasing priority order, where the individual task τ_i is described with the following parameters: P_i, C_i, T_i, D_i. The priority of a task is denoted with P_i, the worst case execution time (WCET) without preemption delay with C_i, the minimum inter-arrival time of each task instance with T_i, and the relative deadline with D_i (Fig. 1).

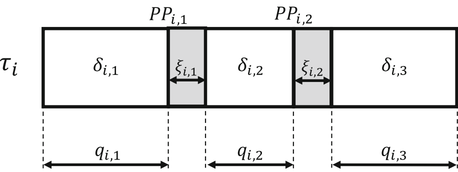


Fig. 1. Task with two preemption points ($PP_{i,1}$ and $PP_{i,2}$ with respective worst case CRPD values $\xi_{i,1}$ and $\xi_{i,2}$) which form three non-preemptive regions ($\delta_{i,1}$, $\delta_{i,2}$ and $\delta_{i,3}$ with respective worst case execution times $q_{i,1}$, $q_{i,2}$ and $q_{i,2}$)

Since in this paper we consider LP-FPP scheduling, each task is divided by d_i predefined preemption points. Individual preemption points are denoted with $PP_{i,k}$, where $1 \leq k \leq d_i$, and the maximum possible preemption delay at each point is denoted with $\xi_{i,k}$. The preemption points separate the task into $d_i + 1$ non-preemptive regions, denoted with $\delta_{i,k}$, each with WCET of $q_{i,k}$, such that $C_i = \sum_{k=1}^{d_i+1} q_{i,k}$.

In order to allow for a more precise CRPD estimation considering a direct-mapped cache, we extend the system model with detailed information about the cache usage within a task. Considering each preemption point $PP_{i,k}$, we define the following:

- $UCB_{i,k}$ – a set of *useful cache blocks* at $PP_{i,k}$. As proposed by Lee et al. [9], and superseded by Altmeyer et al. [10], a memory block m is in $UCB_{i,k}$, if and only if:
  (a) m must[1] be cached at $PP_{i,k}$, and
  (b) m may be reused on at least one control flow path starting at $PP_{i,k}$ without eviction[2] of m on this path.

[1] In the original definition, Altmeyer used *must* to eliminate the cases where the useful cache block eviction is accounted by both: WCET and CRPD analysis, while it should be accounted only by the CRPD analysis. However, a useful cache block still *may* be cached, but its eviction will not be accounted by both analysis.

[2] Refers only to self-eviction by τ_i.

Considering each non-preemptive region $\delta_{i,k}$, we define the following set:

- $ECB_{i,k}$ – a set of *evicting cache blocks* of $\delta_{i,k}$, such that $m \in ECB_{i,k}$ if and only if m may be accessed during the execution of $\delta_{i,k}$. The evicting cache block set of the whole task τ_i is defined with ECB_i, such that:

$$ECB_i = \bigcup_{k=1}^{d_i+1} ECB_{i,k}$$

3 Sources of CRPD Over-Approximation

Many papers in LP-FPP scheduling, e.g., [5–7] assume an estimation of CRPD which accounts for a worst case that all of the preemption points are preempted, and each point exhibits the maximum possible eviction scenario of their useful cache blocks. Thus, for a task τ_i, the worst case execution time with accounted CRPD is computed as: $C_i^{\gamma} = C_i + \sum_{r=1}^{d_i} \xi_{r,k}$. This computation can lead to a significant over-estimation. We describe the two cases which can be a source of the CRPD over-approximation: (1) accounting for the infeasible preemption combinations, (2) accounting for the infeasible useful cache block reloads.

3.1 Infeasible Preemptions

In Fig. 2 we show a task τ_2 with three preemption points and four non-preemptive regions with their worst case execution times. We also show a preempting task τ_1 with $C_1 = 20$ and $T_1 = 65$.

We show a scenario where it is obvious that if τ_1 preempts τ_2 at $PP_{2,1}$, then it cannot preempt also on $PP_{2,2}$. This is the case because the next instance of τ_1 cannot be released before the latest start time of the third non-preemptive region $\delta_{2,3}$. Let us assume that the instance of τ_2 started to execute. If the instance of τ_1 is released during the execution of $\delta_{2,1}$, thus definitely preempting at $PP_{2,1}$, can the next instance of τ_1 preempt τ_2 at $PP_{2,2}$? We first compute the maximum time interval between the start time of $\delta_{2,1}$ and the start time of $\delta_{2,3}$, and we get:

$$q_{2,1} + C_1 + \xi_{2,1} + q_{2,2} = 59$$

However, the next instance of τ_1 comes earliest 65 time units after the start of τ_2, which is less than the latest start time of $\delta_{2,3}$. Therefore, preempting at both $PP_{2,1}$ and $PP_{2,2}$ is infeasible. By analysing the other infeasible preemptions from the example shown in Fig. 2 we would see that τ_1 can preempt τ_2 only in two scenarios: (1) τ_1 preempts τ_2 at $PP_{2,1}$ and $PP_{2,3}$, or (2) τ_1 preempts τ_2 only at $PP_{2,2}$. Assuming that τ_2 is preempted by τ_1 at all three points is safe, but it might be a significant over-approximation. To find which of feasible preemption combinations results in a maximum CRPD is not a trivial problem, especially with the higher number of preempting tasks, and it is solved in Sect. 4.

τ_1 $q_{1,1} = 20$

$C_1 = 20,\ T_1 = 65$

$PP_{2,1}$ $PP_{2,2}$ $PP_{2,3}$

τ_2 $q_{2,1} = 20$ $q_{2,2} = 15$ $q_{2,3} = 15$ $q_{2,4} = 10$

$C_2 = 60,\ T_2 = 300$ $\xi_{2,1} = 4$ $\xi_{2,2} = 3$ $\xi_{2,3} = 2$

Fig. 2. A preempted task τ_2 with three preemption points ($PP_{2,1}$, $PP_{2,2}$ and $PP_{2,3}$), and four non-preemptive regions with worst case CRPD at each point. Top of the figure: preempting task τ_1 with $C_1 = 20$, and $T_1 = 65$.

3.2 Infeasible Useful Cache Block Reloads

In Fig. 3, we show the same tasks: τ_2 with its useful cache block sets, and the preempting task τ_1 with a set of its evicting cache blocks. Inside the non-preemptive regions we show the accessed memory blocks, e.g., during the first non-preemptive region of τ_3 those are: 1, 2, 3, and 4. Since they are all used in the remaining part of the task execution, they belong to the $UCB_{2,1}$ set. Notice that for the second non-preemptive region, the only accessed memory block is 1, but the $UCB_{2,2}$ is equal to $\{2, 3, 4\}$ since the memory blocks 2, 3, and 4 are cached and may be reused in the remaining part of the task execution.

Let us ignore for now the analysis from the previous example and assume that τ_1 may preempt τ_2 at all three preemption points. Considering memory block 2, it may be evicted when τ_1 preempts τ_2 at $PP_{2,1}$, or at $PP_{2,2}$. The existing CRPD analysis would account for two reloads, because memory block 2 is in the useful cache block sets of both points and in ECB_1 of τ_1, but this is an over-approximation, since the analysis would over-approximate the number of reloads. The same holds for memory blocks 3 and 4, since τ_1 may evict them at any of the three points, but at most one reload should be accounted for.

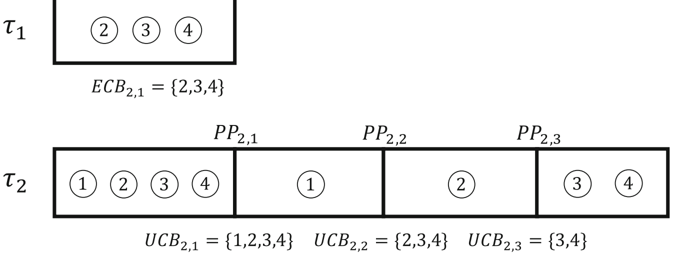


Fig. 3. A preempted task τ_1 with three preemption points with defined UCB sets ($UCB_{2,1}$, $UCB_{2,2}$, and $UCB_{2,3}$), and four non-preemptive regions. The cache block accesses throughout the task execution are shown as circled memory block numbers inside the task.

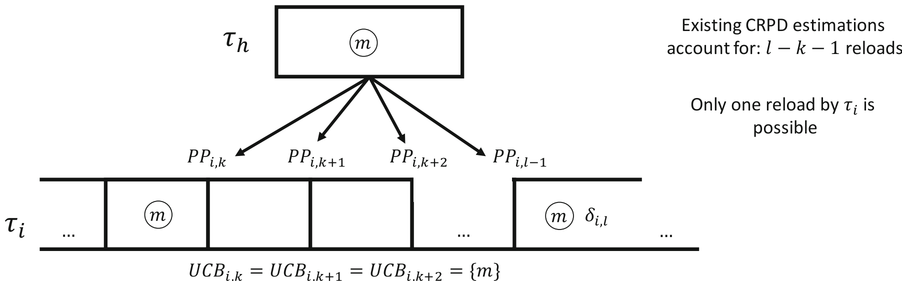


Fig. 4. A preempting task τ_i with a memory block m accessed before preemption point $PP_{i,k}$ and re-accessed at $\delta_{i,l}$. Top: A preempted task τ_h that evicts m and preempts τ_i at all preemption points between $PP_{i,k}$ and $\delta_{i,l}$.

Accounting for the precise number of the useful cache block reloads is the main problem that we address in this paper.

In order to reduce the over-approximation of the useful-cache block reloads in the CRPD estimation, we use the fact that:
Between two accesses of a memory block m, by a task τ_i, at most one reload should be accounted for in the CRPD analysis for τ_i. More detailed:

Let $UCB_{i,k}$ be a useful cache block set at $PP_{i,k}$ of a preempted task τ_i, and let m be a memory block such that $m \in UCB_{i,k}$. Then at most one reload of m should be accounted by CRPD for τ_i, until the first succeeding non-preemptive region $\delta_{i,l}$, where $k < l$, such that $m \in ECB_{i,l}$.

The existing papers in the domain of CRPD estimation account for the reloads based on useful-cache blocks at the preemption points, thus they would account for the $l - k - 1$ cache block reloads from $PP_{i,k}$ to $\delta_{i,l}$. Although m may be useful at all preemption points between two consecutive accesses of τ_i, accounting that τ_i reloaded m at each preemption point can be a significant over-approximation (see Fig. 4).

4 Computing Tighter CRPD Bounds

In this paper we address both sources of over-approximation that are described in the previous section. The previously proposed CRPD method [8] for LP-FPP task model addressed the problem of infeasible preemptions by using a constraint satisfaction model. However, that solution may result in over-approximating CRPD values since it does not consider infeasible reloads. Therefore, in order to jointly solve both problems, we improve the previously proposed constraint satisfaction model such that it even further reduces the pessimism in the resulting CRPD values. The overview of the method is described in Algorithm 1. The algorithm considers a taskset Γ with tasks ordered in a decreasing priority order. For each task τ_i it computes the CRPD γ_i and updates its WCET which accounts for the CRPD C_i^{γ}. The constraint satisfaction model is defined in three steps:

1. First, we generate the variables that represent the possible preemptions from the preempting tasks at different preemption points of the preempted task.
2. Second, we generate the constraints which define infeasible preemption combinations.
3. Third, we generate a goal function which accounts for the infeasible useful cache block reloads, described in Sect. 3, jointly accounting for the infeasible preemption combinations.

In the following subsections we use the example defined (in Sect. 3) in Figs. 2 and 3.

Data: Task set Γ
Result: Set γ of CRPD values for each task from Γ

```
1  γ ← ∅
2  for i ← 2 to n do
3      V_i ← Generate variables that represent all preemptions on τ_i
4      C_i ← Generate constraints which define infeasible preemption combinations
5      G_i ← Generate a goal function
6      γ_i ← Compute CRPD bound solving the constraint problem: V_i, C_i and G_i
7      γ ← γ ∪ γ_i
8      C_i^γ ← C_i + γ_i
9  end
10 return γ
```

Algorithm 1. Algorithm for tightening the upper bounds of CRPD in a taskset.

4.1 Variables

For each preempted task τ_i, we generate $d_i \times (i-1)$ boolean variables, where d_i is the number of preemption points of τ_i, and $(i-1)$ is a number of higher priority (preempting) tasks. Each boolean variable $X_{h,k}$ represents the case where an instance of the preempting task τ_h can affect the preemption cost of the preemption point $PP_{i,k}$. The set V_i of boolean variables is formally defined as:

$$V_i = \left\{X_{h,k} \in \{0,1\} \;\middle|\; (1 \leq h < i) \wedge (1 \leq k \leq d_i)\right\}$$

Considering the running example from Figs. 2 and 3, the set V of τ_2 is:

$$V_2 = \left\{X_{1,1} \in \{0,1\},\; X_{1,2} \in \{0,1\},\; X_{1,3} \in \{0,1\}\right\}$$

4.2 Constraints

The constraints describe the infeasible preemption combinations. Considering two instances of a preempting task τ_h and their potential preemptions at two preemption points $PP_{i,k}$ and $PP_{i,l}$ $(1 \leq l \leq d_i)$ of the preempted task τ_i, we showed in [8] that the following proposition holds:

Proposition 1. *If the maximum time interval from the start time of $\delta_{i,k}$ until the start time of $\delta_{i,l+1}$ is less or equal to the minimum inter-arrival period T_h then task τ_h cannot affect one instance of τ_i at both preemption points $PP_{i,k}$ and $PP_{i,l}$.*

The maximum time interval $I_i^{k,l}$ from the start time of $\delta_{i,k}$ until the start time of $\delta_{i,l+1}$ is defined as the least fixed point of the following recursion:

$$\begin{cases} I_i^{k,l}(0) = \sum\limits_{w=k}^{l}(q_{i,w} + \xi_{i,w}) + \sum\limits_{h \in hp(\tau_i)} C_h^{\gamma} \\ I_i^{k,l}(z) = \sum\limits_{w=k}^{l}(q_{i,w} + \xi_{i,w}) + \sum\limits_{h \in hp(\tau_i)} \left(\left\lfloor \frac{I_i^{k,l}(z-1)}{T_h} \right\rfloor + 1 \right) C_h^{\gamma} \end{cases}$$

Therefore, we generate a constraint whenever the inequality $I_i^{k,l} \leq T_h$ holds. Since the constraint expresses the case when τ_h cannot affect τ_i at the two points $PP_{i,k}$ and $PP_{i,l}$, it has the following form:

$$X_{h,k} + X_{h,l} \leq 1$$

Meaning that only one of the two preemptions is possible, $X_{h,k}$ or $X_{h,l}$. The whole set C_i of constraints is formally defined as:

$$C_i = \left\{ X_{h,k} + X_{h,l} \leq 1 \;\middle|\; (1 \leq h < i) \wedge (1 \leq k < l \leq d_i) \wedge (I_i^{k,l} \leq T_h) \right\}$$

Considering the running example from Figs. 2 and 3, we compute that: $I_2^{1,2} = 62$, and $62 \leq T_1$; $I_2^{2,3} = 55$, and $55 \leq T_1$; and $I_2^{1,3} = 99$, and $99 > T_1$. Considering this, the set C_2 of τ_2 is:

$$C_2 = \{ \underset{\substack{\uparrow \\ I_2^{1,2} \leq T_1}}{X_{1,1} + X_{1,2} \leq 1}; \quad \underset{\substack{\uparrow \\ I_2^{2,3} \leq T_1}}{X_{1,2} + X_{1,3} \leq 1} \}$$

As discussed previously, the scenario where the instances of τ_1 preempt τ_2 at both $PP_{2,1}$ and $PP_{2,2}$ is not possible. Also, the scenario where τ_1 preempts both $PP_{2,2}$ and $PP_{2,3}$ is not possible. Preempting at $PP_{2,1}$ and $PP_{2,3}$ is however possible, as are the scenarios of preempting only at one or none of the three preemption points.

4.3 Goal Function

Finally, we define the goal function of the constraint satisfaction model. The goal function computes the maximum CRPD considering the infeasible preemption combinations defined by the constraints, but also accounts for the precise number of useful cache block reloads which should be accounted by the CRPD estimation.

In order to account for the precise maximum number of useful cache block reloads, we generate a goal function based on observation from Sect. 3.2. For

each useful cache block m at $PP_{i,k}$ ($m \in UCB_{i,k}$) we first need to define a sequence of the non-preemptive regions during which m can be reloaded at most once by τ_i.

Therefore, we define the set $A(i,k,m)$, which for a task τ_i denotes a sequence of preemption points from $PP_{i,k}$ during which we need to account for at most one reload of m. Formally:

$$A(i,k,m) = \{l \mid k < l \leq d_i \wedge \forall l'(k < l' \leq l) \Rightarrow m \notin ECB_{i,l'}\} \tag{1}$$

Considering the example in Fig. 3, $A(2,3,1) = \{2,3\}$, since these preemption points ($PP_{i,2}$, and $PP_{i,2}$) succeed $PP_{i,1}$, and precede the non-preemptive region ($\delta_{i,4}$) where memory block 3 is re-accessed by τ_i. The CRPD estimation should account for maximum one reload of memory block 3 even if there are preempting tasks that may evict m at several of the preemption points.

We propose the following goal function in order to account for both the infeasible preemptions, and the precise number of useful cache block reloads:

$$G_i = Maximize:$$

$$\sum_{PP_{i,k} \in \tau_i} \sum_{m \in UCB_{i,k}} \Bigg[\min\Big(1, \sum\{X_{h,k} \mid \tau_h \in hp(\tau_i) \ \wedge m \in ECB_h\}\Big)$$

$$\times \Big(1 - \min\Big(1, \sum_{r \in A(i,m,k)} \sum\{X_{h,r} \mid \tau_h \in hp(\tau_i) \ \wedge m \in ECB_h\}\Big)\Big)\Bigg] \times BRT$$

The goal function consists of two expressions which compute the maximum number of necessary cache block reloads for τ_i, which multiplied by the block reload time BRT results into the upper bound γ_i on the CRPD.

The first expression (line 1) of the goal function iterates over the useful cache blocks of the preemption points, one by one, accounting for the evictions from the higher priority tasks at those specified points. The minimum function accounts for at most one eviction of a memory block, even if it is in the evicting cache block set of more than one higher priority task. For a single memory block m, where $m \in UCB_{i,k}$ at $PP_{i,k}$, the minimum function on line one will result in:

- ◇ 1: if there is at least one preemption from a task with m in its evicting cache block set.
- ◇ 0: if there are no preemptions from the preempting tasks with m in their evicting cache block sets.

The second expression (line 2) of the goal function accounts for the precise number of useful cache block reloads. It computes if there is an eviction of memory block m, from $PP_{i,k}$ until the first following non-preemptive region where m is re-accessed, given by $A(i,k,m)$. If m is evicted during this interval, the expression results in 1, otherwise it results in 0.

Let us consider a memory block m at the specified preemption point $PP_{i,k}$ such that $m \in UCB_{i,k}$, and let us assume that m is re-accessed by τ_i at $\delta_{i,l+1}$. Next, we need to account for only one reload of m, from $PP_{i,k}$ until $\delta_{i,l+1}$. The goal function will result in:

- ⋄ 0: if there are no feasible preemptions from the preempting tasks with m in their evicting cache block sets. This is the case because the first expression results in 0 and it is multiplied with the second expression.
- ⋄ 0: if m is in the evicting cache block set of at least one task that preempts at $PP_{i,k}$, but also in the evicting cache block set of at least one task that preempts from $PP_{i,k+1}$ until $PP_{i,l}$. This is the case because both of the expressions result in 1, which finally results in: $1 \times (1 - 1) = 0$.
- ⋄ 1: if m is in the evicting cache block set of at least one task that preempts at $PP_{i,k}$, and there are no further preemption points until $\delta_{i,l+1}$ where m is evicted. This is the case because the first expression results in 1 and the second expression results in 0, which finally results in: $1 \times (1 - 0) = 1$.

For any memory block m, the formulation will account for at most one reload from $PP_{i,k}$ until $\delta_{i,l+1}$, accounting it in the last preemption point where m is evicted during this interval. Reloads of the previous preemption points where m is evicted in this interval will not be accounted for since they will result in 0.

Let us take for example memory block 3 from Figs. 2 and 3. First time it is accessed by τ_2 is at the non-preemptive region $\delta_{2,1}$. Then, it is re-accessed by τ_2 at $\delta_{2,4}$. It can be evicted either at $PP_{2,1}$ or $PP_{2,3}$, as shown in Sect. 4.2, but the analysis should account for only one reload. The goal function for this memory block at all preemption points is shown bellow.

$$
\begin{array}{lccccccr}
G = Maximize : BRT \times \Big(& & & & & & & \\
 & ... & + & ... & + \; X_{1,1} \times (1 - \min(1, X_{1,2} + X_{1,3})) & + & ... & \leftarrow k = 1 \\
 & ... & + & ... & + \quad X_{1,2} \times (1 - \min(1, X_{1,3})) & + & ... & \leftarrow k = 2 \\
 & ... & + & ... & + \quad X_{1,3} \times (1 - 0) & + & ...\Big) & \leftarrow k = 3 \\
 & \uparrow & & \uparrow & \uparrow & & \uparrow & \\
 & m = 1 & & m = 2 & m = 3 & & m = 4 &
\end{array}
$$

We omitted the remaining parts of the nested sum for other m and k values of the goal function. Considering the memory block 3 and its possible eviction at $PP_{i,1}$ we get that it is $X_{1,1} \times (1 - \min(1, X_{1,2} + X_{1,3}))$. This means that a preemption at $PP_{2,1}$ contributes to the CRPD only if there is no preemption at $PP_{2,2}$ or $PP_{2,3}$. Similar formulation holds for the same block at $PP_{2,2}$ and $PP_{2,3}$, except in case of $PP_{2,3}$ the set A(2,3,3) results in 0, since there are no preemption points after $PP_{2,3}$ where memory block 3 can be useful.

5 Evaluation

In the following experiments we used the open-source constraint programming solver Choco [11] on a device with 2,9 GHz Intel Core i5 processor and 8 GB 1867 MHz DDR3 RAM memory. In the experiments we evaluated three methods for CRPD estimation: the method proposed in this paper, the method which accounts only for the infeasible preemption combinations [8], and the over-approximation which accounts for the maximum eviction at all preemption points.

Experiment Setup: In all of the experiments, tasksets are generated with the fixed utilisation of 0.8, using the U-unifast algorithm [12]. We randomly generated the minimum inter-arrival times from the uniform distribution $[5ms, 5s]$, as it reasonably corresponds to real systems. The worst case execution times are calculated such that: $C_i = T_i \times U_i$, and the priorities are assigned considering the rate monotonic order. We also randomly generated a number of non-preemptive regions from uniform distribution from 1 until 10.

Regarding the cache setup we used the one described by Altmeyer et al. [13], where memory blocks are represented with integer values from 0 to 256, and the maximum cache size CS is 256. Block reload time BRT is set to 8 µs. The evicting cache block set of a task ECB_i is generated using U-unifast algorithm such that $CU = \sum_1^n |ECB_i|/CS$, where $|ECB_i|$ is the number of cache blocks in the ECB_i. In cases when the algorithm returned values above 1, we assigned all cache blocks to ECB_i. Useful cache block set UCB_i of a task is randomly generated from ECB_i using the reload factor RF, representing the assumed reuse factor which in real system varies from very low (0%), to high (30%). Therefore, $UCB_i = RF \times |ECB_i|$, where RF is uniformly generated from the range $[0, 0.3]$. Next, we randomly generated useful cache block sets $UCB_{i,k}$ for each non-preemptive region from uniform distribution $[0, 100 \times |UCB_i|]$, and accordingly added evicting cache blocks such that the sequence of the consecutive useful cache block sets is bounded by two memory block accesses.

First Experiment: In the first experiment, we evaluated the CRPD estimation average over the generated tasksets, varying the total cache utilization from 20% to 90% (see Fig. 5), and the number of tasks in each taskset was fixed to 10. We show the estimations of the CRPD upper bounds computed by the method proposed in this work IPR (standing for accounting the infeasible preemptions and reloads), method proposed in [8] IP (standing for accounting only the infeasible preemptions), and the over-approximation OA which accounts for the maximum eviction at all preemption points. As expected, IPR estimates considerably less pessimistic upper bounds compared to IP and OA. Moreover, it is evident that the benefit of using IPR is even larger with the increase of cache utilisation

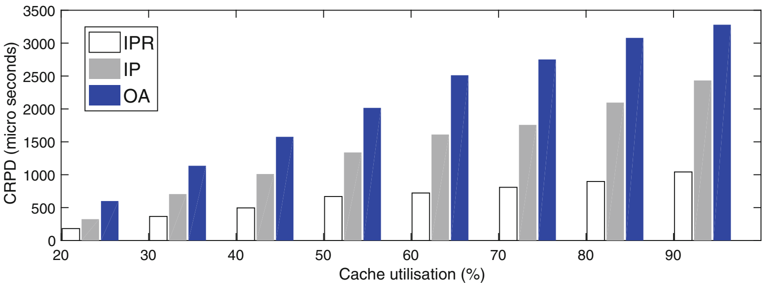


Fig. 5. CRPD estimation per taskset for different levels of cache utilization, calculated as the average over the 2000 generated tasksets.

compared to IP. For example, when $CU = 20\%$, the average taskset CRPD estimation using IPR is 181 µs, which is significantly reduced compared to the estimation of 324 µs when using IP (by 44%), and the estimation of 601 µs when using OA (by 70%). When $CU = 90\%$, the average taskset CRPD estimation using IPR is 1040 µs, which is a reduction ratio of 58% compared with the bound of 2432 µs when using IP, and 70% compared to the estimation of 3280 µs when using OA. This is the case since by increasing cache utilisation, more preempting tasks share the evicting cache blocks, which furthermore increases the number of infeasible reloads that should be accounted by the CRPD analysis.

Second Experiment: In the second experiment, we evaluated the CRPD estimation average over the generated tasksets, varying the number of tasks from 3 to 10 (see Fig. 6). In this experiment, we fixed the total cache utilisation to 40%. The results reveal that the IPR dominates the other two approaches (IP and OA) even when increasing the number of tasks in a taskset. This is the case since by increasing the number of tasks, we also increase the number of preempting tasks and thus the number of possible preemptions. For example, when $n = 7$, the average taskset CRPD estimation using IPR is 470 µs, which is a reduction of 48% compared to the bound of 896 µs when using IP, and even 69% compared to the estimation of 1560 µs when using OA. However, the average analysis time increases with the number of tasks in a taskset, from 48 ms when $n = 3$, to 2772 ms when $n = 10$. Since the analysis time varies a lot among different cases, we used a time limit of 40 s per taskset analysis, and if it failed to provide a value within this time bound, it instead reported the CRPD value computed with OA. However, the proposed solution is an offline method and the analysis time can be significantly improved by using different solvers.[3]

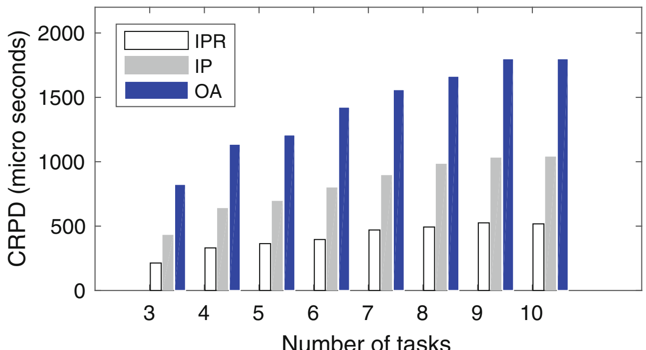


Fig. 6. CRPD estimation per taskset, for different taskset sizes, calculated as the average over the 2000 generated tasksets.

[3] We extracted a few of the most time-consuming (more than 40 s) cases from the evaluation, and with IBM CPLEX [14] they were solved in less than 200 ms.

6 Related Work

The majority of the work about CRPD analysis considers only the fully preemptive real-time systems. The two dominant approaches for computing the CRPD were UCB-union [15] and ECB-union (extended from the ECB-only approach proposed by Busquets-Mataix et al. [16], and later Tomiyama and Dutt [17]). In these approaches it is assumed that the evicting cache block sets of the preempting tasks always definitely evict the useful cache blocks of the preempted task in a same manner, without consideration that the preemptions might result in a smaller CRPD values than the prior accounted ones. In order to account for this possibility, Staschulat et al. [18] proposed a CRPD analysis based on multisets. However, their approach over-approximates the number of intermediate preemptions between the preempting tasks and also the number of preemptions that may impact on the response time of the preempted task. These sources of over-approximation were later addressed by Altmeyer et al. [19] using the ECB- and UCB-Union multiset approaches. However, neither of those works accounts for the infeasible preemptions. Ramaprasad and Miller [20] analysed the feasible preemptions considering the worst case placement of preemptions on each job of a preempted task throughout the hyper-period. However, their work addresses only periodic task systems.

Considering LP-FPP approaches, the majority of the proposed schedulability analysis, e.g., [5–7] use the over-approximation that accounts for the worst case eviction at each preemption point. Cavicchio et al. [21] improved the analysis for such systems, considering the preemption point selection that uses a detailed CRPD computation for each pair of adjacent preemption points, therefore being able to significantly improve the CRPD precision and reduce the pessimism in the analysis compared to previous works. However, they did not account for the infeasible preemption combinations and the infeasible useful cache block reloads, which may be a source of a significant over-approximation in LP-FPP real-time systems, which are addressed in this paper.

7 Conclusions

In this paper, we proposed an improved Cache-Related Preemption Delay (CRPD) analysis for sporadic real-time systems under Fixed Preemption Point Scheduling, which reduces the pessimism in CRPD estimation compared to previous approaches. We first identified two potential sources of CRPD over-approximation: (1) infeasible preemption combinations, and (2) infeasible useful cache block reloads. In order to address those problems and compute more precise CRPD upper bounds, we proposed a constraint satisfaction model. The evaluation results show that the proposed approach significantly reduces the upper bounds on CRPD estimation, compared to previous methods.

In future work, we will investigate the proposed method integrating it with the schedulability analysis for LP-FPP systems, also evaluating the potential benefits of its use considering different task and cache parameters. We will also

address fully preemptive real-time systems, considering the identified sources of over-approximation which hold for such systems as well.

References

1. Pellizzoni, R., Bui, B.D., Caccamo, M., Sha, L.: Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In: Real-Time Systems Symposium, pp. 221–231. IEEE (2008)
2. Wang, Y., Saksena, M.: Scheduling fixed-priority tasks with preemption threshold. In: Sixth International Conference on Real-Time Computing Systems and Applications, RTCSA 1999, pp. 328–335. IEEE (1999)
3. Baruah, S.: The limited-preemption uniprocessor scheduling of sporadic task systems. In: 17th Euromicro Conference on Real-Time Systems (ECRTS 2005), pp. 137–144. IEEE (2005)
4. Burns, A., Son, E.S.: Preemptive priority based scheduling: an appropriate engineering approach. In: Advances in Real-Time Systems, pp. 225–248 (1994)
5. Bertogna, M., Xhani, O., Marinoni, M., Esposito, F., Buttazzo, G.: Optimal selection of preemption points to minimize preemption overhead. In: 2011 23rd Euromicro Conference on Real-Time Systems (ECRTS), pp. 217–227. IEEE (2011)
6. Buttazzo, G.C., Bertogna, M., Yao, G.: Limited preemptive scheduling for real-time systems. A survey. IEEE Trans. Ind. Inform. **9**(1), 3–15 (2013)
7. Peng, B., Fisher, N., Bertogna, M.: Explicit preemption placement for real-time conditional code. In: 2014 26th Euromicro Conference on Real-Time Systems (ECRTS), pp. 177–188. IEEE (2014)
8. Markovic, F., Carlson, J., Dobrin, R.: Tightening the bounds on cache-related preemption delay in fixed preemption point scheduling. Presented at the 17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017). OASIcs-OpenAccess Series in Informatics, vol. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
9. Lee, C.-G., Han, J., Seo, Y.-M., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M., Kim, C.S.: Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans. Comput. **47**(6), 700–713 (1998)
10. Altmeyer, S., Burguiere, C.: A new notion of useful cache block to improve the bounds of cache-related preemption delay. In: 21st Euromicro Conference on Real-Time Systems, ECRTS 2009, pp. 109–118. IEEE (2009)
11. CHOCO: Open Source Java Library for Constraint Programming. http://www.choco-solver.org/. Accessed 13 Apr 2017
12. Bini, E., Buttazzo, G.C.: Measuring the performance of schedulability tests. Real Time Syst. **30**(1–2), 129–154 (2005)
13. Sebastian, A., Roeland, D., Will, L., Robert, I.D.: Evaluation of cache partitioning for hard real-time systems. In: Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), pp. 15–26 (2014)
14. IBM ILOG CPLEX. https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer. Accessed 25 Feb 2018
15. Tan, Y., Mooney, V.: Timing analysis for preemptive multitasking real-time systems with caches. ACM Trans. Embed. Comput. Syst. (TECS) **6**(1), 7 (2007)
16. Busquets-Mataix, J.V., Serrano, J.J., Ors, R., Gil, P., Wellings, A.: Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In: Proceedings of 1996 IEEE Real-Time Technology and Applications Symposium, pp. 204–212. IEEE (1996)

17. Tomiyama, H., Dutt, N.D.: Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In: Proceedings of the Eighth International Workshop on Hardware/Software Codesign, pp. 67–71. ACM (2000)
18. Staschulat, J. Schliecker, S., Ernst, R.: Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005), pp. 41–48. IEEE (2005)
19. Altmeyer, S., Davis, R.I., Maiza, C.: Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. Real Time Syst. **48**(5), 499–526 (2012)
20. Ramaprasad, H., Mueller, F.: Tightening the bounds on feasible preemptions. ACM Trans. Embed. Comput. Syst. (TECS) **10**(2), 27 (2010)
21. Cavicchio, J., Tessler, C., Fisher, N.: Minimizing cache overhead via loaded cache blocks and preemption placement. In: 2015 27th Euromicro Conference on Real-Time Systems (ECRTS), pp. 163–173. IEEE (2015)

Real-Time Scheduling

Combined Scheduling of Time-Triggered and Priority-Based Task Sets in Ravenscar

Jorge Real[1(✉)], Sergio Sáez[2], and Alfons Crespo[1]

[1] Instituto de Automática e Informática Industrial,
Universitat Politècnica de València, Camí de Vera, s/n, 46022 València, Spain
{jorge,alfons}@disca.upv.es

[2] Instituto Tecnológico de Informática,
Universitat Politècnica de València, Camí de Vera, s/n, 46022 València, Spain
ssaez@disca.upv.es

Abstract. Time-triggered and priority-based are the two major approaches for scheduling real-time systems. Both have their own advantages and drawbacks and none is superior in the general case. While time-triggered schedules excel at determinism and jitter control, they are hard to design and lack flexibility. Priority-based scheduling, on the other hand, keeps the logical and timing aspects of real-time applications conveniently separated from each other, at the cost of indeterminism and larger input and output jitter for all but the highest-priority tasks.

In a previous paper, we presented a model and a related Ada implementation to support the combined execution of time-triggered and priority-based task sets, aiming to obtain the best of both worlds. This paper presents continuation of that work in two directions. One is the extension of the original model to support more behavioural patterns; the other is providing a Ravenscar implementation, targeting high-integrity systems. We conclude that Ravenscar is expressive enough to support most of the patterns in the original full-Ada version, and those that require forbidden features (such as dynamic priorities) are not out of reach if the time-triggered scheduler is implemented at the runtime level.

Keywords: Real-time systems · Time-triggered scheduling
Ravenscar profile · High-integrity systems · Embedded systems

1 Introduction

There are two major approaches to real-time task scheduling. One is time-triggered (TT) scheduling, whereby each task is executed during the time intervals dictated by a fixed plan, designed in advance. The other is priority-based

This work has been partly supported by the Spanish Government's project M2C2 (TIN2014-56158-C4-1-P-AR) and the European Commission's projects ENABLE-S3 and AQUAS (ECSEL-JU, Contracts 692455 and 737475).

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 105–122, 2018.
https://doi.org/10.1007/978-3-319-92432-8_7

(PB) scheduling, where the intervals when a task executes are decided at run time, based on the task's priority. The priority of a task can in turn be static or dynamic, leading to several variations of the idea. Another significant variation of PB scheduling schemes stems from whether they are preemptive or not.

There are pros and cons to both approaches. TT scheduling is superior in terms of predictability, correctness by design, runtime simplicity and reduced jitter (i.e., precise and rapid task release), but a TT schedule is difficult to design for non-trivial cases – actually, it is an NP-complete problem [1]. Another interesting property of TT systems is that access to shared resources is simpler, provided all tasks execute non-preemptively; whereas PB preemptive systems require mechanisms to enforce mutual exclusion in the access to shared resources. A fundamental advantage of PB over TT scheduling is that it keeps a convenient separation of concerns between timing and functional requirements, making PB systems easier to modify and maintain. Sporadic and aperiodic tasks are also more naturally incorporated in PB systems than they are in TT systems, which are bound to using polling schemes.

In previous papers we have proposed an architecture to support applications that include a mix of TT and PB workload [2,3]. The idea was to divide the application into two subsets of tasks, one scheduled according to a TT plan, and the other one scheduled by a PB scheduler. The implementation of this architecture reserves the highest priority of a PB scheduler to the TT subset of tasks, and lets the PB subset execute during the spare time left by the TT workload, using the rest of priority levels. The TT schedule is driven by an Ada timing event handler (like a timer interrupt handler), which allows the system to react promptly to the arrival of time events, hence keeping a reduced release jitter for TT tasks.

Combining TT and PB scheduling helps mitigating their drawbacks and taking advantage of their benefits. For example, the TT load can be reduced to just the set of more jitter-sensitive tasks (such as control or communication tasks), hence leading to simpler plan design; whereas the rest of tasks (logging, user interface, optimisation tasks,...) can benefit from the advantages of PB scheduling. The price to pay with this combined scheduling approach is that it requires an underlying PB scheduler, which cannot be as efficient as a TT scheduler, since it has to deal with context switches between tasks. In summary, a combined TT-PB scheduling scheme gives good results for jitter control, slot-based communication and reuse of pre-designed TT schedules. The paper [2] positions this approach with respect to other methods pursuing similar goals.

In this paper, we adapt the technique proposed in [2,3] to the Ravenscar profile [4], towards facilitating its adoption in high-integrity, certifiable, real-time and embedded systems, which are target niches of Ravenscar. The relative simplicity of a Ravenscar scheduler (compared to a full-Ada scheduler), would reduce the overhead imposed by the underlying PB scheduler. Despite the restrictions of Ravenscar compared to the full-Ada tasking model, we have found it possible to replicate and even extend the model with TT mechanisms that we had not considered previously, which in turn enable new TT task patterns.

These new features, however, require the TT support to be implemented at the runtime level, rather than a user level library, hence our proposal for a new Ada package `Ada.Dispatching.TTS`.

The rest of this paper is organised as follows. Section 2 describes the system model. Section 3 proposes a variety of behavioural patterns for TT tasks, and in Sect. 4 we describe design and implementation aspects of the TT scheduler. We present experimental results in Sect. 5, focusing on jitter measurements obtained from running the TT scheduler on an embedded board using ARM's STM32F407VGT6 micro-controller unit. We finally give our conclusions in Sect. 6.

2 System Model: The Time-Triggered Plan

Our system model combines two disjoint subsets of tasks: one subset scheduled according to an offline, static TT plan, and the other subset scheduled under a PB, preemptive scheduler. Both subsets run on a common PB scheduler, but the TT workload uses a higher priority level than any task in the PB subset. Consequently, PB tasks do not interfere the execution of the TT plan[1]. We will therefore limit ourselves to describing the system model for the TT plan. Note that, since we are aiming at a Ravenscar implementation of the model, the PB subset can only be scheduled under a fixed-priority scheme such as Rate Monotonic or Deadline Monotonic [5,6]; but nothing in our model precludes the use of dynamic priority algorithms such as EDF [5] for the PB subset, or even a combination of schedulers using different priority bands, if the underlying runtime supported the full-Ada tasking model.

A TT plan is a cyclic sequence of actions to be executed at particular points in time. The plan is described by means of an ordered list of *time slots*, each of its own *slot duration*. If a slot starts at time t, its lifetime goes from t to t+*slot duration*. There are no gaps between slots: each slot starts just at the end of the previous slot in the plan. In other words, the duration of the plan is the sum of slot durations.

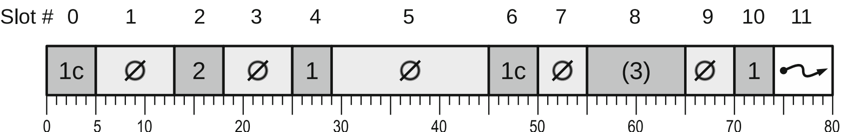


Fig. 1. A 12-slot time-triggered plan. Slots with sequence numbers 2, 4 and 10 are regular slots for works 1 and 2, as indicated; slots 0 and 6 are continuation slots for work 1; slot 8 is an optional slot for work 3; and slot 11 is a mode-change slot. The rest are slots.

Figure 1 shows a 12-slot example plan, with slots sequentially numbered from 0 to 11. Using the time scale in milliseconds at the bottom of the plan, it can be

[1] Blocking is possible between TT and PB tasks if they share resources, but not interference. This aspect is further discussed in Sect. 3.

seen that the plan has a duration of 80 ms, slot 0 has a duration of 5 ms, slot 11 takes 6 ms from time 74 ms to 80 ms, etc. There are five possible types of slots in a plan, all of them represented in Fig. 1:

- A ***regular slot*** defines a time interval reserved for the execution of a TT task (a *work*). It is denoted by a *regular Work_Id*, a positive integer value that identifies the particular work to execute during the slot duration. The underlying TT scheduler will make the work start to execute as soon as feasible after the start time of the slot. In Fig. 1, slots 2, 4 and 10 are regular slots corresponding to works 1 or 2 as indicated.
  The duration of a regular slot must be sufficient, by design, to accommodate the worst-case execution time of the work it serves. If a work overruns its regular slot then the scheduler will resort to raising a Program_Error exception, since an overrun violates the schedulability assumptions of TT scheduling. If, on the contrary, a work completes before the end of the slot duration, then the rest of the slot remains available for PB tasks. A TT task must always be ready to use its allocated regular slots in the plan. Failing this, the scheduler will raise Program_Error as well. The following type of slot is more permissive in this regard.
- An ***optional slot*** is like a regular slot except that it can be omitted. A TT task may decide to use or not to use an assigned optional slot in the plan. If it does use it (the task is ready to start when the optional slot starts) then it has the same semantics as a regular slot, including overrun control at the end of the slot. But if the task is not waiting for the start of the slot when it starts, it is not considered an error and the slot duration is made available for PB tasks. In Fig. 1, slot 8 is an optional slot for work 3, indicated with parentheses.
  Optional slots are useful for tasks that may or may not require to use their allocated slot, such as a communication task when it has nothing to say; or a sporadic task whose activation event has not occurred.
- A ***continuation slot*** can be regarded as a special kind of regular slot, in the sense that it is associated to a particular Work_Id. In Fig. 1, slots 0 and 6 are continuation slots. They are marked with a regular Work_Id plus a letter ‘c’, indicating *continuation*.
  What is special about these slots is that the work they host does not need to be completed by the end of the slot: it can be continued in future slots. Failing to finish by the end of a continuation slot is not an overrun. Instead, the work is held at the end of the slot and resumed at the start of the next slot in the plan that is marked with its Work_Id. There may be a number of consecutive continuation slots for a given work. Overrun will only be checked when the plan reaches a regular slot for this work. We will refer to the last, non-continuation slot of a series, as a *terminal slot*. In Fig. 1, slots 4 and 10 are terminal slots for work 1, given that they are preceded by continuation slots 0 and 6 for work 1, respectively.
  This type of slot is useful to split a large TT task into smaller pieces in a way that is essentially transparent to the task code. We will visit this pattern

in Sect. 3. Continuation slots require asynchronously holding and resuming a running task, which in turn requires support from the runtime system. This is the reason why our implementation of the TT scheduler (Sect. 4) is an extension of the runtime system, in the form of a new package Ada.Dispatching.TTS. The hold/release mechanism is indeed to be taken very carefully, specially with regard to its interaction with protected actions[2]. But it is doable under certain, controlled restrictions as we will show.

The following two types of slots correspond to scheduler actions exclusively and they have no associated TT task to execute, hence they carry no Work_Id value.

- An ***empty slot*** defines a time interval during which no TT work is planned. This is useful for inserting gaps in the plan to make the CPU available to PB tasks. Even though there is no TT task to execute during an empty slot, there will be scheduler actions executed at the beginning of the slot, as described in Sect. 4. In Fig. 1, slots 1, 3, 5, 7 and 9 are empty slots.
- A ***mode-change slot*** is similar to an empty slot in the sense that it has no associated work to execute. But additionally, it defines a time in the plan where it is possible to substitute the current plan with a new one. By placing mode change slots in the plan, the designer determines the exact points in the plan where a mode change can occur. If there is a pending mode-change request to process at the start of a mode-change slot, then the new plan will start at the end of the slot. This ability to change mode at defined points in time introduces a degree of flexibility that off-line, static schedules do not possess by nature. In Fig. 1, slot 11 is a mode-change slot, indicated with a curved arrow.

For comparison with the TT plan model we defined in previous papers [2,3], the model we have just defined introduces the new types of continuation and optional slots. The former are motivated by feedback received from participants at the 18th International Real-Time Ada Workshop suggesting that "*[...] one should be able to divide a long-running time-triggered task into segments that would be executed across several slots [so that] spreading the TT task execution across several slots [would give] chances for other tasks to execute in between these slots.*" [7]. With this type of slots we want to give support to this concept, although it has relevant implications that we will present in Sect. 3.2, in the context of patterns using this type of slots.

Note that in this model we are assuming that a plan is executed on a single CPU: there are no overlapping slots. This assumption will help us keep the rest of this paper as simple as possible. However, note also that in a multiprocessor system, the model is applicable provided that it is fully partitioned, i.e., there is only one plan per processor and tasks are statically assigned to CPUs. With careful synchronisation of plans, it is also conceivable to allow data sharing

[2] This difficulty alone can explain the general lack of support for the standard package Ada.Asynchronous_Task_Control.

between tasks running in different CPUs. Beyond this restrictive setup, we have also suggested to use controlled forms of migration between plans, so that a task can alternate slots in plans on different processors, to balance the overall TT workload [2]. But we will assume a single-processor platform for the rest of this paper.

3 Time-Triggered Task Patterns

The model described in the previous section grants time slots for the execution of TT tasks, leaving time gaps to be used by PB tasks running at lower priority levels. The ability to use regular, continuation and optional slots, opens the possibility to define a number of behavioural patterns for the TT tasks using them. This section proposes a set of such patterns. From some of these patterns we will derive further requirements for the design and implementation of the TT scheduler that will be presented in Sect. 4. We classify the patterns in the categories described in the following four sections.

3.1 Patterns Using Regular Slots

The simplest pattern we can think of is a TT task that accommodates all its execution time within the duration of one slot. We call this a *Simple TT Task* pattern. The task structure is simply an infinite loop where it waits for the arrival of its next slot and then executes its sequence of statements, just before suspending itself again until the arrival of its next slot.

The top half of Fig. 2 represents an example of this pattern, showing the execution of three iterations of a simple TT task. The task uses the scheduler service Wait_For_Activation to wait for the arrival of the next regular slot in the plan for the TT task with Work_Id = 1 (for example). At the beginning of each slot, the scheduler releases the work and lets it run at the highest priority among all application tasks. The priority-based subset is therefore disabled to run, since it must use strictly lower priorities than TT tasks. When the task completes within the slot duration (first and second cases in the top part of Fig. 2), it is suspended by a new call to Wait_For_Activation. The time not used by the TT task becomes available for the lower-priority PB tasks.

If, for whatever reason, the execution time of a simple TT task exceeds the slot duration, this is considered a hard deadline violation and Program_Error is raised. The TT scheduler is thus in charge of making this check at the end of regular slots.

A simple TT task may have its own local state, which is kept across successive releases. It can also share data with other simple TT tasks, because this type of task executes in mutual exclusion with other simple TT tasks (there are no overlapping slots). If the task needs to share data with pre-emptable PB tasks (or sliced TT tasks, as we'll see later), then it needs to do it via protected objects. In such case, it may experience blocking that must be taken into account when deciding the slot duration.

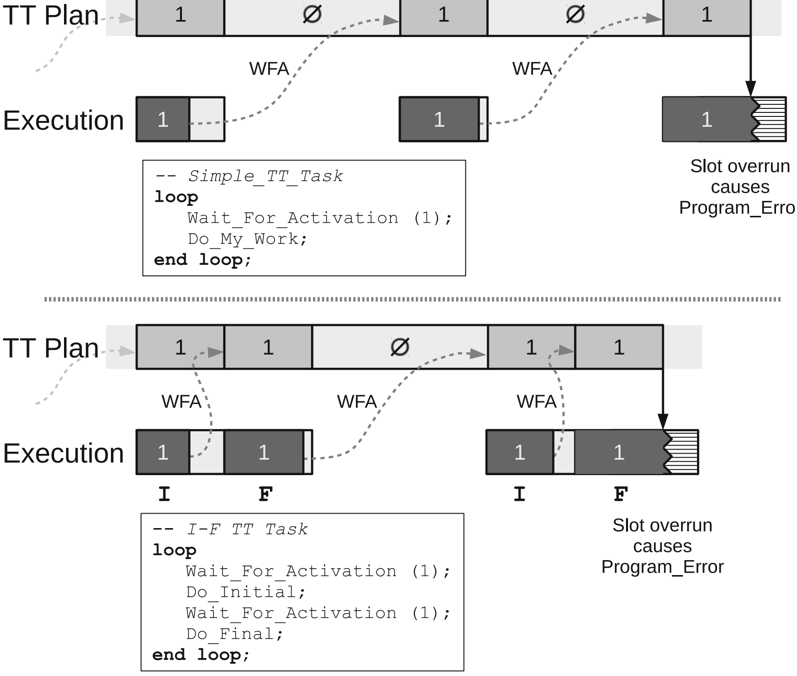


Fig. 2. TT task patterns using only regular slots. The Simple TT Task pattern (top) uses Work_Id = 1. The I-F pattern (bottom) uses two regular slots, one for each part.

The *Initial-Final* pattern (I-F, for short) is for TT tasks that can be subdivided in two parts, both with strict jitter requirements and both requiring overrun control. This pattern can be easily obtained by sequential composition of two simple TT patterns, as shown in the bottom part of Fig. 2, which shows the execution of two iterations of an I-F task. The loop is split in two parts, first the initial and then the final, and both use the same Work_Id when calling Wait_For_Activation – not necessarily as a restriction, but just to keep the plan more human-readable. Note that the slots for the initial and final parts need not have the same duration. Overrun must be checked for both parts.

Regarding data sharing, the considerations we made about simple TT tasks apply also to the case of an I-F task. Note however that communication between the initial and final parts is straightforward, since both parts are carried out by the same task.

A further variation is the *Initial-Mandatory-Final* pattern (I-M-F, for short), which uses three consecutive regular slots to perform a logically related sequence of TT actions. This scheme is typical in embedded control systems, where the initial part acquires some environment data, the mandatory part makes some calculations with the acquired data, and the final part applies the results of the mandatory part to actuators of the controlled system. The logical structure

would be defined by three calls to Wait_For_Activation using the same Work_Id, each preceding the statements of the initial, mandatory and final parts. The same considerations regarding overrun detection and data sharing we made for simple and I-F tasks, apply to I-M-F tasks: overrun is checked for each and every part of the task.

Actually, this pattern can be generalised to a form I-{M}-F, where there are one or more slots dedicated to execute overrun-controlled parts of the mandatory section. If we do not want overrun control in all the intermediate slots, then we need to use continuation slots.

3.2 Patterns Using Continuation Slots

Continuation slots allow one to break a long running TT task into slices in a way that is transparent to the application code, i.e., it does not require the task to make explicit calls to Wait_For_Activation at particular points of its execution. Slicing is dynamic and occurs at run time, rather than statically hardcoded. The TT scheduler will hold and resume the task at points dictated by the plan. Like a simple TT task, it just calls Wait_For_Activation and then performs its work. But execution of the work can be split accross several consecutive continuation slots. A sliced TT task requires the use of one or more continuation slots, ending with a terminal, regular slot.

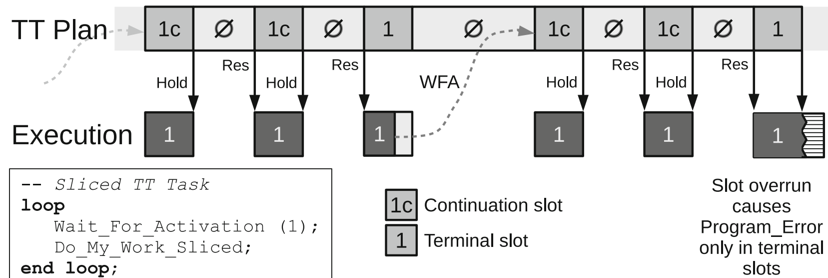


Fig. 3. Sliced Task Pattern.

Figure 3 shows two iterations of execution of a sliced TT task. This task can make use of up to three consecutive slots, which is reflected in the plan as two continuation slots (marked '1c') and one terminal, regular slot (marked simply '1'). The task structure does not differ in structure from the simple TT task described in Sect. 3.1, but the semantics are totally different due to the use of continuation slots. In other words, one needs to look at the plan to distinguish a simple TT task from a sliced TT task.

In the two iterations shown in Fig. 3, the task is held (by the TT scheduler) at the end of exhausted continuation slots, and resumed at the start of its next slot in the plan. Exceeding the lifetime of a continuation slot is not an overrun

situation. In the first iteration, the task completes its work within its terminal slot (a regular slot). In the second, the task overruns the terminal slot, hence Program_Error occurs.

These two cases are relatively simple to consider, but we need to look into more possible situations. Given the pattern structure, the task will call Wait_For_Activation as soon as it is done with the Do_My_Work_Sliced sequence of statements. This may well happen during a continuation slot, before the terminal slot of the sequence. The task *may* use up to three slots to complete, but it could take less. If that is the case, then the scheduler needs to *ignore* the pending call to Wait_For_Activation until the first slot of the next sliced sequence. An early wait for activation must therefore be *propagated* until the next continuation slot after the next terminal slot, i.e., the next start of a sliced sequence.

Figure 4 shows the possible cases of early completion of a three-slot sliced task. In the first case, the task completes in the second slot of a three-slot sequence. When the terminal slot of this sequence arrives, the scheduler has to avoid waking up the task at the start of the slot and checking for overrun at the end. The effect of Wait_For_Activation must be postponed and the task must remain blocked waiting for the start of a new sequence of slots. This is marked as "Propagate" in Fig. 4. In the second case, the task completes even earlier, during the first slot of the sequence. The effects of Wait_For_Activation must be propagated to the next two slots. A new sequence of the sliced TT task starts after the next regular (terminal) slot.

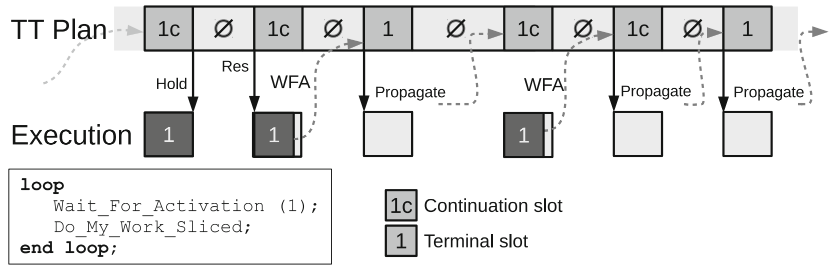


Fig. 4. Two iterations of a three-slot sliced task completing before the terminal slot, requiring propagation of early Wait_For_Activation calls.

As indicated previously, the need to hold and resume a running task has implications that must be taken into account. The problem is specially relevant if the sliced task shares data with other TT tasks or PB tasks. Since the task can be held asynchronously, this data sharing can only be protected. But holding the task while it is running a protected action is not acceptable, and letting it finish a long protected action could enlarge the release jitter of the next slot. To mitigate the effects of these two issues, at a cost, there are two design aspects to consider:

1. A sliced task can only share data with other TT or PB tasks by means of a protected object. To avoid holding the task while it is running a protected action, the ceiling priority of the protected object must be set at a level that effectively disables interrupts. This is the only way to avoid the execution of the (interrupt-driven) TT scheduler while a sliced task is executing a protected action.
2. As a consequence of the previous point, protected actions involving a sliced TT task must be as short as possible. Typically, they should only involve word-sized data exchanges and perhaps a simple condition evaluation. As few cycles as possible, because we are meanwhile blocking interrupts. If the protected action cannot be so short, then there are still alternatives. One is to design the plan so that all continuation slots are followed by empty slots of sufficient duration to absorb the potential blocking time of the runtime system. If this is not possible, because the data exchange required is large, then it is still possible to make use of multiple buffering techniques in order to reduce the need for mutual exclusion to just the time to swap a pointer.

A final consideration regarding continuation slots and their use by sliced TT tasks is that mode changes are not allowed in the middle of sliced sequences, because that would break their logic. This must be avoided by design, because the mode-change slots in the plan determine exactly when mode changes are acceptable.

Sliced sequences can also be combined with parts supported by regular slots. The *Initial-Mandatory_Sliced-Final* pattern (I-Ms-F) is a variant of the I-M-F pattern where the mandatory part is sliced. The *InitialMandatory_Sliced-Final* pattern (IMs-F, note the missing dash between the 'I' and 'M' parts) is a slight, though important modification of I-Ms-F that allows the mandatory part to start executing immediately after the initial part, without waiting for the next slot in the plan. Both patterns have the same representation in the plan, taking one regular slot for the initial part (so that it is subject to overrun control), then one or more continuation slots ending with a terminal slot for the sliced mandatory part, plus one regular slot for the final part.

Note that the IMs-F pattern requires specific support from the TT scheduler. Since IMs-F allows the mandatory sliced part to start as soon as the initial part is done, during the first regular slot, we are effectively transforming the semantics of the Initial part's regular slot into that of a continuation slot. The scheduler must therefore be informed of the termination of the initial part so that, if the initial part is not done by the end of the slot, then there is an overrun; but if it has completed, then the slicing regime has started and the hold/resume mechanism has to apply to the already started sliced mandatory part. This TT scheduler service is called `Continue_Sliced`.

Figure 5 shows these two patterns (I-Ms-F and IMs-F) for a sliced mandatory part of two slots. The top line represents the TT plan, common to both patterns. The structures of the patterns are shown to the left. The middle row represents a normal execution of an I-Ms-F task. After completing the initial part before the end of the first slot, the task waits for the next activation, hence delaying

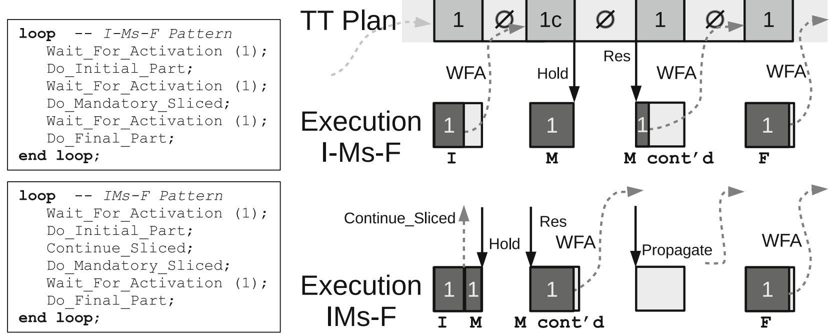


Fig. 5. Two variants of tasks with a sliced mandatory part: I-Ms-F and IMs-F.

the start of the mandatory part to the next slot. Since the first slot is regular, the initial part runs under overrun control. The sliced mandatory part takes the next continuation slot and a part of the second continuation slot and then waits for the arrival of the terminal slot, which it uses to execute the final part. In the IMs-F pattern in Fig. 5, use of this scheduler service is represented by the call to Continue_Sliced. If the scheduler has not received this call by the end of the first slot, then the initial part has overrun; otherwise, the initial part was completed during the first slot and the running task is held/resumed as a sliced subsequence of this pattern. The final part of the pattern requires a previous call to Wait_For_Activation, as already described for other patterns with a final part.

3.3 TT Patterns with Non-TT Parts

This type of pattern makes it possible for a TT task to include parts that are executed in the PB level, in competition with the PB subset of tasks. A TT scheduler service, Leave_TT_Level, allows a TT task to abandon the TT level and continue execution under the PB regime. This is useful to execute parts that are not subject to strict jitter requirements, or that may be difficult to integrate in the TT plan.

As an example, consider a control task with jitter-sensitive initial and final parts. These parts are used for reading the plant state and for sending commands to actuators, respectively. After reading sensors, the initial part rapidly calculates a first approximation to the control output, to be later applied during the final part. Until that time arrives, an intermediate part tries to improve this calculation by means of an optimisation algorithm that may take disparate execution times, depending on changing environment conditions (e.g., the number of objects detected by a radar). If this middle part had to be included in the TT plan, then the plan would have to provide sufficient slots for the worst-case execution of the optimisation algorithm. But if we could execute this optimisation part as any other PB task, with a selected priority below the TT level, then it would not require slots in the plan, hence keeping it simpler. At the end of the

middle part, the task would go back to the TT level to execute the final part with minimal jitter and using the best output possible in the available time.

Figure 6 shows the execution of such *Initial-Priority_Based-Final* pattern, or I-P-F. The pattern requires just two regular slots in the plan for the initial and final parts. In the figure, there is a regular slot for another, unrelated work (Work_Id = 2) in between these two slots of the I-P-F task, which uses Work_Id = 1. The initial part executes during the first slot and, when completed, issues a call Leave_TT_Level to inform the scheduler that the task continues with the execution of the non-TT part at a priority in the PB region. From that moment on, the PB part continues in competition with higher-priority PB tasks and other TT tasks, such as that with Work_Id = 2. The PB part eventually completes with a call to Wait_For_Activation, which makes the task return to the TT level and wait for a regular slot to execute the final part.

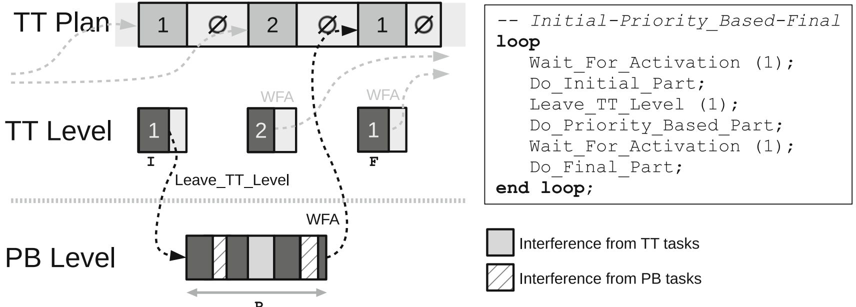


Fig. 6. Initial-Priority_Based-Final pattern (I-P-F)

The implementation of the Leave_TT_Level mechanism requires changing the priority of the TT task at runtime, which, at first sight, appears to be in contradiction with the Ravenscar model of fixed priorities. However, a Ravenscar runtime has to actually support a limited form of dynamic priorities, because it is needed to implement the Ceiling_Locking policy for protected objects. Handling PB parts as required by Leave_TT_Level can be supported as well in Ravenscar. Without going into the implementation details that we will visit in Sect. 4, the problem of scheduling a task with TT and PB parts can be seen as if the task had a base priority in the PB level, at which it runs its PB phase, and an active priority at the TT level when it runs in a TT slot. The mechanism does not need to change the priority of a task other than the running task, as for Ceiling_Locking case. And the changes between base and active priorities occur only as a result of statements executed by the same task that is affected by the priority changes, as is the case for protected actions.

The Leave_TT_Level mechanism can also be used to compose other interesting patterns. For example, a periodic PB task with one TT phase, to be executed during a regular slot in the plan. This slot could be used to synchronise the task

with the arrival of slots in the plan, for mutually exclusive communication or data exchange with other tasks, or for accessing a shared resource in general, such as in a slot-based communication protocol.

3.4 TT Patterns with Optional Slots

The inclusion of optional slots in the model opens the door to other flexible patterns. The *Priority_Based-Optional_Final* pattern (P-[F]) described now, uses a non-TT part and an optional slot. This pattern fits a periodic PB task that may or may not use a TT regular slot, for example to synchronise or communicate with other TT or PB tasks. At the TT level, the task requires just an optional slot per activation in the plan. At the PB level, the task executes as any other periodic or sporadic task.

Figure 7 shows three full iterations of a P-[F] task with a periodic PB part. In the first iteration, the task completes the priority-based part and then calls Wait_For_Activation, because the boolean Needed was True. As a result, the final part is executed at the TT level at the start of the next (optional) slot for this work. In the second iteration, Needed is False and hence the task skips the call to Wait_For_Activation and re-enters the loop to execute the delay sentence instead. Consequently, the task skips its next slot in the plan. If the slot was regular, then this no-show situation would end up in Program_Error. But because the slot is optional, the scheduler knows that this absence of a task waiting for a just started slot is intended and taken care of by the application.

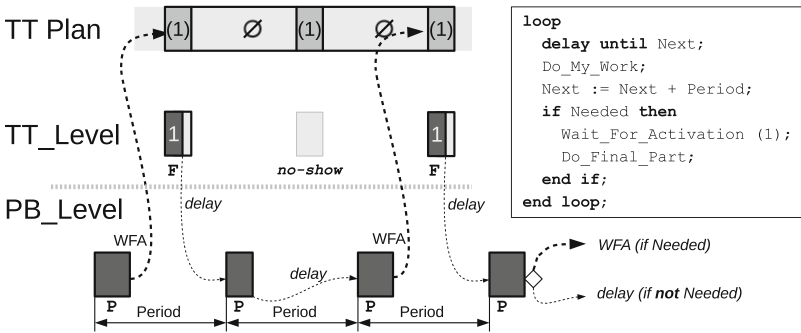


Fig. 7. The P-[F] pattern (Priority_Based-Optional_Final) combines the use of non-TT parts and optional slots.

3.5 Patterns: Looking Back

As we mentioned in the introduction, our aim with this work was to revisit our previously proposed model and implementation of combined TT-PB scheduling [2,3]. The goal was to make our implementation Ravenscar-compatible, and hence making it susceptible for consideration in the high-integrity domain.

Given that Ravenscar is a restrictive subset of the tasking model of Ada, one could think that, *a priory*, something will need to be sacrificed. We have found that this almost not true.

Looking back at the patterns we proposed in [2,3], the only mechanism we have not included is the self cancellation mechanism, which requires the use of Ada's asynchronous transfer of control, a feature that is (wisely) absent in Ravenscar, because it is not an adequate feature in a high-integrity context, due to the indeterminism it introduces. So this is a sacrifice that stems logically from our new context assumptions.

But it is to be noticed that, despite the Ravenscar restrictions, we have been able to replicate all the functionalities provided by the full-Ada scheduler —the cancellation mechanism mentioned above was implemented by the TT tasks, not the scheduler—. In addition, we have extended the model to include continuation and optional slots, which suggest a number of new patterns.

4 Design and Implementation Details

The TT scheduler enforces a TT plan, which is represented by an array of slot descriptors. Since not all slots carry the same information, we represent a slot by means of a variant record with the discriminant determining the type of slot, i.e., whether it hosts a TT work or it is an empty or mode-change slot. All slots have a Slot_Duration field, and regular slots include also:

- Work_Id - The TT work identifier.
- Is_Continuation - A boolean that marks the slot as a *continuation slot.*
- Is_Optional - A boolean indicating whether the slot is an *optional slot* or not.

An important source of information for the scheduler is the dynamic status of all TT tasks. The scheduler uses the work status to determine the actions to be taken during a slot switch. The status of a TT work is represented by the following boolean fields of a Work_Control_Block record:

- Has_Completed - Indicates that a work does not require more time at TT level. This flag is set to True when the TT task calls Wait_For_Activation or Leave_TT_Level.
- Is_Waiting - Indicates whether the work task is waiting for a new slot or not. This flag is set to True when the work calls Wait_For_Activation.
- Is_Sliced - When this flag is True, it means that this work is currently running sliced, hence it may need to be held/resumed. This flag is set to True when the work invokes Continue_Sliced or when it enters a continuation slot, and it is set to False when the work is at the start of a terminal slot.

In our implementation, the TT scheduler is the handler of a timing event that is set to occur at the start of each slot in the plan. As such, it runs at the highest interrupt priority (ARM D.15 14/2 [8]). Based on the slot and work descriptors, the scheduler decides the actions to take during a slot switch. Table 1 describes

some of these actions, limited to the case of regular and continuation slots, for short.

The top part of Table 1 the actions to take at the end of a TT work slot. These are *Hold task*, to hold the running TT task when it exhausts a slot and must continue sliced in future slots; and *raise Program_Error* when overrun is detected, i.e., the task has not completed and it is not running sliced. If Has_Completed is True, then there are no actions to take. To support Hold, our implementation of Ada.Dispatching.TTS uses runtime operations to suspend and extract from the ready queue the low-level thread behind the TT task. This is the reason why a sliced part can only use protected objects with ceiling at the highest interrupt priority, as mentioned in Sect. 3.2.

Table 1. Scheduler actions upon a slot switch, depending on work status and slot type. The top part lists actions with regard to the exhausted slot (Actions at END of slot). Actions related to the immediately starting slot (Actions at START of slot) take the bottom part.

Work status		**Actions at END of slot**
`Has_Completed`	`Is_Sliced`	
False	True	Hold task
False	False	Raise `Program_Error`

Work status			Next Slot	**Actions at START of slot**
`Has_Completed`	`Is_Sliced`	`Is_Waiting`	`Is_Optional`	
True	True			Common Actions (CA)*
True	False	True		Release Task + CA*
True	False	False	True	CA*
True	False	False	False	Raise `Program_Error`
False	True			Resume Task + CA*

*Common Actions ≡ Work.Is_Sliced ← Slot.Is_Continuation; Set_Handler

The bottom part of Table 1 lists scheduler actions related to the immediately starting slot. The actions that are common to most cases in this table (denoted *CA*) are to mark the work as sliced when it enters a continuation slot, and to set the timing event handler to the end of this starting slot. Transferring the Is_Continuation property of the slot to the Is_Sliced attribute of the work effectively propagates the *sliced* condition of the TT task until the terminal slot, for which Is_Continuation will be False.

The scheduler must also perform some actions upon calls to its public services. These may affect the work status and also the active priority of the caller's underlying thread. Note that it is the task itself who changes its own work status and priority, if needed, while running a scheduler protected operation. Table 2 summarises these update operations. For example, when a TT task invokes Leave_TT_Level, its work status is marked as completed and its priority demoted to the task's base priority – so the base priority of the task implementing the

Table 2. How and when the work status and TT task priority are modified by the scheduler.

Invoked procedure	Changes to work status			Changes to task prio
	Has_Completed	Is_Sliced	Is_Waiting	
Wait_For_Activation	True		True	Priority'Last
Continue_Sliced		True		
Leave_TT_Level	True			Task'Base_Priority

TT pattern must be determined according to the required priority level when it runs in the PB level. For readers interested in the full details of the TT scheduler, its code is available on GitHub [9], along with a basic library of TT utilities to ease the construction of TT plans.

5 Experimental Results

We have performed experiments to evaluate the performance of the scheduler, mainly in terms of jitter. The hardware platform is an STM32F4 Discovery board at 168 MHz clock frequency, running a modified version of the GNAT *ravenscar-full* runtime from AdaCore's GNAT GPL 2017. Apart from adding the Ada.Dispatching.TTS package to the runtime, we have changed the timing event and delay resolution of the GNAT runtime from the original 1 ms to 10 µs. We have measured an additional overhead of 3.5% due to this modification.

In order to compare the Ravenscar TT scheduler with the full-Ada version, we have used the same task set we used in [2]. The top part of Fig. 8 shows the cumulative frequency histogram of measured release jitters of three TT tasks, W1, W2 and W3 and two PB tasks, T4 and T5. . W1 is a simple TT task and works 2 and 3 are I-F tasks with parts identified as WI2, WF2, WI3 and WF3. The jitter is measured with respect to the theoretical activation time, i.e. the start time of the slot for TT tasks or the time expression used in the delay statements of the periodic PB tasks. As it happened with our previous full-Ada implementation, the PB tasks suffer variable release jitter, ranging from 7 µs to 400 ms, due to interference from TT tasks and higher-priority PB tasks. Their minimal jitter is however shorter than that of the TT tasks, due to the TT scheduler overhead. Note that the jitter for TT tasks is not only short but also very predictable, always within the range of 22 to 24 µs.

Taking this high predictability into account, we have tested an optimisation strategy that consists in anticipating the start time of slots by a fixed offset (20 µs in our case) so that the TT scheduler overhead is not paid for at the start, but at the end of the TT slot. This overhead is unavoidable and has to be taken into account when the plan is built, but moving it to the end of the slot drastically improves the *release jitter* of TT tasks. The bottom part of Fig. 8 shows the release jitter incurred after applying this slot anticipation idea, which now ranges from 3 to 4 µs. These results clearly outperform those previously

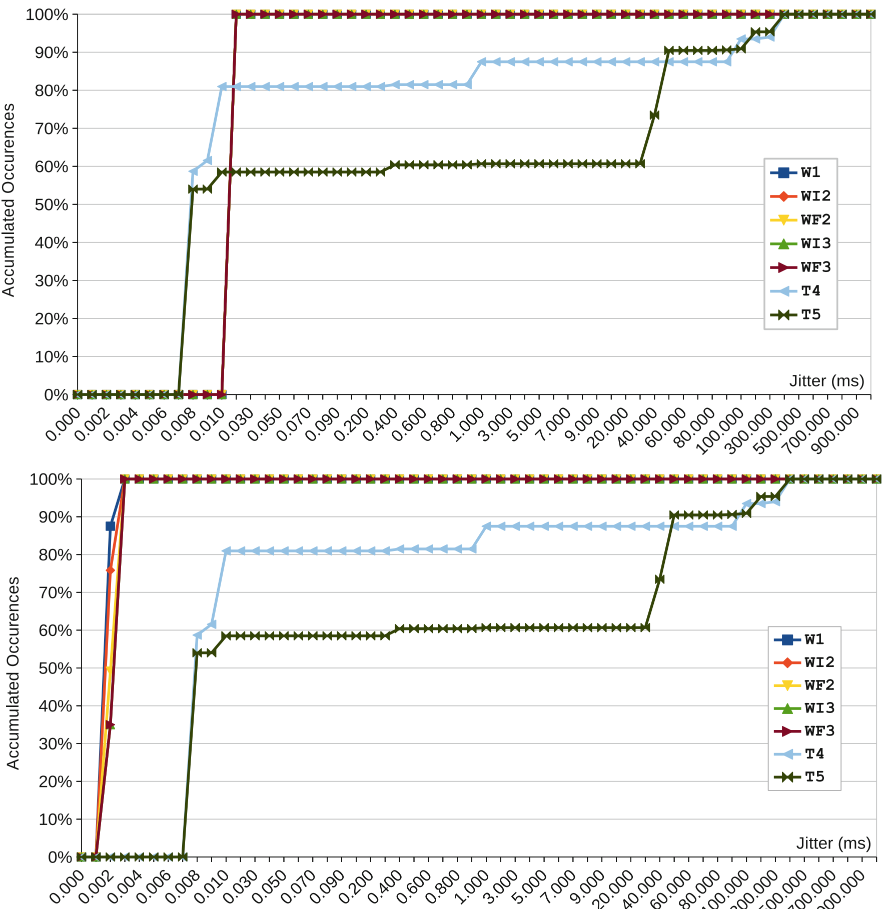


Fig. 8. Release jitter on a STM32F4 Discovery at 168 MHz. The top part reflects the jitter incurred by TT tasks with scheduler overhead. The bottom part shows how the jitter of TT tasks is further reduced with a slot anticipation of 20 µs.

obtained with a full-Ada implementation using MarteOS [10] on a Pentium III processor at 800 MHz, which varied from 30 to 70 µs.

6 Conclusions

This paper has presented the results of transforming a full-Ada architecture for combined TT-PB scheduling [2,3] to make it Ravenscar-compatible. Our aim was to make this scheduling strategy compatible with a more appropriate programming model for high-integrity and embedded systems. Our first efforts focused on a user-level library supporting the scheduler, but we soon moved to a runtime library, so that the scheduler could support continuation slots (via hold/resume)

and non-TT slots (via priority demotion), thus improving expressiveness and making room for more possible patterns. We have also introduced the concept of optional slot, and included provision for, now tolerable, no-show situations.

We have made Ada.Dispatching.TTS a generic package, where the number of TT work identifiers is a generic parameter. This allows us to keep the size of data structures to the minimum necessary for the number of TT tasks to be scheduled. The experimental results are encouraging, even better than those obtained in full-Ada with a much faster processor. No doubt, the simplicity of the Ravenscar runtime has to do with these results.

We are aware that the applicability of TT technology is very much subject to the availability of tools to help building plans and checking their consistency. In line with the former goal, we are incorporating a library of TT utilities to facilitate building the plan and using predesigned TT patterns. An initial version of this work-in-progress library is available in GitHub [9] (version v0.2.0 at the time of writing).

References

1. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, San Francisco (1979)
2. Real, J., Sáez, S., Crespo, A.: Combining time-triggered plans with priority scheduled task sets. In: Bertogna, M., Pinho, L.M., Quiñones, E. (eds.) Ada-Europe 2016. LNCS, vol. 9695, pp. 195–212. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39083-3_13
3. Real, J., Sáez, S., Crespo, A.: Combined scheduling of time-triggered plans and priority scheduled task sets. Ada Lett. **36**(1), 68–76 (2016)
4. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high-integrity systems. Technical report YCS-2017-348, University of York, June 2017
5. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM **20**(1), 46–61 (1973)
6. Leung, J., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform. Eval. (Netherlands) **2**(4), 237–250 (1982)
7. Real, J., Rogers, P.: Session summary: experience. Ada Lett. **36**(1), 101–102 (2016)
8. ISO/IEC-JTC1-SC22-WG9: Ada Reference Manual ISO/IEC 8652:2012(E) (2012). http://www.ada-europe.org/manuals/LRM-2012.pdf
9. Sáez, S., Real, J.: TTS Ravenscar runtime source code. Version v0.2.0, March 2018. https://doi.org/10.5281/zenodo.1206197
10. Rivas, M.A., González Harbour, M.: MaRTE OS: an Ada kernel for real-time embedded applications. In: Craeynest, D., Strohmeier, A. (eds.) Ada-Europe 2001. LNCS, vol. 2043, pp. 305–316. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45136-6_24

Theory and Practice of EDF Scheduling in Distributed Real-Time Systems

J. Javier Gutiérrez(✉) and Héctor Pérez

Software Engineering and Real-Time Group,
Universidad de Cantabria, 39005 Santander, Spain
{gutierjj,perezh}@unican.es

Abstract. The behavior of EDF schedulers has been very extensively studied for single-processor systems and there is also a lot of work on scheduling and schedulability analysis techniques dealing with EDF in homogeneous multi-processor systems. However, if distributed systems are considered, only a small number of schedulability analysis techniques are available and there is only a little information on practical experience with this kind of systems. For distributed systems where a clock synchronization mechanism is not available, a recent work has theoretically shown how a feasible deadline assignment can significantly increase the utilization of processing resources while keeping the system schedulable (i.e., meeting all the timing requirements). On the other hand, Ada provides support for building applications scheduled by EDF. This paper proposes a set of experiments to contrast the theoretical results on scheduling deadline assignment in a distributed real-time application against those obtained through its real execution.

Keywords: Distributed systems · Real-time · EDF · Schedulability analysis Scheduling deadline assignment · Ada applications

1 Introduction

The Earliest Deadline First (EDF) scheduling policy is present in different software layers such as real-time operating systems (SHaRK [1], ERIKA [2] or OSEK/VDX [3]), real-time communication networks (CAN Bus [4] or general purpose networks [5, 6]), real-time distribution middleware (RT-CORBA [7]), or real-time programming languages (Java RTSJ [8], or Ada [9]). Since the initial work in [10], the EDF scheduling policy has been very widely studied for single-processor systems from different perspectives of theory and practice, even in comparison with other industrially accepted scheduling policies such as fixed priority [11, 12]. There are also many works concerning EDF and multiprocessor systems [13], where EDF schedulers are well defined and can be classified into global scheduling (tasks may migrate from one

This work has been funded in part by the Spanish Government under grant TIN2014-56158-C4-2-P (M2C2). We would also like to thank Dr. Mario Aldea for his invaluable help in the implementation of the platform used in this work.

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 123–137, 2018.
https://doi.org/10.1007/978-3-319-92432-8_8

processor to another, e.g. [14]) or partitioned scheduling (fixed allocation of tasks to processors, e.g. [15, 16]). An evaluation of these two kinds of schedulers in a real-time operating system is done in [17].

In the case of real-time distributed systems, there is not an equivalent body of knowledge about the behavior of EDF scheduling. A recent study [18] shows how the utilization of processors that can be achieved while keeping the system schedulable strongly depends on: (1) the availability of a global clock (i.e., a clock synchronization mechanism); and (2) the scheduling deadline assignment technique used when tasks are composed of a sequence of sub-tasks with precedence relations which may be executed in different processors. In [18] two kinds of EDF policies are considered:

- Local-clock EDF (LC-EDF), where scheduling deadlines of each sub-task are referenced to their release times in their own processor, so clock synchronization among processors is not required.
- Global-clock EDF (GC-EDF), where scheduling deadlines of each sub-task are referenced to the release time of the task (first sub-task), possibly in a different processor, and thus requiring clock synchronization.

In general GC-EDF performs better than LC-EDF. One interesting result in [18] is the unexpected behavior found for LC-EDF that was summarized with the phrase "more haste less speed". This means that the assignment of larger scheduling deadlines to sub-task that did not comply with the end-to-end deadline of the task yields lower worst-case response times, thus improving schedulability. In the exploitation of this phenomenon, different techniques for the assignment of scheduling deadlines in LC-EDF were tested and proposed in [18].

On the other hand, Ada has supported EDF in its Real-Time Annex [9] since 2005. Furthermore, this scheduling policy continues to attract the attention of the Ada community. For example, this can be seen in the proposals for defining a new EDF Ravenscar profile [19], or for integrating the new synchronization protocol for EDF called DFP (Deadline Floor inheritance Protocol [20]) into the Ada standard [21].

The unexpected behavior observed for the LC-EDF policy occurs under a worst-case situation when applying response time analysis techniques, which are pessimistic (i.e. the worst-case response times calculated are upper bounds of the real ones). In this paper, we propose to fill the gap between theory and practice for real-time distributed systems scheduled by LC-EDF. The objective is twofold:

1. We want to evaluate the execution of a simple synthetic application under some of the scheduling deadline assignments tested in [18]. The purpose is to find out, for the different assignment techniques, whether the average and worst response times observed follow the same behavior as the worst-case response times obtained by the analysis or not.
2. We also want to verify whether the Ada language is able to support the LC-EDF policy for distributed real-time systems.

It should be noticed that we do not intend to provide the exact characterization of a real distributed application, but just to ascertain the trends in a real execution of the application for comparison with the behavior predicted by theory.

The paper is organized as follows. In Sect. 2 we provide a review of the model that we use for the distributed system as well as a short description of the schedulability analysis and scheduling deadlines assignment techniques for LC-EDF. The simple example of a distributed application that will be used in our tests is introduced in Sect. 3. Section 4 shows the results of the schedulability analysis for the scheduling deadline assignment techniques considered. A description of how the example using the LC-EDF scheduling policy could be implemented in Ada is given in Sect. 5. Section 6 discusses the performance evaluation of the application for the different scheduling deadline assignments. Finally, in Sect. 7 the conclusions of this work are presented.

2 The System Model and the Schedulability Analysis and Optimization Techniques for LC-EDF

This section briefly describes the model, the schedulability analysis technique and some of the scheduling deadline assignment algorithms used in [18] for LC-EDF. We will use MAST (Modeling and Analysis Suite for Real Time Applications) [22], which integrates the tools needed in this work as well as a model [23] aligned with MARTE (Modeling and Analysis of Real-Time Embedded systems) [24], a standard defined by the OMG (Object Management Group) for modeling and analysis of real-time and embedded systems.

The MAST model considers a system composed of distributed *end-to-end flows* (following the terminology of the OMG's MARTE standard) with periodic or sporadic activations. Each end-to-end flow Γ_i is released by a periodic sequence of external events with period T_i, and contains a set of *steps* that model tasks and messages. Each periodic release of an end-to-end flow causes the execution of the set of steps, each step being released when the preceding one in its end-to-end flow finishes its execution. We assume that tasks and messages are statically assigned to processors and networks (migration is not allowed), and that the relative phasing of the activations of different end-to-end flows is arbitrary.

Figure 1 shows an example of one end-to-end flow with three steps. The arrival of the external event that releases the end-to-end flow is represented by a thick horizontal arrow labeled e_i, and has a period of T_i. The thin horizontal arrows represent the release of the following steps in the end-to-end flow; a step cannot be executed before the preceding step has been completed. We assume that events represented in the figure are instantaneous and any activity in the system is modeled as a step. The *j*-th step of end-to-end flow Γ_i is identified as τ_{ij}; it is characterized by its worst-case execution time C_{ij} and its best-case execution time C^b_{ij}. The timing requirements that we consider are end-to-end deadlines, D_i, that start at the end-to-end flow instance's period, and must be met by the final step in the flow. We allow deadlines to be larger than the periods. As a result of the schedulability analysis, each step τ_{ij} also has a worst-case response time (or an upper bound on it) R_{ij}, and a best-case response time (or a lower bound on it) R^b_{ij}. The worst-case response time estimation of the last step can be compared with the end-to-end deadline in order to determine the schedulability of the system.

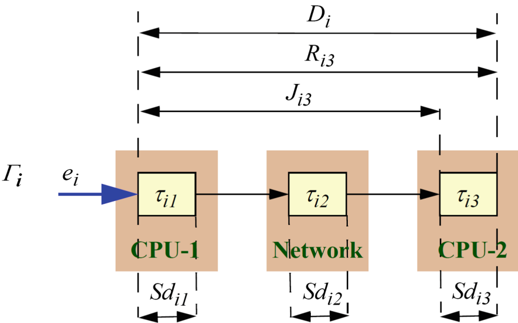


Fig. 1. The model of the end-to-end flow Γ_i

We allow the external event that triggers an end-to-end flow to have a maximum release jitter J_{i1} in relation to the corresponding activation of the end-to-end flow. Other steps τ_{ij} may also have an initial release jitter J_{ij}. Despite this jitter, global deadlines and response times always refer to the theoretical start of their respective instance's period (t_{in}), not to the actual release of the end-to-end flow. We assume that J_{ij} may be larger than the period of its end-to-end flow, T_i. For each step τ_{ij} scheduled by the LC-EDF policy, a local scheduling deadline Sd_{ij} is defined, which is referenced to the release time of its associated step in its own processing resource, thus allowing the use of the local clock.

This subset of the MAST model enables the response time analysis technique developed in [25] for LC-EDF to be applied. This technique is equivalent to the one developed by Spuri [26] for GC-EDF and obtains an estimation of the worst-case response times. The calculation of the best-case response times permits a better estimation of jitter, thus reducing the estimated worst-case response times and increasing the schedulability of the system. We will compute a lower bound on the best-case response time as the sum of the best-case execution times of the current step and all the previous ones in the end-to-end flow.

In this work, we evaluate four of the scheduling deadline assignment techniques described in [18] within their interpretation for LC-EDF:

- Ultimate Deadline (UD) [27], which obtains the scheduling deadlines by assigning the end-to-end deadline of an end-to-end flow (D_i) to each one of its steps.
- Effective Deadline (ED) [27], which obtains the scheduling deadlines by considering that if a step finishes its execution within its assigned deadline, the following steps in the same end-to-end flow will have to complete within their worst-case execution time.
- Proportional Deadline (PD) [27], which calculates scheduling deadlines by distributing the end-to-end deadline (D_i) proportionally to the worst-case execution time of each step (C_{ij}.).
- Proportional Deadline with Global Scheduling Deadline (PD-GSD) [18], which works as PD does, but converts the scheduling deadlines obtained for LC-EDF into GC-EDF ones. Thus, the scheduling deadline of step τ_{ij} equals the sum of all deadlines (assigned by PD) of the preceding steps in the end-to-end flow, including itself.

A priori, PD could be the only algorithm that makes sense for LC-EDF, as we might think that if local deadlines are guaranteed for each step, the end-to-end deadline will be guaranteed. However, [18] showed that the remaining scheduling deadline assignment techniques are able to obtain shorter worst-case response times in general, PD-GSD being the best technique for the majority of the cases. Further details on the description of these algorithms as well as on their scheduling capabilities can be found in [18].

3 The Distributed Application

We present a simple example of a distributed application consisting of two processors (CPU1 and CPU2) connected through a communication network, and executing six end-to-end flows, all of them with the same structure: two tasks allocated in a different processor; the first one activated periodically and triggering the second one through a message on the network. For the sake of simplicity and taking into account that we do not have an LC-EDF-compliant network available, we consider the messages sent through the network instantaneous, so we will not model them. This can be done without loss of generality by selecting sufficiently long periods and execution times to minimize the effect of communications. This selection will also minimize the effects of the overhead due to the operating system or communication drivers, which will be present in the real implementation. Figure 2 shows the architecture of the example.

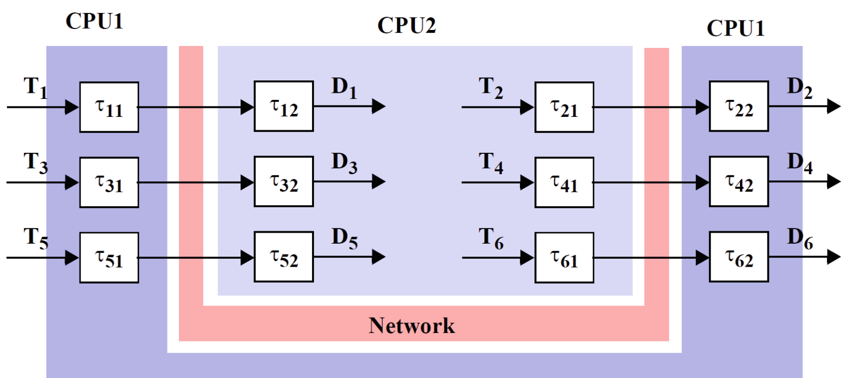


Fig. 2. Architecture of the distributed application

For this example, we propose two different configurations with the characteristics shown in Table 1. Both configurations have the same periods for the end-to-end flows, and we also consider that end-to-end deadlines are as in the case denominated T1 in [25], i.e., D = T + 1/3T for this example. The values of the execution times have been carefully selected to show the differences in the response times obtained by the analysis when it is applied to the different scheduling deadlines assignments. The worst-case execution times expressed in Table 1 are fixed, i.e. the best-case execution times of

Table 1. Characteristics of the simple distributed application (times in ms)

e2e flow	T_i	D_i	Configuration 1		Configuration 2	
			C_{i1}	C_{i2}	C_{i1}	C_{i2}
Γ_1	150	200	22	25	37	32
Γ_2	840	1120	155	217	193	159
Γ_3	1350	1800	346	386	364	366
Γ_4	1650	2200	178	148	158	138
Γ_5	3150	4200	189	223	189	223
Γ_6	30000	40000	1150	1472	1150	1472

tasks equal the worst-case execution times. The CPU utilizations are: 86.01% for CPU1 and 85.41% for CPU2 in Configuration 1, and 89.83% for CPU1 and 91.91% for CPU2 in Configuration 2.

4 Scheduling Deadline Assignments and Schedulability Analysis

After modeling the proposed application with MAST, we applied the scheduling deadline assignment techniques introduced in Sect. 3. Table 2 shows the scheduling deadlines assigned to tasks by the different techniques for both configurations. As can be seen, the trivial UD technique produces the same assignment for both configurations, and PD is the only technique in which the scheduling deadlines assigned to tasks (steps) are within the bounds of the end-to-end deadline of the end-to-end flow.

Table 2. Scheduling deadlines (Sd_{ij}) assigned to tasks by each technique (times in ms)

Task	Configuration 1				Configuration 2			
	UD	ED	PD	PD-GSD	UD	ED	PD	PD-GSD
τ_{11}	200	175	93.617	93.617	200	168	107.246	107.246
τ_{12}	200	200	106.383	200	200	200	92.754	200
τ_{21}	1120	903	466.667	466.667	1120	961	614.091	614.091
τ_{22}	1120	1120	653.333	1120	1120	1120	505.909	1120
τ_{31}	1800	1414	850.82	850.82	1800	1434	897.534	897.534
τ_{32}	1800	1800	949.18	1800	1800	1800	902.466	1800
τ_{41}	2200	2052	1201.23	1201.23	2200	2062	1174.32	1174.32
τ_{42}	2200	2200	998.77	2200	2200	2200	1025.68	2200
τ_{51}	4200	3977	1926.7	1926.7	4200	3977	1926.7	1926.7
τ_{52}	4200	4200	2273.3	4200	4200	4200	2273.3	4200
τ_{61}	40000	38528	17543.86	17543.86	40000	38528	17543.86	17543.86
τ_{62}	40000	40000	22456.14	40000	40000	40000	22456.14	40000

Table 3. Worst-case response times of the end-to-end flows obtained by the analysis for each scheduling deadline assignment (times in ms)

e2e flow	Configuration 1				Configuration 2			
	UD	ED	PD	PD-GSD	UD	ED	PD	PD-GSD
Γ_1	47	47	47	47	69	69	**215.717**	69
Γ_2	466	584	706,56	878	490	651	1091.86	965
Γ_3	1370	1470	1421	1461	1482	1612	1763,23	1678
Γ_4	2153	2153	1786,56	1765,23	**2350**	**2282**	**2247.72**	1882,32
Γ_5	**4349**	**4349**	3793,37	3336	**5306**	**5306**	4156.37	3529
Γ_6	19435	19435	18120	16978	25046	25046	25987	25046

Table 3 shows the results of applying the response time analysis [25] to the different scheduling deadline assignments. Only the worst-case response times of the final tasks (steps) of the end-to-end flow are shown, as they are the ones that should be compared with the end-to-end deadlines. The response times that do not meet the corresponding end-to-end deadlines are in bold face. We can observe that analytical results are as expected from [18]:

- Only the PD-GSD technique enables the end-to-end deadlines to be met for all the end-to-end flows in both configurations, while PD achieves this objective for Configuration 1.
- For Configuration 2, which has higher utilization, PD obtains higher response times in general than the remaining techniques.
- For Configuration 1, PD-GSD outperforms the worst-case response times obtained by PD for the end-to end flows with larger deadlines, without jeopardizing those with shorter deadlines.
- Although UD and ED do not allow the schedulability of both configurations, they obtain the lowest worst-case response times for the end-to end flows with shorter deadlines.

5 Ada Implementation of the LC-EDF Example

Support for EDF scheduling has been included in the Ada standard since the 2005 revision. Among the basic facilities defined in the language are the *EDF_Across_Priorities* task dispatching policy and mechanisms to set tasks' relative and absolute deadlines. Furthermore, Ada facilitates the development of periodic and sporadic EDF tasks by supporting the procedures illustrated in Fig. 3. On the one hand, the *Delay_Until_And_Set_Deadline* procedure is valuable for implementing periodic EDF tasks, as it delays the calling task until the value of *Delay_Until_Time* and the task becomes ready at that point with a deadline set to *Delay_Until_Time* plus *Deadline_Offset.* On the other hand, the *Suspend_Until_True_And_Set_Deadline* procedure is handy for implementing EDF tasks activated by external events such as a network message, as it blocks the calling task until the suspension object passed as a parameter is

```
procedure Delay_Until_And_Set_Deadline
   (Delay_Until_Time: in Time;
    Deadline_Offset : in Time_Span);

procedure Suspend_Until_True_And_Set_Deadline
   (S  : in out Suspension_Object;
    TS : in Ada.Real_Time.Time_Span);
```

Fig. 3. Ada facilities for setting the scheduling deadline in LC-EDF

set to true; at that point, the task becomes ready with a deadline set to *Ada.Real_Time.Clock* plus *TS* (*Time_Span* as shown in Fig. 3).

It is worth noting that the *Suspend_Until_True_And_Set_Deadline* procedure is suitable for simple synchronization scenarios, as it is based on the suspension object mechanism which is a low-level primitive equivalent to a binary semaphore. Therefore, it cannot handle nested activations (i.e., when the suspension object is set to true successive activations are lost). However, nested activations are common in distributed systems where response times are usually higher than periods, thus requiring the use of a higher-level synchronization mechanism.

Figure 4 shows the sequence diagram for the execution of an end-to-end flow in the proposed LC-EDF example. As can be seen, three different kinds of tasks can be defined:

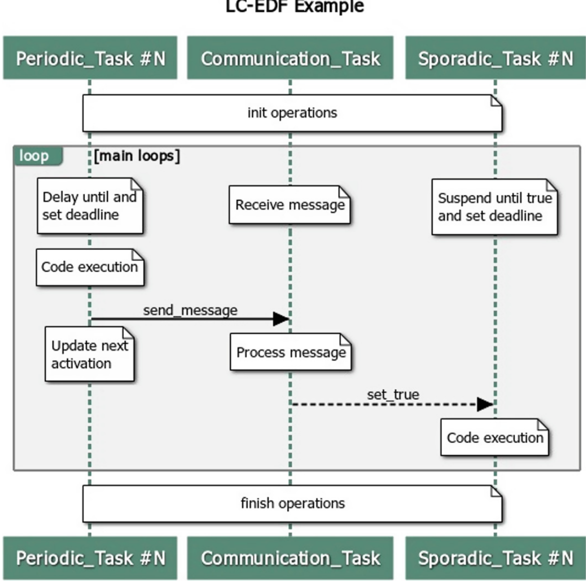


Fig. 4. Sequence diagram for the implementation of the end-to-end flows

- Periodic tasks, which periodically execute their code and send a message through the network in order to activate the remote tasks.
- Sporadic tasks (remote), which wait for the arrival of the external event to start its execution.
- Communication task, which waits for the arrival of incoming network messages to signal the triggering event to the corresponding sporadic task.

In our example, execution times for a given task can be simulated using busy loops based on CPU-time clocks, which have been included in the language since Ada 2005.

6 Performance Measurements

The hardware platform used for the example consists of two industrial embedded computers following the PC/104 architecture. Each computer is based on a single Intel Pentium III processor with a clock rate of 800 MHz, and it communicates through an Ethernet network at 100 Mbits/s. This hardware is suitable for our purpose, as our study deals with distributed systems based on single core processors. The software platform is based on the GNAT-GPL-2014 compiler [28] for the proposed application running on top of the real-time operating system called MaRTE OS (v1.9) [29, 30]. This operating system follows the Minimal Real-Time POSIX.13 subset, and it also implements the EDF scheduling policy that can be used from Ada and C applications [31].

To the best of our knowledge, there are no implementations for those Ada facilities related to the suspension object described in Sect. 5. Although we have tried to implement them in the context of MaRTE OS, a stable version is not available yet so we decided to leave this development for future work. As we wanted to measure the execution times of a distributed application scheduled by EDF, we decided to build an equivalent distributed application written in C, which directly uses the MaRTE OS facilities for EDF. Basically, the Ada tasks were turned into threads and the suspension object was replaced by a semaphore. The latter change has the advantage that semaphores can handle nested activations of the remote part of the end-to-end flow. Even though the Ada code could not be evaluated, it still remains a good example of an elegant implementation for this application.

Without having a global clock and in order to properly compare the measurements of response times in the real application to those obtained by the schedulability analysis (shown in Sect. 4), we will separately estimate the response times of both steps of the end-to-end flow: the first part (a task activated periodically) is measured from its activation until the corresponding message is sent; the second part (the remote task activated by the message) is measure from the arrival of the message until the end of the simulated execution. Then, the end-to-end response time is calculated by adding the measured response times of both parts for each activation of the external event. For this purpose, the response time obtained in the first step of the end-to-end flow is sent to the second step embedded in the message. After finishing the second step, the end-to-end response time is calculated and stored for later processing. It should be noticed that the time spent in the network is not included in these measurements, which is coherent with

the kind of schedulability analysis that we have done. Once the experiment has finished, the best, average and worst response times are calculated for each end-to-end flow.

Each individual time measurement within a step is obtained as the difference of two reads of the monotonic clock. This clock is based on the TSC (Time Stamp Counter) available in the Pentium architecture. On the other hand and in order to be precise in the implementation of the execution times of the steps, a simple library is used to load the CPU with the execution of a given amount of time. This library is based on iteratively checking the CPU-time clock until the specified time is met. Each step has its own CPU-time clock to count only the time that it is using the CPU. Thus, by applying this methodology, we can obtain quite accurate results when measuring the response times of the end-to-end flows without clock synchronization. For the interpretation of the results obtained, the following issues should be taken into account:

- Similarly to the analysis, the contribution of the transmission times of messages to the response times has not been considered. However, these transmissions impact the activation jitter of the remote task, which indirectly influences the response times of the real application. We think that this impact may be negligible since the timing parameters of tasks and the transmission times of messages differ by at least two orders of magnitude.
- In the real application, the execution times of tasks also include an overhead due to the transmission and reception of messages and the management of the timing parameters for each activation. To estimate this overhead, we have measured the round trip execution of an end-to-end flow without the simulated load. Table 4 shows the values of the total response time and also the time spent in the remote part of the end-to-end flow for 100000 executions with a message size of 64 bytes.
- There is also extra overhead introduced by the operating system and the communication task. This overhead should be negligible compared to the execution times of our application.

Table 4. Evaluation of the task's overhead and message transmission (times in μs)

	Best	Average	Worst
Total	184	195	198
Remote task	45	52	74

Tables 5 and 6 show the response times (best, average and worst) of the end-to-end flows for the configurations and scheduling deadline assignments proposed in our distributed application. Each test has been executed for 6 h and the size of the messages interchanged by tasks is 64 bytes. Taking into account these results and also those obtained in the response time analysis (see Table 3), the following observations can be made:

- The response time analysis technique used is pessimistic, i.e. it obtains upper safe bounds of the worst-case response times. On the other hand, it is probable that we have not found the worst-case situation for some end-to-end flows in the execution

Table 5. Response times of the end-to-end flows measured in the Ada application for each scheduling deadline assignment and **Configuration 1** (times in ms)

e2e flow	UD			ED		
	Best	Average	Worst	Best	Average	Worst
Γ_1	47	47	48	47	47	48
Γ_2	419	445	466	419	449	559
Γ_3	873	1202	1315	873	1142	1315
Γ_4	374	1144	2127	373	1176	2128
Γ_5	650	1528	2840	509	1580	3290
Γ_6	11583	14187	16825	11969	14246	16706
e2e flow	PD			PD-GSD		
	Best	Average	Worst	Best	Average	Worst
Γ_1	47	47	48	47	47	48
Γ_2	419	468	698	419	585	856
Γ_3	873	1087	1335	873	1127	1462
Γ_4	373	1121	1747	373	818	1723
Γ_5	507	1619	2995	506	1348	2188
Γ_6	12229	14200	16703	11966	14237	16631

Table 6. Response times of the end-to-end flows measured in the Ada application for each scheduling deadline assignment and **Configuration 2** (times in ms)

e2e flow	UD			ED		
	Best	Average	Worst	Best	Average	Worst
Γ_1	69	69	70	69	69	70
Γ_2	421	464	491	421	474	619
Γ_3	981	1304	1428	981	1232	1428
Γ_4	397	1191	**2281**	397	1231	**2212**
Γ_5	731	1923	3522	593	1781	3552
Γ_6	16159	19925	23365	15843	19882	23183
e2e flow	PD			PD-GSD		
	Best	Average	Worst	Best	Average	Worst
Γ_1	69	69	70	69	69	70
Γ_2	421	479	643	421	629	966
Γ_3	981	1230	1453	981	1301	1654
Γ_4	410	1133	1793	365	887	1789
Γ_5	731	1803	3072	593	1738	2850
Γ_6	16162	19887	23309	16090	20065	23340

of the application. Figure 5 shows the difference between the analytical and observed values for the worst-case response times. This difference is relative to the analytical one and expressed as a percentage. We can see that this difference is low for the end-to-end flows with shorter deadlines (Γ_1 to Γ_4) for both configurations

and all the assignment techniques except for PD in Configuration 2. In this latter case, the analysis obtains particularly pessimistic results that cannot be observed in the real execution. PD-GSD is the technique that produces tighter worst-case response times and all of them are within the deadlines. As predicted by theory (see Table 3), Γ_4 misses the deadline with the ED and UD assignments in Configuration 2 (see Table 6), but the worst-case observed for Γ_5 is much lower than the analytical value.

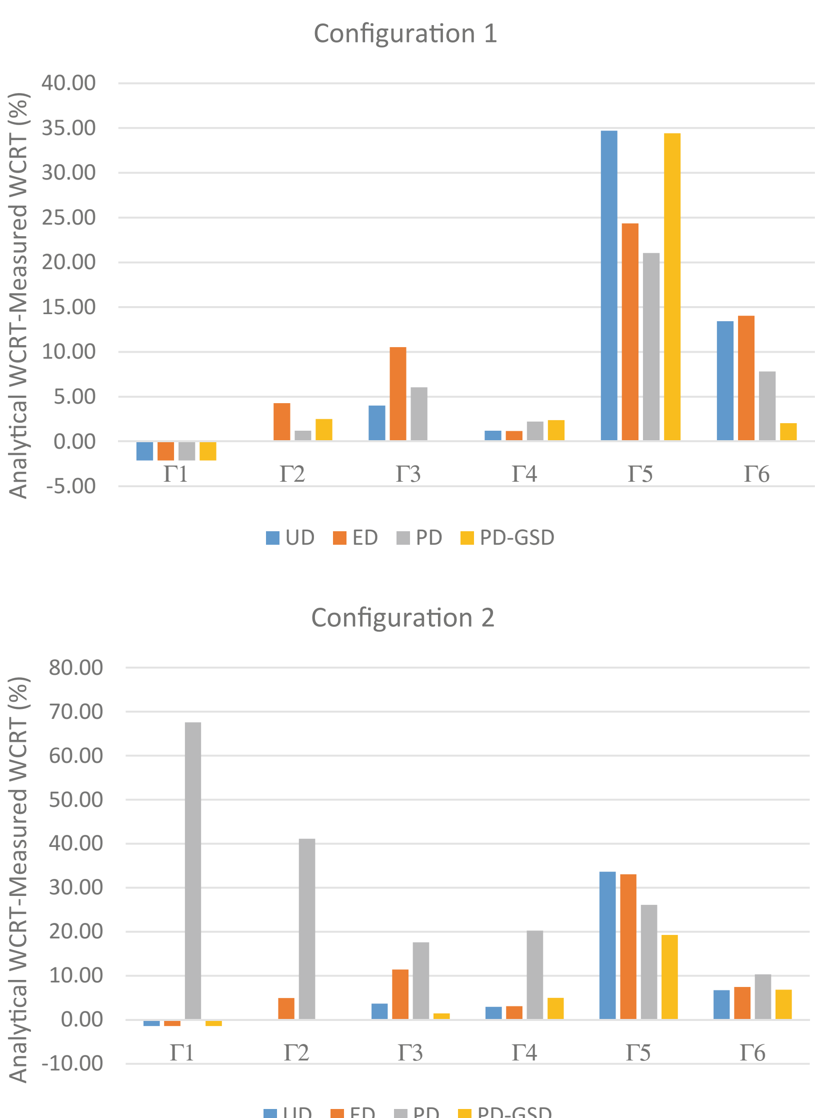


Fig. 5. Differences between analytical and observed values for the worst-case response times relative to the analytical ones

- Negative values in Fig. 5 mean that the analytical value obtained for the worst case is lower than the observed one. This is the case of Γ_1 for all the scheduling deadline assignment techniques and in both configurations except for PD in Configuration 2. It also happens for the PD-GSD assignment in Configuration 1 (Γ_3) and Configuration 2 (Γ_2), and for UD in Configuration 2 (Γ_2). In all these cases, the deviations are low and they can be justified by the overheads that have not been considered in the analysis.
- As a general conclusion about the worst-case response time, we can say that UD and ED benefit end-to-end flows with shorter deadlines to the detriment of the other ones, while PD-GSD balances the worst-case response times of all the end-to end flows, which allow the deadlines to be met. In contrast to theory, PD in the real application does not show a radically different behavior to PD-GSD. Compared to PD-GSD, PD benefits end-to-end flows with shorter deadlines.
- In relation to the average execution times, we cannot obtain a general behavior pattern. For example, ED, UD and PD obtain lower average response times than PD-GSD for Γ_2 in both configurations, but the latter achieves a very low average response time for Γ_4.
- The best-case response times obtained by all the assignment techniques in both configurations are quite similar. The response time analysis technique applied uses the sum of the best-case execution times of steps as a lower bound on the best-case response time of the end-to-end flow. For example, the best-case response time for Γ_3 in Configuration 2 is 364 + 366 = 730 ms (see Table 1). However, the best-case measured is 981 ms for all the assignment techniques. The analysis could be improved with a better estimation of the best-case response times, which would contribute to reducing the activation jitter of the remote tasks.

7 Conclusions

In this paper we have reported findings about using LC-EDF scheduling in distributed real-time systems. This is based on a previous result that showed that the assignment of longer scheduling deadlines could yield lower worst-case response times under some circumstances according to the scheduling theory. To contrast this result, our work has focused on checking whether this unexpected behavior is also reproducible in a practical application. To this end, the proposed application has been executed extensively, using 4 different scheduling deadline assignment techniques: UD, ED, PD and PD-GSD.

This paper shows that PD-GSD is the most precise assignment technique, as well as being the only one that allows deadlines to be met in theory and practice. The poor performance predicted by theory for the PD assignment has not been corroborated by our experiments. The average behavior is quite similar for all circumstances, and it does not enable clear conclusions about whether one technique performs better than the others. Finally, a better estimation of the best-case response times in the analysis could be possible, but it is an open topic for further research. From the results obtained in the paper, we can conclude that the behavior of EDF in distributed systems is still difficult to predict.

Although EDF has been widely studied for single-processor real-time systems, further research is still required in the case of distributed real-time systems in order to extend the use of this scheduling policy. Furthermore, providing Ada implementations supporting EDF may help in its adoption, so we have plans to complete the implementation of the EDF facilities defined by the Ada standard in the MaRTE OS kernel.

References

1. SHaRK (Soft Hard Real-Time Kernel) home page. http://shark.sssup.it/
2. ERIKA Enterprise: Evidence home page. http://www.evidence.eu.com/
3. Diederichs, C., Margull, U., Slomka, F., Wirrer, G.: An application-based EDF scheduler for OSEK/VDX. In: Design, Automation and Test in Europe, DATE 2008, pp. 1045–1050 (2008)
4. Pedreiras, P., Almeida, L.: EDF message scheduling on controller area network. Comput. Control Eng. J. **13**(4), 163–170 (2002)
5. Di Natale, M., Meschi, A.: Scheduling messages with earliest deadline techniques. Real Time Syst. **20**(3), 255–285 (2001)
6. Qian, T., Mueller, F., Xin, Y.: Hybrid EDF packet scheduling for real-time distributed systems. In: Proceedings of the 27th Euromicro Conference on Real-Time Systems, Lund, Sweden, pp. 37–46 (2015)
7. OMG (Object Management Group): Realtime Corba Specification, v1.2 (2005), http://www.omg.org/spec/RT/1.2/
8. RTSJ (Real-Time Specification for Java) home page. http://www.rtsj.org
9. ISO/IEC, 2012: Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012(E) (2012)
10. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
11. Buttazzo, G.: Rate monotonic vs. EDF: judgment day. Real Time Syst. **29**(1), 5–26 (2005)
12. Davis, R.I., Burns, A., Baruah, S., Rothvoß, T., George, L., Gettings, O.: Exact comparison of fixed priority and EDF scheduling based on speedup factors for both pre-emptive and non-pre-emptive paradigms. Real Time Syst. **51**(5), 561–601 (2015)
13. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). https://dl.acm.org/citation.cfm?id=1978814
14. Bertogna, M., Cirinei, M., Lipari, G.: Schedulability analysis of global scheduling algorithms on multiprocessor platforms. IEEE Trans. Parallel Distrib. Syst. **20**(4), 553–566 (2009)
15. Baruah, S., Fisher, N.: Non-migratory feasibility and migratory schedulability analysis of multiprocessor real-time systems. Real Time Syst. **39**(1–3), 97–122 (2008)
16. Baruah, S.: Partitioned EDF scheduling: a closer look. Real Time Syst. **49**(6), 715–729 (2013)
17. Gracioli, G., Fröhlich, A.A., Pellizzoni, R., Fischmeister, S.: Implementation and evaluation of global and partitioned scheduling in a real-time OS. Real Time Syst. **49**(6), 669–714 (2013)
18. Rivas, J.M., Gutiérrez, J.J., Palencia, J.C., González Harbour, M.: Deadline assignment in EDF schedulers for real-time distributed systems. IEEE Trans. Parallel Distrib. Syst. **26**(10), 2671–2684 (2015)
19. Burns, A.: An EDF run-time profile based on Ravenscar. Ada Lett. **XXXIII**(1), 24–31 (2013)

20. Burns, A., Gutiérrez, M., Aldea, M., González Harbour, M.: Deadline-floor inheritance protocol for edf scheduled embedded real-time systems with resource sharing. IEEE Trans. Comput. **64**(5), 1241–1253 (2015)
21. Burns, A., Wellings, A.: The deadline floor protocol and Ada. Ada Lett. **XXXVI**(1), 29–34 (2016)
22. MAST home page. http://mast.unican.es/
23. González Harbour, M., Gutiérrez, J.J., Drake, J.M., López, P., Palencia, J.C.: Modeling distributed real-time systems with MAST 2. J. Syst. Architect. **59**(6), 331–340 (2013)
24. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. OMG Document, v1.1 formal/2011–06-02 (2011)
25. Rivas, J.M., Gutiérrez, J.J., Palencia, J.C., González Harbour, M.: Optimized deadline assignment and schedulability analysis for distributed real-time systems with local EDF scheduling. In: Proceedings of the 8th International Conference on Embedded Systems and Applications, ESA 2010, Las Vegas, Nevada, USA, pp. 150–156 (2010)
26. Spuri, M.: Holistic analysis for deadline scheduled real-time distributed systems. Technical report RR-2873, INRIA, France (1996)
27. Liu, J.: Real-Time Systems. Prentice Hall, Upper Saddle River (2000)
28. GNAT compiler. https://www.adacore.com
29. Rivas, M.A., González Harbour, M.: MaRTE OS: an Ada kernel for real-time embedded applications. In: Craeynest, D., Strohmeier, A. (eds.) Ada-Europe 2001. LNCS, vol. 2043, pp. 305–316. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45136-6_24
30. MaRTE OS home page. http://marte.unican.es
31. Aldea Rivas, M., González Harbour, M., Ruiz, J.F.: Implementation of the Ada 2005 task dispatching model in MaRTE OS and GNAT. In: Kordon, F., Kermarrec, Y. (eds.) Ada-Europe 2009. LNCS, vol. 5570, pp. 105–118. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01924-1_8

New Application Domains

Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP

Sara Royuela[1(✉)], Xavier Martorell[1], Eduardo Quiñones[1], and Luis Miguel Pinho[2]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{sara.royuela,xavier.martorell,eduardo.quinones}@bsc.es
[2] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal
lmp@isep.ipp.pt

Abstract. There is a growing need to support parallel computation in Ada to cope with the performance requirements of the most advanced functionalities of safety-critical systems. In that regard, the use of parallel programming models is paramount to exploit the benefits of parallelism.

Recent works motivate the use of OpenMP for being a de facto standard in high-performance computing for programming shared memory architectures. These works address two important aspects towards the introduction of OpenMP in Ada: the compatibility of the OpenMP syntax with the Ada language, and the interoperability of the OpenMP and the Ada runtimes, demonstrating that OpenMP complements and supports the structured parallelism approach of the tasklet model.

This paper addresses a third fundamental aspect: functional safety from a compiler perspective. Particularly, it focuses on race conditions and considers the fine-grain and unstructured capabilities of OpenMP. Hereof, this paper presents a new compiler analysis technique that: (1) identifies potential race conditions in parallel Ada programs based on OpenMP or Ada tasks or both, and (2) provides solutions for the detected races.

1 Introduction

The parallel computation paradigm has irrupted in all computing domains, including safety-critical systems, to cope with the increasing need of higher levels of performance to implement advanced functionalities (e.g. autonomous driving [1]). Despite the clear benefits of parallel computation, it also introduces hazards regarding safety and reliability, crucial concepts for critical systems.

This trend has also arrived to Ada [2], a language used in safety-critical and high-security domains, and designed to keep safeness. Two main (and complementary) research lines are tackling the extension of Ada to support parallelism: (a) the simple yet powerful *tasklet* model [3–7] 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

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 141–157, 2018.
https://doi.org/10.1007/978-3-319-92432-8_9

structured and unstructured parallelism [8,9] (Sect. 3 provides more details of the two approaches). This paper is framed in the latter case.

OpenMP [10] has become a de facto standard for shared-memory systems as a result of being successfully used for decades in high-performance computing (HPC). The model has recently gained much attention in the embedded field as it addresses key issues for such systems: (a) the coupling of a main processor to one or more accelerators, (b) the tasking model, capable of expressing fine-grain and highly-dynamic parallelism, and (c) its time predictability properties.

Current works have addressed two fundamental pillars towards the adoption of OpenMP into Ada: (1) the compatibility of the OpenMP syntax with the Ada language [8], and (2) the compatibility of the OpenMP and the Ada execution models [9]. The former proves that OpenMP provides an equivalent and compatible interface to that of the tasklet model, guaranteeing the same safety features. The latter analyses the interoperability between the OpenMP and the Ada runtimes with a threefold objective: (a) fulfill both specifications without jeopardizing safety, (b) use OpenMP as an implementation of the tasklet model, and (c) incorporate OpenMP directives in Ada programs. This paper focuses on the third fundamental and yet unaddressed pillar: *ensure functional safety in the presence of parallel computation from a compiler perspective.*

The most frequent errors in parallel computation are deadlocks, race conditions and starvation, among others. Most of these errors can effectively be identified and solved using compiler techniques. Furthermore, compilers may always take a conservative approach overreacting when the solution is not decidable at compile time. In this regard, recent works propose to extend the Ada [4] and the OpenMP [11] specifications to include new aspects and directives, respectively, to address race conditions and deadlocks (among other issues) when whole-program analysis is not possible.

In particular, this paper focuses on race conditions because this kind of error is closely related with the exploitation of the fine-grained and unstructured parallel capabilities of the OpenMP tasking model. In this context, this paper advances one step towards safe parallel execution in Ada by proposing a new compiler analysis technique that: (1) allows identifying race conditions that can potentially appear in Ada programs parallelized with both OpenMP and Ada tasks, and (2) provides solutions for the detected races. The specific contributions of the paper are the following:

1. A control flow graph that represents the semantics of Ada and OpenMP, and allows the analysis of a program combining, or not, such languages.
2. The adaptation of OpenMP compiler analysis techniques developed for sequential languages (C, C++ and Fortran) to the Ada concurrent language.
3. A compiler method based on techniques for enhancing the programmability of OpenMP, that: (1) detects race conditions in Ada programs using or not OpenMP, and (2) provides users with directions to solve the errors.

2 Background

2.1 The Ada Concurrent Model

The Ada concurrency model is based on the notion of *task*, a unit of concurrency that represents an independent thread of control. All, the tasks and the mechanisms for inter-task communication and synchronization, are introduced at language level in order to allow building safer programs. As an illustration, Ada 95 [12] introduced *protected objects* to allow controlling how data is accessed, thus eliminating race conditions.

Additionally, in 1997, Burns et al. introduced the Ravenscar profile [13], a subset of the Ada programming language that allows high integrity applications to be analyzed for their timing properties by pursuing three main goals: (1) ensuring predictable execution, (2) simplifying the runtime support, and (3) eliminating constructs with high overhead. The limitations imposed by the Ravenscar profile have an inevitable impact in the complexity of correctness analyses, e.g. tasks can only communicate through shared objects (tasks entries are not allowed, so the *rendezvous* mechanism cannot be used), tasks are assumed to be non-terminating, and tasks and protected objects cannot be dynamically allocated.

Along the same lines, SPARK [14], a language that subsets Ada to enable the formal verification of programs, eliminates race conditions by forcing any global object referenced from a task to be marked as `Part_Of` that task, or be a *synchronized* object[1] [15].

2.2 The OpenMP Tasking Model

The tasking model appears in OpenMP 3.0 from the need of productively implementing certain types of parallelism: unbounded loops, recursion, unstructured parallelism, etc. It is based on the notion of *task*[2], a specific instance of executable code and its data environment, generated when a thread encounters certain language construct (e.g. `task`, `taskloop` and `parallel`). Other constructs, such as `taskwait` and `depend`, allow for tasks synchronization. The runtime system is responsible of creating and executing the tasks, which can be executed immediately after creation, or deferred. This depends on two factors: (1) task scheduling constraints (e.g. dependencies with other tasks described in the `depend` clauses), and (2) thread availability.

The uncertainty introduced by the tasking model regarding when the tasks are executed represents a challenge with respect to determining which portions of code are concurrent. Furthermore, the relaxed-consistency memory model of OpenMP (allowing `private`, `firstprivate`, `lastprivate` and `shared`

[1] Spark considers the following *synchronized* objects: protected objects, *atomic* objects (all accesses are atomic), and *suspension* objects (a kind of private semaphore).

[2] The term *task* in OpenMP is not related to Ada tasks. OpenMP tasks are lightweight parts of the code that can be executed in parallel by worker threads. In that regard, OpenMP tasks are very similar to Ada tasklets [16].

attributes), and the way data-sharing attributes may be defined[3] add extra complexity for the user, reducing the programmability, and increasing the possibilities of introducing errors.

3 Related Work

In the last years, several works are leading the introduction of fine-grain parallelism in Ada. This is so due to the increasing demand of computational capabilities of the systems using such a programming language. There are two main approaches: (1) the implementation of a parallel model built in the Ada core language, named *tasklet* model [3–7], and (2) the introduction of OpenMP in pure Ada applications [8,9]. The latter is gaining attention lately due to several reasons: (a) OpenMP is a mature parallel programming model, under continuous revision by an expert and experienced committee, (b) OpenMP is flexible yet robust, allowing the definition of both structured and unstructured parallelism, as well as the use of heterogeneous architectures, and (c) most compiler (e.g. GNU, Intel) and chip vendors in HPC (e.g. Intel, ARM, PowerPC, etc.) and the real-time domain (e.g. Kalray MPPA, TI Keystone II) support OpenMP.

In this context, different works have already explored the safety requirements necessary for OpenMP to be used in safety-critical environments, and they point to two main directions: time predictability and functional safety. About the former, the OpenMP tasking model has been proven to be analyzable regarding its time properties [17–20], thus valid to ensure that deadlines can be fulfilled. About the latter, different studies conclude that including some modifications in the OpenMP specification, as well as implementing some guidelines in OpenMP frameworks (including both the compiler and the runtime), may enable OpenMP programs to meet the correctness requirements of a safety-critical system [8,11].

This paper focuses on how the compiler can address functional safety. In this context, several compiler analysis techniques exist to check OpenMP programs for diverse errors, mainly deadlocks and race conditions. Among the former, Kroenig et al. developed a technique for detecting deadlocks in C/Pthreads programs [21] that can easily be applied to OpenMP because Pthreads mutexes (e.g. `pthread_mutex_lock`) are comparable to OpenMP locking routines (e.g. `omp_set_lock`). Among the latter, Ma et al. created a tool for detecting race conditions in OpenMP programs with a fixed number of threads [22], and Basupalli et al. developed a robust technique for detecting race conditions in OpenMP programs using affine constructs [23]. Finally, Royuela et al. developed a series of algorithms focused on the OpenMP tasking model to find incoherences in data-sharing and dependence clauses, as well as race conditions [24].

On another level, several methodologies exist to analyze Ada concurrent programs. These include two important aspects: (1) the representation used to describe the concurrent semantics of Ada programs, and (2) the technique used

[3] OpenMP allows three ways to determine the data-sharing attributes: predetermined, implicitly determined, and explicitly determined. The first two kinds are defined by several rules in the specification, the latter requires explicit definition by the user.

to implement analysis on top of a given representation. Regarding the former, the most common representations used for Ada analytics are Petri nets [25], control flow graphs [26], and different forms of task graphs such as program reachability graphs [27], real-time task digraphs [28] and system dependence nets [29]. Concerning the latter, most analysis techniques for Ada are based on model checking[4], which allows the automatic verification of a system's correctness. In this sense, Faria et al. developed ATOS [30], a tool that automatically extracts a SPIN model [31] from an Ada program, as well as a set of desirable properties from a specification annotated by the user in the program, inspired by the SPARK annotation language. Resembling ATOS, GNATprove [32] is a formal verification tool for Ada, based on the GNAT compiler [33] and Meyer's *design by contract* paradigm [34]. These contracts must be explicitly stated by programmers as preconditions and postconditions for functions and procedures, and loop invariants, all in the syntax of Ada 2012.

4 Motivation

The Ada Reference Manual [35] distinguishes three kinds of errors: (1) those that can be detected at compile time, (2) those that can be detected at run time, and (3) those that do not need to be detected. The nature of Ada is to prevent users from making errors, providing a series of mechanisms for data synchronization and mutual exclusion, among others. Still, it is the responsibility of the programmers to use these mechanisms in order to avoid errors such as race conditions and deadlocks. Section 3 introduces some state-of-the-art techniques for correctness checking. On the one hand, model checking based techniques are very mature, although their usefulness depends on contracts that are also written by programmers, hence are liable to have errors. On the other hand, techniques based on petri-nets or reachability graphs mostly tackle deadlocks, because these representations do not describe data flow information, but states. Hence, there is a lack of static techniques for data race detection in Ada programs.

OpenMP also provides mechanisms for data synchronization and mutual exclusion, but the correct use of these mechanisms relies on the programmer. This is stated in the specification, when it says that *"application developers are responsible for correctly using the OpenMP API to produce a conforming program*[5]*"*. Still, many static and dynamic techniques have been developed for OpenMP correctness checking to enhance productivity in parallel programming, as we introduce in Sect. 3. Two of them are particularly interesting to us because, although developed to enhance the programmability of OpenMP, they are also

[4] *Model checking* mechanisms allow exhaustively and automatically checking a given model regarding a given specification. Typically, hardware or software components are checked against safety requirements such as the absence of deadlocks and other critical states that can cause a system to crash.

[5] An OpenMP *conforming* program is that which follows all rules and restrictions of the OpenMP specification.

useful to detect race conditions. The first technique, named *auto-scope*, automatically defines the scope of the variables in a task construct (i.e. the data-sharing clauses) [36], and the second technique, named *auto-deps*, discovers the dependencies among tasks (i.e. dependence clauses) [37]. If whole program analysis is possible, the only limitation of the algorithms concerns the use of third-party libraries which code is not visible. Anyhow, the algorithms are sound and, when a variable cannot be automatically determined, it is reported to the user.

Overall, despite the specification of both Ada and OpenMP do not require correctness checking mechanisms to ensure programs are free from errors, including those is fundamental to increase productivity in parallel programming. In that regard, we note a lack of mechanisms for detecting race conditions in Ada, which is particularly important in case of safety environments to ensure a correct operation of the system. This paper considers the algorithms developed for OpenMP and propose the adaptation of these to handle Ada semantics. With this, we are able to detect race conditions in pure Ada programs and in mixed Ada/OpenMP programs as well.

The work uses for this paper the Ada Ravenscar profile, due to its simpler concurrency model. This restriction is not related to the safety of the analysis, which is independent from the model, but to the complexity of the control flow graph that needs to be extracted and analyzed. Section 5.4 provides information on how the approach extends to less restrictive models, being the goal that the approach is used with full Ada.

5 Proposal: Compiler Analysis for Mixed Ada and OpenMP Tasks

This section explains our proposal to solve race conditions in mixed Ada and OpenMP programs. It is structured as follows: first we present the singularities of Ada/OpenMP programs, then we show how we represent Ada/OpenMP programs, next we introduce the algorithm used to detect race conditions in such programs, and finally we show the results of applying the algorithm to a particular test case. For illustration purposes, we use the Ada application *Ravenscar*, defined in Sect. 7 of the Ada Ravenscar Profile Guide [38] as test case. The system modeled in this application includes a periodic process (*Regular_Producer*) that handles offers for a variable amount of workload (*Small_Whetstone*). When the requested workload exceeds a given threshold (*Due_Activation*), the excess load is processed by a sporadic process (*On_Call_Producer*). Additionally, interrupts may appear at any point (*External_Event_Server*), and different priorities are used to ensure preference among the different tasks.

Figure 1 shows the HRT-HOOD[6] representation of the *Ravenscar* application. There, red dashed boxes represent tasks, blue dotted boxes represent packages with functions and procedures, and yellow double-lined boxes represent protected objects with entries and procedures. The *Ravenscar* code illustrates the

[6] Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) is an object-based structured design method for hard real-time systems [39].

expressiveness of the Ravenscar profile, for it includes several features of Ada that are of our interest: protected objects, other shared data, synchronous and asynchronous synchronizations, etc.

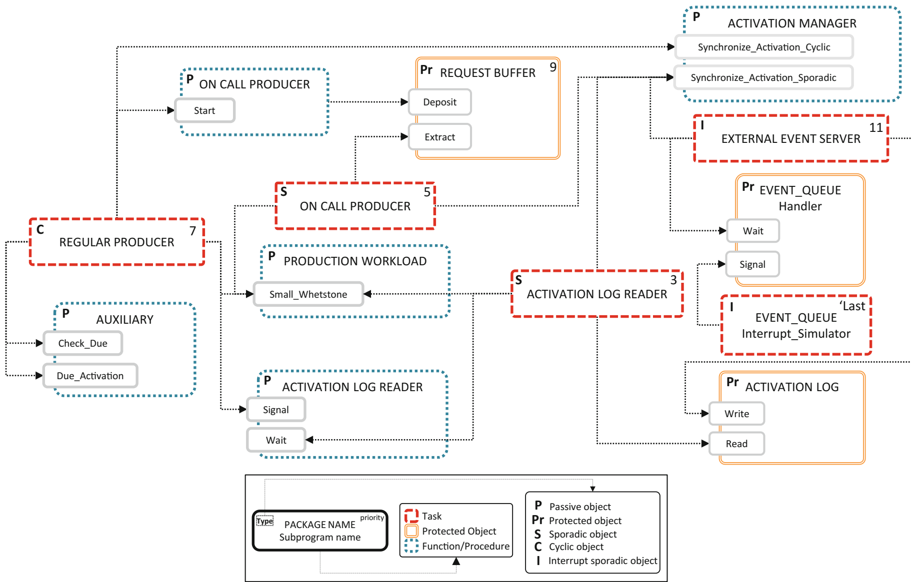


Fig. 1. HRT-HOOD representation of the *Ravenscar* application. (Color figure online)

To exemplify how the analysis handles the two levels of parallelism (Ada coarse grain tasks and OpenMP fine grain tasks), we have introduced an OpenMP computation in the *Small_Whetstone* procedure, which turns into the entry point of a sensor fusion operation. This new functionality is described in Fig. 2 using the syntax proposed in Ada to use OpenMP [8]. There, the `parallel` construct initiates parallel execution by creating a team of threads. Then, the `single` construct indicates that only one thread will execute the inner statements. Finally, the `taskloop` construct indicates that the iterations of the most outer loop are split into chunks that can be executed in parallel by the threads in the current team using OpenMP tasks. In this implementation, the parameter of the *Small_Whetstone* procedure indicates the operation to carry out: 1 means reading sensor A, 2 means reading sensor B, and 3 means fusing the two sensors by adding up its values. Sensor A is read periodically from *Regular_Producer*, sensor B is read sporadically from *On_Call_Producer*, and the fusion is performed sporadically from *Activation_Log_Reader*.

5.1 Mixing Ada and OpenMP

As introduced previously, pure Ada programs define concurrency by means of tasks, while OpenMP creates parallelism by means of the `parallel` construct,

```
1  package body Production_Workload is
2    type Dim is range 1 .. 512;
3    type M is array (Dim, Dim) of Float;
4    M_A, M_B, M_C: M;
5
6    procedure Read_Sensor_A is begin
7      pragma OMP (parallel);
8      pragma OMP (single);
9      pragma OMP (taskloop);
10     for I in Dim loop
11       for J in Dim loop
12         M_A(I,J) := sensor(1, I, J);
13       end loop;
14     end loop;
15   end Read_Sensor_A;
16
17   procedure Read_Sensor_B is begin
18     pragma OMP (parallel);
19     pragma OMP (single);
20     pragma OMP (taskloop);
21     for I in Dim loop
22       for J in Dim loop
23         M_B(I,J) := sensor(2, I, J);
24       end loop;
25     end loop;
26   end Read_Sensor_B;
27
28   procedure Fuse_Sensors is
29   begin
30     pragma OMP (parallel);
31     pragma OMP (single);
32     pragma OMP (taskloop);
33     for I in Dim loop
34       for J in Dim loop
35         M_C(I,J) := M_A(I,J) + M_B(I,J);
36       end loop;
37     end loop;
38   end Fuse_Sensors;
39
40   procedure Small_Whetstone
41             (Workload: Positive) is
42   begin
43     case Workload is
44       when 1 => Read_Sensor_A;
45       when 2 => Read_Sensor_B;
46       when 3 => Fuse_Sensors;
47       when others => null;
48     end case;
49   end Small_Whetstone;
50
51 end Production_Workload;
```

Fig. 2. OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application.

and distributes it by means of worksharing and tasking constructs. When both languages are used together, concurrency may be defined at multiple levels: between Ada tasks, between OpenMP tasks, and between Ada and OpenMP tasks. Table 1 summarizes our approach to resolve race conditions in each case.

Table 1. Solutions for race conditions in an Ada/OpenMP application.

Race condition between		Solution
Ada tasks		Ada mechanisms: protected object
Ada and OpenMP tasks		
OpenMP tasks	Different binding regions	
	Same binding region	OpenMP mechanisms: * Synchronization constructs and clauses: `taskwait`, `barrier`, `depend` * Mutual exclusion constructs: `critical`, `atomic` * Data-sharing attributes: `private`, `firstprivate`, `lastprivate`

Ada protected objects are a robust and lightweight mechanism for mutual exclusion and data synchronization. For this reason, they are to be used whenever possible to solve race conditions, i.e. when race conditions occur between

Ada tasks, between Ada and OpenMP tasks, and between OpenMP tasks that belong to different binding regions[7]. The last case is particularly interesting because in C/C++/Fortran OpenMP[8] programs, tasks belonging to different binding regions cannot be concurrent unless there are nested parallel regions. Tasks in such situation cannot be synchronized, and only data synchronization is available via the flush operation, a highly unrecommended mechanism when safety is essential due to the difficulty of analyzing its behavior. The extra layer of concurrency introduced by Ada unlocks this scenario, hence only protected objects are safe enough to synchronize data. Finally, to exploit the flexibility of OpenMP, race conditions between OpenMP tasks that belong to the same binding region are to be solved using OpenMP mechanisms: mutual exclusion constructs (i.e. `atomic` and `critical` constructs), synchronization constructs (e.g. `taskwait` and `barrier`), synchronization clauses (i.e. `depend`) and data-sharing clauses (e.g. `private`, `firstprivate` and `lastprivate`).

5.2 Representation of an Ada/OpenMP Program

As introduced in Sect. 3, several representations allow expressing the semantics of an Ada program (e.g. reachability graphs, Petri nets, control flow graphs, etc.). However, some representations are not suitable for our purpose, for instance Petri nets and reachability graphs, because these express states whereas data flow information is hidden. Furthermore, these representations have other limitations such as the state explosion problem, and the inability of representing recursive programs. Hence, to represent the behavior of an Ada/OpenMP program we use the classic control flow graph (CFG) representation extended to support Ada concurrency and OpenMP parallelism. Our graph draws from the parallel control flow graph for C/C++ and OpenMP/OmpSs [40] developed by Royuela et al. [24], and the control flow graph for Ada developed by Fechete et al. [26].

To ease the reading we show the CFGs of the original *Ravenscar* application and the new OpenMP code separately, in Figs. 3 and 4 respectively (the complete CFG of the Ada code is displayed in Appendix A). The CFG of the original *Ravenscar* code shows the code executed at elaboration time (top of the figure), and the Ada code run during the execution of the program (rest of the figure). Each partial CFG represents a task (*Regular_Producer*, *On_Call_Producer* and *Activation_Log_Reader*). The special nodes *En* and *Ex* express the entry and the exit points of each task, and the OpenMP code is pointed with dashed-dotted purple lines. Finally, the turquoise square boxes at the bottom represent some significant shared data, and the edges relating this boxes to the CFG nodes symbolize the type of access to the data: read (dotted dark red), write (solid yellow) and read/write (dashed green).

Regarding the OpenMP code, it is independent from the Ada code because the data structures being used are different. However, it is important to note that

[7] In OpenMP, the *binding region* is the enclosing region that determines the execution context. The binding region of a task is the innermost enclosing parallel region.

[8] The OpenMP API is an specification for defining parallelism in C, C++ and Fortran programs.

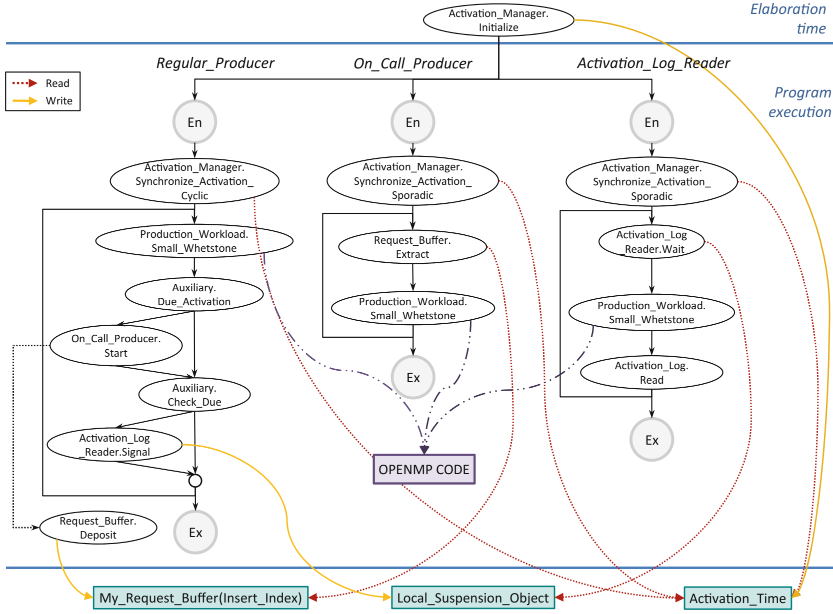


Fig. 3. Simplified CFG of the *Ravenscar* application. (Color figure online)

the OpenMP parallel tasks are inherently concurrent because they are called from within different Ada tasks, which are in turn concurrent.

Definition 1. *A* block of concurrency, *or* concurrent block, *is a set of portions of code that may execute in parallel.*

Since the application meets the Ravenscar profile, the CFG is particularly simple because all tasks are created at library level, meaning that they start executing at the beginning of the program (after elaboration) and terminate when the program ends (task allocators, task termination and abortion, and task hierarchies, among others, are not allowed). Hence, there are only two blocks of concurrency (split by blue lines in the CFG) that correspond to the code executed during elaboration, and the rest of the code.

5.3 Correctness Analysis

Inspired by the algorithms presented in the scope of OpenMP to automatically determine the data-scoping attributes [36] and the dependence clauses [37] of an OpenMP task, we present an algorithm able to find data-race conditions in Ada concurrent programs, containing or not OpenMP tasks. The high-level description of the algorithm is outlined in Listing 1.

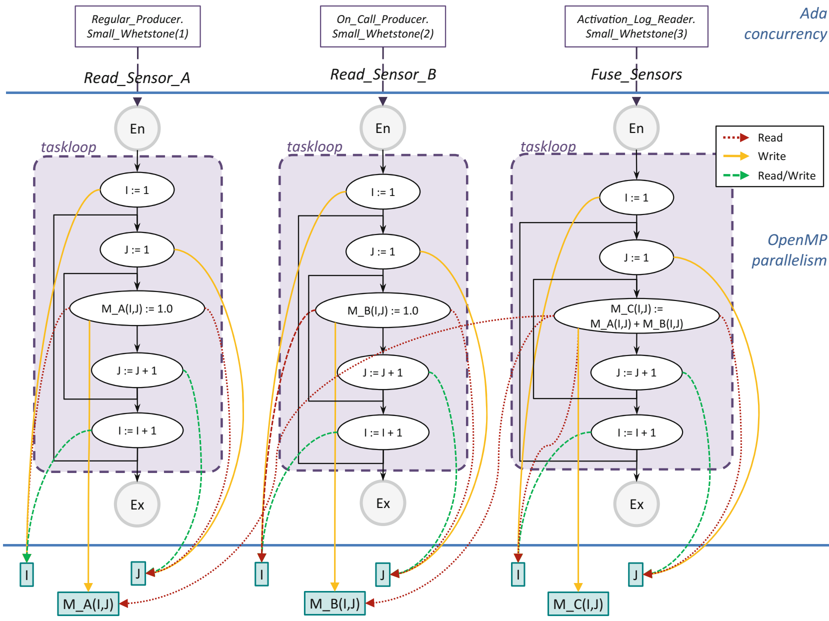


Fig. 4. CFG of the OpenMP code introduced in the *Small_Whetstone* procedure. (Color figure online)

Algorithm 1. High-level description of the race detection algorithm.

---

1. Build the inter-procedural CFG of the program.
2. Recognize the blocks of concurrency (in a Ravenscar application this is as simple as splitting the elaboration code and the rest of the code).
3. For each concurrent block, look for concurrent accesses to shared data, where at least one of the accesses is a write. If that occurs:
   (a) If all accesses to the shared data are within OpenMP tasks that belong to the same binding region, then:
      – If the operations are commutative [41], then protect the accesses with an `atomic` or a `critical` construct.
      – Otherwise, there are two approaches:
         * Use full synchronizations: insert a `taskwait` or `barrier` construct between the two accesses.
         * Use partial synchronizations: follow the algorithm to automatically determine the dependence clauses of an OpenMP tasks [37].
   (b) Otherwise, propose to wrap the shared data in a protected object.

Applying the two first steps of the algorithm to our test case results in the CFGs presented in Sect. 5.2. All Ada and OpenMP tasks correspond to the same block of concurrency, hence potential race conditions may occur among all Ada and OpenMP tasks. However, since OpenMP and Ada tasks manage different share data, we can treat them separately.

Applying the third step on the original *Ravenscar* code reveals that: (a) *Activation_Time* is not in a race condition because the read and the write accesses are in different concurrent blocks, (b) *Local_Suspension_Object* is not in a race condition because the operations performed on it are atomic with respect to each other, as the standard says, and (c) *My_Request_Buffer(Insert_Index)* is not in a race condition because this object is part of the protected object *Request_Buffer*. The results of the algorithm on the original *Ravenscar* application successfully found that the code contains no race conditions.

Regarding the analysis of the OpenMP code note that the OpenMP data-sharing rules indicate that there is a private copy of the induction variable of the taskloop for each thread. As a result, applying the third step of the algorithm on the OpenMP code reveals that accesses to variables *I* and *J* are not in a race condition. On the other hand, accesses to the matrices *M_A* and *M_B* are in a race condition because the write access to *M_A* and *M_B* from *Read_Sensor_A* and *Read_Sensor_B* respectively collide with the read access to both variables from *Fuse_Sensor*. The results of the algorithm indicate the use of partial synchronizations in the form of task dependence clauses, which are shown in Fig. 5.

```
1  procedure Read_Sensor_A is begin
2    pragma OMP (parallel);
3    pragma OMP (single);
4    pragma OMP (taskloop, depend=>in, M_A(0:Dim,0:Dim));
5    ...
6  end Read_Sensor_A;
7
8  procedure Read_Sensor_B is
9  begin
10   pragma OMP (parallel);
11   pragma OMP (single);
12   pragma OMP (taskloop, depend=>in, M_B(0:Dim,0:Dim));
13   ...
14 end Read_Sensor_B;
15
16 procedure Fuse_Sensors is
17 begin
18     pragma OMP (parallel);
19     pragma OMP (single);
20     pragma OMP (taskloop, depend=>in, M_A(0:Dim,0:Dim), M_B(0:Dim,0:Dim),
21                           depend=>out, M_C(0:Dim,0:Dim));
22     ...
23 end Fuse_Sensors;
```

Fig. 5. Snippet of the OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application including the dependence clauses proposed by the correctness analysis.

5.4 Safe Parallelism Beyond the Ravenscar Profile

This work currently assumes a restricted model, where Ada applications follow the Ravenscar profile [38], and considering only the sharing of variables declared in the same scope. This restriction is not related to the approach per se, but to

the complexity of the CFG as well as the program code visibility required for the analysis. Hence, to support the full Ada concurrency model, the CFG must be extended as to include further edges between tasks (e.g. master dependencies, task termination, rendezvous, etc.). These edges must be taken into account to determine the concurrency blocks (considering when tasks come to life and terminate), and also to tune the accuracy of the results of the race condition algorithm (considering when data is actually accessed, if possible). The compiler approach in this analysis must always be conservative in the sense that false positives are acceptable, but false negatives are inadmissible.

Another important consideration is the introduction of full program analysis to allow the algorithm addressing the data sharing of variables declared in any scope. In this sense, we consider the proposals for both Ada [4] and OpenMP [11] to cope with this limitation, both consisting in annotations added to APIs of those applications which are to be used as third-party libraries. The Ada annotations include the aspects `Global` and `Potentially_Blocking` to resolve race conditions and deadlocks respectively, and the OpenMP annotations include the directives `globals` and `usage` to resolve race conditions and illegal nesting[9] (including nested regions that can cause deadlocks).

6 Conclusions

This paper provides one step further in the work to enable OpenMP fine-grained parallelism in Ada, by addressing the safety of the code in the presence of parallel computation. For this, the paper proposes compiler analysis techniques that can identify potential race conditions in Ada, both considering Ada tasks and parallel OpenMP code. These techniques are built on top of three components: (a) a graph representation that includes both control- and data-flow dependencies of concurrent and parallel code, (b) an adaptation of existent compiler techniques developed for sequential languages to consider Ada tasks, and (c) compiler methods that detect data races and guide the programmer in solving them.

Together with previous works, this paper provides a solution to enable the use of the OpenMP fine-grained tasking model, which can be used together with, or supporting, the existent Ada parallel tasklet model.

Acknowledgments. This work was supported by the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P, and by the FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

A Complete CFG of the Ravenscar Application

This appendix includes the complete CFG of the Ada code used to illustrate the proposal of this paper, extracted from the Ada Ravenscar Profile Guide [38] (Fig. 6).

[9] The OpenMP specification (Sect. 2.17 [10]) defines a series of rules that determine which constructs cannot be nested within each other.

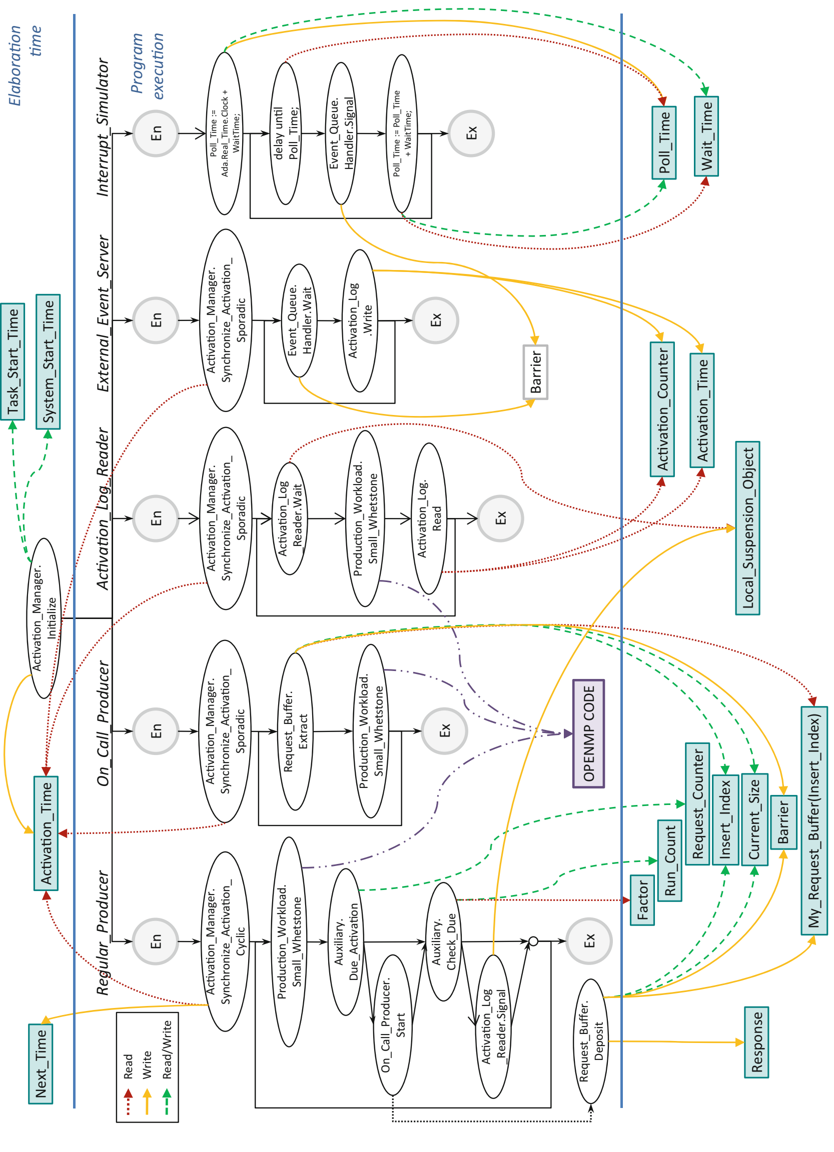


Fig. 6. Control flow graph of the *Ravenscar* application defined in Sect. 7 of the Ada Ravenscar Profile Guide, containing accesses to shared data.

References

1. NVIDIA: Automotive (2017). https://www.nvidia.com/en-us/self-driving-cars
2. AdaCore: Automotive (2018). https://www.adacore.com/industries/automotive
3. Pinho, L.M., Moore, B., Michell, S.: Parallelism in Ada: status and prospects. In: George, L., Vardanega, T. (eds.) Ada-Europe 2014. LNCS, vol. 8454, pp. 91–106. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08311-7_8
4. Taft, S.T., Moore, B., Pinho, L.M., Michell, S.: Safe parallel programming in Ada with language extensions. ACM SIGAda Ada Lett. **34**(3), 87–96 (2014)
5. Pinho, L.M., Moore, B., Michell, S., Tucker Taft, S.: An execution model for fine-grained parallelism in Ada. In: de la Puente, J.A., Vardanega, T. (eds.) Ada-Europe 2015. LNCS, vol. 9111, pp. 196–211. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19584-1_13
6. Pinho, L.M., Moore, B., Michell, S., Taft, S.T.: Real-time fine-grained parallelism in Ada. ACM SIGAda Ada Lett. **35**(1), 46–58 (2015)
7. Taft, T., Moore, B., Pinho, L.M., Michell, S.: Reduction of parallel computation in the parallel model for Ada. ACM SIGAda Ada Lett. **36**(1), 9–24 (2016)
8. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: OpenMP tasking model for Ada: safety and correctness. In: Blieberger, J., Bader, M. (eds.) Ada-Europe 2017. LNCS, vol. 10300, pp. 184–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60588-3_12
9. Royuela, S., Pinho, L.M., Quiñones, E.: Converging safety and high-performance domains: integrating OpenMP into Ada. In: Design, Automation & Test in Europe, March 2018
10. OpenMP Architecture Review Board: OpenMP Application Programming Interface 4.5 (2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
11. Royuela, S., Duran, A., Serrano, M.A., Quiñones, E., Martorell, X.: A functional safety OpenMP* for critical real-time embedded systems. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 231–245. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_16
12. Ada Resource Association: Ada 95 Reference Manual. ISO/IEC 8652:1995(E) with COR.1 (2000). http://www.adaic.org/resources/add_content/standards/95lrm/RM.pdf
13. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In: Asplund, L. (ed.) Ada-Europe 1998. LNCS, vol. 1411, pp. 263–275. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055011
14. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security: Sample Chapters. Pearson Education, London (2003)
15. Taft, S.T., Schanda, F., Moy, Y.: High-integrity multitasking in SPARK: static detection of data races and locking cycles. In: 17th International Symposium on High Assurance Systems Engineering, pp. 238–239. IEEE (2016)
16. Michell, S., Moore, B., Pinho, L.M.: Tasklettes – a fine grained parallelism for Ada on multicores. In: Keller, H.B., Plödereder, E., Dencker, P., Klenk, H. (eds.) Ada-Europe 2013. LNCS, vol. 7896, pp. 17–34. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38601-5_2
17. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In: International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 157–166. IEEE Press, October 2015

18. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Design, Automation & Test in Europe, pp. 1066–1071. IEEE, March 2016
19. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quiñones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical OpenMP applications. In: 22nd Asia and South Pacific Design Automation Conference, pp. 659–665. IEEE, January 2017
20. Sun, J., Guan, N., Wang, Y., He, Q., Yi, W.: Scheduling and analysis of real-time OpenMP task systems with tied tasks. In: Proceedings of Real-Time Systems Symposium (2017)
21. Kroening, D., Poetzl, D., Schrammel, P., Wachter, B.: Sound static deadlock analysis for C/Pthreads. In: 31st International Conference on Automated Software Engineering, pp. 379–390. IEEE, September 2016
22. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in OpenMP programs. In: 42nd International Conference on Parallel Processing, pp. 510–516. IEEE, October 2013
23. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: polyhedral analysis for the OpenMP programmer. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 37–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_4
24. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: 12th International Conference on Computing Frontiers, p. 7. ACM, May 2015
25. Evangelista, S., Kaiser, C., Pradat-Peyre, J.-F., Rousseau, P.: Quasar: a new tool for concurrent Ada programs analysis. In: Rosen, J.-P., Strohmeier, A. (eds.) Ada-Europe 2003. LNCS, vol. 2655, pp. 168–181. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44947-7_12
26. Fechete, R., Kienesberger, G., Blieberger, J.: A framework for CFG-based static program analysis of Ada programs. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 130–143. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68624-8_10
27. Qi, X., Xu, B.: An approach to slicing concurrent Ada programs based on program reachability graphs. Int. J. Comput. Sci. Netw. Secur. **6**(1), 29–37 (2005)
28. Mohaqeqi, M., Abdullah, J., Guan, N., Yi, W.: Schedulability analysis of synchronous digraph real-time tasks. In: 28th Euromicro Conference on Real-Time Systems, pp. 176–186. IEEE, July 2016
29. Wang, B., Gao, H., Cheng, J.: Definition-use net and system dependence net generators for Ada 2012 programs and their applications. Ada User J. **38**(1), 37–55 (2017)
30. Faria, J.M., Martins, J., Pinto, J.S.: An approach to model checking Ada programs. In: Brorsson, M., Pinho, L.M. (eds.) Ada-Europe 2012. LNCS, vol. 7308, pp. 105–118. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30598-6_8
31. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
32. AdaCore, Altran, Astrium Space Transportation, CEA-LIST, ProVal at INRIA and Thales Communications: Project Hi-Lite: GNATprove (2017). http://www.open-do.org/projects/hi-lite/gnatprove
33. GNU: GNAT (2016). https://www.gnu.org/software/gnat
34. Meyer, B.: Object-Oriented Software Construction, vol. 2. Prentice Hall, New York (1988)

35. Ada Resource Association: Ada Reference Manual, ISO/IEC 8652:2012(E) (2012). http://archive.adaic.com/standards/83lrm/html
36. Royuela, S., Duran, A., Liao, C., Quinlan, D.J.: Auto-scoping for OpenMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 29–43. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_3
37. Royuela, S., Duran, A., Martorell, X.: Compiler automatic discovery of OmpSs task dependencies. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 234–248. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_16
38. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. ACM SIGAda Ada Lett. **24**(2), 1–74 (2004)
39. Burns, A., Wellings, A.J.: HRT-HOOD: a structured design method for hard real-time systems. Real-Time Syst. **6**(1), 73–114 (1994)
40. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
41. Lippe, E., van Oosterom, N.: Operation-based merging. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, SDE 5, pp. 78–87. ACM, New York (1992)

Microservice-Based Agile Architectures: An Opportunity for Specialized Niche Technologies

Stefano Munari, Sebastiano Valle(✉), and Tullio Vardanega

Department of Mathematics, University of Padua,
Torre Archimede, Via Trieste, 63, 35121 Padova, Italy
stefanomunari.sm@gmail.com, valle.sebastiano93@gmail.com,
tullio.vardanega@math.unipd.it

Abstract. This work discusses lessons learned from the development of a medium-size peer-to-peer distributed software system centered around asynchronous computation and message-/stream-oriented communication. Albeit foreign to traditional high-integrity systems, these architectural characteristics are making rapid headway into large-scale mission-critical and business-critical software infrastructures, thus becoming candidate solutions for the design of reliable systems. We wanted our software architecture to be agile, that is, versatile, easy to evolve and modify, and resilient enough not to degrade across changes. To meet this goal, we adopted the microservices style, which afforded us the choice of best-of-breed technology to implement the individual system parts. Embracing heterogeneity while seeking agility however challenged our ability to design effective solutions for component coordination and interaction, as well as the goodness of fit of the used technologies for system integration and testing. Reflecting on our experience, we distill the lessons we learned in terms of architectural patterns, highlighting the pros and cons we saw in the microservices style and in our technologies selection.

Keywords: Microservices architecture · Distributed systems
Agility · Scalability · Heterogeneity · Patterns

1 Introduction

Nowadays, software systems are ubiquitous: Internet of Things (IoT) devices, swarms of drones and self-driving cars are perhaps the most prominent examples of computer-based services that are becoming part of everyone's life [1].

It is not difficult to predict that the degree of criticality of modern computing systems will grow proportionally to their pervasiveness. This unstoppable trend will likely turn into critical infrastructure several systems that were not regarded as such at their origin. Amazon and Google are two most notable sources of applications and services infrastructures progressively expanding into the business-critical and mission-critical systems camp. Some of those systems,

© Springer International Publishing AG, part of Springer Nature 2018
A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 158–174, 2018.
https://doi.org/10.1007/978-3-319-92432-8_10

such as autonomous cars, air traffic control systems, medical systems, already have a significant impact on human society: a downtime in their infrastructure could cost billions of dollars [2], and their faulty behavior can cost lives [3]. These observations suggest that most future systems will need to take availability and *reliability* into account as non-negotiable qualities.

It is interesting to look back at how software engineering practices, designated to pursue measurable conformance with requirements, evolved, in the face of the explosion in size and criticality of software systems.

In the late 1970s, Myers [4] defined software testing principles and practices (e.g., black-box testing) to limit the quantity of residual software faults in delivered products, while acknowledging that not all of them could be eradicated in that manner. Testing acquired greater prominence in the V model, to help assert requirements coverage and earn confidence in the system reliability. Sadly, waterfall-cycle experience has shown that (large) fractions of the planned test campaign are liable to omission as their execution occurs late in the development process, maximally exposed to the risk of time and cost overruns.

In the early 2000s, agile methodologies made their grand entrance into software engineering practice, rapidly becoming the dominant choice among industrial practitioners. Seeking a cure to the vulnerability of prior solutions (too rigid and, consequently, too fragile), Beck [5,6] proposed extreme programming (XP) principles and test-driven development (TDD). One trait of XP that has not derailed since, recommends automating testing *and* integration to render the development process more efficient. These very principles are now being taken up in DevOps practices [7]. TDD instead reverses the waterfall model by lifting tests to steer software design and thus provide instant feedback to developers. In TDD, each test has to be written *before* implementing the respective functionality, therefore focusing more deeply (as driven by immediate verification) on the decomposition of the possible use-cases to distill the behavior of every individual software unit. TDD can be applied to both top-down (integration tests with mock objects) and bottom-up (unit tests) approaches. For instance, a top-down approach combined with Continuous Integration (CI) helps uncover system integration problems earlier in the development process. According to Martin [8], TDD helps design organic architectures, getting rid of major bugs early in the software development life-cycle, thereby reducing the final costs of the software project.

The massive growth in size (lines of code), scale (distribution and deployment), and liveness of modern software systems poses a further massive challenge to software engineering practice by making traditional big-bang integration increasingly more impractical, if at all possible. This trend widens the gap between development (and test) and production environment, to the point that the former can hardly capture the (extreme) characteristics of the latter [9]. As a testimony to the relevance of this problem and its need for "lateral thinking", Netflix, supplier of large-scale live, business-critical distributed software services, conceived Chaos Engineering [9], a practice that aims to discover unpredictable failures of a complex distributed system running in production mode by injecting faults in isolated parts of it and monitoring the outcome of it.

Contribution. Since the most interesting distributed systems are private for obvious reasons of intellectual property, little knowledge is available on their software architecture and the best practices that were applied to build them [10]. Open-source and academic projects that experiment in this field therefore are a useful means to raise the awareness and the quality of discussion on the state of distributed computing. With this work, we present our design and implementation experience on a long-running academic project, which we developed exploring several state-of-the-art features and technologies. In particular, we designed, built and deployed a medium-size distributed system that realizes a city-traffic simulator usable through a streaming service. Even if smaller-scale than any of the systems we mentioned earlier, this work of ours exhibits traits that overlap with modern distributed services. We trust that the practices we experienced and the lessons we learned in our project may provide some insight on recurrent problems that industrial practitioners encounter when dealing with modern, heterogeneous, agile and reliable distributed systems.

The remainder of this paper is organized as follows. Section 2 surveys the state-of-the-art of distributed systems, focusing on the principles that we followed. Section 3 presents our solution in terms of architectural principles and technology selection. Section 4 reviews and evaluates our choices after the experience on our reference scenario. Finally, Sect. 5 distills some lessons learned.

2 Requirements

Hardware is intrinsically prone to wear and tear due to its tangible nature. Software instead does not decay per se over time, but is equally subject to faults. Where continuity of service is paramount, utmost attention has to be given to the supremely important quality of software *reliability*, and reflect in consequent architectural choices and implementation decisions. Reliability reflects the continuity of service warranted to end users. The common metrics used for it (e.g., MTBF, failure rate) consider downtime periods and number of failures incurred over time. While very familiar to the niche of cognizant developers, this seemingly obvious dimension of concern has for a long time escaped the attention of those working without the pressure of certification or qualification bodies. Recent events show how far this oblivion had spread.

In 2017, Amazon experienced an outage that triggered a four-hour downtime to a large number of commercial websites deployed on its cloud infrastructure. The failure occurred as an operator debugging the billing process on some distributed servers, hence underneath the services provisioned to the affected customers, accidentally shut down almost an entire region of servers. This event resulted in service disruption costs of ∼$150 million, seriously undermining the company's credibility as a reliable cloud provider.

In 2008, a corruption in the main database forced Netflix to shut down the whole system for 4 days. This catastrophe taught Netflix the lesson to depart from their earlier monolithic system and move toward a microservice-based architecture. That change-over took 8 years to complete for good, and, reportedly,

allowed them to reduce the downtime-per-year duration to 52 min, while being able to apply thousands of production changes per day.

In the past, mission-critical systems (those whose failure would crush the corresponding investment) used to be a small niche in software business. Presently, numerous business-critical systems are acquiring mission-critical traits as well, since a single failure in them can imperil the future of a whole enterprise. As a consequence, software reliability has returned to be a top concern, and motivated market leaders to develop and deploy a new host of best practices [11].

2.1 Architectural Principles

Our vision encompasses an evolving architecture, which is capable of mutating over time as need arises, instead of crystallizing in a fixed structure. In this section, we define the principles that inspired our architectural decisions.

(P1) Agility. An agile architecture is able to evolve over time while assuring the continuity of the services exposed by its interface. If the system is able to serve user requests incurring no downtime during updates, we can say that the system is *agile*. For a proof-of-concept that explores the agility of a microservices-based architecture, see [12]. Big players in the Cloud computing, messaging and video streaming service businesses, such as Amazon AWS, Google, Netflix and Facebook, have already embraced this principle.

(P2) Versatility. A distributed system is versatile if its composing parts can be used in different contexts. This principle implicitly embodies portability, whereby a portable component does not make any assumption on the platform(s) on which it is deployed. At the same time, being versatile also means being potentially reusable in different software systems, application context and not just execution platforms. A distributed system that uses versatile components can mutate its composition over time while maintaining (or extending) its goal, like a pattern that can have different instantiations, but just a single meaning, and one and the same interface to the outside.

(P3) Scalability. A distributed system is scalable when it is able to assure stable yield in the face of dynamic variation in the user load within one and the same active configuration [13]. When the system infrastructure is subject to a pay-per-use cost policy, scalability needs to become elastic, and the system components should be able to grow (scale out) or shrink (scale in) as needed, thus avoiding both under- and over-provisioning.

According to [14], scalability can be measured over the 3 axes shown in Fig. 1.

- *Vertical scalability (Y-axis)*: this property is achieved when the system's performance can elastically adapt to the current workload by adding or removing resources from a running node transparently to the end user.

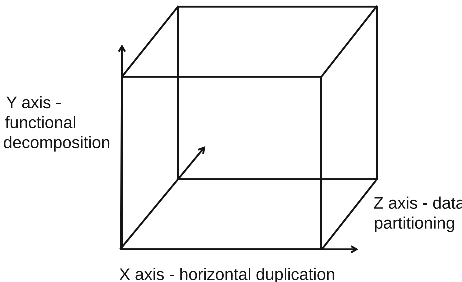


Fig. 1. The scale cube [14]

- *Horizontal scalability (X-axis)*: a system scales horizontally, aka scales out, when its design allows cloning one or more services (replicas) and distributing load among them in full transparency to the user. A system able to scale out can also be fault-tolerant, as the effects of local crashes in individual replicas can be kept hidden to the user, as long as one or more (hot) replicas are able to instantaneously replace the failing node.
- *In-depth scalability (Z-axis)*: the need for this property arises in the face of large amounts of stored data. A database is partitioned in disjoint subsets of data (i.e., shards) based on records attributes used as key when routing requests to shards. This approach improves fault isolation and transaction scalability since a failure makes inaccessible only part of the data and requests for multiple transactions may be distributed among several nodes.

In our project we did not have important storage needs. Our architecture did therefore focus only on vertical and horizontal scaling.

(P4) Simplicity. Seeking simplicity helps avoid bloated and unmaintainable architectures, by designing just the essential components and "small", focused, interfaces for them. Simplicity has a more retrospective connotation; it therefore often fails to catch the developer's attention early enough. Performing periodic refactoring may be a useful discipline to not lose sight of it. Refactoring, however, postulates agility.

2.2 Technology Requirements

No one software technology fits all needs. Conversely, different technologies are versed to solve different problems. Similarly, each microservice carries out a distinct, self-contained functionality of the system. The decoupled nature of the microservices architectural style allows integrating multiple best-of-breed technologies (each with their own stack) into a reliable whole, without entangling the corresponding needs and dependencies.

Hereafter, we outline the main technology requirements that guided the realization of our architecture as an aggregation of several heterogeneous and multi-

threaded components, which frequently interact with each other under the same umbrella, for delivering a reliable service.

(T1) High-level abstractions. Mature technologies capable of expressing concurrency at program level and of controlling it for predictability and performance are essential to implement multi-threading *reliably*. This is an enabler to distributed computing, and consequently also in demand for microservices architectures.

(T2) Modern testing framework. An agile architecture (in the sense of Sect. 2.1) should rest on a framework that can nimbly sustain increments. For example, incremental and continuous integration practices along with automated tests, executable in reproducible and isolated scenarios, provide rapid and valuable feedback for the developer and help steer the course of development.

(T3) Beware dependencies. Versatile systems leverage flexibility and seek interoperability among heterogeneous parts. Achieving this goal is massively aided by making components interact at the highest possible level, e.g., HTTP(S).

(T4) Economy matters. Development concerns should not overlook efficiency, which relates to the quantity of human resources that the project needs to deploy. This consideration suggests narrowing the spectrum of technologies that have to be mastered, while also selecting them based on the value that they can deliver to the final product (which looks back at the principles discussed in Sect. 2.1.

Table 1 provides a synoptic view of our requirements.

Table 1. Project requirements

Code	Name	Purpose
P1	Agility	Sustain increments without giving up stability
P2	Versatility	Facilitate reuse
P3	Scalability	Cope with variable amounts of demand
P4	Simplicity	Design easy-to-change systems
T1	Abstraction	Enable Inversion of Control
T2	Testing	Automatically verify software properties
T3	Interoperability	Connect heterogeneous technologies
T4	Economy	Bear technical debt to stay on schedule

3 Solutions

We now discuss the principal architectural patterns that we used in our project. Subsequently, we review the technology selection that preluded implementation.

3.1 Patterns

Owing to space limit, we focus on two patterns, which turned out to be especially useful in meeting the principles recalled in Sect. 2.1.

The *Layering* pattern facilitates decomposing the system into a hierarchical structure since handling the intricacy of a distributed system can be very challenging.

The *Pipe and Filter* pattern instead helps guide the design of the data-flow traits of the system.

Layering. This pattern is inspired by the UNIX protection rings and the TCP/IP model (which are veritable evergreens), and advocated by Martin [8] as a vector of a "clean" architecture. The fundamental idea behind this pattern is to *separate and isolate* the responsibilities of each part of the system.

A distributed system can be viewed as a cohesive set of functionalities which operate at different levels of abstraction (Fig. 2a). To mitigate the inordinate growth of complexity, concentrating (for implementation, verification and maintenance) on one specific concern should allow transparently ignoring other concerns. Yet, if the system parts do not have a clear (distinct, self-contained, not overlapping) role in the intended collaborations, the system architecture will suffer coupling, and the interface between components will be disorderly.

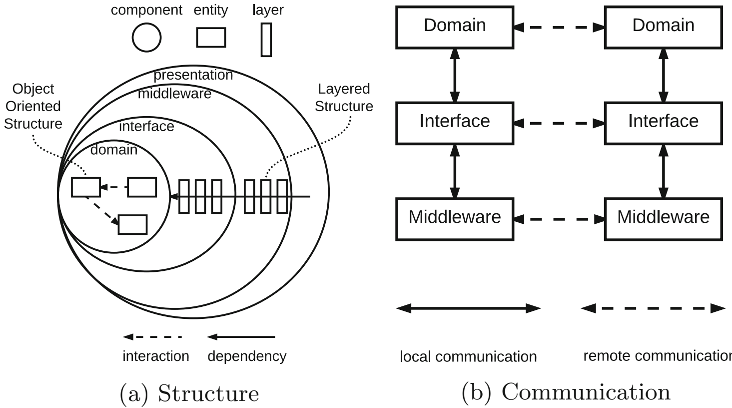
One useful recipe in this regard is to decompose the system into isolated layers whose adjacent pairs (typically co-located) communicate vertically, and the symmetrical peers (typically remote) communicate horizontally, as shown in Fig. 2b. Vertical communication between heterogeneous concerns needs well-defined call interfaces. Horizontal communication between homogeneous remote peers needs protocols.

Arguably, layering helps achieve *versatility* (P2) as it allows individual parts (layers) to be considered in isolation and be deployed or reused individually so long as their required interfaces are satisfied in place.

Pipe and Filter. This idea derives from the UNIX pipe-and-filter architectural style and has a significant positive impact on distributed systems. Its core concept is *separating data from behavior.* As Hohpe and Woolf [15] observe, this pattern divides a large processing task in a sequence of small and reusable steps of progress, thereby yielding natural parallelism.

A great variety of services imply continuous streams of data and messages to process, e.g., social networks, messaging services and video streaming providers. These systems need to process streams of data which can vary in size and amount, while being able to scale and to stay available and operative as the load increases. Moreover, these systems may not have a full a-priori knowledge of the features which characterize the processed data.

The whole processing can be decomposed in atomic steps called *filters*, each of them responsible for exactly one task. Filters expose a common interface and are connected among them by pipes, which provide basic producer-consumer



(a) Structure (b) Communication

Fig. 2. Layering pattern

capabilities, e.g., queues or channels. This decomposition process can be recursively applied starting from the higher levels of the architecture, as shown in Fig. 3. System throughput is limited by the slowest filter in the chain, which becomes its *bottleneck*. Furthermore, messages and streams of any sort can be wrapped in a uniform-type bundle that can be processed by all pipes and filters.

The resulting processing pipeline guarantees *scalability* (P3): filters can be parallelized since each data flow is independent of the others. Also, the Pipe and Filter architecture embodies *simplicity* (P4), because each part (filter) is responsible for a single task, and they can be composed as needed.

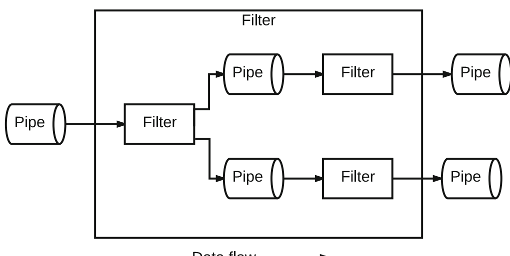


Fig. 3. Pipe and Filter architecture

3.2 Technology Selection

Our reference domain represents the traffic of a city (e.g., commuters, vehicles, transits), with inherent traits of (independent) parallelism and (collaborative) concurrency. We want our system to be able to sustain variable traffic load and balance it across service nodes as needed. This means designing it as a

distributed system. To be scalable and agile, our system needs to employ asynchronous communications. To this end, we chose a message-oriented approach, leveraging a message broker. Finally, to manage a software development that uses microservices, we need automated provisioning.

Based on these premises, we now illustrate how the scouted technologies respond to the requirements elicited in Sect. 2.2.

Concurrency. C++, Java and Ada implement the shared memory model, i.e., the interaction among threads is ruled by the state of shared memory regions. The former two languages do not provide complete and reliable concurrency abstractions [16]. Conversely, Ada guarantees thread-safe, polling-free access to shared (logical) resources in a concurrent environment through *Protected Objects*. Recently, C++17 has improved in this direction by introducing readers-writer locks[1], which mimic the semantics of Protected Objects.

The actor model is a completely different concurrency paradigm, initially implemented by Erlang [17] for soft real-time systems. This paradigm leverages asynchronous message passing between reactive entities, because each actor executes only after receiving a message. Concurrency hazards are avoided by processing one message at a time, hence allowing only single-threaded changes to the state of an actor. Since actors communicate with each other via messages, there is no restriction on their location.

Go provides another interesting concurrency model, inspired on the long-known communicating sequential processes (CSP) concept [18]. In this paradigm, goroutines (i.e., lightweight threads in Go) exchange information by means of typed and thread-safe queues called *channels*. This concurrency model slightly differs from the one implemented by Erlang since channels in Go can be consumed in an arbitrary order. On the other hand, channels may exchange data over several machines, therefore removing the need of sharing memory.

We choose Ada to implement our *business logic*. We found it offering a reliable and mature technology to handle concurrency (T1), with the extra bonus of strong typing (with a large range of static checks), object-oriented programming, and real-time events.

As a token to the rationale of our choice, at the core of our simulator a controller schedules and then executes activities, which represent the actions of travellers in our system. Ada's task abstraction and its accompanying `Ada.Real_Time` package enabled us to develop and test the controller with relatively little effort.

Distribution. A software technology intended for distribution should natively (in its runtime) include high-level primitives such as *send* or *receive* (T1), catering for process isolation and statelessness.

Ada or Go do not come with handy dependency-injection frameworks to manage distribution, so they do not represent a convenient choice in terms of economy (T4). Conversely, actor model-based solutions match our idea of distributed computing. Hence, we consider two state-of-the-art implementations of

[1] http://en.cppreference.com/w/cpp/thread/shared_mutex.

this model: Open Telecom Platform (OTP) [17] and Akka[2]. OTP is an Erlang standard library that natively supports location-transparent primitives and distributed computation. Akka is a Scala/Java framework inspired by OTP. Erlang and Java run on two different virtual machines (i.e., BEAM and JVM) and this difference significantly affects the behavior of their actor model implementations. With Akka, concurrent modifications may occur to the internal state of an actor. This is not possible in OTP. Moreover, as the JVM shares a common heap for all threads, garbage collection causes global pauses in the system run time. Conversely, thanks to the process isolation guaranteed by BEAM, the independent heap of each Erlang actor (i.e., lightweight thread) is garbage-collected separately without affecting the execution of any other actor. Furthermore, BEAM supports hot code swapping, allowing updates to the actor's code without stopping running processes (P1).

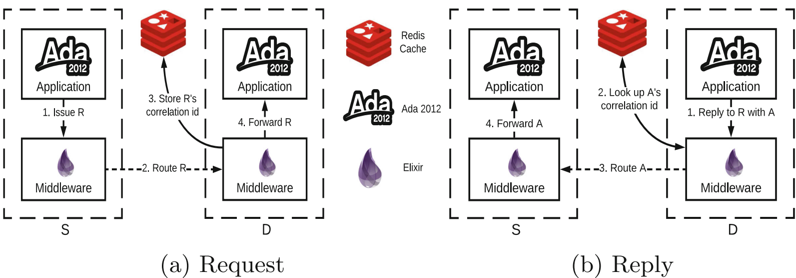
On these grounds, we chose Erlang and OTP in favor of Scala and Akka. In particular, we chose Elixir[3], an Erlang dialect, which we deemed to offer a gentler learning curve, and a number of features like the Mix build tool that can improve overall productivity (T4).

We wanted the application-level entities in our system to use asynchronous and reliable communication services, provided for by a middleware layer completely decoupled from the application logic. Instances of where this need arises in our design are the travellers who commute from one area (i.e., a node in our system infrastructure) to another, or wish to book a parking spot at their point of destination, which has to happen as determined by the city fabric. Let us illustrate how such as an application scenario develops in our system implementation. When an application-level entity resident on node S issues a request R for transport to a destination d (a building or a street), the following steps occur, as depicted in Fig. 4).

1. the middleware routes R to the destination node D of the system, where d is located, by looking it up in its routing table;
2. before passing R to the upper (application) layer, the middleware of D stores the correlation id and the source node S of R in its cache (i.e., Redis[4] in Fig. 4);
3. when the middleware of node D receives the answer A to R, it checks whether a request with a matching correlation id exists in its cache and, if it does, it subsequently routes A back to the node S where R originated;
4. the middleware of S sends A to its application layer, where it will be handled by an event loop.

We implemented this and other services by composing several Elixir actors. Our implementation leveraged advanced high-level abstractions (e.g., supervision trees), which enabled us to concentrate on realizing the desired service policies achieving reliability without losing efficiency (T4).

[2] https://akka.io/.
[3] https://elixir-lang.org/.
[4] https://redis.io/.



(a) Request (b) Reply

Fig. 4. Handling of a request

Message Brokers. There exists a broad variety of message brokers with different delivering, storing and filtering capabilities. Kafka [19] and RabbitMQ[5] represent two solutions characterized by high-throughput capabilities [20]. Kafka was initially developed at LinkedIn as a log processor with particular emphasis on partitioning and durability, whereas RabbitMQ offers more flexibility and advanced filtering options, which makes it better suited for real-time processing.

On balance, we chose RabbitMQ because: (i) it supports several standard protocols (e.g., AMQP, MQTT), which allows us to meet requirement (T3); and (ii) it is easy to deploy, as it supports dynamic scaling (P3), as well as to use, as it provides fresh and exhaustive documentation and tutorials (T4).

Another factor that made us lean toward RabbitMQ is its seamless integration with Erlang and therefore with Elixir. In particular, several libraries exist that weave RabbitMQ with GenStage, i.e., a recently introduced Elixir dataflow oriented abstraction. RabbitMQ and AMQP also provide a neat separation between direct (synchronous) queues, which can be used for inter-node communication, and topic (asynchronous) queues, which can be used in a publish-subscribe fashion to generate events for the frontend component of the system.

Microservices. Docker[6] is an open-source container platform, which has become the de-facto standard for the implementation of microservices.

Docker offers an agile way to package code and related dependencies in an isolated "runnable" environment that guarantees low start-up times (P1), typically in the order of ms [12]. Docker helps optimize the infrastructure costs by instantiating individual containers as lightweight and isolated user-space processes, which share the OS kernel among them, without need for hypervisor-based arbitration, hence without hypercall-API or HW virtualization.

Docker also provides Docker Compose, which assists the user in configuring different deployment environments, addressing specific development and test needs.

[5] https://www.rabbitmq.com/.
[6] https://www.docker.com/.

Docker Swarm[7] is an even higher-level tool, shipped with the standard installation from *Docker* 1.12.0 onwards, as a native mode to manage clusters of Docker Engines[8]. The gist of Docker Swarm is to automate most part of the features that it provides (e.g., service discovery, load balancing, TLS authentication), only requiring very little configuration on the user side. Being a native Docker solution, Docker Swarm enjoys fast deployment and excellent integration with direct use by means of the Docker API.

Kubernetes is a technology-agnostic (T3) cluster orchestration system, which embodies replication and service discovery as its core primitives. Google conceived Kubernetes after over 10 years of experimentation with container management systems [21], and that solution has been subsequently adopted by OpenStack and other big players, e.g., Ebay, Yahoo, Comcast.

Both Docker Swarm and Kubernetes are open-source. They support declarative configuration recipes (YAML), provide high-availability through replication, and are supported by common Cloud platforms like AWS and emerging container-OS technologies like Rancher[9]. Kubernetes is complex to configure, but more mature and fault-tolerant than Docker Swarm, which is very tightly tied to the underlying Docker Engine.

With attention to requirement T4, we chose Docker to implement our microservice architecture, and Docker Swarm for orchestration. Docker helps achieve agility (P1), scalability (P3) and versatility (P2), while Docker Swarm is a simple-to-use solution that does not incur further dependencies (cf. T4).

Docker was one of the key technologies in our system: it was essential to isolate and package code dependencies in standalone containers. Hence, by leveraging the container abstraction, our system can smoothly mix several best-of-breed niche technologies.

Configuration Management and Automation. Among the available testing frameworks, we choose xUnit [6] because it is simple to use (P4, T4), thanks to support for an assertion-based verification of test cases. xUnit is flexible as it allows grouping together independent unit tests by means of suites based on test fixtures, which define the required set-up and tear-down scenarios.

We chose this framework as it is widely supported as a plug-in by a large number of project management tools (e.g., Eclipse or Apache Maven), which is a token of its maturity. In particular, we used its *AUnit* and *ExUnit* instances to respectively test Ada and Elixir code.

For the purposes of Continuous Integration (CI), we considered Travis[10] and Jenkins[11], which are the dominant solutions in the professional market. We eventually choose the latter, due to our satisfactory previous experience with it (T4), with the additional bonus of being open-source as opposed to the proprietary nature of the former.

[7] https://docs.docker.com/engine/swarm/.
[8] A Docker Engine is the OS-level service which handles the container runtime.
[9] https://rancher.com/rancher-os/.
[10] https://travis-ci.org/.
[11] https://jenkins.io/.

In terms of Cloud services, we have had previous experience with Digital Ocean and AWS, with the latter faring better for reliability and maturity.

We therefore used an Amazon Elastic Compute Cloud (or EC2) instance to install our CI server. Using the Github integration plug-in, we configured a Github webhook to trigger our Jenkins master node hosted on AWS after each commit on selected repository branches (Fig. 5). The Jenkins master schedules a job to a specific Jenkins slave which draws changes from a remote branch and then it notifies through e-mail the intended developer in case of failure.

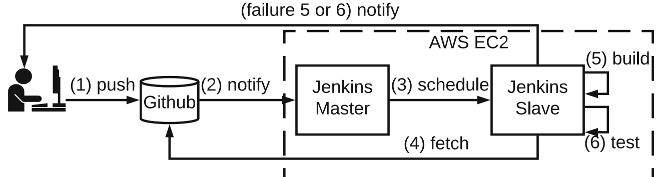


Fig. 5. CI workflow

We extensively used the Make tool and GNAT Project Manager (GPR) for building and testing of the Ada component. The presence of numerous of external libraries (e.g., GNATColl, dynamically linked, and an AI library written in C++, statically linked) increased the complexity of our build process.

We also used Mix, the software project management tool bundled with the Elixir installation, which eases defining different deployment environments, handling dependencies, configuring the project structure and the test environment.

The aggregation of these automation tools helped us greatly to achieve agility and reliability, reducing the cost of change in the long run (T2).

4 Evaluation

In conclusion, we discuss our retrospective assessment of how far much our design *and* implementation decisions were able to meet the architectural *and* technology requirements discussed in Sects. 4.1 and 4.2 respectively. The *and* conjunction in between the two parts of this challenge is the key trait of this discussion: selecting technologies that are best-of-breed in their own camp is a viable decision only as long as they can integrate seamlessly into a system (Fig. 6), and the corresponding development process, that have the sought properties.

4.1 Principles

Pursuing the architectural principles singled out in Sect. 2.1 was hard.

We learned that agility (P1) can be reached thanks to the combination of adopting the microservices architectural style and using the technologies that associate well with it, e.g., Docker and the Erlang ecosystem. We also found that

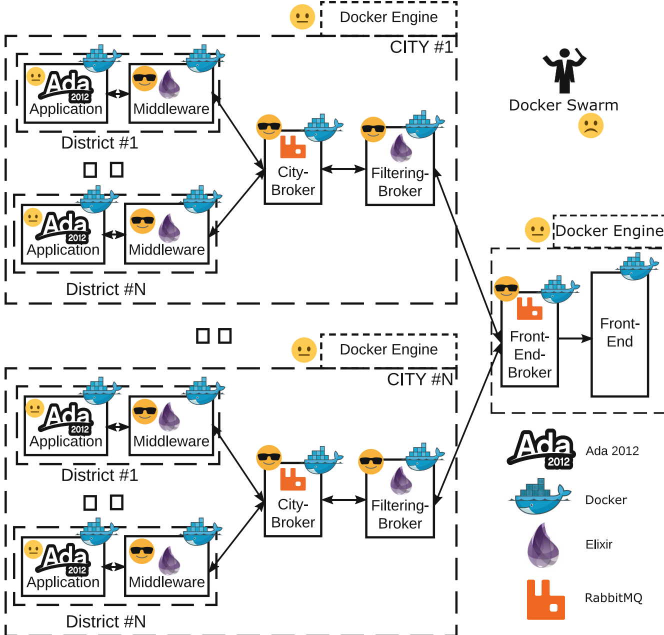


Fig. 6. The software architecture and the corresponding technology choices

managing container updates within a medium-large codebase cleanly, requires experience and becomes extremely hard for early adopters. Admittedly, our solution instance has much room for improvement in this respect.

Versatility (P2) is possible even in large systems, and the layering pattern (Sect. 3.1) promotes cohesion and reusability of each single tier. Retrospectively, however, we realized that architectural principles alone do not assure sufficient portability, as technology support plays a large part (best if gently, but firmly proactive) in making that possible. For example, Docker forces (as opposed to loosely suggests) one to implement in ways that actively pursue portability and reusability, owing to the decoupling that stems from embracing the microservices style.

Simplicity (P3) can be evaluated subjectively, hence without standard metrics. Nevertheless, using cut-to-the-bone interfaces in the communication between our business logic and middleware layers, allowed us to define a clear boundary between major architectural parts of our system.

Our hands-on experience has taught us that scalability (P4) can be achieved concretely, and in fact rather easily, with a microservices architecture. Scalability was also aided by adopting the Pipe and Filter pattern, which we adhered to assuring the statelessness of the individual parts of it.

To ripe the full yield of our architectural principles, however, part of the project effort (especially its initial segment) should be devoted to putting into

place a robust and productive deployment system, without which incremental integration would be a hard-to-reach objective.

4.2 Technologies

Ada is most effective to handle concurrency. Yet, having been designed with embedded systems in mind, it is closed to interoperability and tooling. These shortcomings make Ada's learning curve steeper than other comparable languages (Table 2). Elixir is "a modern Erlang", hence it earns all the Erlang benefits plus an easy-to-use feel and good support by a thriving community. Nevertheless, Elixir has a serious drawback in dynamic typing, which hinders debugging. RabbitMQ provides support for standard protocols, it actually turned out to be simple and reliable, and integrates perfectly with Docker containers. Docker is not fully supported by Windows and its higher level tools have a perceptibly limited flexibility. Finally, Jenkins is straightforward to use, but it lacks working plug-ins for Docker Compose; nevertheless, this limitation can be obviated with a custom (even though unpractical) solution, e.g., shell scripts.

Table 2. Evaluation summary

Ambit	Technology	Pros	Cons
Concurrency, Testing, IDE, Project Management	Ada, AUnit, GPS, GPR	Reliability, Inversion of Control, Modularity	Low interoperability, Steep learning curve
Middleware, Testing, Project Management	Elixir, exUnit, Mix	Resiliency, Simplicity, Inversion of Control, Performance, Productivity	Dynamic Typing, Awkward error reporting
Broker	RabbitMQ	Simplicity, Monitoring tools	-
Containerization, Configuration, Orchestration	Docker, Docker Compose, Docker Swarm	Agility, Efficiency, Simplicity	Limited portability, Lack of flexibility
CI	Jenkins	Simplicity, Flexibility	Poor Docker support

5 Lessons Learned

Our system comprised $\sim$100 k SLOC (50% of which written in Ada) and took us one year to develop, from design to implementation, continuous integration, and deployment, for a 3-student team working remotely outside of academic hours, equivalent to 250 person-days of effort. We trust this quantity of effort qualifies

our project experience as a sound case study for interested practitioners to draw concrete indications on the present and the future of reliable distributed systems in relation to current state-of-the-art of microservices solutions.

Ultimately, in our endeavor we learned that container-based solutions allow niche technologies to emerge as a real alternative for in-the-large software systems, as long as these technologies come with *high-level support for interoperability*, such as e.g. HTTP(S) and *seamless integration with automated production tools* such as e.g., Jenkins.

References

1. Vashi, S., Ram, J., Modi, J., Verma, S., Prakash, C.: Internet of things (IoT): a vision, architectural elements, and security issues. In: 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), pp. 492–496, February 2017
2. Charette, R.N.: Why software fails [software failure]. IEEE Spectr. **42**(9), 42–49 (2005). https://doi.org/10.1109/MSPEC.2005.1502528
3. Blair, M., Obenski, S., Bridickas, P.: Patriot missile defense: Software problem led to system failure at Dhahran. Report GAO/IMTEC-92-26 (1992)
4. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, New York (2011)
5. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, Boston (2000)
6. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Boston (2003)
7. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston (2015)
8. Martin, R.C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, Englewood Cliffs (2017)
9. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos engineering. IEEE Softw. **33**(3), 35–41 (2016)
10. Pääkkönen, P., Pakkala, D.: Reference architecture and classification of technologies, products and services for big data systems. Big Data Res. **2**(4), 166–186 (2015). https://doi.org/10.1016/j.bdr.2015.01.001
11. Maurer, B.: Fail at scale. Queue **13**(8), 30:30–30:46 (2015). http://doi.acm.org/10.1145/2838344.2839461
12. Simioni, A., Vardanega, T.: In pursuit of architectural agility: experimenting with microservices. Submitted to IEEE International Conference on Services Computing (2018)
13. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis (2000). aAI9980887
14. Abbott, M.L., Fisher, M.T.: The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Pearson Education (2009)
15. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Reading (2004)
16. Goetz, B., Peierls, T.: Java Concurrency in Practice. Pearson Education (2006)
17. Armstrong, J.: Erlang. Commun. ACM **53**(9), 68–75 (2010). http://doi.acm.org/10.1145/1810891.1810910

18. Hoare, C.A.R.: Communicating sequential processes. In: Hansen, P.B. (ed.) The Origin of Concurrent Programming, pp. 413–443. Springer, New York (1978). https://doi.org/10.1007/978-1-4757-3472-0_16
19. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
20. Dobbelaere, P., Esmaili, K.S.: Kafka versus RabbitMQ: a comparative study of two industry reference publish/subscribe implementations: industry paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, DEBS 2017, pp. 227–238. ACM, New York (2017). http://doi.acm.org/10.1145/3093742.3093908
21. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016). http://doi.acm.org/10.1145/2890784

Author Index

Alonso, Alejandro 73

Blieberger, Johann 53
Burgstaller, Bernd 53

Carlson, Jan 19, 87
Crespo, Alfons 105

de la Puente, Juan A. 73
Dobrin, Radu 87

Gallina, Barbara 19
Garrido, Jorge 73
Gutiérrez, J. Javier 123

Hansson, Hans 19

Jaradat, Omar 3

Maalej, Maroua 37
Marković, Filip 87
Martorell, Xavier 141

Moy, Yannick 37
Munari, Stefano 158

Pérez, Héctor 123
Pinho, Luis Miguel 141
Punnekkat, Sasikumar 3
Puri, Stefano 19

Quiñones, Eduardo 141

Real, Jorge 105
Royuela, Sara 141

Sáez, Sergio 105
Sljivo, Irfan 19

Taft, Tucker 37

Valle, Sebastiano 158
Vardanega, Tullio 158

Zamorano, Juan 73