Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, NY 10121-0701

April 2013 Ada Letters • Volume XXXII, Number 1

Ada LETTERS

Volume XXXII Number 1 April 2013

Table of Contents

A Publication of SIGAda,
the ACM Special Interest Group on Ada

ACM Digital Library

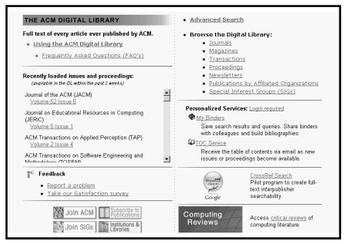
www.acm.org/dl

The Ultimate Online INFORMATION TECHNOLOGY Resource!

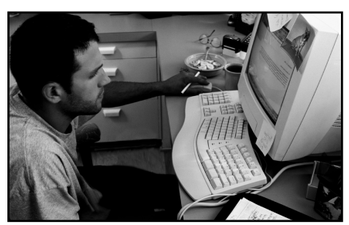
Powerful and vast in scope, the **ACM Digital Library** is the ultimate online resource offering unlimited access and value!

The **ACM Digital Library** interface includes:

• **The ACM Digital Library** offers over 45 publications including all ACM journals, magazines, and conference proceedings, plus vast archives, representing over two million pages of text. The ACM DL includes full-text articles from all ACM publications dating back to the 1950s, as well as third-party content with selected archives. **www.acm.org/dl**



• **The Guide to Computing Literature** offers an enormous bank of more than one million bibliographic citations extending far beyond ACM's proprietary literature, covering all types of works in computing such as journals, proceedings, books, technical reports, and theses! **www.acm.org/guide**



• **The Online Computing Reviews Service** includes reviews by computing experts, providing timely commentary and critiques of the most essential books and articles.

Available only to ACM Members.
Join ACM online at **www.acm.org/joinacm**

To subscribe to the ACM Digital Library, contact ACM Member Services:

Phone: 1.800.342.6626 (U.S. and Canada)
+1.212.626.0500 (Global)
Fax: +1.212.944.1318
Hours: 8:30 a.m.-4:30 p.m., Eastern Time
Email: acmhelp@acm.org
Mail: ACM Member Services
General Post Office
PO Box 30777
New York, NY 10087-0777 USA

ACM Professional Members can add the ACM Digital Library for only $99 (USD). Student Portal Package membership includes the Digital Library. Institutional, Corporate, and Consortia Packages are also available.

Association for Computing Machinery

Advancing Computing as a Science & Profession

AD10

join today!

SIGAda & ACM

www.acm.org/sigada **www.acm.org**

The **ACM Special Interest Group on Ada Programming Language** (SIGAda) provides a forum on all aspects of the Ada language and technologies, including usage, education, standardization, design methods, and compiler implementation. Among the topics that SIGAda addresses are software engineering practice, real-time applications, high-integrity & safety-critical systems, object-oriented technology, software education, and large-scale system development. SIGAda explores these issues through an annual international conference, special-purpose Working Groups, active local chapters, and its *Ada Letters* publication.

The **Association for Computing Machinery** (ACM) is an educational and scientific computing society which works to advance computing as a science and a profession. Benefits include subscriptions to *Communications of the ACM*, *MemberNet*, *TechNews* and *CareerNews*, full and unlimited access to online courses and books, discounts on conferences and the option to subscribe to the ACM Digital Library.

- ❑ SIGAda (ACM Member) ..... $ 25
- ❑ SIGAda (ACM Student Member & Non-ACM Student Member) ..... $ 10
- ❑ SIGAda (Non-ACM Member) ..... $ 25
- ❑ ACM Professional Membership ($99) & SIGAda ($25) $124
- ❑ ACM Professional Membership ($99) & SIGAda ($25) & ACM Digital Library ($99) $223
- ❑ ACM Student Membership ($19) & SIGAda ($10) ..... $ 29
- ❑ *Ada Letters* only ..... $ 53
- ❑ Expedited Air for *Communications of the ACM* (outside N. America) ..... $ 58

payment information

Name _____

ACM Member # _____

Mailing Address _____

_____

City/State/Province _____

ZIP/Postal Code/Country _____

Email _____

Mobile Phone _____

Fax _____

Credit Card Type: ❑ AMEX ❑ VISA ❑ MC

Credit Card # _____

Exp. Date _____

Signature _____

Make check or money order payable to ACM, Inc

ACM accepts U.S. dollars or equivalent in foreign currency. Prices include surface delivery charge. Expedited Air Service, which is a partial air freight delivery service, is available outside North America. Contact ACM for more information.

Mailing List Restriction
ACM occasionally makes its mailing list available to computer-related organizations, educational institutions and sister societies. All email addresses remain strictly confidential. Check one of the following if you wish to restrict the use of your name:

- ❑ ACM announcements only
- ❑ ACM and other sister society announcements
- ❑ ACM subscription and renewal notices only

Questions? Contact:
ACM Headquarters
2 Penn Plaza, Suite 701
New York, NY 10121-0701
voice: 212-626-0500
fax: 212-944-1318
email: acmhelp@acm.org

Remit to:
ACM
General Post Office
P.O. Box 30777
New York, NY 10087-0777

SIGAPP

www.acm.org/joinsigs

Association for Computing Machinery

Advancing Computing as a Science & Profession

Volume XXXIII Number 1, April 2013

Table of Contents

A Publication of SIGAda,
the ACM Special Interest Group on Ada

For advertising information contact:
Advertising Department
2 Penn Plaza, Suite 701, New York, NY 10121-0701
Phone: (212) 869-7440; Fax (212) 869-0481

Is your organization recognized as an Ada supporter? Become a SIGAda INSTITUTIONAL SPONSOR! Benefits include having your organization's name and address listed in every issue of Ada Letters, two subscriptions to Ada Letters and member conference rates for all of your employees attending SIGAda events. To sign up, contact Rachael Barish, ACM Headquarters, 2 Penn Plaza, Suite 701, New York, NY 10121-0701, and email: MEETING@ACM.ORG, Phone: 212-626-0603.

Interested in reaching the Ada market? Please contact Jennifer Booher at Worldata (561) 393-8200 Ext. 131, email: platimer@worldata.com. Please make sure to ask for more information on ACM membership mailing lists and labels.

Ada Letters (ISSN 1094-3641) is published three times a year by the Association for Computing Machinery, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA. The basic annual subscription price is $20.00 for ACM members.

POSTMASTER: Send change of address to Ada Letters:
ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA

From the Editor's Desk

Alok Srivastava

Welcome to this issue of ACM Ada Letters. In this issue you will find very remarkable papers presented at 15th International Real-Time Ada Workshop (IRTAW-15) in Liébana (Cantabria), Spain. Since the late Eighties the International Real-Time Ada Workshop series has provided a forum for identifying issues with real-time system support in Ada and for exploring possible approaches and solutions, and has attracted participation from key members of the research, user, and implementer communities worldwide. Recent IRTAW meetings have significantly contributed to the Ada 2005 standard and to the proposals for Ada 2012, especially with respect to the tasking features, the real-time and high-integrity systems annexes, and the standardization of the Ravenscar profile. The summary and outcome of various focused sessions during the IRTAW-15 are listed here.

In this issue you will find the Call for Technical Contributions for the High-Integrity Language Technology SIGAda 2013 conference to be held from November 10-14, 2013 in Pittsburgh, Pennsylvania (USA). The conference will feature three outstanding keynote speakers Edmund M. Clarke from Carnegie Mellon University and 2007 ACM Turing Award winner, Jeannette Wing from Microsoft Research and Ada veteran John Goodenough from Software Engineering Institute. Another major Ada event, the 18th International Conference on Reliable Software Technologies Ada-Europe 2013 to be held from June 10 to14, 2013 in Berlin, Germany.

In this issue you will also find one very interesting research paper "Investigating SystemAda: TLM_FIFO Detailed Characteristics Proof, TLM2.0 Interfaces Implementation, Simulation Time Comparison to SystemC" by our regular contributor Negin Mahani. In another paper by Takeo Ekiba, Yuichi Goto, and Jingde Cheng from Saitama University, Japan have presented some new types of tasking deadlocks concerning the new synchronization waiting relations defined in the Ada 2012.

Ada Letters is a great place to submit articles of your experiences with the language revision, tips on usage of the new language features, as well as to describe success stories using Ada. We'll look forward to your submission. You can submit either a MS Word or Adobe PDF file (with 1" margins and no page numbers) to our technical editor:

Pat Rogers, Ph.D.
AdaCore, 207 Charleston, Friendswood, TX 77546 (USA)
+1 281 648 3165, Rogers@AdaCore.Com

We look forward to hearing from you!

Alok Srivastava, Ph.D.
Technical Fellow, TASC Inc.
475 School St, SW; Washington, DC 20024 (USA)
+1 202 314 1419 Alok.Srivastava@TASC.Com

Editorial Policy (from Alok Srivastava, Managing Editor)

As the editor of ACM Ada Letters, I'd like to thank you for your continued support to ACM SIGAda, R&D in the areas of High Reliability and Safety Critical Software Development and encourage you to submit articles for publication. In addition, if there is some way we can make **ACM Ada Letters** more useful to you, please let me know. Note that Ada Letters is now on the web! See http://www.acm.org/sigada/ada_letters/index.html. The two newest issues are available only to SIGAda members. Older issues beginning March 2000 are available to all.

Now that Ada is standing on its own merits without the support of the DoD, lots of people and organizations have stepped up to provide new tools, mechanisms for compiler validation/assessment, and standards (especially ASIS). The Ada 2012 language version is fulfilling the market demand of robust safety and security elements and thereby generating a new enthusiasm into the software development. Ada Letters is a venue for you to share your successes and ideas with others in the Ada and specifically in High Reliability Safety Critical Software Development community. Be sure to take advantage of it so that we can all benefit from each other's learning and experience.

As some of the other ACM Special Interest Group periodicals have moved, Ada Letters also transitioned to a **tri-annual publication**. With exception of special issues, Ada Letters now is going to be published three times a year, with the exception of special issues. The revised schedules and submission deadlines are as follows:

Deadline	**Issue**	**Deadline**	**Issue**
June 1st, 2013	August, 2013	October 1st, 2013	December, 2013
February 1st, 2014	April, 2014	June 1st, 2014	August, 2014

Please send your article to Dr. Pat Rogers at rogers@adacore.com

Guidelines for Authors

Letters, announcements and book reviews should be sent directly to the Managing Editor and will normally appear in the next corresponding issue.

Proposed articles are to be submitted to the Technical Editor. Any article will be considered for publication, provided that topic is of interest to the SIGAda membership. Previously published articles are welcome, provided the previous publisher or copyright holder grants permission. In particular, keeping with the theme of recent SIGAda conferences, we are interested in submissions that demonstrate that "Ada Works." For example, a description of how Ada helped you with a particular project or a description of how to solve a task in Ada are suitable.

Although Ada Letters is not a refereed publication, acceptance is subject to the review and discretion of the Technical Editor. In order to appear in a particular issue, articles must be submitted far enough in advance of the deadline to allow for review/edit cycles. Backlogs may result in an article's being delayed for two or more issues. Contact the Managing Editor for information on the current publishing queue.

Articles should be submitted electronically in one of the following formats: MS Word (preferred) Postscript, or Adobe Acrobat. All submissions must be formatted for US Letter paper (8.5" x 11") with one inch margins on each side (for a total print area of 6.5" x 9") with no page numbers, headers or footers. Full justification of text is preferred, with proportional font (preferably Times New Roman, or equivalent) of no less than 10 points. Code insertions should be presented in a non-proportional font such as Courier.

The title should be centered, followed by author information (also centered). The author's name, organization name and address, telephone number, and e-mail address should be given. For previously published articles, please give an introductory statement (in a distinctive font) or a footnote on the first page identifying the previous publication. ACM is improving member services by creating an electronic library of all of its publications. Read the following for how this affects your submissions.

Notice to Contributing Authors to SIG Newsletters:

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library
- to allow users to copy and distribute the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will make every effort to refer requests for commercial use directly to you.

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

Back Issues

Back issues of Ada Letters can be ordered at the price of $6.00 per issue for ACM or SIGAda members; and $9.00 per issue for non-ACM members. Information on availability, contact the ACM Order Department at 1-800-342-6626 or 410-528-4261. Checks and credit cards only are accepted and payment must be enclosed with the order. Specify volume and issue number as well as date of publication. Orders must be sent to:

ACM Order Department, P.O. Box 12114, Church Street Station, New York, NY 10257 or via FAX: 301-528-8550.

KEY CONTACTS

Technical Editor
Send your book reviews, letters, and articles to:

Pat Rogers
AdaCore
207 Charleston
Friendswood, TX 77546
+1-281-648 3165
Email: rogers@adacore.com

Managing Editor
Send announcements and short notices to:

Alok Srivastava
TASC Inc.
475 School Street, SW
Washington DC 20024
+1-202-314-1419
Email: Alok.Srivastava@tasc.com

Advertising
Send advertisements to:

William Kooney
Advertising/Sales Account Executive
2 Penn Plaza, Suite 701
New York, NY 10121-0701
Phone: +1-212-869-7440
Fax: +1-212-869-0481

Local SIGAda Matters
Send Local SIGAda related matters to:

Greg Gicca
AdaCore
1849 Briland Street
Tarpon Springs, FL 34689, USA
Phone: +1-646-375-0734
Fax: +1-727-944-5197
Email: Gicca@AdaCore.Com

Ada CASE and Design Language Developers Matrix
Send ADL and CASE product Info to:

Judy Kerner
The Aerospace Corporation
Mail Stop M8/117
P.O. Box 92957
Los Angeles, CA 90009
+1-310-336-3131
Email: kerner@aero.org

Ada Around the World
Send Foreign Ada organization info to:

Dirk Craeynest
c/o K.U.Leuven, Dept. of Computer Science,
Celestijnenlaan 200-A, B-3001 Leuven (Heverlee)
Belgium
Email: Dirk.Craeynest@cs.kuleuven.be

Reusable Software Components
Send info on reusable software to:

Trudy Levine
Computer Science Department
Fairleigh Dickinson University
Teaneck, NJ 07666
+1-201-692-2000
Email: levine@fdu.edu

SIGAda Working Group (WG) Chairs

See http://www.acm.org/sigada/ for most up-to-date information

Ada Application Programming Interfaces WG
Geoff Smith
Lightfleet Corporation
4800 NW Camas Meadows Drive
Camas, WA 98607
Phone: +1-503-816-1983
Fax: +1-360-816-5750
Email: gsmith@lightfleet.com

Ada Semantic Interface Specification WG
http://www.acm.org/sigada/wg/asiswg/asiswg.html
Bill Thomas
The MITRE Corp
7515 Colshire Drive
McLean, VA 22102-7508
Phone: +1-703-983-6159
Fax: +1-703-983-1339
Email: BThomas@MITRE.Org

Education WG
http://www.sigada.org/wg/eduwg/eduwg.html
Mike Feldman
420 N.W. 11th Ave., #915
Portland, OR 97209-2970
Email: MFeldman@seas.gwu.edu

Standards WG
Robert Dewar
73 5th Ave.
New York, NY 10003
Phone: +1-212-741-0957
Fax: +1-232-242-3722
Email: dewar@cs.nyu.edu

Ada Around the World
(National Ada Organizations)

From: http://www.ada-europe.org/members.html

Ada-Europe
Tullio Vardanega
University of Padua
Department of Pure and Applied Mathematics
Via Trieste 63
I-35121, Padova, Italy
Phone: +39-049-827-1359
Fax: +39-049-827-1444
E-mail: tullio.vardanega@math.unipd.it
http://www.ada-europe.org/

Ada-Belgium
Dirk Craeynest
C/o K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200-A, B-3001
Leuven (Heverlee), Belgium
Phone: +32-2-725 40 25
Fax : +32-2-725 40 12
E-mail: Dirk.Craeynest@cs.kuleuven.be
http://www.cs.kuleuven.be/~dirk/ada-belgium/

Ada in Denmark
Jørgen Bundgaard
E-mail: Info at Ada-DK.org
http://www.Ada-DK.org/

Ada-Deutschland
Peter Dencker, Steinackerstr. 25
D-76275 Ettlingen-Spessartt, Germany
E-mail: dencker@parasoft.de
http://www.ada-deutschland.de/

Ada-France
Association Ada-France
c/o Jérôme Hugues
Département Informatique et Réseau
École Nationale Supérieure des Télécommunications, 46, rue Barrault
75634 Paris Cedex 135, France
E-mail: bureau@ada-france.org
http://www.ada-france.org/

Ada Spain
J. Javier Gutiérrez
P.O. Box 50.403
E-28080 Madrid, Spain
Phone: +34-942-201394
Fax : +34-942-201402
E-mail: gutierjj@unican.es
http://www.adaspain.org/

Ada in Sweden
Rei Stråhle
Box Saab Systems
S:t Olofsgatan 9A
SE-753 21 Uppsala, Sweden
Phone: +46-73-437-7124
Fax : +46-85-808-7260
E-mail: Rei.Strahle@saabgroup.com
http://www.ada-i-sverige.se/

Ada in Switzerland
Ahlan Marriott
White Elephant GmbH
Postfach 327
CH-8450 Andelfingen, Switzerland
Phone: +41 52 624 2939
Fax : +41 52 624 2334
E-mail: ada@white-elephant.ch
http://www.ada-switzerland.org/

Italy
Contact: tullio.vardanega@math.unipd.it

Ada-Europe Secretariat
e-mail: secretariat@ada-europe.org

Support for Multiprocessor Platforms

A. Burns and A.J. Wellings
Department of Computer Science
University of York, UK
burns@cs.york.ac.uk, andy@cs.york.ac.uk

Abstract

Multiprocessor platforms are becoming the norm for more powerful embedded real-time systems. Although Ada allows its tasks to be executed on such platforms, until recently it has provided no explicit support. The new revision of Ada now provides facilities for allocating and scheduling tasks on SMP platforms. The notions of a CPU and Dispatching-Domain have been introduced. We summaries these facilities and review the extra support that could be provided in the future for SMP and non-SMP platforms.

1 Introduction

The support that Ada provides, or should provide, for programs executing on multiprocessor or multicore platforms has been the subject of a number of IRTAW papers [11, 10, 6, 12, 8] and discussion sessions [7, 5] at the last two workshops. This is in addition to the many papers there have been, since the very first workshop, on distributed systems.

At the last workshop a number of recommendation were made for extensions to be included in the current revision to the Ada standard. Many, but not all, of these suggestions have been incorporated into the draft language definition that is currently been finalised. In this paper we briefly review the features that will be supported in Ada 2012, and revisit the many other topics that have been discussed previously but have are not made it into the Standard. Our aim is to inform a possible workshop discussion on what should be included in future amendments of the language, and what features might better be addressed by other formal or defacto standards.

2 Basic Requirements

The primary requirement for supporting the execution of Ada tasks on symmetric multiprocessors (SMPs) is to manage the mapping of tasks to processors [4]. We assume that we are concerned with real-time code, in which case the execution of any task can be view as a sequence of invocations or *jobs*. Between jobs, the task is blocked, waiting either for an event (typically an external interrupt) or for a future time instance. In this paper we do not discuss support for resource sharing between jobs. This topic is discussed in depth in an accompanying paper to the workshop.

To cater for the allocation/mapping of tasks/jobs to processors, two basic approaches are possible:

1. **Fully Partitioned** – each task is allocated to a single processor on which all its jobs must run; and
2. **Global** – all tasks/jobs can run on all processors, jobs may migrate during execution.

There are many motivation for choosing either global or partitioned allocation, some of these motivations come from issues of scheduling [2]. These details are not significant here, what is important is that the Ada language is able to support both schemes.

From these schemes, two further variants are commonly discussed: for global scheduling, tasks are restricted to a subset of the available CPUs; and for partitioned scheduling, the program can explicitly change a task's affinity and hence cause it to be moved at run-time.

Restricting the set of CPUs on which a task can be globally scheduled supports scalability – as platforms move to contain hundreds of CPUs, the overheads of allowing full task migration become excessive and outweighs any advantage that might accrue from global scheduling. Controlled changing of a task's affinity has been shown to lead to improved schedulability for certain types of application [9, 1, 3].

In the following discussions, in keeping with the terminology in the new Ada Standard, we will use the term *dispatching domain* to represent a group of processors across which global scheduling occurs. A task is said to be assigned to a dispatching domain and may also be restricted to a particular CPU.

For non-SMPs there are many other issues to consider. For cc-NUMA architectures these include the memory map, partitioning of the heap and control over object location. We will return to these issues in Section 4.2.

3 Features in Ada 2012

The following packages allows the group of CPUs to be partitioned into a finite set of non-overlapping 'Dispatching_Domains'. One dispatching domain is defined to be the 'System' dispatching domain; the environmental task and any derived from that task are allocated to the 'System' dispatching domain. Subprograms are defined to allow new dispatching domains to be created.

Tasks can be assigned to a dispatching domain and be globally scheduled within that dispatching domain; alternatively they can be assigned to a dispatching domain and restricted to a specific CPU within that dispatching domain. Tasks cannot be assigned to more than one dispatching domain, or restricted to more than one CPU.

The first package just defines a parent unit for all microprocessor facilities, and give a range for an integer representation of each CPU. Note the value 0 is used to indicate `Not_A_Specific_CPU`; so the CPUs are actually numbered from one. The function `Number_Of_CPUs` will, for any execution of the program, return the same value. In effect the number of CPUs is a constant and reflects the number of available processors at system start up.

```
package System.Multiprocessors is
  pragma Preelaborate(Multiprocessors);
  type CPU_Range is range 0 .. <implementation-defined>;
  Not_A_Specific_CPU : constant CPU_Range := 0;
  subtype CPU is CPU_Range range 1 .. CPU_Range'Last;
  function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

The second package provides the support for dispatching domains.

```
with Ada.Real_Time;
package System.Multiprocessors.Dispatching_Domains is
  pragma Preelaborate(Dispatching_Domains);
  Dispatching_Domain_Error : exception;
  type Dispatching_Domain (<>) is limited private;
  System_Dispatching_Domain : constant Dispatching_Domain;
  -- initially all CPUs are in System_Dispatching_Domain
  function Create(First, Last : CPU) return Dispatching_Domain;
  -- removes specified CPUs from System_Dispatching_Domain
  function Get_First_CPU(Domain : Dispatching_Domain) return CPU;
  function Get_Last_CPU(Domain : Dispatching_Domain) return CPU;
  function Get_Dispatching_Domain(T : Task_Id := Current_Task)
    return Dispatching_Domain;
  procedure Assign_Task(Domain : in out Dispatching_Domain;
    CPU : in CPU_Range := Not_A_Specific_CPU;
    T : in Task_Id := Current_Task);
  procedure Set_CPU(CPU : in CPU_Range; T : in Task_Id := Current_Task);
  function Get_CPU(T : in Task_Id := Current_Task) return CPU_Range;
  procedure Delay_Until_And_Set_CPU(
    Delay_Until_Time : in Ada.Real_Time.Time; CPU : in CPU_Range);
private
  ... -- not specified by the language
end System.Multiprocessors.Dispatching_Domains;
```

The semantics of the features contained in this package are as one would expect, and are similar to those defined and recommended during the last IRTAW. However, it should be noted that a number of more general features identified during

the last workshop did not get included in the language amendment; in particular, the dispatching rules for each dispatching domain cannot be individually defined.

Although a pragma can be used to define `CPU` and `Dispatching_Domain`:

```
pragma Dispatching_Domain(expression);
pragma CPU(expression);
```

for example,

```
task type T (Pri : System.Priority; Core : System.Multiprocessors.CPU) is
  pragma Priority(Pri);
  pragma CPU(Core);
  pragma Dispatching_Domain(Hard_Real_Time);
  entry SomeEntry(...); ...
end T;
```

where `Hard_Real_Time` is a defined `Dispatching_Domain`, the use of such pragmas is now deprecated (deemed obsolescent) and *aspects* should be employed. The above code should now be written as:

```
task type T (Pri: System.Priority; Core : System.Multiprocessors.CPU) with
  Priority => Pri,
  CPU => Core,
  Dispatching_Domain => Hard_Real_Time;
  is
    entry SomeEntry(...); ...
end T;
```

Although this is a quite difference syntactical form, there is no difference in semantics. With both forms a simple task declaration:

```
Example : T(13,3);
```

would place a task with priority 13 on CPU 3 within the `Hard_Real_Time` dispatching domain.

In addition to the above packages an extension to Ada.Interrupts (C.3.2) is made:

```
with System.Multiprocessors;
function Get_CPU(Interrupt: Interrupt_Id)
  return System.Multiprocessors.CPU_Range;
```

Taken together the above set of facilities will allow multitasking Ada programs to be allocated and scheduled on to a basic multicore platform in which all cores are identical. Although each dispatching domain (DD) has the same dispatching (scheduling) rules, there is ample opportunity to program many different paradigms. For example, the range of priorities could be split between EDF and fixed priority (FP). In one DD only the EDF range is used, in another only the FP range. As a result, although the overall dispatching strategies are the same in each DD, in effect one DD is totally EDF scheduled (and could employ, for example, global partitioning) whilst the other is FP (and perhaps uses a fully partitioned approach).

In addition to these facilities that are all focused on the control of task affinities, there is a new *task control barrier* package that allows a set of tasks to block on a condition and then be released in parallel. Such a facility is essential for many forms of data-centric parallelism (for example, a large array being searched by parallel tasks – one per core).

Finally, the definition of *Volatile* was clarified to enable lock-free synchronisation algorithms to be implemented.

4 Review of Previous Suggestion

Here we review the suggestions that have been voiced at previous workshop, but which are not yet incorporated into Ada. We distinguish between SMPs and other more general platforms. In both of theses cases one of the issues that remain significant is the degree to which any of the Ada facilities can be efficiently mapped on to platforms that conform to standards provided by Posix or Linux. We do not however investigate this issue further here.

4.1 SMP architectures

Two restrictions are apparent with the current proposal to extend the facilities provided in the Ada language for SMP platforms. One concerns the control over the number of CPUs, the other with control over scheduling. First the issue of the

number of CPUs and how they might map to dispatching domains. In the facilities described above, the set of CPUs is fixed and there is a static 'once and for all' allocation of CPUs to DDs.

Previously a more abstract API was proposed [11] in which sets of processors are defined with the availability of the CPUs potentially changing over time:

```
with Ada.Task_Identification; use Ada.Task_Identification;
package System.Processor_Elements is
  Affinity_Error : exception;
  Unsupported_Operation : exception;
  type Processors is range 0 .. <<implementation-defined>>;
    -- The number of processors available on this system.
    -- Each processor has a logical Id in the range.
    -- On a single processor system, the range is 0..0
  type Processor_Set is array(Processors) of Boolean;
    -- A set of processors. A boolean set to True, indicates
    -- that the logical processor is included in the set
  function Available_Processors return Processor_Set;
    -- Indicates which of the processors in the system are
    -- current available to the program. In some
    -- systems this will never change, others it may.
...
end System.Processor_Elements;
```

A package like this could support a more dynamic platform in which CPUs may alter their availability in a manner that would allow the programmer to produce adaptable code.

It was mentioned earlier that scheduling on a per-DD basis was consider at an earlier workshop – for example using code such as [4]:

```
with System; use System;
package Ada.Dispatching is
   Dispatching_Policy_Error : exception;

   type Dispatching_Domain_Policy is private.

   type Scheme is (Priority_Specific_Dispatching,
                   Non_Preemptive_FIFO_Within_Priorities,
                   FIFO_Within_Priorities,
                   Round_Robin_Within_Priorities,
                   EDF_ACross_Priorities);

   subtype Priority_Specific_Scheme is Scheme range
     FIFO_Within_Priorities .. EDF_ACross_Priorities;

   procedure Set_Policy(DDP : in out Dispatching_Domain_Policy;
                        P : Scheme);

   procedure Set_Priority_Specific_Policy(
               DDP : in out Dispatching_Domain_Policy;
               P : Priority_Specific_Scheme;
               Low : Priority; High : Priority);
   -- raises Dispatching_Policy_Error if
   --   DDP has not  been set to Priority_Specific_Dispatching, or
   --   High is not greater than Low, or
   --   any priority from Low to High has already been set

private
  -- not defined by language
end Ada.Dispatching;
```

However it is now not clear that this level of control over dispatching is needed and, as indicated above, sufficient flexibility can probably be extracted from the current facilities.

To support more stream-based parallelism, Ada 2012 has introduced the task control barrier. However further support is possible in this area. Ward and Audsley [10] argue for a broadcast primitive (to write to many protected objects in parallel),

and a new type of function for protected objects (POs). Such a function would have a barrier and hence could be blocked. But once the barrier is lowered all blocked function calls would proceed in parallel. In effect this would be a high level abstraction of the task control barrier; in a similar way that an ordinary PO is a more structured form of synchronous task control.

4.2 Non-SMP architectures

Once the hardware platform can no longer be consider to be a pure SMP (where all processors are identical, work at the same speed and have the same relationship with memory) then many new challenges arise. The follow is a possible classification of such platform architectures:

- SMP-like – processors are identical but run at different speeds,
- cc-NUMA – cache-coherent Non Uniform Memory Architecture,
- NUMA – no cache coherence, single address space,
- Heterogeneous – multiple address spaces.

What can a future definition of Ada contain to help program applications for these emerging platforms?

For SMP-like, more knowledge about each processor needs to be available to the program. A simple ordering of CPU is insufficient. Some form of map must be provided and a representation beyond that of a simple integer will probably be required.

Wellings et al [12] investigated the needs of cc-NUMA. With this architecture, although there is cache coherence, the time it takes to access memory will vary significantly for different tasks/objects in the program. These access times must be managed for any real-time system. They concluded that at least four issues must be addressed:

1. understanding the address map,
2. using storage pools to partition the heap,
3. using representation aspects to control object location, and
4. using task affinities to control thread migration.

Once we move beyond cc-NUMA towards heterogenous processors and memory, the problems become more complex. With a single address space, regions of cache coherence must be identified and have some form of representation in the program. The partition model within Ada (primarily to support distributed systems) may have a role here. Once we have multiple address spaces then a 'distributed systems' model does seem appropriate. As assessment of the Ada partition model for such situations is necessary. Extensions may be required.

5 Conclusions

In this paper we have attempted to review all the topics relating to multiprocessor platforms that were raised and discussed in previous workshops. We note the progress that has been made in the support that Ada now provides. But it is clear that for non-SMP architectures there are many challenges for any programming language that is attempting to support the production of efficient and predicable programs. Smart compilers might be able to remove some of these problems, but it is likely that some abstract representation of the platform will need to be available at the code level.

References

[1] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 243–252, 2008.

[2] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, 2000.

[3] A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a C=D scheme. In *Proceedings of 18th International Conference on Real-Time and Network Systems (RTNS)*, pages 169–178, 2010.

[4] A. Burns and A.J. Wellings. Dispatching domains for multiprocessor platforms and their representation in Ada. In J. Real and T. Vardanega, editors, *Proceedings of Reliable Software Technologies - Ada-Europe 2010*, volume LNCS 6106, pages 41–53. Springer, 2010.

[5] A. Burns and A.J. Wellings. Multiprocessor systems: Session summary. *Ada Letters – Proceedings of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14)*, XXX(1):16–25, 2010.

[6] A. Burns and A.J. Wellings. Supporting execution on multiprocessor platforms. *Ada Letters – Proceedings of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14)*, XXX(1):16–25, 2010.

[7] J. Real and S. Mitchell. Beyond Ada 2005 session report. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 124–126, 2007.

[8] J. Ruiz. Towards a ravenscar extension for multiprocessor systems. *Ada Letters – Proceedings of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14)*, XXX(1):86–90, 2010.

[9] K. Shinpei and Y. Nobuyuki. Portioned EDF-based scheduling on multiprocessors. In *EMSOFT*, pages 139–148, 2008.

[10] M. Ward and N.C Audsley. Suggestions for stream based parallel systems in Ada. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 33–138, 2007.

[11] A.J. Wellings and A. Burns. Beyond Ada 2005: allocating tasks to processors in SMP systems. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 75–81, 2007.

[12] A.J. Wellings, A.H. Malik, N.C. Audsley, and A. Burns. Ada and cc-NUMA architectures. what can be achieved with Ada 2005? *Ada Letters – Proceedings of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14)*, XXX(1):125–134, 2010.

TTF-Ravenscar: A Profile to Support Reliable High-Integrity Multiprocessor Ada Applications

A. Burns, A.J. Wellings and A.H. Malik
Department of Computer Science, University of York
Heslington, York YO10 5GH, UK
(burns,andy,haseeb)@cs.york.ac.uk

Abstract

Although the Ravenscar profile of Ada has achieved a measure of success in the development of high-integrity system, it is often criticised for not having enough expressive power to deal with common real-time programming patterns. This has led to a call for more facilities to be added to the profile. These have been turned down by the Ada standardization body for fear of "feature creep" and the lack of clear and consistent motivation. This paper proposes a coherent profile to support the construction of fault-tolerant high-integrity real-time programs.

1 Introduction

The success of the Ravenscar profile has resulted in the call for more facilities to be added to the profile [10, 11, 13]. However, often these facilities do not have an underlying coherent model and have been turned down by the Ada standardization body for fear of "feature creep".

The alternative approach to providing a more expressive profile is to start from the computation model that Ravenscar[1] was intended to support and to extend that model. A new profile can then be defined to support this new model. In this paper, we propose an augmented Ravenscar profile whose goal is to support applications that are required to tolerate timing faults (we define the term TTF-Ravenscar for this profile). The motivations for defining a new profile come from:

- the increased use of multicore platforms in high-integrity systems and the accompanying greater uncertainty that is present with current timing (schedulability and worst-case execution) analysis for these platforms,
- the desire to use a Ravenscar-like sets of features (i.e. not the full language) to program high-integrity systems that can identify and respond to faults even if full fault recovery is not possible (as this typically requires use of language features only available via the full language).

Any new profile should ideally retain the key properties of Ravenscar – that is, its run-time support system should be small, efficient and potentially certifiable.

The paper is structured as follows. Section 2 discusses our assumptions and presents our fault-tolerance framework within which programs execute. Section 3 then considers the use of Ravenscar within this framework and identifies where support is lacking. The TTF-Ravenscar profile is then defined and discussed in section 4. Finally conclusions are drawn.

2 Assumptions

Before suggesting any new profile that addresses fault-tolerance issues, it is important to define the failure assumptions and the overall framework within which fault tolerance is to be provided. Here, we assume failures in the underlying execution

[1] In this paper, we will use the term Ravenscar to indicate the subset of Ada that is allowed when conforming to the Ada Ravenscar profile.

platform (processor, memory and bus) are handled outside of the Ada program, and that they are either masked or result in the whole system crashing: hence we assume crash failures [5] only. We use the traditional definitions for failures, errors and faults: **failures** result from unexpected problems internal to the system which ultimately manifest themselves in the system's external behaviour; these problems are called **errors** and their mechanical or algorithmic cause are termed **faults**. A fault is **active** when it produces an error, until this point it is **dormant**. Once produced, the error can be transformed into other errors via the computational progress through the system. Eventually, the error manifests itself at the boundaries of the system causing a service delivery to fail [3]. This failure, then becomes a fault in any other systems that uses the first system as a component. The fault-error-failure-fault chain is one of the underlying assumptions of fault-tolerance.

According to the Anderson and Lee framework [1], the provision of software dynamic redundance consists of the following phases [4, 1].

1. **Error detection** – A fault will ultimately manifest itself in the form of an error; no fault tolerance scheme can be utilized until that error is detected.

2. **Damage confinement and assessment** – When an error has been detected, it is necessary to decide to what extent the system has been corrupted (this is often called *error diagnosis*); the delay between a fault occurring and the detection of the associated error means that erroneous information could have spread throughout the system.

3. **Error recovery** – Error recovery techniques aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality).

4. **Fault treatment and continued service** – An error is a symptom of a fault; although the damage may have been repaired, the fault may still exist, and therefore the error may recur unless some form of maintenance is undertaken.

From our perspective, we are concerned with tolerating timing faults. In this case, the above phases can be re-interpreted as follows [4].

1. **Error detection** – Most timing faults will eventually manifest themselves in the form of missed deadlines.

2. **Damage confinement and assessment** – When deadlines have been missed, it is necessary to decide which tasks in the system are at fault. Ideally, confinement techniques should ensure that only faulty tasks miss their deadlines.

3. **Error recovery** – The response to deadline misses requires that the application undertakes some forward error recovery.

4. **Fault treatment and continued service** – Timing errors often result from transient overloads. Hence they can often be ignored. However, persistent deadline misses may indicate more serious problems and require some form of maintenance to be undertaken.

The faults that may result in missed deadlines include the following [4]:

- worst-case execution time (WCET) calculations were inaccurate (optimistic rather than pessimistic),
- blocking times were underestimated,
- assumptions made in the schedulability checker were not valid,
- the schedulability checker itself had an error,
- the scheduling algorithm could not cope with a load even though it is theoretically schedulable,
- the system is working outside its design parameters, for example sporadic events occurring more frequently than was assumed in the schedulability analysis.

Assuming the schedulability analysis is correct, the following chains are possible in the context of priority-based systems [7]:

1. *Fault* (in task τ_i's WCET calculation or assumptions) $\rightarrow$ *error* (overrun of τ_i's WCET) $\rightarrow$ *error propagation* (deadline miss of τ_i) $\rightarrow$ *failure* (to deliver service in a timely manner);

2. *Fault* (in task τ_i's WCET calculation or assumptions) $\rightarrow$ *error* (overrun of τ_i's WCET) $\rightarrow$ *error propagation* (greater interference on lower priority tasks) $\rightarrow$ *error propagation* (deadline miss of lower priority tasks) $\rightarrow$ *failure* (to deliver service in a timely manner);

3. *Fault* (in task τ_i's minimum inter-arrival time assumptions) $\rightarrow$ *error* (greater computation requirement for τ_i) $\rightarrow$ *error propagation* (deadline miss of τ_i) $\rightarrow$ *failure* (to deliver service in a timely manner);

4. *Fault* (in task τ_i's minimum inter-arrival time assumptions) $\rightarrow$ *error* (greater interference on lower priority tasks) $\rightarrow$ *error propagation* (deadline miss of lower priority tasks) $\rightarrow$ *failure* (to deliver service in a timely manner);

5. *Fault* (in task τ_i's WCET calculation or assumptions when using a shared resource) $\rightarrow$ *error* (overrun of τ_i's resource usage) $\rightarrow$ *error propagation* (greater blocking time of higher priority tasks sharing the resource) $\rightarrow$ *error propagation* (deadline miss of higher priority tasks) $\rightarrow$ *failure* (to deliver service in a timely manner).

To be tolerant of timing faults, it is necessary to be able to detect:

- miss of a deadline – the final error in all the above error propagation chains;
- overrun of a worst-case execution time – potentially causing the task and/or lower priority tasks to miss their deadlines (error chains 1 and 2);
- a sporadic event occurring more often than predicted – potentially causing the task and/or lower priority tasks to miss their deadlines (error chains 3 and 4);
- overrun in the usage of a resource – potentially causing higher priority tasks to miss their deadlines (error chain 5).

Of course the last three error conditions do not necessary indicate that deadlines will be missed; for example, an overrun of WCET in one task might be compensated by a sporadic event occurring less often than the maximum allowed. Hence, the damage confinement and assessment phase of providing fault tolerance must determine what actions to take.

The following section will discuss the extent to which Ravenscar can support: error detection mechanisms for the above timing faults, possible error confinement approaches, and strategies for recovery. Fault treatment typically involves maintenance, which is a topic outside the scope of this paper.

3 Ravenscar and Dynamic Software Fault Tolerance

In section 2, the class of faults that could result in an application missing its deadline were identified. This section first considers, the Ravenscar facilities that allow the resulting errors to be detected. This is followed by a brief discussion on damage/error confinement. Finally. the error recovery phase is considered.

3.1 Error detection

From a timing perspective, there are three primitive errors that can result in a deadline miss.

1. WCET (Budget) overrun – Modern multicore processors make accurate and tight WCET analysis extremely difficult. To reflect this, we will assume that each Ada task has a CPU budget and not be concerned with whether this a worst-case measure. It will be up to the application to decide how to respond to budget overrun during its error recovery phase.

   Ravenscar supports execution-time clocks but not the `Ada.Execution_Time.Timers` package. Hence, the only possible approach to detect budget overrun is to poll for it. Clearly, this is very inefficient and, in practice, the overrun is likely to be detected well after the fact.

2. Overrun of sporadic events – A sporadic event firing more frequently than anticipated can have an enormous impact on a system attempting to meet hard deadlines. Where the event is the result of an interrupt, the consequences can be potentially devastating.

   Ada has no notion of a release event (job) so any detection of this overrun condition must be provided by the application. In general, this can be achieved using protected objects and timers as illustrated in Section 13.4.1 of Burns and Wellings [4]. Note that this code is Ravenscar-compliant.

3. Overrun of worst-case blocking time – In Ada, tasks cannot self-suspend while hold protected object locks. Assuming the protected-object usage has been accurately determined during the analysis phase, the only way a blocking overrun can occur is if the budget allocated to each protected action overruns, or blocking has been introduced by the Ada run-time environment. Support for detecting blocking overruns is traditionally not found in real-time operating systems, and consequently is not supported by Ada. The error will, therefore, propagate and either a) be transformed into a budget overrun, or b) go undetected until its affect is lost or c) result in a missed deadline. As an aside, Ada does not define the task that executes an entry body in a protected object. Hence, calculating the execution time of a protected procedure, which results in an entry becoming open, may require that entry and (in the general case) several other entries to be taken into account.

   Given no support in Ada, and the catch-all detection capability of a missed deadline, we will not consider this case further.

The fall-back position for all timing faults is to detect a deadline miss. **In Ravenscar, a deadline miss can be detected by using timers attached to the Real-Time clock.**

3.2 Damage confinement

The role of damage confinement of time-related faults is to prevent propagation of the resulting errors to other components in the system. There are two aspects of this that can be identified.

- Protecting the system from the impact of sporadic task overruns and unbounded aperiodic activities. The problem of overruns in sporadic objects and unbounded demand on the processor's time from aperiodic activities is usually handled by *Aperiodic Servers* such as *Sporadic Servers* and *Deferrable Servers*.
- Supporting composability and temporal isolations. When composing systems of multiple components, whether dynamically or statically, it is often required that each component be isolated from one another. Typically, this is achieved by *hierarchy schedulers* and *reservation-based systems*. Usually, two levels of scheduling are used. A global (top-level) scheduler and multiple application-level (second-level) scheduler. Usually, the application-level scheduler is also called called a *Server* or *Execution-time Server* or *Group Server*.

Although the above confinement techniques are similar, they have slightly different emphasis. For temporal isolation, the key requirement is that the Group Server be *guaranteed its budget each period*. To support sporadic and aperiodic execution, it is sufficient that the server *consumes no more than its budget each period.*

To implement both execution-time servers and aperiodic server typically requires a group budget facility. Ravenscar does not support the `Ada.Execution_Time.Group_Budget` package.

3.3 Error recovery

The greater the accuracy by which the fault and its resulting error can be detected and confined to the errant task, the easier it is to program recovery. Hence detecting budget overrun in a task indicates that it is that tasks that has the fault. In contrast, detecting deadline miss in a task does not necessarily mean that task is at fault.

This subsection considers the appropriate strategies for hard (deadlines must be met), soft (deadlines can occasional be missed and there is still some value in delivering a service late) and firm (deadlines can occasional be missed but there is no value in delivering a service late).

3.3.1 Strategies for handling budget overrun

Once detected, the task response will depend on whether it is a hard, soft or firm deadline.

Budget overrun in hard real-time tasks One possibility, is that the WCET values used in the schedulability analysis consists of the addition of two components. The first is the time allocated for the primary algorithm and the second is the time for recovery (assuming a fault hypothesis of a single failure per task per release). The first time is the time that is used by the system when monitoring. When this time passes, recovery occurs and the alternative algorithm can be executed. This can either be within the same task and the budget increased, or by releasing a dedicated recovery task. Typically, these alternative algorithms try to provide a degraded service. Another possibility is simply to do nothing. This assumes that there is enough slack in the system for the task (and other lower priority tasks) to still meet their deadlines.

Budget overruns in soft/firm real-time tasks Typically overruns in soft and firm real-time tasks can be ignored if the isolation techniques guarantee the capacity needed for the hard real-time tasks. Alternatively, the tasks priorities can be lowered, or the current releases can be terminated and the tasks re-released when their next release event occurs.

3.3.2 Strategies for handling sporadic event overruns

There are several responses to the violation of minimum inter-arrival time of a sporadic task: the release event can be ignore, an exception can be raised, the last event can be overwritten, (if it has not already been acted upon) or the actual release of the thread can be delayed until the MIT[2] has passed. Of course, the violation could be ignored and the task release allowed.

3.3.3 Strategies for handling deadline misses

Although the early identification of potential timing problems facilitate damage assessment, many real-time systems just focus on the recovery from missed deadlines. Several strategies are possible.

Deadline miss of hard real-time tasks – Again it is possible to set two deadlines for each task. An early deadline whose miss will cause the invocation of forward or backward error recovery. A later deadline is the deadline used by the schedulability test. In both cases, the recovery should aim to produce a degraded service for the task.

Deadline miss of soft real-time task – Typically this can be ignored and treated as a transient overload situation. A count of missed deadlines can be maintained, and when it passes a certain threshold a health monitoring system can be informed.

Deadline miss of a firm real-time task – As a firm task produces no value passed its deadline, its current release can be terminated. A count of terminated release can be maintained, and when it passes a certain threshold a health monitoring system can be informed.

3.3.4 Mode changes and event-based reconfiguration

In the above discussions, it has generally been assumed that a missed deadline and other timing errors can be dealt with by the task that is actually responsible for the problem. This is not always the case. Often the consequences of a timing error are as follows.

- Other tasks must alter their deadlines or even terminate what they are doing.
- New tasks may need to be started.
- Critically important computation may require more processor time than is currently available; to obtain the extra time, other less significant tasks may need to be 'suspended'.
- Tasks may need to be 'interrupted' in order to undertake one of the following (typically):
  - immediately return their best results they have obtained so far;
  - change to quicker (but presumably less accurate) algorithms;
  - forget what they are presently doing and become ready to take new instructions: 'restart without reload'.

These actions are sometimes known as *event-based reconfiguration*.

Some systems may additionally enter anticipated situations in which deadlines are liable to be missed. A good illustration of this is found in systems that experience *mode changes*. This is where some event in the environment occurs which results in certain computations that have already been initialized, no longer being required. If the system were to complete these computations then other deadlines would be missed; it is thus necessary to terminate prematurely the tasks that contain the computations.

To perform event-based reconfiguration and mode changes requires communication between the tasks concerned. Due to the asynchronous nature of this communication, it is necessary to use the asynchronous notification mechanisms. Any reconfiguration may also require tasks to be moved between processors in order to balance the load.

[2] Minimum Inter-arrival Time.

3.3.5 Ravenscar and error recovery

There are several common techniques that are used to implement the above error-recovery strategies

1. Reduce the priority of the task to a priority level where it will not jeopardize the execution of other tasks.

   Ravenscar does not support the `Ada.Dynamic_Priorities` package.

2. Move a task from one processor to another.

   Ravenscar does not support the `System.Multiprocessor` package, and its associated aspects.

3. Suspend the task completely.

   Ravenscar does not support the `Ada.Asynchronous_Task_Control` package.

4. Start a new task.

   Ravenscar does not support the dynamic creation of tasks, so all tasks must be created during the initialization phase. They can then queue on a mode changer (error recovery) protected object waiting for their mode to become active. Unfortunately, multiple tasks cannot be placed on the same entry queue. This results in obfuscated code, the problem is compounded by Ravenscar not supporting multiple entries in the same protected objects.

5. Request an asynchronous transfer of control in the task and then, possibly, also change its priority level.

   Ravenscar does not support the select-then-abort facility.

6. Terminate the current release (job) of the task and allow it to start afresh at the next release.

   Ada does not recognise the notion of releases. Hence there are no direct facilities to allow this. It has to be implemented by the program. The only way to do this would be to poll for the termination request.

4 Supporting a TTF-Ravenscar Profile

To make it worthwhile to define a new profile, there must be a clear application need, a computational model that reflect this need, and an implementation strategy that leads to a run-time footprint significantly smaller than that needed by the full language. Earlier is this paper we have attempted to define a need (to tolerate timing faults) and a set of facilities that together form a coherent model whilst still requiring a run-time footprint much closer to Ravenscar than the full Ada language.

However, to be successful, any proposed new profile must also be easily implementable on a wide range of current operating system. In the past, we have always used the POSIX standard as our measure of wide-spread support. However, the POSIX standards have been a bit slow responding to the increase in use of multicore/multiprocessor platforms. For this reason, we use Linux (and its various patches) as our measure.

4.1 Linux support relevant to the proposed profile

Linux was never designed to support real-time systems. Early versions of the Linux kernel were non-preemptible i.e. any task running inside a kernel (or a kernel driver) needed to finish its execution before anything else was allowed to execute [12]. Linux 2.0 supported symmetric multiprocessors by using a single spin lock (the big kernel lock) which allowed only one task to be running kernel code at one time. Later, locks were distributed based on critical regions rather than having a single lock for the whole kernel. However, any task inside the critical region was still non-preemptible and response times were still very high. The realtime-preempt patch was included in the mainline kernel in Linux 2.6.18[3] as a configuration option; it tries to minimize the amount of non-preemptible code.

On multiprocessors, ideally the OS should provide support for different scheduling policies from fully partitioned to global scheduling. In Linux, each processor has its own run queue. Once a thread is allocated to a processor it stays in its run queue until it is either moved to another processor by changing its affinity or through load balancing. Although Linux does not have a single queue for multiple processors for true global scheduling, however, it uses load balancing on real time tasks to make sure that the highest priority threads present in all the run queues are running at any particular instant of time [2]. When there is more than one real time thread in a run-queue, the lower priority real time thread is migrated to a CPU on which it can run. This migration happens at the following events:

[3] http://www.Linuxfordevices.com/c/a/News/Linux-kernel-gains-new-realtime-support/

1. a real time thread becomes schedulable, however, it finds a higher priority thread already running in the run-queue on which it has been allocated. In such a case it is pushed to another run queue where it can run;

2. when a task is suspended (or completes) on one processor (processor A), and a task exists in the run queue of another processor (processor B) with a higher priority than the highest priority in processor A's queue, the kernel pulls the higher priority task from B's queue to A's queue, and the higher priority task is dispatched.

However, the migration does respect the affinities of the tasks and any such migrations are not allowed where the CPU is out of the affinity set of the task.

In Linux, control groups [9] provide a mechanism to partition groups of tasks and then manage resources allocated to each partition. Control group (cgroup) is a file system, where tasks can be added and then be arranged hierarchically. By default all tasks belong to the root cgroup. Each resource (CPU, memory etc.) has an associated controller which limits resources to tasks based on the cgroup they belong to.

Building the mainline kernel by enabling the `CONFIG_RT_GROUP_SCHED` option, provides the support for real-time group scheduling. The real-time group scheduling can be used along with the cgroup filesystem to provide group budget for a set of tasks(process/threads). The real time group scheduling allows explicit allocation of CPU bandwidth to task groups [6]. The bandwidth can be set by the following two parameters

1. `cpu.rt_runtime_us` to set the budget of the task group

2. `cpu.rt_period_us` to set the period of the task group

In every period T, the group is allocated the budget. Once the budget expires, the group is blocked until the arrival of the next period. In Linux, the scheduler `scheduler_tick` function is called periodically after a time period T which is called by a high resolution timer. This function maintains the different CPUs accounting parameters and checks for budget exhaustion. On each scheduling event (at the `scheduler_tick`, task deactivation, task pre-emption) the collective execution times of all tasks are compared to the budget value until it exceeds the budget at which point all tasks are descheduled.

A sporadic server patch has also been presented in [8] with minimum changes to the real-time group scheduling of Linux. The changes mainly involve behaviour of threads; instead of blocking, threads are re-entered in a lower priority queue once the budget has expired. It also makes changes to how the budget is replenished after each period.

4.2 The proposed TTF-Ravenscar profile

In section 3, several Ada facilities were needed that were not available in the Ravenscar. In this section, we discuss the feasibility of an augmented Ravenscar profile and whether the added functionality is supported by Linux.

`Multiprocessor` and `Multiprocessors.Dispatching_Domains`

These two Ada packages allow the affinity of a task to be set to a dispatching domain. To be in keeping with Ravenscar, we propose the following restrictions.

- No dispatching domains can be created by the program.
- Each task, including the environment task, should be set to run on a single CPU (using the `Set_CPU` subprogram).
- A program can explicitly move a task from one processor to another[4].
- Each interrupt handler should only run on a single processor – but this may not be under the control of the program.

We note that Linux can support all of these facilities with its CPU affinity API that has been available since Kernel version 2.5.8.

[4]The circumstances in which this is allowed could be restricted.

`Ada.Execution_Time.Timers`

This package essentially allows event handlers (protected procedures) to be executed when the amount of CPU time a task has consumed reaches a specified value. The handler is run by the underlying clock/timer interrupt handler on the processor(s). Only one handler per task is needed.

Linux supports the POSIX CPU clocks and timers facilities, which can be used to implement this facility.

`Ada.Execution_Time.Group_Budgets`

Group budgets are challenging to implement, particularly on a multicore platform. Consequently, to simplify the support, we suggest that group budgets can only be set for tasks that reside on the same processor.

The nearest functionality in this area, which is provided by mainstream operating systems, is the support for sporadic servers. Linux also support the notion of control groups, as described in Section 4.1.

`Ada.Dynamic_Priorities`

Adding dynamic priorities would undermine the static nature of Ravenscar. However, they are a valuable tool for error confinement and recovery.

If dynamic priorities are supported, then either ceiling must be set to the ceiling of the ceilings for all priority settings, or the dynamic ceilings must be included in the new profile.

`Ada.Asynchronous_Task_Control`

Although asynchronous task control would add significant expressive power to the profile, we have concerns over its widespread support by Ada vendors. As far as we are aware, no one implements this package. The problem is that it is difficult to get the required semantics without the underlying OS providing a base priority level that will guarantee that no progress is made by the task. The Ada semantics are defined in terms of dynamic priorities, and consequently supporting dynamic priorities would obviate the need for asynchronous task control.

Entry queues or multiple entries

Having more flexible support for protected objects would make some programs much easier to implement, including support for mode change algorithms. At a minimum, having a single entry but multiple tasks queued (or multiple entries but with only one task allowed to queue) is required.

The select-then-abort statement

Any error recovery that requires a task to have asynchronous transfer of control requires use of the select-then-abort statement. The semantics of this language facility, even in a restricted tasking model, are still complex (as, in effect, threads can be aborted), and the run-time facilities are likely to lead to a footprint comparable with that required by the full language. We feel that support for this is a step too far for a restricted high-integrity profile. Full fault tolerance requires the full language.

5 Conclusions

Although pragma `Restrictions` allows an Ada program to specify a restricted use of the language, it does not guarantee the provision of a simplified run-time system; however, this is encourage by the language's reference manual (Section 2.7, par 21). By specifying a predefined profile, the Ada language signals a stronger intention that an appropriate subset of the run-time system should be developed. The profile identified in this paper can be specified using the current `Restrictions` pragma but this would not have the desired effect. Hence, we recommend the definition of a new profile whose goal is to support the implementation of fault-tolerant high-integrity real-time programs. In short, this TTF-Ravenscar profile would be defined to be Ravenscar plus (or modified by):

1. Ability to change a task affinity at run-time (use of `Set_CPU`).
2. Use of `Timers` – one per task.
3. Use of `Group_Budgets` – tasks on same CPU
4. `Dynamic_Priorities` – or possible. `Ada.Asynchronous_Task_Control`.
5. Entry queues for protected objects – or multiple single-task queues per PO.

References

[1] T. Anderson and P.A. Lee. *Fault Tolerance Principles and Practice:*. Prentice-Hall International, 2nd edition, 1990.

[2] Jeremy Andrews. Balancing real time threads, http://kerneltrap.org/linux/balancing_real_time_threads, 2007.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan-March 2004.

[4] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 4th edition, 2009.

[5] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34:56–78, 1993.

[6] Linux Kernel Documentation. Real-time group scheduling, http://www.mjmwired.net/kernel/documentation/scheduler/sched-rt-group.txt.

[7] O. Marchi dos Santos and A.J. Wellings. Run time detection of blocking time violations in real-time systems. *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[8] Dario Faggioli, Antonio Mancina, Fabio Checconi, and Giuseppe Lipari. Design and implementation of a POSIX compliant sporadic server. In *Proceedings of the* 10^{th} *Real-Time Linux Workshop (RTLWS 2008)*, Colotlan, Mexico, October 2008.

[9] Paul Menage. Cgroups, http://www.mjmwired.net/kernel/documentation/cgroups.txt, 2004.

[10] Enrico Mezzetti, Marco Panunzio, and Tullio Vardanega. Temporal isolation with the Ravenscar profile and Ada 2005. *Ada Lett.*, 30:45–55, May 2010.

[11] José F. Ruiz. Towards a Ravenscar extension for multi-processor systems. *Ada Lett.*, 30:86–90, May 2010.

[12] Sven thorsten Dietrich and Daniel Walker. The evolution of Real-Time Linux, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.151.6125.

[13] Tullio Vardanega. Ravenscar design patterns: reflections on use of the Ravenscar profile. *Ada Lett.*, XXIII:65–73, September 2003.

An EDF Run-Time Profile based on Ravenscar

A. Burns
Department of Computer Science
University of York, UK
burns@cs.york.ac.uk

Abstract

*This paper presents a definition of a Ravenscar-like profile (*EDF_Ravenscar*) that incorporates EDF scheduling. Rather than just replace the dispatching policy, an argument is made for simplifying the support that is provided in full Ada for controlling access to protected objects. As a result all tasks and protected objects have their default priorities, and only one ready queue is needed (ordered by absolute deadline). The paper also outlines the static scheduling analysis that can be applied to applications conforming to the profile.*

1 Introduction

Earliest deadline first (EDF) is a popular scheduling paradigm as it has been proven to be the most efficient scheme available on monoprocessors. If a set of tasks, on a single processor, is schedulable by any dispatching policy then it will also be schedulable by EDF. To support EDF requires two language features (now supported in Ada 2005):

- representation of the deadline for a task,
- representation of the preemption level for a protected object.

The first is obviously required; the second is the EDF equivalent of the priority ceiling protocol and allows protected objects to be 'shared' by multiple tasks [1] (see Section 2.1). The support for the second requirement does however give rise to a more complex set of language rules and implementation. This complexity may not be compatible to a Ravenscar-like profile.

The Ravenscar run-time profile [6, 5, 14, 12, 11] is an important feature of the Ada language, it allows simple real-time systems to be analysed and implemented on a run-time system that is itself simple, small and efficient – and is capable of being engineering to the highest level of integrity. The profile, whose definition was developed and refined in previous workshops of the IRTAW series, is defined in Ada 2005 via a collection of restrictions on the full Ada language. It is defined to support applications that have a fixed set of tasks scheduled by the fixed priority scheme - known as `FIFO_Within_Priority` in Ada. Ravenscar is designed to be used by those applications that have tight timing requirements and high integrity (eg. safety critical) constraints.

For some high integrity applications, however, the extra performance that can be obtained from utilising EDF scheduling[1] is significant, and hence the motivation to define a profile similar to Ravenscar but with its fixed priority dispatching scheme replaced by EDF dispatching. Unfortunately the level of simplicity required of any Ravenscar-like profile does not necessarily mean that full EDF scheduling is necessary or desirable. In this paper we indeed argue that a restricted model is needed for, what we term in this paper, the *EDF_Ravenscar* run-time profile.

The paper is organised as follows. First, in the following section, we outline the support that Ada 2005 gives for EDF scheduling. In section 3, the scheduling analysis for EDF-based systems is reviewed. The issues raised in these two sections leads to a restricted definition of an EDF-based run-time profile. This is described in Section 4. Finally conclusions are given in Section 5.

[1] EDF scheduling can be up to 1.763 times more efficient than fixed priority scheduling when preemptive dispatching is employed. That is, a system that is schedulable with EDF might need a processor running 1.763 times faster to be schedulable with fixed priority dispatching [9].

2 Ada's Support for EDF scheduling

In this section we note the provisions available in Ada 2005 for programming EDF-scheduled systems. As indicated earlier, to support EDF requires two language features:

- representation of the deadline for a task,
- representation of preemption level for a protected object.

A predefined package provides support for deadlines:

```
with Ada.Real_Time; with Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
       Ada.Real_Time.Time_Last;
  procedure Set_Deadline(D : in Deadline;
       T : in Ada.Task_Identification.Task_ID :=
       Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline(
       Delay_Until_Time : in Ada.Real_Time.Time;
       Deadline_Offset : in Ada.Real_Time.Time_Span);
  function Get_Deadline(
       T : Ada.Task_Identification.Task_ID :=
       Ada.Task_Identification.Current_Task)
       return Deadline;
end Ada.Dispatching.EDF;
```

These facilities are easily used to program behaviors such as periodic tasks [7]. The basic semantics of EDF scheduling is that the task with the most urgent (earliest) deadline is the one chosen for execution. This task will remain executing until it either completes (and delays for its next release) or is preempted by the release of a task with a shorter absolute deadline.

We now turn to the second requirement. This concerns the use of shared protected objects between tasks executing under EDF rules.

2.1 Preemption Levels and Baker's Protocol

With standard fixed priority scheduling, *priority* is actually used for two distinct purposes:

- to control dispatching
- to facilitate an efficient and safe way of sharing protected data.

The latter is known as the priority ceiling protocol. In Baker's stack-based protocol two distinct notions are introduced for these policies[1]:

- earliest deadline first to control dispatching[2],
- preemption levels to control the sharing of protected data.

With preemption levels (which is a very similar notion to priority) each task is assigned a static preemption level, and each protected object is assigned a ceiling value that is the maximum of the preemption levels of the tasks that call it. At run-time a newly released task, τ_1,can preempt the currently running task, τ_2, if and only if the:

- absolute deadline of τ_1 is earlier (i.e. sooner) than deadline of τ_2, and the
- preemption level of τ_1 is higher than the preemption of any locked protected object.

With this protocol it is possible to show that mutual exclusion (over the protected object on a single processor) is ensured by the protocol itself (in a similar way to that delivered by fixed priority scheduling and ceiling priorities). Baker also showed, for the classic problem of scheduling a fixed set of periodic tasks, that if preemption levels are assigned according to each task's relative deadline then a task can suffer at most a single block from any task with a longer deadline. Again this result is identical to that obtained for fixed priority scheduling. Note for this property to hold, preemption levels must be assigned inversely to relative deadline (ie. the smaller the relative deadline, the higher the preemption level).

[2]His paper actually proposes a more general model of which EDF dispatching is an example.

2.2 Supporting EDF Preemption Levels in Ada

Ada 95 did not support EDF scheduling but new facilities were added to Ada 2005; and it was desirable for fixed priority and EDF scheduling to work together. To this end the Ada 2005 definition did not attempt to define a new locking policy but uses the existing ceiling locking rules without change. Priority, as currently defined, is used to represent preemption levels. EDF scheduling is defined by a new dispatching policy.

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

The basic preemption rule given earlier for Baker's protocol, whilst defining the fundamental behaviour, does not give a complete model. For example, it is necessary to define what happens to a newly released task that is not entitled to preempt. A complete model, within the context of Ada's ready queues, is defined by the following rules (see paragraphs 17/2 to 28/2 of section D.2.6 of the Ada 2005 Reference Manual).

When EDF_Across_Priorities is specified for priority range Low..High all ready queues in this range are ordered by deadline. The task at the head of a queue is the one with the earliest deadline.

A task dispatching point occurs for the currently running task T to which policy EDF_Across_Priorities applies:

- *when a change to the deadline of T occurs;*
- *there is a task on the ready queue for the active priority of T with a deadline earlier than the deadline of T; or*
- *there is a non-empty ready queue for that processor with a higher priority than the active priority of the running task.*

In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority.

For a task T to which policy EDF_Across_Priorities applies, the base priority is not a source of priority inheritance; the active priority when first activated or while it is blocked is defined as the maximum of the following:

- *the lowest priority in the range specified as EDF_Across_Priorities that includes the base priority of T;*
- *the priorities, if any, currently inherited by T;*
- *the highest priority P, if any, less than the base priority of T such that one or more tasks are executing within a protected object with ceiling priority P and task T has an earlier deadline than all such tasks and all other tasks on ready queues with priorities strictly less than P.*

When a task T is first activated or becomes unblocked, it is added to the ready queue corresponding to this active priority. Until it becomes blocked again, the active priority of T remains no less than this value; it will exceed this value only while it is inheriting a higher priority.

When the setting of the base priority of a ready task takes effect and the new priority is in a range specified as EDF_Across_Priorities, the task is added to the ready queue corresponding to its new active priority, as determined above.

In addition, there is a rule (paragraph 30/2 of section D.2.6) that no protected object can have a ceiling of value *Low* – when `EDF_Across_Priorities` is specified for priority range *Low..High*.

2.3 Implications for EDF_Ravenscar

The virtue of the current language definition is that it works with the Ada 95 definition of protected objects and the `Ceiling_Locking` protocol. It also allows mixed systems (of EDF and fixed priority scheduled tasks) to be programmed where protected objects can be shared between tasks in the two dispatching worlds. However, the protocol is complex. Evidence of this comes from the fact that language definition itself had to be modified due to a problem with the first approach [16] and that early implementations have also had difficulties with the protocol. It is hard to test all possible interactions (of tasks with different preemption levels arriving at different times and with various protected objects themselves making calls upon each other). Also the overheads at run-time are not low. The arrival of a task may require a number of ready queues to be checked.

The implication is therefore: the full EDF facility, in terms of its support for a general model of Baker's algorithm, is too complex for a Ravenscar-like profile for safety critical applications. Overheads are not insignificant and proof of correctness may be expensive to obtain.

3 Analysis for EDF Scheduling

First the standard means of analysing EDF scheduled systems is described. This can be used with the proposed `EDF_Ravenscar` Profile.

3.1 System Model

We use a standard system model in this paper, incorporating the preemptive scheduling of periodic and sporadic task systems. A real-time system, $\mathcal{A}$, is assumed to consist of N tasks (τ_1 .. τ_N) each of which gives rise to a series of jobs that are to be executed on a single processor. Each task τ_i is characterized by several parameters:

- A *period* or *minimum inter-arrival time* T_i; for *periodic* tasks, this defines the exact temporal separation between successive job arrivals, while for *sporadic* tasks this defines the minimum temporal separation between successive job arrivals.
- A *worst-case execution time* C_i, representing the maximum amount of time for which each job generated by τ_i may need to execute. The worst-case utilization (U_i) of τ_i is C_i/T_i. The total utilisation of the task set in denoted by U, with $U = \sum_{i=1}^{N} C_i/T_i$.
- A *relative deadline* parameter D_i, with the interpretation that each job of τ_i must complete its execution within D_i time units of its arrival. In this analysis we assume $D_i \leq T_i$. The *absolute deadline* of a job from τ_i that arrives at time t is $t + D_i$.

Once released, a job does not suspend itself. We also assume, initially, that tasks are independent of each other and hence there is no blocking term to be incorporated into the scheduling analysis. This term will be included shortly – in Section 3.3.

There are no restrictions on the relative release times of tasks (other than the minimum separation of jobs from the same task). Hence we assume all tasks start at the same instant in time – such a time-instant is called a *critical instant* for the task system[13]. In this analysis we assume tasks do not experience release jitter.

3.2 Processor-Demand Analysis

Processor-Demand Analysis (PDA) [4, 3] considers the entire task system in one sequence of tests. It uses the property of EDF that at any time t only jobs that have an absolute deadline before t need to execute before t. So the test takes the form (the system start-up is assumed to be time 0):

$$\forall t > 0 : \;\; h(t) \;\leq\; t \tag{1}$$

where $h(t)$ is the total load/demand on the system (all jobs that have started since time 0 and which have a deadline no greater than t). A simple formulae for $h(t)$ is therefore (for $D \leq T$):

$$h(t) \;=\; \sum_{j=1}^{N} \left\lfloor \frac{t + T_j - D_j}{T_j} \right\rfloor C_j \tag{2}$$

The need to check all values of t is reduced by noting that only values of t that correspond to job deadlines have to be assessed. Also there is a bound on t. An unschedulable system is forced to fail inequality (1) before the bound L. A number of values for L have been proposed in the literature, here we give two. First is the *synchronous busy period* [8, 15] which is denoted here by L_B. It is calculated by forming a similar recurrence relationship to that used for response time analysis for fixed priority systems:

$$s^{m+1} \;=\; \sum_{i=1}^{N} \left\lceil \frac{s^m}{T_i} \right\rceil C_i \tag{3}$$

The recurrence stops when $s^{m+1} = s^m$, and then $L_B = s^m$. Note that the recurrence cycle is guaranteed to terminate if $U \leq 1$ for an appropriate start value such as $s^0 \;=\; \sum_{i=1}^{N} C_i$.

If the system utilization (U) is strictly less than 1 then a simpler formulae for L is possible [10]:

$$L_A \ = \ Max \left\{ D_1, ..., D_n \frac{\sum_{j=1}^{N} (T_j - D_j) U_j}{1 - U} \right\} \tag{4}$$

Either formulation for L can be used (or the minimum of the two values).

With all available estimates for L there may well be a very large number of deadline values that need to be checked using inequality (1) and equation (2). This level of computation has been a serious disincentive to the adoption of EDF scheduling in practice. Fortunately, a new much less intensive test has recently been formulated [17]. This test, known as QPA (Quick Processor-demand Analysis), starts from time L and integrates backwards towards time 0 checking a small subset of time points. These points are proved [17] to be adequate to provide a necessary and sufficient test.

A version of the QPA algorithm [18] optimised for efficient implementation is encoded in the following pseudo code in which `D_min` is the smallest relative deadline in the system, and `Gap` is the common divisor of the computation times and deadlines. The value of `Gap` is such that no two significant points in time (eg interval between two adjacent deadlines) is less than `Gap`. For example, if all task parameters are given as arbitrary integers then `Gap` will have the value 1.

```
t := L - Gap
while h(t) <= t and t >= D_min loop
  t := h(t) - Gap
end loop
if t < D_min then
  -- task set is schedulable
else
  -- task set is not schedulable
end if;
```

In each iteration of the loop a new value of `t` is computed. If this new value is less than the computed load at that point, the task set is unschedulable. Otherwise the value of `t` is reduced during each iteration and eventually it must become smaller than the first deadline in the system and hence the system is schedulable.

Theorem 1 *([17]) A general task set is schedulable if and only if $U \leq 1$, and the iterative result of the QPA algorithm is $s \leq D_{min}$ where D_{min} is the smallest relative deadline of the task set.*

It is also possible to undertake sensitivity analysis via QPA [19].

3.3 Incorporating Blocking

The above analysis assumes that tasks are independent. If tasks interact via protected objects (PO) then a task can be delayed not just by tasks with shorter absolute deadlines, but also by tasks that are executing within a PO. For fixed priority systems the blocking times are static values that are easily computed once task and ceiling priorities have been assigned and the use-pattern (of tasks and POs) is known.

For EDF scheduling the basic processor demand inequality (1) becomes [15, 2]

$$\forall t > 0 : \ \ h(t) + b(t) \ \leq \ t \tag{5}$$

where $b(t)$ is the maximum blocking that any task can suffer at time t. It must be recomputed each time the equation is applied.

3.4 Implications for EDF_Ravenscar

With fixed priority scheduling a task τ_i can be blocked by the existence of a PO that is used both by a task with a lower priority, and a task with a high or equal priority (to that of τ_i). For EDF the situation is much more complex, the blocking relation changes with time. Two tasks that share a PO will, at different times, block each other. Even a task with the shortest relative deadline that does not access a PO may suffer blocking.

The implications are therefore that all tasks may suffer blocking and that in many situations the blocking terms are of similar magnitude. To try and compute the exact values (for use in equation 5) may therefore be error-prone and not of great utility.

4 An EDF Version of the Ravanscar Run-Time Profile

The definition of the new profile, `EDF_Ravenscar`, clearly needs access to the `Ada. Dispatching.EDF` package. It can designate EDF dispatching by use of the pragma:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

This implies that all priority levels (if used) are subject to EDF dispatching. In particular the '*Low*' parameter used in the definition of the locking protocol (see Section 2.2) has the value `Priority'First`.

From the consideration of the implications of the Ada model (see Sections 2.3 and 3.4), this paper concludes that the full Ada 2005 language facilities for EDF scheduling is not compatible with a Ravenscar-like profile. The implications of this is that a simpler relationship between tasks and protected objects is required.

Fortunately this can be obtained with a straightforward restriction to the general model. If all tasks have the same preemption level, and all POs have a ceiling level above this value then in effect protected subprograms become non-preemptive. All runnable tasks are on a single ready queue (at priority `Priority'first`). If when a task is released the current task is executing in a PO then the released task will be placed in this ready queue (at the position dictated by its absolute deadline). A released task will never preempt a task executing in a PO. If however the current task is executing outside of a PO then a simple comparison of absolute deadlines will determine which task then executes.

The resulting run-time behaviour is straightforward, the implementation can be efficient, and be open to formal proof. The drawback is that blocking may be more than would otherwise occur. But as noted earlier, this may be a small impact with EDF as blocking is more evenly encountered.

To prevent the use of preemption levels and different PO ceilings it is sufficient to simply ban the use of the priority pragma and the use of the dynamic priorities package (which is already outside of the Ravenscar provisions). If tasks cannot specify their priorities then all tasks will be assigned the default priority of `Default_Priority`. Moreover, all protected objects will be assigned the ceiling priority of `Priority' Last`. As a result, with `EDF_Within_Priority` dispatching there is only one ready queue (at priority `Priority'First`); it is ordered by the absolute deadline of the tasks. Only an executing task can be 'inside' a PO – and the executing task is not on a ready queue.

To analyse an application executing under `EDF_Ravenscar` the processor-domain equation is modified as follows:

$$\forall t > 0 : \;\; h(t) + B \;\leq\; t \tag{6}$$

where B is the maximum execution time of any protected subprogram.

It is possible to postulate more exact analysis, for example taking into account those task that could be executing at time t, but equation (6) provides a simple yet effective basis for analysis (when combined with the QPA approach outlined earlier).

4.1 Full Definition

For completeness the full definition of the proposed profile is now given. First, a new restriction is required to prevent the use of the Priority pragma (or its equivalent aspect): `No_Specified_Priorities`. The definition of `EDF_Ravenscar` is thus:

```
pragma Task_Dispatching_Policy (EDF_Across_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
               No_Specified_Priorities,
               No_Abort_Statements,
               No_Dynamic_Attachment,
               No_Dynamic_Priorities,
               No_Implicit_Heap_Allocations,
               No_Local_Protected_Objects,
               No_Local_Timing_Events,
               No_Protected_Type_Allocators,
               No_Relative_Delay,
               No_Requeue_Statements,
               No_Select_Statements,
               No_Specific_Termination_Handlers,
               No_Task_Allocators,
```

```
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes);
```

5 Conclusions

The Ravenscar profile has proved to be an important aspect of the Ada language. It has been widely adopted by researchers, implementers and practitioners. One key feature of Ravenscar is its use of fixed priority dispatching – a mature scheduling approach with significant support analysis.

The main alternative to fixed priority scheduling is EDF (Earliest Deadline First); this scheme is more efficient (in terms of processor utilization) but has only recently become a mainstream technology appropriate to high integrity applications. To capitalize on these developments this paper has argued that an EDF-based version of the Ravenscar profile be defined for Ada. In producing this, a simplified run-time behaviour is defined. By restricting the use of the `Priority` pragma, protected operations become, in effect, non-preemptive, and only one ready queue is needed in the run-time. Some small loss of performance may result, but this is balanced by a reduction in the run-time overheads and complexity. Future work will attempt to measure this performance degradation.

References

[1] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[2] S.K. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 379–387, 2006.

[3] S.K. Baruah, R.R. Howell, and L.E. Rosier. Feasibility problems for recurring tasks on one processor. *Theorectical Computer Science*, 118:3–20, 1993.

[4] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptive scheduling of hard real-time sporadic tasks on one processor. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[5] A. Burns. The Ravenscar Profile. *ACM Ada Letters*, XIX(4):49–52, Dec 1999.

[6] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, pages 263 – 275. Springer Verlag, 1998.

[7] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.

[8] I. Ripoll A. Crespo and A.K. Mok. Improvement in feasibilty testing for real-time tasks. *Journal of Real-Time Systems*, 11(1):19–39, 1996.

[9] R.I. Davis, T. Rothvo, S. . Baruah, and A. Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority pre-emptive scheduling. *Journal of Real Time Systems*, 43(3):211–258, Nov 2009.

[10] H. Hoang, G.C. Buttazzo, M. Jonsson, and S. Karlsson. Computing the minimum EDF feasible deadline in periodic systems. In *RTCSA*, pages 125–134, 2006.

[11] ISO/IEC. Information technology - programming languages - guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report TR 24718, ISO/IEC, 2005.

[12] M. Kamrad and B. Spinney. An Ada runtime system implementation of the Ravenscar profile for high speed application-layer data switch. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Santander*, pages 26–38. Springer Verlag, 1999.

[13] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[14] K. Lundqvist, L. Asplund, and S. Michell. A formal model of the Ada Ravenscar tasking profile; protected objects. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Santander*, pages 12–25. Springer Verlag, 1999.

[15] M. Spuri. Analysis of deadline schedule real-time systems. Technical Report 2772, INRIA, France, 1996.

[16] A. Zerzelidis, A. Burns, and A.J. Wellings. Correcting the EDF protocol in Ada 2005. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 18–22, 2007.

[17] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transaction on Computers*, 58(9):1250–1258, 2008.

[18] F. Zhang and A. Burns. Improvement to quick processor-demand analysis for EDF-scheduled real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 76–86, 2009.

[19] F. Zhang, A. Burns, and S.K. Baruah. Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems. In *Proceedings of 16th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 61–70, 2010.

Ada 2012: Resource Sharing and Multiprocessors

S. Lin and A.J. Wellings* and A. Burns
Department of Computer Science, University of York, York,YO10 5GH, UK
(charlie,andy,burns)@cs.york.ac.uk

Abstract

As from Ada 2005, the language has supported different dispatching policies including fixed priority (FP) and earliest-deadline first (EDF). EDF dispatching can be applied across the whole range of priorities or across a restricted range. In this way, EDF scheduling is integrated into a FP framework. Baker's Stack Resource Policy (SRP) is also integrated with Ada's immediate priority ceiling protocol to support resource sharing using protected objects. However, with the introduction of support for global, partitioned and cluster scheduling for multiprocessor systems, the use of protected objects in applications needs to be re-evaluated. This is because Ada does not completely define how protected objects are accessed in a multiprocessor environment and because the SRP cannot be directly applied to multiprocessors.

This paper reviews the currently available multiprocessor resource allocation policies and analyzes their applicability to Ada. It proposes some new Ada mechanisms that would facilitate the programming of a variety of protected object access protocols.

1 Introduction

Ada 2012 provides more support for multiprocessor systems than previous versions of the language. However, issues surrounding the use of protected objects are still, to a large extent, implementation defined. This is mainly because there are no agreed standards for the best way to access resources in a multiprocessor environment.

In this paper, we first review the multiprocessor resource control protocols that have been proposed in the literature. We then review Ada's support for multiprocessors focusing on the issue of accessing protected objects. From these reviews, we summarize the main characteristics of the protocols in the context of Ada to determine their compatibility with Ada 2012. We consider both global, partitioned and cluster systems, and fixed priority (FP) and earliest-deadline first (EDF) dispatching. We then propose some new functionality to the language which would help the programmer implement their own protected object access protocols.

2 Multiprocessor Resource Control Protocols

Resource control policies for single processor systems are well understood. In particular:

- Priority Inheritance Protocol (PIP) – defined for preemptive fixed priority-based (FP) systems but also can be used with any system with static task-level measures of execution eligibility.
- Priority Ceiling Protocol (PCP) – defined for FP systems but also can be used with any system with static task-level measures of execution eligibility.
- Nonpreemptive Critical Sections (NCS) – can be used with any dispatching policy and any measure of execution eligibility.

- Stack Resource Policy (SRP) – defined as an extension to PCP for EDF systems but can be used with any static job-level[1] execution eligibility. The SRP when applied to FP systems is also called the Immediate Priority Ceiling Protocol (ICP) (also called highest locker protocol or priority protected protocol or priority ceiling emulation).

Note that the usual assumption with resource control policies is that tasks do not self suspend while holding a resource. This is true with Ada only if resources are encapsulated within protected objects.

In this section, we review the resource control protocols that have been defined for multiprocessors systems and extract their main characteristics.

Throughout the paper we consider a shared memory multiprocessor system and use the following terms:

- Fully partitioned scheduling – tasks are assigned to a single processor and are bound to that processor for their lifetime.
- Global scheduling – tasks can be executed of any processor
- Cluster scheduling – processors are grouped into clusters. A processor can only be contained in exactly one cluster. Tasks are assigned to a single cluster and are bound to that cluster for their lifetime. Global scheduling occurs within each cluster.

2.1 MPCP and DPCP

There have been several versions of the Multiprocessor Priority Ceiling Protocol (MPCP) that extend the single processor PCP. Initially, Rajkumar et al's [13] proposed that all global resources were assigned to a single synchronization processor. This was then generalized in the same paper to allow multiple synchronization processors, but again each resource was assigned to one synchronization processor. A task that wishes to access a global resource migrates to the synchronization processor for the duration of its access. Each global resource is also assigned a ceiling equal to: $P_H + 1 + \max_i\{\wp_i | \tau_i \; uses \; R^k\}$, where P_H is the highest priority of all tasks that are bound to that processor, $\wp_i$ is the priority of task i and R^k is a resource. The protocol is defined below.

Rule 1 Tasks are partitioned between processors. PCP is used for local resource sharing.

Rule 2 When a task τ is blocked by a global resource, it is added to a prioritized queue and suspends. The resource holding task will continue its execution at the inherited highest priority of the tasks being blocked on that resource (if higher than its own priority).

Rule 3 If task τ locks a free global resource, it will execute at its current active priority.

Rule 4 When task τ leaves the resource, the next highest priority task in the queue will be granted access to the resource.

For simplicity of implementation, Rajkumar et al [13] suggest that the priority when accessing a global resource can be raised to the ceiling immediately. We also note that with the basic MPCP there was a single synchronization processor and nested resource accesses are allowed. The protocol ensured deadlock free access. With the generalized MPCP, nested resources are not allowed. This is perhaps too strong a constraint. What is really required is that the group of resources involved in a nesting chain is assigned to the same synchronization processor.

Later, Rajkumar et al [14, 12] renamed the above protocol as the Distributed Priority Ceiling Protocol (DPCP) and clarified the remote access mechanism. In order to access a global resource, a task must acquire a local agent first. A local agent is a task on the remote processor where the global resource is being held. Any remote request for the global resource must be accomplished through the agent. When a global resource request is granted through the agent, the agent executes at an effective priority higher than any of the normal tasks on that processor. Hence, the protocol was targeted at distributed shared memory systems. For globally shared memory systems, the need for remote agents is removed and all global resources can be accessed from all processors. This protocol is now generally referred to as the MPCP.

Unlike in MSRP (see below), with MPCP, the highest priority task can be blocked, once started, by requesting resources being held by lower priority tasks. This results in potentially a double context switch on each resource access, while MSRP only requires one [15]. However, this comes at a cost of MSRP wasting processor cycles by busy-waiting.

[1] The term job is used in scheduling to indicate a single invocation (release) of a task: be it periodic or sporadic. Hence, with a static job-level execution eligibility, a task does not change its base execution eligibility during its release. However, its execution eligibility can differ from one release to another.

2.2 MSRP

The Stack Resource Policy, proposed by Baker [1], emphasizes that: "*a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling.*" The *system ceiling* is defined to be the highest ceiling of any currently locked resource. With this approach, all the tasks in the system can use the same run-time stack (hence the name of the policy).

In order to preserve this property, Gai et al. [9] proposed the Multiprocessor Stack Resource Policy(MSRP) for partitioned EDF scheduling. Resources are divided into two groups: local and global. Local resources are only accessed by tasks which execute on the same processor. Global resources are those which can be accessed by tasks running on different processors. Unlike SRP, global resources have different ceilings: one for each processor. The MSRP protocol is thus defined as:

Rule 1 For each individual processor, SRP is used for sharing local resources. Every tasks has a preemption level, and ceilings are given to local resources.

Rule 2 Nested resource access is allowed for local resource, and a local resource can use a global resource. However, nested global resources are prohibited – in order to prevent deadlocks.

Rule 3 For each global resource, every processor defines a ceiling greater than or equal to the maximum preemption level of the tasks allocated on that processor.

Rule 4 When a task requires a global resource R from processor k, it sets k's system ceiling to resource R's ceiling for that processor. Then, if the resource is free, it executes the resource. Otherwise, the task busy waits in a FCFS queue.

Rule 5 When a global resource is released by task τ_i executing on processor k, the task at the head of the FCFS queue (if any) will be granted access to the resource. The system ceiling for k is restored to its previous value.

Following these rules, global resources are shared across processors in a FCFS manner. Of course, a task on another processor will be able to acquire another global resource. Hence, it is possible that a task, once executing, will find some of its global resources already allocated. Hence, in order to maintain the ability for all tasks on a processor to share the same stack, it is necessary to non-preemptively busy-wait on the resource. The alternative approach would be to not allow the other tasks to executes (that is, implement a global system ceiling). This would results in processors potentially being kept idle.

2.3 FMLP

The Flexible Multiprocessor Locking Protocol (FMLP) proposed by Block et al [2] supports both global and partitioned systems. Resources under FMLP are divided by the programmer into long and short resources. The protocol introduces the notion of a *resource group*, which can contain either short or long resources but not both. A group containing short resources is protected by a non-preemptive FIFO queue lock. A group containing long resources is protected by a semaphore lock. Groups contain resources that are nested, so that a task can hold more that one resource at a time. Non nestable resources are grouped individually. Group locks have the unfortunate side effect of reducing parallelism and are potentially a impediment to composability. They do, however, allow deadlocks to be avoided (see Section 4).

The protocol is explained below:

- **Long Resource**. – A task τ that requires a long resource must acquire the group semaphore lock first. The group lock is served in FIFO order. Under global scheduling, while holding the group lock, it will inherit the maximum priority of any higher-priority task blocked on a resource in the group. For partitioned scheduling, global long resources are served non-preemptively and only local priority is inherited.

- **Short Resource**. – Local resource accesses can use SRP. Again, the group lock should be obtained by a task τ requiring nested short resources. For global resources, it does this in a non-preemptive manner and remains non-preemptive until it releases the lock.

To avoid added context switches when accessing short resources with global scheduling, Block et al [2] introduce the concept of linking a job to a processor. A job T is linked to a processor i at time t if it would be scheduled on processor i at t if all jobs were fully preemptible. Hence, a job would be linked to a processor where that processor is executing a lower-priority job non-preemptively.

2.4 PPCP

The main contribution of the PPCP (Parallel PCP) algorithm proposed by Easwaran and Andersson [8] is to increase the parallelism of the system by allowing low priority tasks to lock shared resources in some situations which are not permitted by PCP. Tasks are globally scheduled across processors, and priority inheritance is used when a task holding a resource is blocking a higher priority task. Each resource also has a ceiling value, which is the highest effective priority at which that resource can be requested. Hence, when a task is holding a resource, the task's effective priority is the priority inherited from other tasks it is blocking. However, the resource-holding tasks also has the notion of a "pseudo effective-priority". This is the maximum priority its effective priority can be boosted to. This is, of course, the ceiling of the resource. Details of the PPCP is given below:

Rule 1 A task τ_j is allowed to lock a shared resource if the total number of tasks with base priority less than j and pseudo effective-priority greater than j is at most α_i. α_i is an implementation defined value that determines the maximum length of the blocking chain of the conventional priority inheritance protocol. By setting α_i to 1, PPCP works similar to PCP. If set to the total number of tasks in the system, the protocol will have the same semantics as Priority Inheritance Protocol (PIP).

Rule 2 If there are any unassigned processors and a task τ_j does not require any resource, the free processor is assigned to τ_j. Otherwise, Rule 3 applies.

Rule 3 If the current executing task τ_j requires a resource and the resource is locked, PIP applies. If the resource is free, rule 1 applies and the resource is either assigned to τ_j or τ_j is suspended.

The value of α_i needs to be carefully chosen. The blocking chain is reduced at the price of reducing the response time of high priority tasks.

2.5 OMLP

The $O(m)$ locking protocol (OMLP) is a suspension-based resource sharing protocol proposed by Brandenburg et al [4]. The algorithm is proposed to reduce the interference from further priority inversion in FMLP caused by suspended tasks being blocked by long resources. Resources are either global or local. Local resources are shared between tasks on the same processor, and can be managed by any particular uniprocessor resource sharing protocols. OMLP can be used in either global or partitioned systems.

- **Global OMLP** – Each global resource, k, is managed by two queues: a FIFO queue (FQ_k) and a Priority Queue (PQ_k). A resource is always granted to the head element of FQ_k. The FQ_k is restricted to contain only m (the number of processors in the system) tasks waiting to access the resource. If the number of waiting task exceeds m, the rest are placed in PQ_k. Tasks in PQ_k are priority ordered, where the highest priority task will be the next task dispatched to FQ_k if the number of task in FQ_k at any time is less than m. When a task releases the resource k, it is dequeued from the FQ_k queue and the task at the new head of the queue is resumed at a priority which is equal to the highest priority of any task in FQ_k and PQ_k.

- **Partitioned OMLP** – Partitioned OMLP uses contention tokens to control access to global resources. There is one token per processor, which is used by all tasks on that processor when they wish to access a global resource. Associated with each token there is a priority queue PQ_m. There is now only one queue per global resource, a FIFO queue, again of maximum length m. In order to acquire a global resource, the local token must be acquired first. If the token is not free, the requiring task is enqueued in PQ_m. If free, the token is acquired, its priority is then raised to the highest priority on that processor, and the task is added to the global FQ_k and, if necessary, suspended. When the head of FQ_k finishes with the resource, it is removed from the queue, releases its contention token, and the next element in the queue (if any) is granted the resource.

Recently, Brandenburg and Anderson [5] have developed a new resource sharing scheme for clustered multiprocessor systems based on OMLP. The aim is to limit multiple repeated priority-inversion blocks that occur with a simple priority boosting scheme. Essentially, a high priority thread donates its priority to a lower priority thread to ensure that the lower priority thread is scheduled when holding a resource. Such a donation can only be performed once.

	Scheduling	Resources	Nested Resources	Access Priority	Queueing
MPCP	Partitioned	Yes	No	Ceiling (priority boosting)	Suspends in a priority-ordered queue
DPCP	Partitioned	Yes	No	Ceiling (priority boosting)	Suspends in a priority-oredered queue
MSRP	Partitioned	Yes	No	Non preemptive	Spins in a FIFO queue
FMLP	Both	No	Yes (Group lock)	Non preemptive for short	Short: Spins in a FIFO queue; Long: Suspends in FIFO queue
PPCP	Global	No	No	Inheritance	Suspends in a Priority queue
OMLP	Both	Yes	Yes (Group lock)	Inheritance for global, Non preemptive for partitioned	Suspends in FIFO and Priority-ordered (or Contention token) queues
Clustered OMLP	Clustered	No	Yes (Group lock)	Priority donation	Suspends in a FIFO queue

Figure 1. Summary of Resource Sharing Protocols

2.6 Queue Locks

For globally scheduled systems, spin locks usually form part of the resource access protocol. Queue locks are spin locks where resources are granted in FCFS order. Tasks waiting for the resource are inserted into a FIFO queue. Waiting tasks are generally non-preemptive. These spin lock resource sharing methods are preferable for small and simple resources where the time tasks spent in a resource is shorter than the time taken to perform a context switch [3][10]; FIFO queueing is usually preferred to priority queuing because the estimation of blocking time is easier (with more complex approaches, conservative assumptions of task arrival must be made which can lead to pessimistic blocking times) [3]. The disadvantage of preempting resource-holding jobs is the side-effect of substantially delaying other jobs waiting for the same resource. Also, in terms of partitioned scheduling, priorities across different processors may not be comparable to each other [3].

The main issue with spin locks is determining the worst-case time that a task can be held in the FIFO queue. This is beyond the scope of this paper; see Yang et al for a discussion on this issue [7, 6].

2.7 Summary

Figure 1 captures the main characteristics of the resource sharing protocols. The column headings have the following meaning.

Scheduling – whether the approach can be applied to global, partitioned or cluster scheduling.

Resources – whether the approach needs to distinguish between local and globally accessed resources.

Nested Resources – whether nested global resource access is allowed, and if so, what is the approach.

Access Priority – how the priority is set for accessing the resource: is it ceiling, inheritance, or non preemptive.

Queueing – are waiting tasks suspended or do they busy wait.

3 Ada and Multiprocessors

Ada, since Ada 2005, has supported different dispatching policies including FP and EDF. EDF dispatching can be applied across the whole range of priorities or across a restricted range. In this way, EDF scheduling is integrated into a FP framework. Baker's Stack Resource Policy [1] (SRP) is also integrated with Ada's ICP protocol to support resource sharing using protected objects.

The approach that Ada adopted for EDF scheduling was novel, and the protocol's correctness was not formally verified. As a result the initial definition was found to contain errors [17] and had to be corrected. Although the protocol is now believed to be correct, its properties on a multiprocessor platform are unclear. This is partly because Ada does not completely define how protected objects are accessed in a multiprocessor environment (see Section 3.2). Also the SRP cannot be directly applied to multiprocessors [9]. According to Gai et al [9], the SRP was designed for single processor systems and cannot be directly applied to multiprocessors.

> "The Stack Resource Policy has several interesting properties. It prevents deadlock, bounds the maximum blocking times of tasks, reduces the numer of context switches and can easily extend to multi-unit resources. ... However, the SRP does not scale to multiprocessor systems."

In this section, we first briefly summarise the the range of scheduling options available for multiprocessors systems as from Ada 2012. We then focus on accessing protected objects.

3.1 Scheduling

Ada 2012 will allow a wide range of scheduling approaches on multiprocessors, including

- Global preemptive priority-based scheduling
- Fully partitioned preemptive priority-based scheduling
- Global EDF scheduling
- Partitioned EDF scheduling

In addition to the above, Ada allows groups of processors to form "dispatching domains" (clusters), where each processor can only be part of one dispatching domain. Tasks can be globally scheduled within a single dispatching domain and it is also possible to fix a task to run on a single processor within a dispatching domain.

3.2 Accessing Protected Objects

The Ada Reference Manual (ARM) and its annotated companion (AARM) does not fully define the access protocol for a protected object on a multiprocessor system. The following points summarise the current position.

- Where there is contention for a protected object's lock, the recommendation to use spin-locks is a discussion point to a note.

  > "If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-defined ordering or queuing presumed for tasks competing to start a protected action – on a multiprocessor such tasks might use busy-waiting; for monoprocessor considerations, see D.3, "Priority Ceiling Locking". Discussion: The intended implementation on a multi-processor is in terms of "spin locks" the waiting task will spin." AARM Note 18, Section 9.5.1.

- It is implementation defined whether spinning occurs non-preemptively or, if not, at what priority. Furthermore, it is not defined whether there are queues (FIFO or priority) associated with the spin-lock.

  > "It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy." AARM D.2.1. par 3.

- The task which executes a protected action is not specified.

  > "An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to

complete. Reason: These permissions are intended to allow flexibility for implementations on multiprocessors. On a monoprocessor, which thread of control executes the protected action is essentially invisible, since the thread is not abortable in any case, and the "current_task" function is not guaranteed to work during a protected action (see C.7.1)." AARM 9.5.3 pars 22 and 22.a.

- The ceiling locking policy must be used with EDF scheduling.

  "If the EDF_Across_Priorities policy appears in a Priority_Specific_Dispatching pragma (see D.2.2) in a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition." AARM D.2.6 par 11/2.
  "The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities". AARM D.3 6/2.

- The value of the ceilings can be altered by the implementation.

  "Every protected object has a ceiling priority, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object." AARM D.3. par 8/2.

  "The implementation is allowed to round all ceilings in a certain subrange of System.Priority or System.Interrupt_Priority up to the top of that subrange, uniformly.

  Discussion: For example, an implementation might use Priority'Last for all ceilings in Priority, and Interrupt_Priority'Last for all ceilings in Interrupt_Priority. This would be equivalent to having two ceiling priorities for protected objects, "nonpreemptible and "noninterruptible, and is an allowed behavior. Note that the implementation cannot choose a subrange that crosses the boundary between normal and interrupt priorities." AARM D.3 par 14/14a.

4 Resource Sharing Protocols and Ada

Table 1 captures the main characteristics of the resource sharing protocols. The following summarises whether the current protocols are compatible with Ada.

- MPCP – No: suspends on an unavailable lock.
- DPCP – No: suspends on an unavailable lock.
- MSRP – Yes: if use ceiling of ceilings for resources.
- FMLP – Partial: short resources only.
- PPCP – No: suspends on an unavailable lock and no immediate inheritance.
- OMLP – No: suspends on an unavailable lock.

As can be seen, the MPCP, DPCP, PPCP and OMLP are not a good match for Ada as they suspend on access to a resource. In a fully partitioned system, that might be enforced by a Ravenscar-like profile, a slight variant of the MSRP can be used. If FMLP is constrained to short resources, it is essentially equivalent to using MSRP on the group lock and treating all resources as global. It, perhaps should be noted as an aside, that use of the Ada rendezvous could be considered as a long resource.

In Ada, nested resource access is allowed. In most of the protocols, either nested resource accesses are disallowed, or the notion of groups are introduced (as in FMLP) and a group lock must be obtained. This is to avoid deadlocks. In standard concurrency theory, deadlock occurs when the following four condition holds:

- mutual exclusion – only one task can use a resource at once (that is, the resource is non sharable or at least limited in its concurrent access);
- hold and wait – there must exist tasks which are holding resources while waiting for others;

- no preemption – a resource can only be released voluntarily by a task; and
- circular wait – a circular chain of tasks must exist such that each task holds resources which are being requested by the next task in the chain.

There are three possible approaches to dealing with deadlock:

- deadlock prevention – ensure that at least one of the above conditions does not hold at all times;
- deadlock avoidance – allow the four conditions to occur but undertake measures to ensure the system never deadlocks;
- deadlock detection and recovery – allow deadlocks to be detected and recovery processes to be performed.

Ada on a single processor adopts the deadlock prevention approach by preventing the *circular wait* condition. It does this by implicitly imposing an order on the resource accesses using ceiling priorities. A protected object with ceiling priority i cannot call a protected object of ceiling priority j if $i > j$. Unfortunately, this combines the notion of ordering with that of execution eligibility. We have already seen that for EDF scheduling it is necessary to separate out execution eligibility from preemption control. Here, Ada's priority is now interpreted to be preemption level, but again it is integrated into the scheduling framework.

For multiprocessor systems, global resources may need to be accessed non-preemptively. The Ada approach for achieving this is to use the priority mechanism to run tasks at the highest priority. By doing this, we have lost the ordering information that was implied by the use of ceilings. Hence, with multiprocessor Ada, deadlock is re-introduced.

In summary, in order to bound blocking and prevent deadlock

- for global FP or global EDF system, all protected objects run non-preemptively with queue locks;
- for fully partitioned systems, all global protected objects run non-preemptively with queue locks and local protected objects use ICP or SRP for FP and EDF respectively;
- for cluster systems, all global protected objects run non-preemptively with queue locks and local protected objects use ICP or SRP for FP and EDF respectively;
- nested protected object calls must be refactored to follow the FMLP protocol, or a new facility is required to re-introduce order into protected object accesses.

An application that wants to use Ada for predictable resource requires:

1. the ability to identify the global protected objects and set the ceiling to be higher than the highest priority of **all** the tasks running on **all** the processors from which the protected object can be accessed – this is achievable using the standard mechanisms for setting the ceilings of a protected object;
2. the implementation to support FIFO spin locks when accessing a protected object – this is not prohibited by the ARM. However, there is currently no documentation requirement for an implementation to specify its access protocol;
3. the ability to prevent deadlocks.

Note that 1) is a stronger requirement than that specified by MSRP for partitioned systems, which allows the ceilings to be dependent of the current processor executing the protected action. This is not implementable in Ada even with dynamic ceilings. Hence, the ceiling has to be set to the highest priority in the system. The effect on the task is the same, the task executes non-preemptively on that processor. The alternative approach is not to set the ceiling at all for global protected objects, as the default is for them to be executed non-preemptively.

The biggest problem facing an implementation is how to prevent deadlocks from occurring. Clearly applications can write their code to ensure deadlocks do not occur and use, for example, model checking techniques to show the absence of deadlock. The programmer can also program their own notion of the FMLP group lock (a *deadlock avoidance* approach) and make resources within a group non-protected.

The preferred way, however, would be to regain the order information that is present in Ada, perhaps using a new protected object aspect. This would allow more parallelism into the system compared to FMLP's group locks. However, it does introduce transitive blocking chains which complicate the blocking analysis.

4.1 EDF Scheduling and Fully Partitioned Systems

We note, the ARM wording is not very clear on how local protected objects are accessed in partitioned systems, section D.2.6 says:

> "the active priority when first activated or while it is blocked is defined as the maximum of the following:
>
> - the lowest priority in the range specified as `EDF_Across_Priorities` that includes the base priority of T;
> - the priorities, if any, currently inherited by T;
> - the highest priority P, if any, less than the base priority of T such that one or more tasks are executing within a protected object with ceiling priority P and task T has an earlier deadline than all such tasks.
>
> Ramification: The active priority of T might be lower than its base priority. AI95-00357-01 When a task T is first activated or becomes unblocked, it is added to the ready queue corresponding to this active priority. Until it becomes blocked again, the active priority of T remains no less than this value; it will exceed this value only while it is inheriting a higher priority. Discussion: These rules ensure that a task executing in a protected object is preempted only by a task with a shorter deadline and a higher base priority. This matches the traditional preemption level description without the need to define a new kind of protected object locking."

The above wording needs to be made tighter so that it refers to the processor on which a task is allocated.

5 Adding Flexibility to Protected Object Access

It is clear that techniques for multiprocessor scheduling and resource access are still in their infancy. The new facilities that Ada 2012 provides for global, partitioned and cluster scheduling have added a great deal of flexibility into the language. The hope is that as new hybrid models become available, programs adhering to these models can be programmed in Ada. Unfortunately, the access protocol to protected objects have been left as implementation-defined. In this section, we consider how easy it would be to support a variety of protocols. We include even those which might suspend the calling tasks, even though Ada considers the call of protected functions and procedures as nonblocking (see section 3.2). There is evidence in the literature [4] that blocking protocols might be more optimal.

Of course, general techniques for user-defined scheduling (such as those proposed by Rivas and Harbour [11]) would allow the programmer to control the protected object access protocol. Alternatively, one could suggest some form of interface using reflection [16]. More generally, we could separate out the sharing of data between parallel tasks and provide some abstract type different from a protected object.

Here, however, we provide a targeted proposal for the access protocol for protected objects. The goal is not to present a full proposal but to illustrate the type of facilities that could be supported.

5.1 Queue Locks

First, we suggest that the Real-Time Annex provides two new locking mechanisms. One where tasks spin waiting for the lock, the other where they are suspended.

Spin Lock – There are two main characteristics of a spin lock. The priority at which the spinning occurs and the order of the associated queue.

Suspension Lock – The main characteristic of a suspension lock is the queuing order. Here we assume that no automatic priority inheritance is performed.

```
with ...; use ...;
package System.Multiprocessors.Queue_Locks is

  type Queue_Order is (FIFO_Ordered, Priority_Ordered); -- for example
  type Spinning_Priority is (Active_Priority_Of_Task, Non_Preemptively);
  type Spin_Lock(Length : Positive := 1; Order : Queue_Order := FIFO_Ordered;
```

```
                At_Pri : Spinning_Priority := Non_Preemptively) is private;
  function Acquire(L : Spin_Lock) return Boolean;
    -- returns False if Queue full, True when lock acquired
  procedure Release(L : Spin_Lock);

  type Suspension_Lock(Length : Positive := 1;
                       Order : Queue_Order := FIFO_Ordered ) is private;
  function Acquire(L : Suspension_Lock)  return Boolean;
    -- returns False if Queue full, True when lock acquired

  function Remove_Head(L : Suspension_Lock) return Task_Id;
  procedure Add(L : Suspension_Lock; T : Task_Id);
  function Highest_Queued_Priority(L : Suspension_Lock) return Any_Priority;
  procedure Release(L :  Suspension_Lock);
private ...
end System.Multiprocessors.Queue_Locks;
```

The above package is useful in its own right.

5.2 Protected Object Access

In Table 1, the various approaches to resource sharing were summarized. The goal here is to allow an application to define its own access protocol. First, we assume the introduction of a new protected object aspect called `Global`; when set to true, this indicates that a) the PO can be accessed from more than one processor and b) the ceiling priority should be interpreted as an order requirement. A new locking policy, called `User_Protected` is also introduced. When this policy is enforced, every protected object can have an associated controller object which implements the access protocol. The following package illustrates the approach.

```
with ...; use ...;
package System.Multiprocessors.Protected_Object_Access is
  type Lock_Type is (Read, Write);
  type Lock_Visibility is (Local, Global);
  type Protected_Controlled is new Limited_Controlled with private;

  overriding procedure Initialize (C : in out Protected_Controlled);
  overriding procedure Finalize (C : in out Protected_Controlled);
  procedure Lock(C : in out Protected_Controlled; L : Lock_Type; V : Lock_Visibility;
                 Ceiling : Priority; Tid :  Task_Id);
  procedure Unlock(C : in out Protected_Controlled; Tid :  Task_Id);
private ...
end System.Multiprocessors.Protected_Object_Access;
```

When an object of `Protected_Controlled` is associated with a protected object, every time the Ada run-time system wants to lock/unlock the object it calls the associated `Lock`/`Unlock` procedure. A protected object can be associated with the controller, for example, as follows:

```
protected type X (PC : access Protected_Controlled) with
  Locking_Policy => (User_Protected, Lock_Visibility => Global, Locking_Algorithm => PC) is
  procedure A;
end X;
```

Now suppose an application wants to implement, for example, the global OMLP protocol. It does this by providing the following package

```
with ...; use ...;
package OMLP is
  type Global_OMLP(Num_Of_Additional_Tasks : Natual) is new Protected_Controlled with private;
  overriding procedure Initialize (C : in out Global_OMLP);
  overriding procedure Finalize (C : in out Global_OMLP);
  overriding procedure Lock(C : in out Global_OMLP; L : Lock_Type; V : Lock_Visibility;
                  Ceiling : Priority; Tid :  Task_Id );
  overriding procedure Unlock(C : in out Global_OMLP; Tid :  Task_Id);
private
  type Global_OMLP(Num_Of_Additional_Tasks : Natural) is new Protected_Controlled  with
  record
```

```
    PQ : Suspension_Lock(Num_Of_Additional_Tasks, Priority_Ordered);
    SQ : Suspension_Lock(Positive(Number_Of_CPUs), FIFO_Ordered);
    Original_Priority : Any_Priority;
   end record;
end OMLP;
```

The OMLP protocol makes use of two suspension queues. The FIFO queue's length is limited to the number of processors in the system, and the priority queue's length is limited to the number of tasks that can call the protected object minus the number of processors in the system (or zero, if there are less that the number of processors). As the OMLP implements priority inheritance on the task owning the lock, then the original priority of the task must be saved.

A simplified structure of the body of this package is shown below:

```
with ...; use ...;
package body OMLP is
  overriding procedure Initialize (C : in out Global_OMLP) is
  begin -- not shown; end Initialize;

  overriding procedure Finalize (C : in out Global_OMLP) is
  begin  -- not shown; end Finalize;

  overriding procedure Lock(C : in out Global_OMLP; L : Lock_Type; V : Lock_Visibility;
                Ceiling : Priority; Tid :  Task_Id ) is
    PF, PP : Any_Priority;
  begin
    -- For simplicity, the following assumes all locks require write access.
    -- We also omit the keeping track, using task attributes, of the
    -- locks already owned by the tasks, and the raising of ceiling violations
    -- if the order is violated.

    if not Acquire(C.SQ) then
      if not Acquire(C.PQ) then
        -- error
      end if;
    end if;
    -- lock acquired
    PF := Highest_Queued_Priority(C.SQ);
    PP := Highest_Queued_Priority(C.PQ);
    if PF > PP then Set_Priority(PF);
    else Set_Priority(PP); end if;
  end Lock;

  overriding procedure Unlock(C : in out Global_OMLP; Tid :  Task_Id) is
    T : Task_Id;
  begin
    -- need this to be an atomic action
    T := Remove_Head(C.PQ);
    Release(C.SQ);
    if T /= Null_Task_Id then Add(C.SQ, T); end if;
  end Unlock;
end OMLP;
```

5.3 Open Issues

The goal of this section was to consider the impact of having more control over accessing protected objects. Having explored one approach, several issues emerge.

- Does the approach given in Section 5.2 generalize to support entries? Or is it necessary for the run-time system to expose a low-level interface defining its interaction between guard evaluation and lock acquisition?
- Ada defines a call to a protected procedure/function (from the language viewpoint) as non blocking – what impact does it have if the application changes this? For example, if the program wanted to time out on access to the PO, it could use the following:

```
select
  delay X;
then abort
  PO.procedure_call; -- note, once the call is in progress it is abort deferred
end select;
```

- In Ada, when a protected action completes there is some epilogue code that is executed that services all the outstanding calls. Is this the most appropriate approach for multiprocessor systems?
- If a PO is called with an inherited priority, how can the application code determine this?
- The issue of interrupt handling and `User_Protected` protected objects.

6 Conclusions

This paper has examined the various multiprocessor resource control policies to determine which are compatible with Ada. The results can be summarized as follows:

1. Ada recommends spinning when accessing already-locked global protected objects, hence those protocols that use suspension are incompatible.
2. To bound blocking times, any spinning must be non preemptive and the queuing policy must be defined. Documentation requirements need to be added in the Real-Time Annex.
3. Deadlock with nested protected object calls can only be avoided if the programmer implements their own concurrency control protocol (such as FMLP resource locks);
4. For global protected objects, no ceiling priority should be set as the default is `System.Priority'Last`.

Given that real-time resource allocation protocols for multiprocessors are still in their infancy, there may be some merit is providing more primitive support for locking so that programmers can implement their our strategies.

References

[1] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3:67–99, March 1991.

[2] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

[3] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, Washington, DC, USA, 2008. IEEE Computer Society.

[4] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 49 –60, 30 2010-dec. 3 2010.

[5] B.B. Brandenburg and J.H. Anderson. Real-time resource sharing under cluster scheduling. In *EMSOFT*, page submitted, 2011.

[6] Y. Chang, R. I. Davis, and A. J. Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *Proceedings of 18th International Conference on Real-Time and Network Systems (RTNS)*, pages 99–108, 2010.

[7] Y. Chang, R.I. Davis, and A.J. Wellings. Improved schedulability analysis for multiprocessor systems with resource sharing. techreport YCS-2010-454, University of York, 2010.

[8] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *IEEE Real-Time Systems Symposium*, pages 377–386, 2009.

[9] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 73 – 83, 2001.

[10] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 189–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. G. Harbour M. A. Rivas. Application-defined scheduling in Ada. In *Proceedings of IRTAW 12, Ada Letters, Vol XXIII(4)*, pages 42–51, 2003.

[12] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[13] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259 –269, 1988.

[14] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing*, pages 116–125, 1990.

[15] J. Ras and A. M. K. Cheng. An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an SMP system. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, RTAS '09, pages 13–22, Washington, DC, USA, 2009. IEEE Computer Society.

[16] P. Rogers and A.J. Wellings. Openada: Compile-time reflection for Ada 95. In Albert Llamos and Alfred Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 166–177. Springer Berlin / Heidelberg, 2004.

[17] A. Zerzelidis, A. Burns, and A. J. Wellings. Correcting the EDF protocol in Ada 2005. *Ada Letters*, XXVII:18–22, April 2007.

Going real-time with Ada 2012 and GNAT

José F. Ruiz

AdaCore
46 rue d'Amsterdam
75009 Paris, France

Phone: +33 (0)1 49 70 67 16
Fax: +33 (0)1 49 70 05 52

ruiz@adacore.com

Abstract

This paper provides an overview of the real-time additions to Ada 2012 and their implementation status in GNAT. Most of these additions are related to multiprocessors, helping to exploit parallelism in an efficient and predictable way, and this is the area where GNAT has made more progress. There is also execution-time accounting of interrupts that has been added to bare board targets, where that GNAT run-time library has fine-grained control. The area of new scheduling policies is the one which has received less attention in GNAT, because of the lack or required support in underlying operating systems.

1 Introduction

New Ada versions use to come with attractive additions for the real-time community, and Ada 2012 could not be an exception. The new language revision addresses new hardware architectures, new capabilities provided by operating systems, and new aspects in real-time analysis. GNAT tries to follow these new additions swiftly, providing a testbed for early testing and verification of these new features, as well as a platform for their early adoption.

This paper describes the implementation status and future plans in GNAT Pro regarding the new real-time features added to Ada 2012. These are the main real-time areas where Ada 2012 provides new things:

- Ravenscar
- Multiprocessors
- Execution-time monitoring
- Scheduling and dispatching policies

The area with more activity is that of multiprocessors (including the use of Ravenscar on multiprocessors), but there has been evolutions in other areas. The following sections discuss these areas in detail. Many other Ada 2012 additions are already implemented in GNAT, but this paper focuses on those more closely tied to real-time systems.

2 Ravenscar for multiprocessors

During the last IRTAW 14, the author [18] presented some ideas for additions to the existing monoprocessor Ravenscar profile to support multiprocessor systems using a fully partitioned approach. An offspring of that proposal was the definition of Ada Issue 171 [7], where CPU affinity is defined statically using *pragma CPU*.

These ideas were implemented in GNAT bare board targets [12], initially targeting the LEON3 multiprocessor board [1], with the idea to extend its support to other multiprocessor bare board targets, such as PowerPC.

Scheduling is a simple extension to the monoprocessor fixed-priority preemptive scheduling algorithm supported by the Ravenscar profile, with tasks allocated statically to processors. Each processor implements a preemptive fixed-priority scheduling policy with separate and disjoint ready queues.

There is a single run-time system, where the only per-processor information is the ready and delay queues (those containing the tasks ready to execute and waiting for a delay respectively). Some run-time data is common and shared among tasks on different processors (such as the time reference).

When internal data in the run-time system can be modified by tasks executing on different processors, inter-processor locking mechanisms (fair locks [19], a cooperative mutual exclusion algorithm where the ownership of the lock is transferred in a round-robin fashion among waiting processors) are used to guarantee mutual exclusion.

Per-processor data (such as the ready queue) can only be modified by operations executing on the owner processor, so these operations can be performed simply disabling interrupts on the affected processor. When operations on one processor require actions on a different processor (such as waking up a task), a special interrupt is triggered in the target processor, which will them perform the actual operations.

The restricted library-level protected objects defined by the Ravenscar profile are used for inter- and intra-processor communication and synchronization. The same restrictions that exist in the Ravenscar profile for single processors apply to the case of a multiprocessor (a maximum of one protected entry per protected object with a simple boolean barrier using ceiling locking access).

Protected objects for inter-processor communication use multiprocessor synchronization mechanisms (fair locks). When a task waiting on an entry queue is awaken by another tasks executing on a different processor than the waiting task, an inter-processor interrupt facility is used to modify the ready queue in the target processor. Protected objects used only by tasks within the same processor always get access to the fair lock, so the multiprocessor overhead in this case is negligible.

The only supported Ravenscar-compliant mechanism to specify interrupt handlers is to attach a protected procedure. Interrupts can be configured to be handled by any number of processors, and the affinity mask for every interrupt is set up by the startup routine (there is no Ada high-level mechanism to do so). Affinity for interrupts is not part of AI05-0171 or any other Ada 2012 proposal, but it would be interesting to be able to use the same *pragma CPU* defined for tasks. The pragma could be attached to the definition of the protected handler.

A single common hardware clock and a single shared hardware timer are used for all processors. It provides a common reference for all the tasks in the system, thus avoiding time drifting problems, and limiting the amount of hardware resources required from the underlying platform. Each processor implements a separate and disjoint delay queue. When a task with an expired delay is in a different processor from the one handling the interrupt then an inter-processor interrupt is triggered on the target processor, and the handler for this inter-processor interrupt will traverse the list of local expired events, executing the required actions in the local queues. The implementation of the time-keeping functionality (clock) does not differ when migrating to a multiprocessor architecture.

In a nutshell, the implementation for multiprocessor LEON3 was achieved without adding much complexity to the run-time library. The same source code is used for both monoprocessor and multiprocessor, and the selection of the number of processors is simply a matter of changing a constant (the multiprocessor-specific code becomes deactivated code in monoprocessor systems, which is never present in the final application binary). The associated run time overhead remains small.

3 General purpose multiprocessor architectures

Previous section 2 described the implementation of a statically partitioned extension to Ravenscar. Obviously, the capability of statically allocating task affinities is of interest (and makes sense) outside the Ravencar restrictions.

Multiprocessor architectures in Ada were largely discussed during IRTAW 14 [10], including a flexible and general-purpose mechanism to handle task affinities in the form of dispatching domains [11].

The notion of task affinity is supported by mainstream operating systems (such as *Linux*, *Windows*, *Solaris*, *VxWorks*, ...) so GNAT implements *pragma CPU* on top of these operating systems. This pragma can appear in the task definition or in the declarative part of the main subprogram (for the environment task).

One of the challenges was to address smoothly versions of the target operating system not supporting task affinities. For example, *Linux* added the support for task/thread affinities to the 2.5 version. In order to be able to provide the same precompiled run-time library working on versions with and without task affinity support, weak symbols were used.

pragma Weak_External applied to a library-level entity marks its associated symbol as a weak symbol for the linker. It is equivalent to *__attribute__((weak))* in GNU C, and causes the entity to be emitted as a weak symbol instead of a regular symbol, that is to say a symbol that does not have to be resolved by the linker if used in conjunction with a *pragma Import*. When a weak symbol is not resolved by the linker, its address is simply set to zero. If a program references at execution time an entity to which this pragma has been applied, and the corresponding symbol was not resolved at link time, then the execution of the program is erroneous. It is not erroneous to take the *Address* of such an entity.

On *Linux*, function *pthread_attr_setaffinity_np* is the one used to set the CPU affinity mask of a task/thread (*sched_setaffinity* changes affinity for processes), but this is not available on relatively old versions.

```
function pthread_attr_setaffinity_np
  (attr       : access pthread_attr_t;
   cpusetsize : size_t;
   cpuset     : access cpu_set_t) return int;
pragma Import (C, pthread_attr_setaffinity_np,
               "pthread_attr_setaffinity_np");

--  Do nothing if required support not provided by the operating system

if pthread_attr_setaffinity_np'Address /= System.Null_Address and then
   T.Common.Base_CPU /= System.Multiprocessors.Not_A_Specific_CPU
then
   declare
      CPU_Set : aliased cpu_set_t := (bits => (others => False));
   begin
      CPU_Set.bits (Integer (T.Common.Base_CPU)) := True;
      Result :=
        pthread_attr_setaffinity_np
          (Attributes'Access,
           CPU_SETSIZE / 8,
           CPU_Set'Access);
      pragma Assert (Result = 0);
   end;
end if;
```

Support for multiprocessor dispatching domains [4] has recently been added to GNAT, using the primitives for handling affinities provided by *Linux*, *Windows*, *Solaris*, and *VxWorks*.

The Ada model is more restricted than the generic mechanism provided by the operating systems: dispatching domains are non-overlapping, they can only be created by the environment task, and they can only be created before calling the main subprogram. However, this more static model is flexible enough to support many different partitioning schemes, while at the same time providing for the definition of analyzable software architectures.

This implementation showed a couple of aspects worth fixing in the current definition of package *System.Multiprocessors.Dispatching_Domains*. First, and foremost, this unit cannot be *Preelaborate* because it depends on package *Ada.Real_Time* which is not *Preelaborate*. Second, the default dispatching domain *System_Dispatching_Domain* is defined as constant, but the dispatching domain it represents is actually not constant (it is implicitly modified by the creation of other dispatching domains).

4 Parallel task synchronization

With the widespread arrival of concurrent/parallel software architectures and parallel machines, effective parallel task synchronization is an interesting goal, not to be neglected by Ada.

At the last IRTAW 14, Michell et al. [15] proposed the addition of a new synchronization mechanism allowing a set of tasks to be blocked and be released at once. This functionality addresses the situation where

a group of tasks must wait until all of them reach a synchronization point, and then be released together to process concurrently.

The POSIX barrier mechanism [14], where a number of tasks wait on a barrier and are released simultaneously when there is a given number of waiting tasks, is the source of inspiration of this new functionality.

Ada Issue 174 [9] acknowledges the interest of this barrier paradigm, and a *Synchronous_Barrier* type has been added to Ada 2012 to allow many tasks to be blocked and be released together:

```
package Ada.Synchronous_Barriers is
   pragma Preelaborate(Synchronous_Barriers);
   subtype Barrier_Limit is Positive range 1 .. <implementation-defined>;
   type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is
      limited private;
   procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
                               Notified    :    out Boolean);
private
   -- not specified by the language
end Ada.Synchronous_Barriers;
```

When a variable of type *Synchronous_Barrier* is created, there are no waiting tasks and the barrier is set to block tasks. When the number of waiting tasks reaches the *Release_Threshold*, all tasks are simultaneously released and the *Notified* out parameter is set to *True* in an arbitrary one of the callers. All other callers result in *Notified* being set to *False* upon returning from the call.

On systems supporting POSIX barriers, such as Linux or IRIX, the GNAT implementation relies on the POSIX functionality (the Ada package is a straightforward binding to POSIX):

```
procedure Wait_For_Release
  (The_Barrier : in out Synchronous_Barrier;
   Notified    : out Boolean)
is
   Result : int;

   PTHREAD_BARRIER_SERIAL_THREAD : constant := -1;
   --  Value used to indicate the task which receives the notification for
   --  the barrier open.

begin
   Result := pthread_barrier_wait
     (barrier => The_Barrier.POSIX_Barrier'Access);
   pragma Assert
     (Result = 0 or else Result = PTHREAD_BARRIER_SERIAL_THREAD);

   Notified := (Result = PTHREAD_BARRIER_SERIAL_THREAD);
end Wait_For_Release;
```

On systems where the POSIX barrier functionality is not available, a functionally equivalent implementation is provided in GNAT using a protected entry:

```
protected type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is
   entry Wait (Notified : out Boolean);
private
   Keep_Open : Boolean := False;
end Synchronous_Barrier;

protected body Synchronous_Barrier is

   --  The condition "Wait'Count = Release_Threshold" opens the barrier
   --  when the required number of tasks is reached. The condition
   --  "Keep_Open" leaves the barrier open while there are queued tasks.
   --  While there are tasks in the queue no new task will be queued
   --  (no new protected action can be started on a protected object
   --  while another protected action on the same protected object is
   --  underway, RM 9.5.1 (4)), guaranteeing that the barrier will
   --  remain open only for those tasks already inside the queue when
   --  the barrier was open.

   entry Wait (Notified : out Boolean)
     when Keep_Open or else Wait'Count = Release_Threshold
   is
   begin
      Keep_Open := Wait'Count > 0;
      Notified := Wait'Count = 0;
   end Wait;
end Synchronous_Barrier;
```

The drawback of the implementation using the protected entry is the lack of the potential parallel release of waiting tasks, because tasks in an entry queue can only be released in a sequential manner, with the entry conditions being reevaluated each time.

5 Memory barriers

On multiprocessor systems, non-blocking algorithms (such as lock-free and wait-free) are extensively used to effectively exploit hardware parallelism. For example, an implementation like the following:

```
task Producer is
   pragma CPU (0);
end Producer;

task Consumer is
   pragma CPU (1);
end Consumer;

Shared_Data : Integer := 0;
pragma Volatile (Shared_Data);

Barrier : Boolean := False;
pragma Volatile (Barrier);

task body Producer is
begin
   loop
      -- Produce data
      Shared_Data := ...;

      -- Notify data is ready
      Barrier := True;
   end loop;
end Producer;

task body Consumer is
   Received_Data : Integer;
begin
   loop
      -- Wait for data
      while not Barrier loop
         null;
      end loop;

      Received_Data := Shared_Data;
      Barrier := False;

      -- Work with produced data
      ...
  end loop;
end Consumer;
```

The use of *Volatile* ensures (both in Ada 2005 and Ada 2012) the strict ordering within tasks because reads and updates of volatile or atomic objects are defined as external effects of the program (RM C.6 (20)), and the execution of the program must be consistent with the order imposed to these external events (RM 1.1.3 (15)). It means that, in task *Producer*, *Shared_Data* will always be updated before setting *Barrier* to *True*.

This feature makes things work as expected (*Consumer* reading *Shared_Data* when it has already been written) on monoprocessors, according to Ada 2005 rules. However, in Ada 2005, nothing guarantees that this will work on multiprocessors. Even when *Producer*'s processor (*0*) sees the correct order of updating the data and then updating the barrier, the effect of multi-level caches may make that *Consumer*'s processor observes a different order, hence reading an obsolete value of *Shared_Data*.

This problem is addressed by Ada Issue 117 [5], that modifies the semantics of *pragma Volatile* to require that all tasks of the program (on all processors) see the same order of updates to volatile variables.

Depending on the memory consistency model implemented by the target, ensuring this new order may require the insertion of special machine instructions, called memory barriers or memory fences (at the cost of reduced performance).

In Ada 2005 there were two aspects addressed by *pragma Volatile*: one was about the order of execution when reading or updating volatile variables, which has been made stricter in Ada 2012 to address the need

of controlling reordering on multicore architectures; the second was about reading and writing directly to memory, which has been left out of Ada 2012.

Note that the semantic change to *pragma Volatile* is not appropriate for memory-mapped devices, that require a means to enforce reads and updates to be made directly to memory. *pragma Atomic* should be used in these cases, which forces loads or stores of atomic objects to be implemented, where possible, by a single load or store instruction (RM C.6 (23/2)).

In GNAT, the intention is to keep the notion of "volatile" consistent with that existing in the gcc backend. However, we are actively working on this issue. The GCC *__sync_synchronize* built-in seems to be what is needed for implementing the new Ada 2012 semantics: it inserts a barrier after an atomic store, and both before and after an atomic load. GCC knows the right primitives for each architecture, transforming the built-in into the required object code.

6 Execution-time accounting of interrupts

During the last IRTAW 14, two independent proposals were submitted on the issue of execution time accounting of interrupt handlers (Gregertsen et al. [13] for Ravenscar and Aldea et al. [16] for the general case). They both agreed that charging the execution time of interrupts into the currently running task does not help accurate accounting. Ada Issue 170 [6] was define from these works, to enable the time spent in interrupt handlers to be accounted for separately from time spent in tasks.

Apart from MaRTE [16], separate accounting of interrupt execution time is not available in operating systems targeted by the GNAT development environment. On these systems, both *Interrupt_Clocks_Supported* and *Separate_Interrupt_Clocks_Supported* are set to *False* to reflect that the execution time of interrupt handlers is not accounted separately.

GNAT for Ravenscar bare board targets, where the Ada run-time library provides all the required support, accounts separately the execution time of individual interrupt handlers. The interrupt wrapper (the one which calls the user-defined interrupt handler) has been modified to update CPU clocks before and after the execution of the interrupt handler.

The model implemented is that the elapsed execution time is charged to the executing task or interrupt when a dispatching event occur. Dispatching events are defined as the following occurrences:

- Context switch between tasks
- Interrupt delivery (start interrupt handler). It can be either an interrupt preempting a running task or another interrupt (for proper tracking of individual interrupt handlers when interrupts are nested)
- Interrupt return (end interrupt handler). It can return to either a preempted task or interrupt (symmetric to the previous point)
- Task preempting an interrupt. Task can have a priority in the *Interrupt_Priority* range so this case must be taken into consideration
- Task returning to a preempted interrupt (symmetric to the previous point)
- Task or interrupt preempting the idle activity. When no task is ready to execute and there is no interrupt to be delivered, an idle loop is executed whose execution time is obviously not charged to any task or interrupt. However, this event requires start accounting for the preempting entity
- Task or interrupt returning to the idle activity. At this point, the elapsed execution time is charged to the running entity and then execution time accounting is temporarily disabled (symmetric to the previous point)

The overhead associated to monitoring execution time is not huge, but is pervasive. The implementation minimizes this overhead when execution-time clocks are not used (whenever package *Ada.Execution_Time* is not with'ed) using weak symbols (see section 3 for details about this mechanism).

7 New scheduling policies

There was a considerable amount of effort in Ada 2005 related to the introduction of new scheduling policies (non-preemptive, round-robin, EDF, priority-specific dispatching). Ada 2012 does not add any new scheduling policy, but it provides some new capabilities to already existing dispatching policies [8, 2] and some fixes [3].

No work has been performed yet on GNAT for these new functionalities, because they rely on functionality which is not provided by any GNAT supported targets (only implemented in MaRTE [17]).

8 Conclusions

Ada 2012 comes with interesting additions for real-time programming, and GNAT already supports an important part of it. Some others are in the pipeline.

The biggest addition in GNAT is the support for Ravenscar on multiprocessor bare board platforms, and the support for task affinities and multiprocessor dispatching domains on platforms supporting it (such as Linux, Windows, Solaris and VxWorks). A couple of issues worth fixing in the current definition of package *System.Multiprocessors.Dispatching_Domains* were found, but the Ada 2012 model seems to provide the appropriate support and flexibility for designing analyzable systems taking advantage of multiprocessor hardware.

Task barriers for parallel task synchronization have been implemented, using the underlying POSIX barriers when available, increasing the potential parallelism of applications using it.

Execution-time monitoring in GNAT for Ravenscar bare board targets has been extended with the support for accounting separately the execution time of individual interrupt handlers. Other targets do not support it because the required functionality is not present in the underlying operating system.

An interesting and important point in defining the semantics of Ada applications running on multiprocessor (parallel) architectures is the mechanism to ensure memory consistency. *pragma Volatile* has been modified to make the life of Ada users on multiprocessors easier, at the cost of some performance penalty, and some work on compiler vendor's plate. In GNAT, we are working on the addition of the required memory barriers with the help of the GCC back-end.

Finally, the area of modifications to new scheduling and dispatching policies has not yet received much attention in GNAT. The reason is that the required capabilities are not provided by mainstream operating systems.

References

[1] Aeroflex Gaisler. *LEON3 Multiprocessing CPU Core*, 2010. Available at `http://www.gaisler.com/doc/leon3_product_sheet.pdf`.

[2] ARG. Extended suspension objects. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0168-1.txt`.

[3] ARG. Glitch in edf protocol. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0055-1.txt`.

[4] ARG. Managing affinities for programs executing on multiprocessors. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0167-1.txt`.

[5] ARG. Memory barriers and volatile objects. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0117-1.txt`.

[6] ARG. Monitoring the time spent in interrupt handlers. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0170-1.txt`.

[7] ARG. Pragma CPU and Ravenscar Profile. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0171-1.txt`.

[8] ARG. Yield for non-preemptive dispatching. Technical report, ISO/IEC/JTC1/SC22/WG9, 2010. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0166-1.txt`.

[9] ARG. Implement task barriers in Ada. Technical report, ISO/IEC/JTC1/SC22/WG9, 2011. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0174-1.txt`.

[10] Alan Burns and Andy J. Wellings. Multiprocessor systems: Session summary. *Ada Letters*, XXX(1):147–151, April 2010.

[11] Alan Burns and Andy J. Wellings. Supporting execution on multiprocessor platforms. *Ada Letters*, XXX(1):16–25, April 2010.

[12] Fabien Chouteau and José F. Ruiz. Design and implementation of a Ravenscar extension for multiprocessors. In A. Romanovsky and T. Vardanega, editors, *Ada-Europe 2011, 16th International Conference on Reliable Software Technologies*, volume 6652 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011.

[13] Kristoffer Nyborg Gregertsen and Amund Skavhaug. Execution-time control for interrupt handling. *Ada Letters*, XXX(1):33–44, April 2010.

[14] IEEE. *Portable Operating System Interface (POSIX)*, 2008. IEEE Std 1003.1.

[15] Stephen Michell, Luke Wong, and Brad Moore. Realtime paradigms needed post Ada 2005. volume XXX of *Ada Letters*, pages 62–67. ACM, April 2010.

[16] Mario Aldea Rivas and Michael González Harbour. Execution time monitoring and interrupt handlers, position statement. *Ada Letters*, XXX(1):68–72, April 2010.

[17] Mario Aldea Rivas, Michael Gonzlez Harbour, and José F. Ruiz. Implementation of the Ada 2005 task dispatching model in MaRTE OS and GNAT. In *Reliable Software Technologies — Ada Europe 2009, 14th Ada-Europe International Conference on Reliable Software Technologies*, volume 5570 of *Lecture Notes in Computer Science (LNCS)*, 2009.

[18] José F. Ruiz. Towards a Ravenscar extension for multi-processor systems. *Ada Letters*, XXX(1):86–90, April 2010.

[19] S. Swaminathan, J. Stultz, J. F. Vogel, and Paul E. McKenney. Fairlocks — a high performance fair locking scheme. In *International Conference on Parallel and Distributed Computing Systems*, pages 241–246, 2002.

Adapting the end-to-end flow model for distributed Ada to the Ravenscar profile

Héctor Pérez Tijero, J. Javier Gutiérrez, and Michael González Harbour

Grupo de Computadores y Tiempo Real, Universidad de Cantabria
39005-Santander, SPAIN
{perezh, gutierjj, mgh}@unican.es

Abstract[1]

This paper presents a proposal to develop High-integrity Distributed Real-Time (HDRT) systems by integrating the real-time end-to-end flow model with the Ravenscar profile. Although this profile is being widely used in the development of single-processor critical hard real-time systems, further research is required to apply this profile in a distributed environment. This work is built upon the endpoints pattern, a technique that we have used to integrate the end-to-end flow model into Ada's Distributed Systems Annex (DSA). We adapt our previous work to the requirements of the Ravenscar profile. Therefore, this paper gives a step forward and discusses the modifications needed to make a specific instance of the endpoints pattern compatible with Ravenscar.

1. Introduction

High-integrity Real-Time (HRT) systems have reliability requirements that must be guaranteed throughout the execution of application. In order to guarantee those requirements, a thorough analysis of the system is required, resulting in a challenging process that must cope with strong restrictions and compliance to standards and formal development methods. For this purpose, Ada provides a set of facilities that include the "Guide for the use of Ada in high integrity systems" (GA-HI) [1], SPARK [2] and the Ravenscar profile [3]. While both of the former mainly focus on the sequential part of the language, the latter deals with the concurrent aspects of Ada.

Traditional approaches to build real-time applications are based on the knowledge of the system workload at design time. The schedulability of static distributed real-time systems can be analysed using techniques to determine a priori if their timing requirements can be met or not. The model that fits best in the analysis of distributed real-time systems is the transactional model [4][5], which is currently known as the end-to-end flow model in the MARTE specification [6]. The work presented at the 14th IRTAW [7] discussed how this model could be integrated into the Distributed Systems Annex of Ada (DSA) in order to support the development of analyzable real-time distributed applications in Ada.

The focus of this paper is the integration of the real-time end-to-end flow model into the development of High-integrity Distributed Real-Time (HDRT) systems. This topic was suggested for discussion at the 14th IRTAW [19], where it was considered desirable to standardize a real-time distributed model that could be applied both to the full version of Ada or to a restricted profile. In particular, this paper will explore how to adapt the Ada API proposed in [7] to the Ravenscar profile. Furthermore, a practical implementation is also provided as a proof of concepts to validate the usefulness of the approach. Finally, the suitability of a second approach is discussed for those systems that follow not only the guidelines included in the Ravenscar profile but also the restrictions defined in the GA-HI and SPARK documents.

This document is organized as follows. Section 2 introduces the previous work done to support the end-to-end flow model in distributed Ada following the DSA. Section 3 analyses in detail the new approach, describing the aspects required to make the interface compatible with the Ravenscar profile. Section 4 deals with the implementation of the approach within a High-Integrity (HI) middleware. Other solutions to use the end-to-end flow model with Ravenscar are considered in Section 5. Section 6 introduces complementary research work in this field by other authors and points out how to fit our approach in their proposals. Finally, Section 7 draws the conclusions.

1. This work has been funded in part by the Spanish Ministry of Science and Technology under grant number TIN2008-06766-C03-03 (RT-MODEL) and by the European Commission under project FP7/NoE/214373 (ArtistDesign).

2. The end-to-end flow model and the endpoints pattern

A key element in the development of distributed real-time systems is the schedulability analysis, normally based on a model that represents the timing behaviour of the application. With this aim, a real-time system can be modelled as a set of end-to-end flows that describe parts of an application consisting of multiple threads executing code in processing nodes, and exchanging messages through communication networks. Each end-to-end flow may be triggered by the arrival of one or more external events thereby causing the execution of activities in the processor or in the networks. These activities, in turn, may generate additional events, possibly with associated end-to-end timing requirements, which could trigger other activities involved in the end-to-end flow.

The work presented in [7] proposed mechanisms to express complex scheduling and timing parameters in a distributed system: the *endpoints* pattern. This proposal was aimed at supporting the event-driven end-to-end flow model and was integrated in different middlewares (for Ada DSA over GLADE [8] and for CORBA and Ada DSA over PolyORB [9]) enabling the use of interchangeable scheduling policies.

The endpoints pattern identifies two different schedulable entities. For the processing nodes, handler tasks intended to execute remote calls, and for the network, endpoints or communication points which are used to transport messages through the network. To customize the communication layer and handler tasks, the proposal in [7] defines a set of interfaces to explicitly create and configure both schedulable entities with the appropriate scheduling information.

Complete support for the end-to-end flow model is provided by a single parameter, called *Event_Id*, which represents the event that triggers the execution of an activity within its end-to-end flow. Under this approach, middleware is responsible of updating the scheduling parameters and managing the chain of events within the end-to-end flow, providing a separation of concerns between the logic of application and the real-time aspects, from a software engineering perspective.

In the end-to-end flow model, external events trigger the end-to-end flows, and the setting of an identifier of those events is the only operation that our model requires the application code to perform. The rest of the end-to-end flow elements, including the communication endpoints, the handler tasks, and all the scheduling parameters, can be described as a part of a configuration operation. After this configuration phase, the system will remain in a steady state. Furthermore, the real-time configuration of a distributed application following this model can be generated and validated automatically using CASE (Computer-Aided Software Engineering) tools.

Figure 1 illustrates a linear end-to-end flow performing one asynchronous remote procedure call through three processors and using one communication network. This example can represent a system that has to take some data from the environment, send it to another processor to be analyzed, and finally cause an action to occur in another place. The end-to-end flow is composed of five activities: (1) reading some data, (2) sending the data to be analyzed, (3) processing the data to perform a control actuation, (4) sending the actuation command and (5) performing an action depending on the command received. Figure 1-A shows the sequence of APCs (Asynchronous remote Procedure Calls, as defined in the DSA [3]) while Figure 1-B illustrates the equivalent real-time model.

Figure 1-C shows the details of the events, activities and processing resources (the processors or the network) according to the endpoints pattern. By setting the initial event *e1*, the application task triggers the activation of the end-to-end flow that generates the output event *e2*. This event causes a message to be sent to perform the first remote call. CPU-2 identifies the incoming event in the receive endpoint and transforms it to the corresponding event *e3*. Then, the middleware sets the appropriate scheduling parameters for the handler task accordingly to the event received. After processing the data, a message is sent to make the remote call to CPU-3 with event *e3*. Finally, the middleware in CPU-3 will transform the incoming event into event *e4* and will set the corresponding scheduling parameters for the actuation. This action completes the execution of the end-to-end flow.

The set of interfaces included in [7] provides support for the real-time model by means of the specification of all the resources involved in the end-to-end flow. This API consists of the following Ada packages:

- *Event_Management*, supporting the description of the end-to-end flow architecture via event transformations performed automatically at the transformation points.

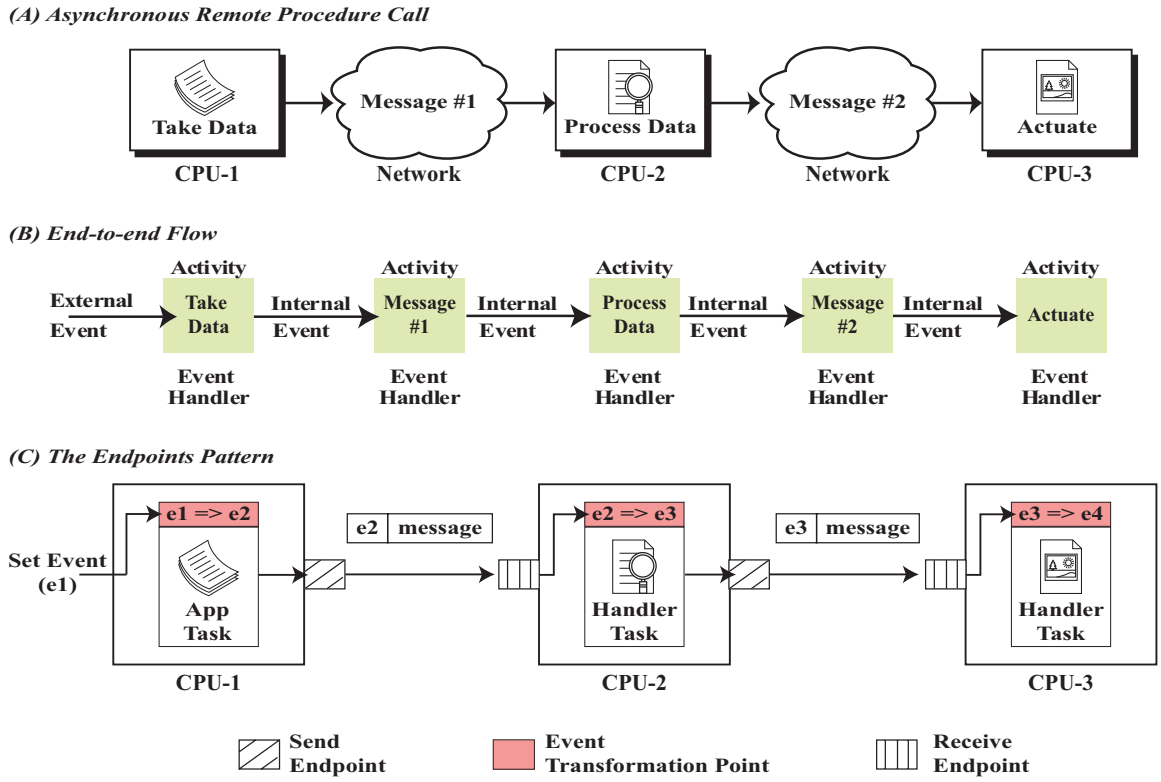


Figure 1: End-to-end flow model and the endpoints pattern

- *Processing_Node_Scheduling*, containing the operations to configure the handler tasks and to assign their scheduling parameters.
- *Network_Scheduling*, containing the endpoint configuration and assigning the scheduling parameters for the messages in the communication networks.

A detailed description of those interfaces can be found in [7]. Although this API is useful to build distributed real-time systems, it has not been designed to consider the restrictions imposed by the Ravenscar computational model, so the following sections will explore the strategy to allow the integration of the end-to-end flow model in the development of HDRT systems.

3. Adapting the endpoints pattern to the Ravenscar profile

The Ravenscar profile defines a set of restrictions in the system concurrency model, thus imposing a review of the original proposal [7]. In particular, the following restrictions must be considered:

- **Only the FIFO_Within_Priorities dispatching policy is allowed**

  The approach proposed in [7] is independent of the selected policy. However, this flexibility does not violate this restriction because the choice of available policies remains implementation-defined. Furthermore, keeping this flexibility enables future profile extensions as proposed in [10].

- **The set of tasks in the system is fixed and created at library level**

  The *Processing_Node_Scheduling* interface provides operations to create handler tasks at configuration time, which it is not Ravenscar compliant, and thus requires a revision of the proposal in [7].

- **Tasks have static scheduling parameters**

  Although the current approach allows handler tasks to update their scheduling parameters at runtime according to the retrieved *Event_Id,* this feature is not compatible with Ravenscar and must be disabled. However, the

transmission of the *Event_Id* parameter remains necessary in order to allow sharing the same handler task among multiple end-to-end flows, if needed.

Although the end-to-end flow model does not violate any further Ravenscar restrictions, there are some other aspects that middleware implementations should take into account:

- **Prevent the use of task attributes**

  The middleware implementation for Ada DSA and CORBA [9] uses task attributes to store the *Event_Id* parameter.

- **All tasks are non-terminating**

  The *Processing_Node_Scheduling* interface provides operations to destroy handler tasks that must be disabled.

The Ravenscar profile has mainly been applied for the static analysis of applications running within a single or multiple nodes but without considering the communication networks in the analysis. However, the real-time distributed model proposed in this work includes the endpoints as schedulable entities representing the communication points within the network. The ARINC 653 [20] specification follows a similar approach through the definition of *ports*, entities that enable the interpartition and intrapartition communication and behave as the input points to a real-time network (e.g., AFDX [21]).

Each of these considerations must be addressed within the set of interfaces described in the previous section. The modifications proposed over the original API in [7] are detailed in the following subsections.

3.1. Event Management Interface

End users should configure the sequence of events within an end-to-end flow, and the middleware will be in charge of automatically setting the appropriate event at the transformation points of the remote call as shown in [7]. This interface is Ravenscar compliant and therefore it does not require any modification from the original, which is presented below.

```
package Ravenscar.Distributed.Event_Management is

   procedure Set_Event_Assocation (Input_Event : Event_Id; Output_Event : Event_Id);

   function Get_Event_Association (Input_Event : Event_Id) return Event_Id;

   procedure Set_Event_Id (New_Event : Event_Id);

   function Get_Event_Id return Event_Id;

end Ravenscar.Distributed.Event_Management;
```

3.2. Network Scheduling Interface

The overall response time of a distributed system is strongly influenced by the underlying networks and therefore networks are required to be scheduled with appropriate techniques. This API addresses this aspect by making the communication endpoints visible, and by associating scheduling parameters to the messages sent through them. The approach in [7] is already Ravenscar compliant and thus it can remain unaltered. However, the use of class-wide types and operations are disallowed in HI systems [1][2]. As a result, the corresponding Network Scheduling Interface is defined as follows:

```
package Ravenscar.Distributed.Network_Scheduling is

   type Message_Scheduling_Parameters is abstract tagged private;

   procedure Create_Receive_Endpoint
     (Net          : Generic_Network;
      Port         : Generic_Communication_Port;
      Endpoint     : out Receive_Endpoint_Id) is abstract;
```

```
   procedure Create_Send_Endpoint
      (Param        : Message_Scheduling_Parameters;
      Dest_Node     : Generic_Network_Address;
      Event         : Event_Id;
      Net           : Generic_Network;
      Dest_Port     : Generic_Communication_Port;
      Endpoint      : out Send_Endpoint_Id) is abstract;

   procedure Create_Reply_Receive_Endpoint
      (Net          : Generic_Network;
      Event_Sent    : Event_Id;
      Port          : Generic_Communication_Port;
      Endpoint      : out Receive_Endpoint_Id) is abstract;

   procedure Create_Reply_Send_Endpoint
      (Param        : Message_Scheduling_Parameters;
      Dest_Node     : Generic_Network_Address;
      Event         : Event_Id;
      Net           : Generic_Network;
      Dest_Port     : Generic_Communication_Port;
      Endpoint      : out Send_Endpoint_Id) is abstract;

   procedure Destroy_Receive_Endpoint (Endpoint : Receive_Endpoint_Id) is abstract;

   procedure Destroy_Send_Endpoint (Endpoint : Send_Endpoint_Id) is abstract;

private
   type Message_Scheduling_Parameters is abstract tagged ...;

end Ravenscar.Distributed.Network_Scheduling;
```

The operations provided to destroy endpoints are not strictly necessary in this kind of static systems and can be removed. However, in order to keep similar interfaces for full and restricted Ada, both operations have been included.

Extensions of the *Message_Scheduling_Parameters* tagged type will contain the specific network scheduling parameters that must be associated to a specific endpoint. Furthermore, each scheduling policy must implement operations to map its own scheduling parameters (e.g., priorities) onto extensions of this private type. However, the use of abstract types in HI systems can be controversial, as they are forbidden by SPARK [2] but allowed by GA-HI [1].

3.3. Processing Node Scheduling Interface

Handler tasks are responsible for awaiting remote requests and processing them. As we stated before, the proposal in [7] to create and manage handler tasks relied on the dynamic creation of tasks which is forbidden in Ravenscar systems. The new API uses a set of Ada packages instead: a *Processing_Node_Scheduling* package to perform the registration and identification of tasks in the system, and a set of child packages to create tasks with the appropriate scheduling parameters.

One child package per scheduling policy is required. Since handler tasks must be created explicitly at library level, the new API considers the creation of tasks through a generic package which has been demonstrated to be a suitable approach [11]. This generic package includes the following parameters and operations:

- *Task scheduling parameters:* The scheduling parameters are set statically via a pragma.
- *Handler_Task_Callback:* Procedure used as callback for each middleware implementation. There is only one input parameter: the endpoint where the handler task will wait for an incoming request. How to create and obtain the endpoint is implementation-defined.
- *Create_Handler_Task_Endpoint:* This function returns the receive endpoint where the calling handler task will wait for incoming requests. Since the communication endpoints are created through the API (i.e. during the configuration phase), and handler tasks are created at library level, both entities must be created in a predefined order.

Furthermore, this generic package can be completed by including several optional subprograms and parameters; for instance, to execute the basic initialization operations required within each middleware implementation, to execute recovery procedures when any error is detected or to specify basic properties associated to a task (e.g., the stack size).

Finally, the use of generics, although used in many safety-critical research Ada projects [11][12], is debatable, as it is forbidden by SPARK but allowed by GA-HI. As an example, the following package represents the generic unit used for fixed priorities scheduling:

```
-- Dependences are omitted
generic

   Handler_Task_Priority   : System.Priority;

   with procedure Handler_Task_Callback (Endpoint : Receive_Endpoint_Id);

   with function Create_Handler_Task_Endpoint return Receive_Endpoint_Id;

package Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities is

   task FP_RPC_Handler_Task is
      pragma Priority (Handler_Task_Priority);
   end FP_RPC_Handler_Task;

end Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities;
```

3.4. Example of real-time configuration

The real-time configuration file shall include the code required to create and configure each end-to-end flow defined in the system. This section describes the configuration required for the scenario illustrated in Figure 1. As an example, we will show how to generate the configuration code for CPU-2, but a similar procedure could be followed for CPU-1 and CPU-3. The configuration code for CPU-2 should include the following elements:

- *Handler_Task_1* package: It is responsible of the creation of a handler task, the creation/retrieval of the associated receive endpoint and the callback to the main loop defined in the middleware implementation.
- *Set_Partition_Configuration* procedure: It performs the configuration of the rest of the end-to-end flow elements (communication endpoints, scheduling parameters and event association).

As a consequence, the Ada specification for the configuration of CPU-2 is shown below:

```
package Partition_2_Configuration_File is

   package Handler_Task_1 is new Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities
      (Handler_Task_Priority          => Handler_Task_1_Priority,
      Create_Handler_Task_Endpoint    => Partition_2_Configuration_File.New_Receive_Endpoint,
      Handler_Task_Callback           => MW_Implementation.RPC_Main_Loop);
   ...
   procedure Set_Partition_Configuration;
end Partition_2_Configuration_File;
```

On the other hand, the configuration code for CPU-2 (*Set_Partition_Configuration* procedure) should include the following procedure calls:

- *Set_Event_Association*: As can be seen in Figure 1, CPU-2 receives *e*2 as incoming event and transforms it to event *e*3. Such mapping between events remains registered in the middleware.
- *Create_Send_Endpoint*: This partition considers two network transmission activities: one of them is for sending messages and thereby associated with one send endpoint with its specific scheduling information. The other network activity corresponds to the incoming messages and therefore it must be mapped to a receive endpoint. This endpoint is now created by the associated handler task to avoid the complexity of synchronization issues during the creation of endpoints (configuration phase) and handler tasks (library level).

Using the procedures mentioned above, the Ada package body for the configuration of Partition 2 is shown below:

```
package body Partition_2_Configuration_File is

   procedure Set_Partition_Configuration is
   -- The definition of variables is omitted
   begin
      -- Set event associations
       Set_Event_Assocation  (Input_Event  => e2, Output_Event => e3);

      -- Create one send endpoint
    Create_Send_Endpoint
       (Param          => Msg_Scheduling_Parameters_e3,
       Dest_Node       => CPU-3,
       Event           => e3,
       Net             => Default_Network,
       Dest_Port       => Rcv_Port_Partition_e3,
       Endpoint        => Snd_Endpoint_Id_e3);
   end Set_Partition_Configuration;

end Partition_2_Configuration_File;
```

Finally, these real-time configuration files can be generated and validated automatically using CASE tools to ease the development of large scale systems. Next section briefly describes the implementation of our proposal in a middleware for HI systems. This implementation can be integrated within an MDE (Model-Driven Engineering) process as it is described in [17].

4. Implementation of the endpoints pattern within a HI middleware

This section describes how the endpoints pattern has been integrated into Ocarina and PolyORB-HI [12], a software tool suite that includes a complete framework to build HDRT systems.

The architecture of Ocarina comprises two different parts: a frontend, which processes the system model described in the input file, and a backend, which implements the strategies to generate the source code for different targets. The current version supports the AADL modeling language as input and several targets, such as those based on the PolyORB-HI middleware, as output.

PolyORB-HI is a lightweight distribution middleware compatible with the restrictions specified by the Ravenscar profile. It is distributed with the Ocarina tool as an AADL runtime that provides all the required resources (i.e. stubs, skeletons, marshallers and concurrent structures) to build high-integrity distributed systems. The current software release provides three runtimes depending on the target system: PolyORB-HI-C, PolyORB-HI-Ada and PolyORB-HI-QoS.

To validate the proposed approach, Ocarina and PolyORB-HI have been extended to provide a new backend or code generation strategy called PolyORB-HI-Endpoints, which, together with the implementation of the endpoints pattern, automatically generate the distribution source code. This Ravenscar-compliant source code is based on the real-time end-to-end flow model and makes use of the proposed interfaces.

This architecture can be extended to include the automatic generation of the real-time configuration code as illustrated in Figure 2. Under this approach, the MAST [18] model can be auto generated from the system model (e.g., through a new Ocarina backend or an external AADL2MAST conversion tool [17]) and thus providing the necessary support to perform a static verification of the end-to-end deadlines. As a result of the analysis, a real-time configuration file is created containing all the timing parameters required. Further details can be found in [17].

A prototype implementation of the PolyORB-HI-Endpoints backend on x86 architecture and an UDP-based network has been developed as a proof of concepts. This prototype uses fixed-priority scheduling policies for both schedulable resources: processors and networks. The network uses the 802.1p specification (which is included in the document 802.1q [13]) to prioritize different message streams. The real-time configuration file is currently generated by hand, although automatic generation of the real-time configuration code is under development following the approach described in Figure 2.

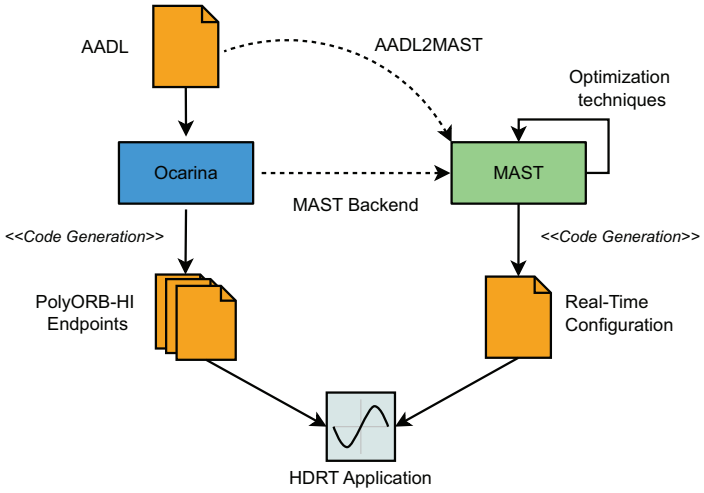


Figure 2: Ada Toolchain for HDRT systems

5. The endpoints pattern and static distributed Ada

In Section 3, we have proposed the adaptation to Ravenscar of an API designed to support the timing analysis of distributed real-time systems in full Ada. Although the changes suggested are compatible with the Ravenscar profile, they are not exempt from some points of controversy, summarized as follows:

- **The use of abstract types and generic units.** Although both features are used in some HI research projects, they are not included in SPARK.
- **The use of a configuration interface.** While the Ravenscar profile seems to fit better in an absolute static system, the use of the API presents a more dynamic nature: the system is static but only after the configuration phase.

As a consequence, it is reasonable to make a step forward and ask if the adaptation is worthy enough. One of the most important advantages of defining the interface is that it represents a quite similar solution for full and restricted Ada. However, another kind of solution could fit better in HI systems. The next list briefly summarizes the minimum requirements to develop HDRT systems following the end-to-end flow model. As a consequence, the configuration of the application can be divided into three phases:

- Configuring the partition of the program, as required by the DSA. This step is implementation-defined.
- Identification and configuration of the schedulable entities. This step should contain the following semantics:
  - A distributed real-time application defines two kind of schedulable entities: the tasks for the processor, and the messages for the networks.
  - The implementation shall provide means to explicitly create and configure each schedulable entity and associate the handler tasks with the appropriate receive endpoint.
- Identification and configuration of the different end-to-end flows. This step should contain the following semantics:
  - The end-to-end flow is a conceptual entity that describes a sequence of processing steps (e.g. the execution of a piece of code or the transmission of a network message).
  - The implementation shall provide means to explicitly set the initial event for the user task. This can be achieved by defining a new pragma associated to tasks to identify the starting end-to-end flow.
  - The implementation shall provide means to explicitly perform the event mapping.
  - The implementation shall include the *Event_Id* parameter as part of the network message if a single handler task may process several requests matching different end-to-end flows.

6. Related work

In this section we briefly present some of the research which aims to extend the Ravenscar profile to address HDRT systems. These works can be classified as follows:

- **Ravenscar and Ada DSA.** These works are mainly focused on the adaptation of the Ada DSA to be Ravenscar compliant as discussed in [15] and [16]. The main disadvantage of this option is the lack of a standard real-time distributed framework.
- **Ravenscar and a custom mechanism to perform the distribution.** Under this approach the HDRT systems are built by automatically generating source code from architectural descriptions (i.e. system models). A representative work is the tool suite Ocarina, which has been used in this paper.

Our approach does not only rely on the Ada DSA to make the distribution but it can be applied to different distribution models [14]. Nevertheless, it can be considered as a complementary work to [15] and [16] since the end-to-end flow model provides a feasible architecture to perform a static timing analysis in which the timing requirements of the applications can be validated. Our approach does not deal with other important features required in HI systems and detailed in such references, like the coordinated elaboration of partitions or the bounded size of network messages. Moreover, our real-time distributed model still provides support for some of the restrictions discussed in the previous IRTAWs:

1. **The use of synchronous remote calls (RPCs).** This kind of communication is amenable to timing analysis and current techniques have actually reduced the pessimism introduced in a wide range of scenarios. However, there exist typical situations in distributed systems whose analysis can be still improved; for instance, the existence of simultaneous activations owning to the same end-to-end flow (i.e. a linear end-to-end flow with the response time greater than the period). The endpoints pattern supports the use of RPCs by defining the *reply* endpoints.

2. **The use of concurrent remote calls.** In distributed systems, the buffering of incoming requests usually relies on the services provided by communication networks, such as in AFDX [21] or SpaceWire [22]. Furthermore, the problem of dimensioning a wait queue to hold the incoming requests is already considered by the timing analysis techniques. Therefore our approach does not preclude the reception of concurrent remote calls.

3. **The use of nested RPCs.** The end-to-end flow model is able to compute the waiting times associated to the nested remote calls and thus, from the real-time perspective, the response time analysis can be performed except when the response time of the end-to-end flow is greater than the period.

7. Conclusions

In this work, we have presented a specific model of the endpoints pattern compatible with the Ravenscar profile to provide the means to build and analyse real-time end-to-end flows in HDRT systems. The initial pattern, which was proposed for full Ada at the 14th IRTAW, was not compatible with the Ravenscar profile and required several changes to make a compatible version.

The adaptation of the endpoints pattern has involved a set of modifications mainly related to the way in which handler tasks are created and assigned their scheduling parameters. However, the solutions provided in this paper have considered the restrictions included not only in the Ravenscar profile but also in the “Guide for the use of Ada in high integrity systems” and SPARK. As a consequence, we have identified some points of controversy between those documents, including:

- **The use of abstract types.** In this case, our API defines a set of abstract subprograms intended to be overridden by each scheduling policy implemented. Although this approach is allowed by GA-HI, SPARK explicitly excludes the use of abstract types.
- **The use of generic units.** Under our approach, the API considers the creation of tasks through a generic package. Although this feature has been used in several safety-critical research Ada projects, the current version of SPARK does not allow it.

Therefore, it would be convenient to clarify both controversial features before submitting a final version of our proposal. Although the solution is based on a set of APIs, this paper has also explored the use of an even more static approach that can be applied at compilation time and thus can fit better in the development of HDRT systems. However, the definition of a generic solution for full and restricted Ada would be desirable.

References

1. ISO/IEC TR 15942, "Guide for the Use of the Ada Programming Language in high integrity Systems", Technical Report, Mars 2000.
2. SPARK LRM, "SPARK - The Spade Ada Kernel", November 2010.
3. Taft, S. T., Duff, R. A., Brukardt, R., Ploedereder, E. and Leroy, P. (Eds.). "Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1". LNCS 4348, Springer, (2006).
4. Tindell K.: Adding Time-Offsets To Schedulability Analysis. Technical Report YCS_221 (1994).
5. Palencia J.C., and González Harbour, M, "Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems". 20th IEEE Real-Time Systems Symposium, pp. 328-339 (1999).
6. Object Management Group. "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems". MARTE specification version 1.0 (formal/2009-11-02).
7. Pérez Tijero, H., Gutiérrez, J.J., González Harbour, M.: "Support for a Real-Time Transactional Model in Distributed Ada". Proc. of the 14th International Real-Time Ada Workshop (IRTAW 14), Portovenere (Italy), ACM Ada-Letters XXX(1), pp. 91-103 (2010).
8. López, J., Gutiérrez, J.J., González Harbour, M. "Interchangeable Scheduling Policies in Real-Time Middleware for Distribution". 11th International Conference on Reliable Software Technologies, Porto (Portugal), LNCS, Vol. 4006, Springer, pp. 227-240 (2006).
9. Pérez Tijero, H., Gutiérrez, J.J., Sangorrín, D. and González Harbour, M. "Real-Time Distribution Middleware from the Ada Perspective". 13th International Conference on Reliable Software Technologies, Ada-Europe, Venice (Italy), LNCS 5026, pp. 268-281 (2008).
10. White, R.: Providing Additional Real-Time Capability and Flexibility for Ada 2005. Proc. of the 14th International Real-Time Ada Workshop (IRTAW 14), Portovenere (Italy), ACM Ada-Letters XXX(1), pp. 135-146 (2010).
11. Bordin, M., Vardanega, T.: Correctness by construction for high-integrity real-time systems: A metamodel-driven approach. Proc. of the 12th International Conference on Reliable Software Technologies - Ada- Europe 2007, number 4498 in LNCS, pp. 114-127. Springer-Verlag (2007).
12. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Tr. Embedded Computer Systems, 7(4), pp. 1-25 (2008).
13. IEEE Std 802.1Q: Virtual Bridged Local Area Networks. Annex G. IEEE Document (2006).
14. Pérez Tijero, H. and Gutiérrez, J.J. "Experience in integrating interchangeable scheduling policies into a distribution middleware for Ada". Proc. of the ACM SIGAda Annual International Conference on Ada and Related Technologies, SIGAda 2009, Saint Petersburg, Florida, USA, ACM Ada-Letters XXIX(3), pp. 73–78 (2009).
15. Audsley, N. and Wellings, A. "Issues with using Ravenscar and the Ada distributed systems annex for high-integrity systems". Proc. of the 10th International Real-Time Ada Workshop (IRTAW 10), Las Navas del Marqués (Ávila), Spain, ACM Ada-Letters XXI(1), pp. 33–39 (2001).
16. Urueña, S., Zamorano, J. "Building High-Integrity Distributed Systems with Ravenscar Restrictions". Proc. of the 13th International Real-Time Ada Workshop (IRTAW 13), Woodstock, VT, USA, ACM Ada-Letters XXVII(2): 29-36 (2007).

17. Pérez Tijero, H., Gutiérrez, J.J., Asensio, E., Zamorano, J. and de la Puente, J.A. “Model-Driven Development of High-Integrity Distributed Real-Time Systems Using the End-to-End Flow Model”. Proc. of the 37th Euromicro Conference on Software Engineering and Advanced Applications, Oulu, Finland, pp. 209-216 (2011).
18. González Harbour, M., Gutiérrez, J.J., Palencia, J.C. and Drake, J.M. “MAST: Modeling and Analysis Suite for Real Time Applications,” Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, (2001).
19. Vardanega, T., Harbour, M. G. and Pinho, L. M. "Session summary: language and distribution issues", Proc. of the 14th International Real-Time Ada Workshop (IRTAW 14), Portovenere (Italy), ACM Ada Letters, XXX(1): pp. 152-161 (2010).
20. ARINC. “Avionics Application Software Standard Interface”. ARINC Specification 653-1 (2006)
21. Airlines Electronic Engineering Committee, Aeronautical Radio INC., “ARINC Specification 664P7: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network,” (2005).
22. European Corporation for Space Standardization (ECSS), “SpaceWire - links, nodes, routers and networks”, pp. 1-129 (2008).

Charting the evolution of the Ada Ravenscar code archetypes

Marco Panunzio and Tullio Vardanega
Department of Pure and Applied Mathematics, University of Padova
via Trieste 63, 35121 Padova, Italy
{panunzio, tullio.vardanega}@math.unipd.it

Abstract

In this paper we present the rationale, the status and the planned enhancement of a set of code archetypes that implement common programming patterns suited for the development of Ravenscar-compliant real-time systems. There have been other attempts at building software frameworks that ease the construction of real-time software systems. Ours is not intended for direct access by the user, but for deployment in the back-end code generation engine of a model-based tool environment. A further distinguishing characteristic of our patterns is that they foster the principle of separation of concerns, whereby the functional code of the system (which we require to be purely sequential) stays under the responsibility of the user, whereas the code that realizes the intended concurrency and real-time semantics is obtained by instantiation of predefined, correct by construction, archetypes.

1 Context and motivations

The development of high-integrity real-time systems is subject to stringent process and product requirements. The nature of those systems requires designers to prove the correctness of the system behaviour in the time dimension, which is typically sanctioned by use of some schedulability test or analysis.

Modern development paradigms, such as component-oriented approaches [1] and Model-Driven Engineering [2] foster the early analysis of the system under development, based on an architectural description of it.

Following the MDE development paradigm, the designer decorates the architectural elements with attributes in the dimensions of interest to analysis, e.g., concurrency and time for the purposes of this work. Those attributes express either requirements on the timing behaviour of the software, such as the period of a periodic operation or the minimum inter-arrival time (MIAT) of a sporadic operation, or estimates and bounds to characteristics that depend on system execution, such as the worst-case execution time (WCET) of all units of scheduling.

Early analysis is an important engineering bonus, for it enables the designer to rapidly iterate, at negligible cost, before committing the implementation, until the predictions on the timing behaviour meet all the applicable requirements.

The wisdom of this approach is increasingly accepted within the industrial community. Yet, the prevalent application of that approach overlooks the problem that guarantees must be had that the system at run time actually behaves in accord with the analysis predictions. It is commonly experienced in fact that the implementation may in many, possibly subtle, ways violate assumptions or constraints depended upon by the analysis theory. Notorious examples include the accuracy of the WCET bounds, the guarantee of minimum inter-arrival, the absence of task self-suspension. What the user often fails to appreciate is that the attributes used as input for the analysis equations form the ”frame” in which the analysis results hold; for a system validated by analysis, those attributes shall be regarded as ”properties” to be preserved: any deviation from them incurred at run time may jeopardize the analysis guarantees.

It is therefore necessary to complement the provisions for early analysis with a methodology that caters for property preservation, building on the following constituents:

1. an analysis framework to statically analyse the system;

2. a programming model that enforces the analysis assumptions, permits to express exclusively the semantics imposed by the analysis theory and conveys in the implementation the desired properties;

3. an execution platform that is capable of warranting the properties of interest.

The programming model, which is the subject of this paper, can be realized by adopting a subset of a programming language and, within it, an array of code archetypes whose concurrency semantics conform to the analysis model.

We are interested in the Ravenscar profile (RP) [3], which was one of the outputs of the 8th International Real-Time Ada Workshop (IRTAW). With the Ada 2005 revision [4] it became a standard part of the Ada language. The RP is built on the intent to reject all language constructs that are exposed to unbounded execution time or non-determinism. Its reduced tasking model matches the semantic assumptions and communication model of real-time theory, the response-time analysis strand [5] in particular. For this reason, the RP is an ideal backbone to our programming model.

To complement the RP in the definition of our reference programming model, we need to adopt a set of code archetypes that help us convey in the implementation the desired properties and support the methodological choices that underpin our development process.

The cornerstone of our development methodology is the adoption of a long-known yet often neglected practice, first advocated by Dijkstra in [6]: *separation of concerns*. This principle aims to cleanly separate distinct aspects of software design and implementation, as well as to enable separate reasoning and focused specification.

In our approach, we strive to attain as much separation as possible between functional and extra-functional concerns. At implementation level, we seek separation of the functional/algorithmic code of a "component" (our unit of design), from the code that manages the realization of the extra-functional requirements regarding tasking, synchronization and time-related aspects.

This choice earns us a number of interesting advantages.

Firstly, the separation of the functional part from extra-functional concerns permits to reuse the sequential code of the functional part independently of the realization of the extra-functional concerns that applied when the component was created; this may considerably increase the reuse potential of the software (i.e., the opportunity of its possible reuse under different functional and extra-functional requirements).

Secondly, the separation of sequential algorithmic code from the code that manages all other concerns facilitates timing analysis, as the impact of the sequential code on the worst-case execution time can be more easily and accurately evaluated when isolated from language constructs related to tasking, synchronization and time.

The code archetypes adopted to complete the formulation of the programming model of our interest shall then support this vision, while also striving to fit the needs of systems with the limitation and resource constraints typical of real-time embedded platforms.

One of the results of the ASSERT[1] project was the development of a set of RP-compliant code archetypes adhering to our vision on separation of concerns and amenable to automated code generation. Those archetypes were developed as an evolution of previous work on code generation from HRT-HOOD to Ada (see for example [7] and [8]).

In the CHESS[2] follow-up project, we are revising those code archetypes along two axes: (i) increasing their expressive power, by adding features and lifting some of their current limitations; (ii) revising their structure to better express them with the applicable elements of the latest evolution of Ada.

The rest of the paper is organized as follows. In section 2 we provide a general description on the structure of the code archetypes, as they were finalized for the ASSERT project. We highlight their advantages, disadvantages and limitations. In section 4 we comment on the evolution of these archetypes both as regards their supported feature and improvement from the Ada language standpoint. Finally in section 6 we draw some conclusions.

[1] "Automated proof-based System and Software Engineering for Real-Time systems", FP6 IST-004033 2004-8

[2] "Composition with Guarantees for High-integrity Embedded Software Components Assembly", ARTEMIS JU grant nr. 216682, 2009-2012

2 Structure of the code archetypes

As in ASSERT, our code generation targets the Ada Ravenscar Profile [3] and uses the property-preserving archetypes [9] that we developed at the end of that project.

The task structure that we use as a base for the code mapping approach takes inspiration from HRT-HOOD [10], from which we also inherit the terms OBCS and OPCS.

An important mark of our archetypes is to achieve complete separation between the algorithmic sequential code of the software system and the code that manages concurrency and real-time.

The satisfaction of this goal confirms at implementation level and at run time our claim that it is possible to develop the algorithmic contents of the software (that is, the component behaviour) independently of tasking and real-time issues. The latter are exclusively derived from the extra-functional attributes declaratively specified on component instances and completely managed by the code generated by the design environment.

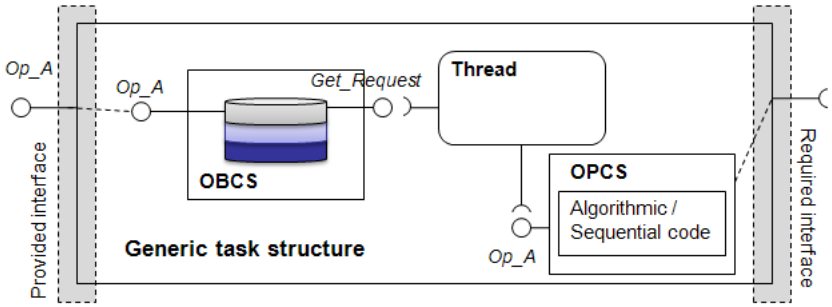


Figure 1: Generic task structure

We achieved the above goal by encapsulating the sequential algorithmic code in a specialized task structure. Figure 1 depicts the generic structure of our task archetypes.

The library of sequential code, which may encompass as many cohesive operations as the designers wishes to attribute to a single executing component, is enclosed in a structure that we term the Operational Control Structure (OPCS). The code in the OPCS library is executed by a Thread, which represents a distinct flow of control of the system. The task structure may be optionally equipped with an Object Control Structure (OBCS). The OBCS represents a synchronization agent for the task; the OBCS is mandatory for sporadic tasks. The OBCS consists of a protected object that stores requests for services issued by clients and destined to be executed by the Thread. The interface exposed by the OBCS to the Thread reflects the *Adapter* pattern [11], which allows the Thread's invocation to the OBCS to stay unchanged regardless of the provided interface exposed by the task structure to the outside. The combined effect of that pattern and the delegation of the OPCS interface to the OBCS makes sure that the task internals are completely hidden from the outside, but the provided services invoked by external clients are executed with the desired concurrency semantics.

As multiple clients may independently require any of a range services to be executed by that Thread, the operations that post execution requests in the OBCS are protected. Upon each release, the Thread fetches one of those requests (with FIFO ordering in the default setting) and then executes the sequential code, stored in the OPCS, which corresponds to the request.

In summary, our archetypal entities hide their internal structure, which realize the concurrency semantics required of them, and expose to the external world solely an interface that matches the signature of the operations embedded in the OPCS. The different concerns dealt with by each such entity are separately allocated to its internal constituents: the sequential behaviour is handled by the OPCS; tasking and execution concerns by the Thread; interaction with concurrent clients and handling of execution requests by the OBCS.

In the following we provide some additional details on the structure of the three constituents of the task structure.

2.1 OPCS

The OPCS is an incarnation of the *Inversion of Control* (IoC) design pattern[3] (a.k.a. *Dependency Injection*), which is a popular practice in lightweight container frameworks in the Java community (e.g. Spring). The bindings used by the OPCS to call operations of its required interface (RI) are not created by the OPCS, but are simply provided with setter operations at system start up. (see line 9 of listing 1 and line 6 of listing 2).

The use of the IoC pattern warrants separation between the configuration of the OPCS (the bindings of the RI and their communication endpoints) and the use of the OPCS; re-configuration of the connected end of the component bindings is thus straightforward, as it only requires to change the binding to the OPCS.

2.2 Thread: periodic release pattern

Our code archetype for cyclic tasks allows the creation of a cyclic Thread by instantiating an Ada generic package, passing the operation that it has to execute periodically (see listings 3 and 4).

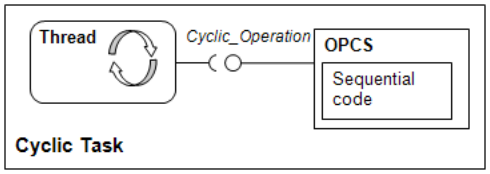


Figure 2: Structure of a cyclic task

The archetype is quite simple. In fact, we only need to create a task type that cyclically executes a given operation with a fixed period.

The specification of the task defines the task type for the cyclic thread. Each thread is instantiated with a statically assigned priority and a period which stays fixed throughout the whole lifetime of the thread. The task type is created inside a generic package, which is used to factorize the code archetype and make it generic on the cyclic operation.

The body of the task is composed by an infinite loop. Just after elaboration, the task enters the loop and is immediately suspended until a system-wide start time (*System_Start_Time*). This initial suspension is used to synchronize all the tasks that are to execute in phase and let them have the first release at the same absolute time.

When resuming from the suspension (which notionally coincides with the release of the task), the task contends for the processor and executes the *Cyclic_Operation* specified at the time of instantiation of its generic package. Then it calculates the next time it has to be released (*Next_Time*) and as first instruction of the subsequent loop, it issues a request for absolute suspension until the next period.

Whereas the code archetype is simple to understand, a few observations are in order.

Firstly, the reader should note that the *Cyclic_Operation* is parameterless. That is not much of a surprise, as it is consistent with the very nature of cyclic operations which are not requested explicitly by any software client. Secondly, this version of the cyclic task assumes that all tasks are initially released at the same time (*System_Start_Time*). Support for a task-specific offset (phase) is easy to implement: we just need to specify an additional *Offset* parameter on task instantiation, which is then added to *System_Start_Time* to determine the time of the first release of the task. The periodic release of the task will then assume the desired phase with respect to the synchronized release of the tasks with no offset.

Finally, we must stress that – as mandated by the Ravenscar Profile – the use of *absolute time* and thus of the construct **delay until** (as opposed to *relative* time and the construct **delay**) is essential to prevent the actual time of the periodic release to drift.

[3] http://www.martinfowler.com/articles/injection.html

2.3 Thread: sporadic release pattern

A sporadic task is a task such that any two subsequent activations of it are always separated by no less but possibly more than a minimum guaranteed time span, known as the MIAT for minimum inter-arrival time.

As in fig.1, our sporadic task is composed of: (i) an OBCS, which external clients use to post their requests for execution; (ii) a thread of control that waits for incoming requests, fetches the first of them from the protected object in the OBCS (according to a given queuing policy, which defaults to FIFO) and executes with sporadic guarantees the requested operation as provided by the OPCS. For the moment we focus on the structure of the Thread. Later we elaborate on the OBCS and the queuing policies for incoming requests.

Listings 5-6 show the code for the sporadic Thread.

The sporadic task enters its infinite loop and suspends itself until the system-wide start time. After that: (i) it calls *Get_Request(Request_Descriptor_T, Time)*, which is an indirection to the single entry of the OBCS; (ii) after the execution of the entry (off which, as we show later on, it obtains a timestamp of when release actually occurred and a request descriptor), the task executes the sporadic operation requested by the client task; (iii) it calculates the next earliest time of release (Next_Time) so as to respect the minimum separation between subsequent activations. At the subsequent iteration of the loop therefore the task issues a request for absolute suspension until that time and thus it will not probe the OBCS for execution requests until the required minimum separation will have elapsed.

In the simplest case, the sporadic task is used to execute a single operation. However, the descriptor of the fetched request can be used to discriminate the operation to perform according to the operation ”type”. This can be done in a case statement (see line 13 in listing 6). In our case, if we fetch a request of type *START_REQ* or *ATC_REQ* we simply execute *My_OPCS*, that will dispatch to the requested target operation. We clarify the dispatching mechanisms as we complete the picture for our task structure by describing the OBCS.

2.4 OBCS and modifiers

The OBCS is the synchronization agent for incoming requests destined for execution by the Thread. The OBCS is also the place where we deal with queueing concerns and we specify the release protocols for the task.

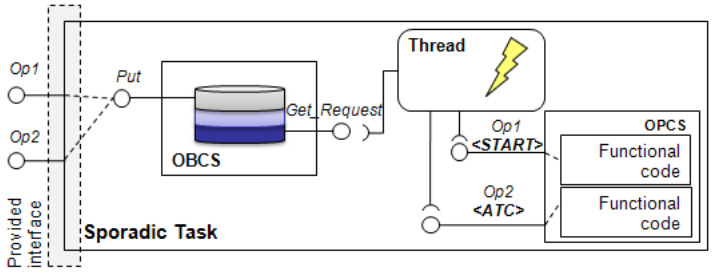


Figure 3: Sporadic Task with modifiers

Suppose that we want to create a sporadic task that at each release can execute either operation *Op1* or operation *Op2*, according to the incoming request issued by clients (see Fig. 3). Gathering different operations into a single cohesive library, to have them executed by the same task is a fairly common need in real-time systems, especially when the execution platform has scarce computational power and memory resources.

Furthermore, we may also want to establish a relative ordering of importance between Op1 and Op2. We consider Op1 as the nominal operation of the sporadic (the operation normally called by clients) and we term it START operation; Op2 is instead an alternative operation that can be called by clients, termed ATC. Then, we stipulate that pending requests for execution of Op1 are served by the sporadic task in FIFO ordering, but requests for Op2 take precedence over pending requests of Op1. Op2 is nicknamed ”modifier” operation, and causes a one-time (as opposed to permanent) modification of the nominal execution behavior of the task.

We encapsulate the implementation of this policy and simply expose to clients of this sporadic task a set of procedures with the signature of Op1 and Op2; the role of these procedures is to reify (akin to serializing,

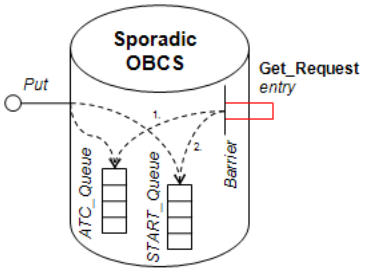


Figure 4: OBCS with dedicated queue for modifiers

only at a higher level of abstraction) the corresponding execution requests. The invocation (type and actual parameters) is recorded in a language-level structure and stored in the OBCS. When the sporadic task fetches a request, it decodes the original request and calls the appropriate operation with the correct parameters (see again line 13 in listing 6).

The code for the request descriptor and the implementation of the queueing policy in the OBCS are included in listings 7 and 8.

2.5 Completing the task structure

To conclude the description of the task structure, we need to complete the definition of the OBCS, and we need to encapsulate the OBCS, Thread and OPCS in a structure that exposes to the clients only the provided interface.

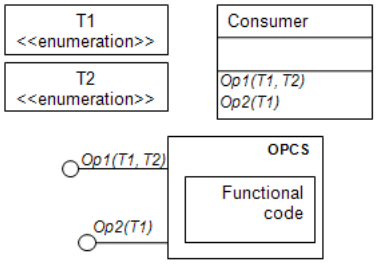


Figure 5: Functional view of the OPCS

Suppose that we want to create a sporadic task for the OPCS of figure, which exposes operation Op1 (nominal) and Op2 (modifier), as specified in Listings 1 and 2.

The complete task structure (called Op1_Op2_Sporadic_Consumer) is shown in Listing 9. It comprises a generic package that exposes to clients exclusively two subprograms with the same signature of Op1 and Op2. Several instances (with different values for the ceiling of the protected object at the OBCS, priority and MIAT) of the same task can then be instantiated.

The private part of the specification contains the OBCS, the declaration of the request descriptors to store and reify inbound requests for execution of Op1 and Op2 and the queues for the descriptors at the OBCS.

In the body, we redefine operation My_OPCS for the request descriptor of Op1 and Op2, so that a call to My_OPCS (in the body of the sporadic Thread, see listing 6) dispatches to the desired OPCS operation.

We instantiate a sporadic Thread for the task (lines 94-95 of listing 10) and set the forwarding to the OBCS of incoming calls to Op1 and Op2 (lines 97-102 of listing 10).

2.6 Catering for property preservation

The above archetypes can, and in fact should, be extended to cater for property preservation.

As we discussed in previous work [12, 9], we want to achieve the following goals:

1. enforcement of the timing properties that are to stay constant during execution;

2. monitoring of the timing properties that are inherently variable in so far as they are influenced or determined by system execution;

3. trapping and handling of the violations of asserted properties.

Goals (2) and (3) can be achieved by use of a simple yet efficient mechanism to monitor the execution time of tasks, to detect WCET overruns and to react in a timely fashion to a violation event: *execution-time timers*, provided by Ada 2005. We refer the reader to [12, 9] where we elaborated specifically on this topic.

3 Discussion

The code archetypes described in the previous sections were used during the ASSERT project, where one industrial partner used them in the (model-based) re-engineering of one reference satellite subsystem of theirs.

The clean separation between the functional/sequential code and the code that manages concurrency and real-time aspects is evident from the task structure: the sequential code is segregated in the OPCS.

The use of Ada generics comes handy to create a reusable archetype structure that is used to instantiate several tasks that are parametric on the operations at the OPCS and real-time parameters (period, MIAT, priority).

Generics are very useful to reduce the size of the source code for the software system and to promote reuse of the same concurrent framework for the construction of real-time systems.

We are also quite satisfied as the current use of generics provides us a flexible mechanism to allow the Thread of the task structure (whose archetype stays fixed) to call operations (at the OPCS) with arbitrarily different signatures.

Unfortunately however the use of generics in our context has important drawbacks too.

First and foremost, they make it harder to perform *timing analysis* [13]. In fact, the Ada generic model permits to use a shared version of the generic body code and at least two Ada compilers take stock of this possibility (GNAT by AdaCore, and – to the best of our knowledge – Janus/Ada by RR Software), and do so with good reasons.

Unfortunately however, when generic instantiations share object code, then for the purposes of timing analysis, each instantiation represents a distinct context of execution. In fact, the analysis needs to discriminate the impact of the different parameters passed for the instantiation of the generic. Consequently, the type of each parameter influences the worst-case execution of the code, to the extent that the WCET may have considerable fluctuations for the range of allowed parameters.

Further, subtler impacts are incurred when the instantiation parameters influence the WCET of loops that are added at the level of object code and that do not have a counterpart at source level; for example, loops added for operations to manipulate arrays. This kind of data-dependent influence is hard to deal with.

The direct consequence is that the state space for the analysis greatly increases, adding inordinate complexity to the problem. In order to mitigate the problem, the designer is normally required to add potentially numerous annotations to guide the analysis, thus increasing even more the burden on the designer who wants to use this kind of analysis.

For the reasons explained above, and in the intent of adding some important ”feature” improvements (which we discuss in the following section), we have resolved to develop an upgrade of our archetypes.

From the Ada language perspective, we want to reduce the amount of generics used in the archetypes and to make use of interfaces in their place. While interfaces were first introduced in Ada 2005 and have therefore been around for quite some time now, they did not have full and satisfactory support in the cross compiler at our disposal during our initial development.

Furthermore, we favour this transition because our primary objective is the creation of code archetypes that are used by *code generation* engines from a model-based design environment. While the use of generics might be a decisive advantage in *code-centric* approaches (thanks to the reduction of source code to be manually written), that benefit is less important in our context.

4 Directions and future work

After some technical discussion with some reference industrial users, we singled out a number of important improvements that our revised archetypes incorporate. The goal of those improvements is to lift some of the current limitations and introduce new capabilities.

The areas we are investigating are:

1. to allow the change of the period or MIAT of a task;
2. to support a more flexible allocation of operations to tasks;
3. to support configurable release protocols at the OBCS;
4. to support the management of the task ”context”;
5. to support software updates at run time;
6. to support different operational modes.

In the following we elaborate on each of them in isolation.

4.1 Allocation of multiple operations to tasks

In our original archetypes we typically allocate the execution of a single operation to a single executor Thread. However, a common need in resource-constrained systems is to contain the proliferation of threads, so as to reduce complexity and memory footprint (primarily, but not exclusively, off task stacks).

The ”modifier” archetype was a tentative (and admittedly partial) solution to this need, especially for sporadic tasks. The queueing policy at the OBCS, which determines how execution requests issued from outside clients are to be served by the executor Thread, was however arbitrarily chosen, without the user being able to specify it, and inflexible. The provision for ”configurable release protocols” (see next paragraph) would complete the archetype, under the condition that it is possible to enforce the MIAT of each sporadic operations.

We would like to complement that provision with the support in our archetypes for the execution of multiple periodic operations by the same task executor, under the condition of retaining their periodic nature. That could be achieved – under the condition that all the grouped operations have the same period or all their periods are harmonic – by introducing appropriate offsets in the release of each single operation.

4.2 Configurable release protocols at the OBCS

User-defined release protocols would offer the user some flexibility in specifying a complex release conditions, in addition to the classical periodic, sporadic and bursty[4] release patterns. We are contemplating to define a small closed language, similar to a state-machine specification, by which the designer can specify (at the level of the architectural description of the software system) the desired release protocol.

The Ravenscar-compliant realization of the release protocol at the OBCS (e.g., maintaining a simple boolean condition for the guard of an entry) would control the release of jobs for that specific task of interest.

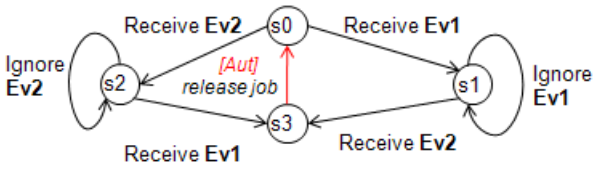


Figure 6: Example of state machine for the release of a new task’s job

In the example of fig. 6, a new job for the task is released after the arrival of event Ev1 and Ev2 (after the arrival of one instance of one of the two event, the protocol ignores other notifications of the same type).

[4]Where we allow a maximum number of job releases within a bounded interval of time.

4.3 Management of the task ”context”

In the current structure of code archetypes there is no explicit support for the management of the task ”context”, i.e., its internal user-level state, as determined by the collection of the values of its typed members, and the bindings with other tasks of its required interfaces.

We ought to introduce facilities for the storage, retrieval and restoration of the task ”context”. This facility is especially important for long-lived and multi-moded software systems, in which one and the same task structure may be used to carry out a range of activities over the lifetime of the system.

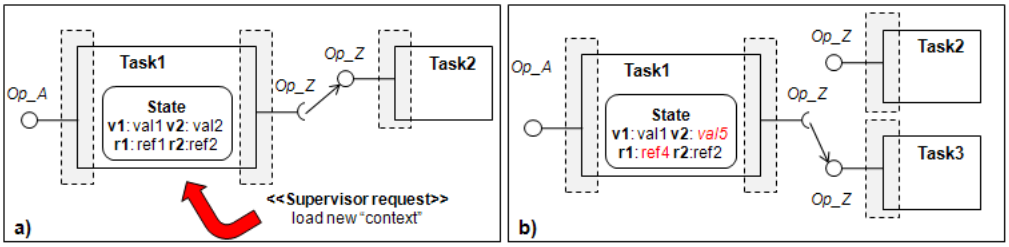


Figure 7: a) A request from a supervisor authority commands the load of a new ”context” for the task. b) The new ”context” contains modified values for internal members and the binding to other tasks

In fig. 7 for example, a supervisory authority requests a task to substitute the values of some of its internal members and its required interface binding. Of course, the use of this facility requires coordination in order that the ”context” update (and similarly, the context ”storage”) be performed when it is safe for the system to execute it.

4.4 Software updates at run time

Support for software updates at run time shall mainly aim at replacing the algorithmic sequential code executed by a Thread. Our strive for separation of concerns facilitates the job, as that portion of the code is cleanly isolated in the OPCS.

Therefore, the replacement of the functional code executed by a Thread is just a matter of finding the most suitable design (with the appropriate language construct) to change the link between the Thread and the OPCS on an as-needed basis, which we currently cannot due to the specific structure of the generic archetype.

Before performing the software update, the task would be prevented from executing by enqueueing it at the OBCS with a blocked guard (see the paragraph on ”Operational modes”) and then updating the link. Additionally, when the Thread is enqueued, we could also directly update the memory region where the functional code resides (assuming of course that we have suitable linker support for it).

4.5 Operational modes and mode change

Finally, as the last item of our extension list, the archetypes shall provide support for multiple operational modes [14]. What would be required is the definition of a framework able to coordinate the mode changes required by single tasks.

The possible actions of interest when changing operational mode are to: (i) change the period or the MIAT of a task of interest; (ii) stop the execution of a task in the new operational mode; (iii) start the execution (of a previously stopped task); (iv) terminate the execution of a task; (v) change the priority of a task.

Item (i) is already in our todo list. Item (ii) can be easily integrated in the archetypes, and achieved by closing the guard of the entry of the task at the OBCS, so that at the end of the execution of the last released job the task is prevented from servicing further requests. For example in listing 10, at line 25, it is easy to modify the Update_Barrier procedure so that the new value for the barrier is set to *false* whenever and as long as the task must be prevented from executing. Item (iii) can be achieved in the same manner, i.e., by re-opening the guard of a task whose thread was permanently enqueued at the entry of its OBCS.

Item (iv) and (v) instead cannot be realized in Ravenscar-compliant systems, as the RP forbids task termination and the use of dynamic priorities for tasks due to the high overhead incurred by those features.

In any case, item (iv) is not fundamental to our partners, as the only need to abruptly terminate the execution of a task would be after a severe software/system failure, an occurrence that is small in probability due to the extensive verification and validation activities performed on the class of systems of our interest, and that most probably would require more drastic remedies (switch to redundant systems or reboot of the computer).

Item (v) would be quite interesting to experiment with fine-grained policies to remedy to timing faults [12]; unfortunately, by adhering to the Ravenscar profile, we have to accept that our mode management framework will be forcedly less expressive than other approaches which leverage on full-fledged Ada (e.g., [15]).

5 Related work

The most known Ada framework for the development of real-time software is by Wellings and Burns [16].

The framework is based on a hierarchy of "tasks states" (realized as tagged types) which are used to specify the information relative to the release pattern of a task (periodic, sporadic and aperiodic). The "task state" is then coupled with a concrete release mechanism and possibly with mechanisms for the detection and reaction to WCET overruns and deadline misses. The different release mechanisms are specified in a hierarchy of synchronized interfaces, and then realized with support of protected objects with entries, timing events and execution-time timers. The framework also offer support for the execution of aperiodic tasks under a server.

The main differences between that framework and ours are the following:

- The framework of Wellings and Burns supports parameter-less sporadic tasks only;
- Their framework is not – and does not intend to be – Ravenscar-compliant (which instead was a goal in ours since its outset), as it uses language constructs that are forbidden by the profile (not considering execution-time timers and group budgets, it nevertheless leverages on protected objects with multiple entries, requeues and "select then abort") and supports task termination;
- Their framework is expression of a task-centric notion typical in real-time systems development, and the sequential code to be executed by the task is embedded in the task structure itself: the user must implement the desired task operation to fully realize the task's tagged type provided by the framework. Our archetypes instead are designed after methodologies (e.g. component-oriented approaches) where the functional (i.e. sequential) code to execute is separately specified (typically in a "component" and possibly by a software supplier) and tasking is a later concern under the responsibility of the software integrator: tasks are just executors of code allocated on them (at software deployment time) and not the basic design entities. This difference is well represented in our separation between the Thread and the OPCS.

6 Conclusions

In this paper we illustrated the structure of a set of Ravenscar-compliant code archetypes suited for the construction of real-time systems.

The archetypes fully support our strive for separation of concerns, for which we want as much as possible separation between the functional/sequential part of the software and the code that deals with concurrency and real-time aspects. Therefore, they complement the Ravenscar Profile in the definition of the programming model that we use in our development process.

We discussed the advantages, the limitations and the drawbacks of those archetypes, which curiously are all centred around our use of Ada generics.

The archetypes were created during the ASSERT project. In a follow-up project CHESS, we are investigating how to evolve these archetypes, to lift their limitations and introduce a number of important advancements as a response to the requests of our reference industrial partners.

Acknowledgements

This work was supported by the CHESS project, ARTEMIS JU grant nr. 216682. The views presented in this paper are however those of the authors' only and do not necessarily engage those of the CHESS partners.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Addison-Wesley Professional, Boston, 2002.

[2] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[3] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs," in *Reliable Software Technologies - Ada Europe*, 1998.

[4] ISO SC22/WG9, "Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1," 2005.

[5] M. Joseph and P. K. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[6] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859 – 866, 1972, ISSN 0001-0782.

[7] J. A. de la Puente, A. Alonso, and A. Alvarez, "Mapping HRT-HOOD Designs to Ada 95 Hierarchical Libraries," in *Proc. of the International Conference on Reliable Software Technologies - Ada-Europe*, 1996, pp. 78–88.

[8] M. Bordin and T. Vardanega, "Automated Model-Based Generation of Ravenscar-Compliant Source Code," in *Proc. of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[9] E. Mezzetti, M. Panunzio, and T. Vardanega, "Preservation of Timing Properties with the Ada Ravenscar Profile," in *Proc. of the 15th International Conference on Reliable Software Technologies - Ada-Europe*, 2010.

[10] A. Burns and A. J. Wellings, *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, 1995.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison - Wesley, 1995.

[12] E. Mezzetti, M. Panunzio, and T. Vardanega, "Temporal Isolation with the Ravenscar Profile and Ada 2005," *Proc. of the 14th International Real-Time Ada Workshop, published on ACM SIGAda Ada Letters*, vol. XXX, no. 1, pp. 45–54, 2010.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G.Bernat, C. Ferdinand, R.Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenström, "The worst-case execution time problem: overview of methods and survey of tools," *IEEE Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[14] J. Real and A. Crespo, "Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal," *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.

[15] ——, "Incorporating Operating Modes to an Ada Real-Time Framework," *Proc. of the 14th IRTAW. SIGAda Ada Letters*, vol. XXX, no. 1, pp. 161–197, 2009.

[16] A. J. Wellings and A. Burns, "Real-Time Utilities for Ada 2005," in *Reliable Software Technologies - Ada-Europe*, 2007, pp. 1–14.

Appendix: Code excerpts

Listing 1: Application of the Inversion of Control pattern in the specification of an OPCS.

```
1  -- in producer.ads
2  package Producer is
3
4     type Producer_FC is new Controlled with private;
5     type Producer_FC_Ref is access all Producer_FC'Class;
6     type Producer_FC_Static_Ref is access all Producer_FC;
7     -- [code omitted]
8
9     -- Called on initialization of the OPCS
10    procedure Set_x(This : in out Producer_FC; c : in Consumer.Consumer_FC_Ref);
11
12    -- Operation of the OPCS
13    procedure Produce(This : in out Producer_FC);
14
15    private
16
17     type Producer_FC is new Controlled with record
18        x : Consumer.Consumer_FC_Ref;
19     end record;
20
21 end Producer;
```

Listing 2: Application of the Inversion of Control pattern in the body of an OPCS.

```
1  -- in producer.adb
2  package body Producer is
3
4     -- [code omitted]
5
6     procedure Set_x(This : in out Producer_FC; c : in Consumer.Consumer_FC_Ref) is
7     begin
8        This.x := c;
9     end Set_x;
10
11    procedure Produce (This : in out Producer_FC) is
12    begin
13       -- do useful stuff
14       This.x.Consume([Parameters]);  -- The actual call is performed on the
15                                      -- "connector" passed upon initialization
16                                      -- of the OPCS
17    end Produce;
18
19 end Producer;
```

Listing 3: Archetype for the cyclic Thread (specification)

```
1  with System; use System;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  generic
5     with procedure Cyclic_Operation;
6  package Cyclic_Task is
7
8     task type Thread_T
9        (Thread_Priority : Priority;
10        Period : Positive) is
11       pragma Priority (Thread_Priority);
12    end Thread_T;
13 end Cyclic_Task;
```

Listing 4: Archetype for the cyclic Thread (body)

```
1  with Ada.Real_Time;
2  with System_Time;
3
4  package body Cyclic_Task is
5
6      task body Thread_T is
7        use Ada.Real_Time;
8        Next_Time : Time := System_Time.System_Start_Time;
9        begin
10          loop
11             delay until Next_Time;
12             Cyclic_Operation;
13             Next_Time := Next_Time + Milliseconds (Period);
14          end loop;
15    end Thread_T;
16 end Cyclic_Task;
```

Listing 5: Sporadic Thread (spec).

```
1  with System_Types; use System_Types;
2  with System; use System;
3  with Ada.Real_Time; use Ada.Real_Time;
4
5  generic
6     with procedure Get_Request(Req : out Request_Descriptor_T;
7                                Release : out Time);
8  package Sporadic_Task is
9
10    task type Thread_T (Thread_Priority : Any_Priority;
11                        MIAT : Integer) is
12       pragma Priority (Thread_Priority);
13    end Thread_T;
14 end Sporadic_Task;
```

Listing 6: Sporadic Thread (body).

```
1  with System_Time; use System_Time;
2  package body Sporadic_Task is
3
4     task body Thread_T is
5        Req_Desc : Request_Descriptor_T;
6        Release : Time;
7        Next_Time : Time := System_Start_Time;
8     begin
9        loop
10          delay until Next_Time;
11          Get_Request (Req_Desc, Release);
12          Next_Time := Release + Milliseconds (MIAT);
13          case Req_Desc.Request is
14             when NO_REQ => null;
15             when START_REQ | ATC_REQ =>
16                      My_OPCS (Req_Desc.Params.all);
17             when others => null;
18          end case;
19       end loop;
20    end Thread_T;
21 end Sporadic_Task;
```

Listing 7: System types (spec).

```
1  with System;
2  with Ada.Real_Time; use Ada.Real_Time;
3  with System_Time;
4  with Ada.Finalization; use Ada.Finalization;
5
6  package System_Types is
7
8     -- Abstract Parameter type --
9     type Param_Type is abstract tagged record
10       In_Use : Boolean := False;
11    end record;
12    -- Abstract functional procedure --
13    procedure My_OPCS (Self : in out Param_Type) is abstract;
14
15    type Param_Type_Ref is access all Param_Type'Class;
16    type Param_Arr is array(Integer range <>) of Param_Type_Ref;
17    type Param_Arr_Ref is access all Param_Arr;
18    -- Request type --
19    type Request_T is (NO_REQ, START_REQ, ATC_REQ);
20
21    -- Request descriptor to reify an execution request
22    type Request_Descriptor_T is
23    record
24       Request : Request_T;
25       Params : Param_Type_Ref;
26    end record;
27
28    type Param_Buffer_T(Size : Integer) is
29    record
30       Buffer : aliased Param_Arr(1..Size);
31       Index : Integer := 1;
32    end record;
33
34    type Param_Buffer_Ref is access all Param_Buffer_T;
35    procedure Increase_Index(Self : in out Param_Buffer_T);
36
37    -- Abstract OBCS --
38    type OBCS_T is abstract new Controlled with null record;
39    type OBCS_T_Ref is access all OBCS_T'Class;
40
41    procedure Put(Self : in out OBCS_T; Req : Request_T; P : Param_Type_Ref)
42       is abstract;
43
44    procedure Get(Self : in out OBCS_T; R : out Request_Descriptor_T)
45       is abstract;
46
47    -- Sporadic OBCS --
48    type Sporadic_OBCS(Size : Integer) is new OBCS_T with
49    record
50       START_Param_Buffer : Param_Arr(1..Size);
51       START_Insert_Index : Integer;
52       START_Extract_Index : Integer;
53       START_Pending : Integer;
54       ATC_Param_Buffer : Param_Arr(1..Size);
55       ATC_Insert_Index : Integer;
56       ATC_Extract_Index : Integer;
57       ATC_Pending : Integer;
58       Pending : Integer;
59    end record;
60
61    overriding
62    procedure Initialize(Self : in out Sporadic_OBCS);
63
```

```
64      overriding
65      procedure Put(Self : in out Sporadic_OBCS; Req : Request_T; P : Param_Type_Ref);
66
67      overriding
68      procedure Get(Self : in out Sporadic_OBCS; R : out Request_Descriptor_T);
69
70      -- Parameter buffer
71      type Param_Buffer_T(Size : Integer) is
72         record
73            Buffer : aliased Param_Arr(1..Size);
74            Index : Integer := 1;
75         end record;
76
77      type Param_Buffer_Ref is access all Param_Buffer_T;
78
79      procedure Increase_Index(Self : in out Param_Buffer_T);
80
81   end System_Types;
```

Listing 8: System types (body).

```
1   package body System_Types is
2
3      -- Sporadic OBCS --
4      procedure Initialize (Self : in out Sporadic_OBCS) is
5      begin
6         Self.START_Pending        := 0;
7         Self.START_Insert_Index   := Self.START_Param_Buffer'First;
8         Self.START_Extract_Index  := Self.START_Param_Buffer'First;
9         Self.ATC_Pending          := 0;
10        Self.ATC_Insert_Index     := Self.ATC_Param_Buffer'First;
11        Self.ATC_Extract_Index    := Self.ATC_Param_Buffer'First;
12     end Initialize;
13
14     procedure Put(Self : in out Sporadic_OBCS; Req : Request_T; P : Param_Type_Ref) is
15     begin
16        case Req is
17           when START_REQ =>
18              Self.START_Param_Buffer (Self.START_Insert_Index) := P;
19              Self.START_Insert_Index := Self.START_Insert_Index + 1;
20              if Self.START_Insert_Index > Self.START_Param_Buffer'Last then
21                 Self.START_Insert_Index := Self.START_Param_Buffer'First;
22              end if;
23              --increase the number of pending requests but do not
24              --overcome the number of buffered ones
25              if Self.START_Pending < Self.START_Param_Buffer'Last then
26                 Self.START_Pending := Self.START_Pending + 1;
27              end if;
28           when ATC_REQ =>
29              Self.ATC_Param_Buffer (Self.ATC_Insert_Index) := P;
30              Self.ATC_Insert_Index := Self.ATC_Insert_Index + 1;
31              if Self.ATC_Insert_Index > Self.ATC_Param_Buffer'Last then
32                 Self.ATC_Insert_Index := Self.ATC_Param_Buffer'First;
33              end if;
34
35              if Self.ATC_Pending < Self.ATC_Param_Buffer'Last then
36                 --increase the number of pending requests but do not
37                 --overcome the number of buffered ones
38                 Self.ATC_Pending := Self.ATC_Pending + 1;
39              end if;
40
41           when others => null;
42        end case;
```

```
43          Self.Pending := Self.START_Pending + Self.ATC_Pending;
44       end Put;
45
46       procedure Get(Self : in out Sporadic_OBCS; R : out Request_Descriptor_T) is
47       begin
48          if Self.ATC_Pending > 0 then
49             R := (ATC_REQ, Self.ATC_Param_Buffer(Self.ATC_Extract_Index));
50             Self.ATC_Extract_Index := Self.ATC_Extract_Index + 1;
51             if Self.ATC_Extract_Index > Self.ATC_Param_Buffer'Last then
52                Self.ATC_Extract_Index := Self.ATC_Param_Buffer'First;
53             end if;
54             Self.ATC_Pending := Self.ATC_Pending - 1;
55             else
56                if Self.START_Pending > 0 then
57                   R := (START_REQ, Self.START_Param_Buffer(Self.START_Extract_Index));
58                   Self.START_Extract_Index := Self.START_Extract_Index + 1;
59                   if Self.START_Extract_Index > Self.START_Param_Buffer'Last then
60                      Self.START_Extract_Index := Self.START_Param_Buffer'First;
61                   end if;
62                   Self.START_Pending := Self.START_Pending - 1;
63                end if;
64          end if;
65          R.Params.In_Use := True;
66          Self.Pending := Self.START_Pending + Self.ATC_Pending;
67       end Get;
68
69       procedure Increase_Index(Self : in out Param_Buffer_T) is
70       begin
71          Self.Index := Self.Index + 1;
72          if Self.Index > Self.Buffer'Last then
73             Self.Index := Self.Buffer'First;
74          end if;
75       end Increase_Index;
76    end System_Types;
```

Listing 9: Complete task structure (spec).

```
1  with System;
2  with Types;
3  with System_Types;
4  with Ada.Real_Time;
5  with Consumer;
6  with Ada.Real_Time; use Ada.Real_Time;
7
8  package Op1_Op2_Sporadic_Consumer is
9
10    use System; use Types;
11
12    use System_Types;
13
14    -- generic child package to instantiate a sporadic task
15    generic
16       Thread_Priority : Priority;
17       Ceiling : Priority;
18       MIAT : Integer;
19       --The OPCS instance
20       OPCS_Instance : Consumer.Consumer_FC_Static_Ref;
21    package My_Sporadic_Factory is
22
23       procedure Op1(a : in T1; b : in T2);
24       procedure Op2(a : in T1);
25     private
26
27    end My_Sporadic_Factory;
28
29    private
30
31     Param_Queue_Size : constant Integer := 3;
32     OBCS_Queue_Size : constant Integer := Param_Queue_Size * 2;
33
34     --create data structures to reify invocations to Op1
35     type Op1_Param_T is new Param_Type with record
36        OPCS_Instance : Consumer.Consumer_FC_Static_Ref;
37        a : T1;
38        b : T2;
39     end record;
40     type Op1_Param_T_Ref is access all Op1_Param_T;
41
42     type Op1_Param_Arr is array(Integer range <>) of aliased Op1_Param_T;
43
44     overriding
45     procedure My_OPCS(Self : in out Op1_Param_T);
46
47     --create data structures to reify invocations to Op2
48      type Op2_Param_T is new Param_Type with record
49        OPCS_Instance : Consumer.Consumer_FC_Static_Ref;
50        a : T1;
51     end record;
52     type Op2_Param_T_Ref is access all Op2_Param_T;
53
54     type Op2_Param_Arr is array(Integer range <>) of aliased Op2_Param_T;
55
56     overriding
57     procedure My_OPCS(Self : in out Op2_Param_T);
58
59     -- create an OBCS that match the interface of the OPCS (FC)
60     protected type OBCS(Ceiling : Priority;
61       Op1_Params_Arr_Ref_P : Param_Arr_Ref;
62       Op2_Params_Arr_Ref_P : Param_Arr_Ref) is
63        pragma Priority(Ceiling);
```

```
64          entry Get_Request(Req : out Request_Descriptor_T; Release : out Time);
65          procedure Op2(a : in T1);
66          procedure Op1(a : in T1; b : in T2);
67       private
68          -- The queue system for the OBCS
69          OBCS_Queue : Sporadic_OBCS(OBCS_Queue_Size);
70          -- Arrays to store a set of reified invocations for Op1 and Op2
71          Op1_Params : Param_Buffer_T(Param_Queue_Size) :=
72             (Size => Param_Queue_Size, Index => 1, Buffer => Op1_Params_Arr_Ref_P.all);
73          Op2_Params : Param_Buffer_T(Param_Queue_Size) :=
74             (Size => Param_Queue_Size, Index => 1, Buffer => Op2_Params_Arr_Ref_P.all);
75          Pending : Standard.Boolean := False;
76       end OBCS;
77    end Op1_Op2_Sporadic_Consumer;
```

Listing 10: Complete task structure (body).

```
1  with Ada.Real_Time; use Ada.Real_Time;
2  with Sporadic_Task;
3  with Types; use Types;
4
5  package body Op1_Op2_Sporadic_Consumer is
6
7     use System_Types;
8     -- Redefinition of My_OPCS. Call Consumer_FC.Op1 and set In_Use to false.
9     procedure My_OPCS(Self : in out Op1_Param_T) is
10    begin
11       Self.OPCS_Instance.Op1(Self.a, Self.b);
12       Self.In_Use := False;
13    end My_OPCS;
14
15    -- Redefinition of My_OPCS. Call Consumer_FC.Op2 and set In_Use to false.
16    procedure My_OPCS(Self : in out Op2_Param_T) is
17    begin
18       Self.OPCS_Instance.Op2(Self.a);
19       Self.In_Use := False;
20    end My_OPCS;
21
22    protected body OBCS is
23       procedure Update_Barrier is
24       begin
25          Pending := (OBCS_Queue.Pending) > 0;
26       end Update_Barrier;
27       -- Get_Request stores the time of the release of the task,
28       -- gets the next request (according to the OBCS queuing policy),
29       -- and updates the guard.
30       entry Get_Request (Req : out Request_Descriptor_T; Release : out Time)
31             when Pending is
32       begin
33          Release := Clock;
34          Get(OBCS_Queue, Req);
35          Update_Barrier;
36       end Get_Request;
37
38       -- When a client calls Op1, the request is reified and put in the OBCS queue.
39       procedure Op1(a : in T1; b : in T2) is
40       begin
41         if Op1_Params.Buffer(Op1_Params.Index).In_Use then
42            Increase_Index(Op1_Params);
43         end if;
44
45         Op1_Param_T_Ref(Op1_Params.Buffer(Op1_Params.Index)).a := a;
46         Op1_Param_T_Ref(Op1_Params.Buffer(Op1_Params.Index)).b := b;
```

```
47            Put(OBCS_Queue, START_REQ, Op1_Params.Buffer(Op1_Params.Index));
48            Increase_Index(Op1_Params);
49            Update_Barrier;
50         end Op1;
51
52         -- When a client calls Op2, the request is reified and put in the OBCS queue.
53         procedure Op2(a : in T1) is
54         begin
55            if Op2_Params.Buffer(Op2_Params.Index).In_Use then
56               Increase_Index(Op2_Params);
57            end if;
58
59            Op2_Param_T_Ref(Op2_Params.Buffer(Op2_Params.Index)).a := a;
60            Put(OBCS_Queue, ATC_REQ, Op2_Params.Buffer(Op2_Params.Index));
61            Increase_Index(Op2_Params);
62            Update_Barrier;
63         end Op2;
64      end OBCS;
65
66      package body My_Sporadic_Factory is
67
68         Op1_Par_Arr : Op1_Param_Arr(1..Param_Queue_Size) := (others =>
69                        (false,
70                         OPCS_Instance,
71                         T1_Default_Value,
72                         T2_Default_Value));
73
74         Op1_Ref_Par_Arr : aliased Param_Arr := (Op1_Par_Arr(1)'access,
75             Op1_Par_Arr(2)'access, Op1_Par_Arr(3)'access);
76
77         Op2_Par_Arr : Op2_Param_Arr(1..Param_Queue_Size) := (others =>
78                       (false,
79                        OPCS_Instance,
80                        T1_Default_Value));
81
82         Op2_Ref_Par_Arr : aliased Param_Arr := (Op2_Par_Arr(1)'access,
83              Op2_Par_Arr(2)'access, Op2_Par_Arr(3)'access);
84
85         -- Creation of the OBCS
86         Protocol : aliased OBCS(Ceiling, Op1_Ref_Par_Arr'access,
87                                              Op2_Ref_Par_Arr'access);
88         -- Indirection to Get_Request of the OBCS
89         procedure Getter(Req : out Request_Descriptor_T; Release : out Time) is
90         begin
91            Protocol.Get_Request(Req, Release);
92         end Getter;
93
94         -- Instantiate the generic package using the procedure above
95         package My_Sporadic_Task is new Sporadic_Task(Getter);
96
97         Thread : My_Sporadic_Task.Thread_T(Thread_Priority, MIAT);
98         -- When a client calls Op1, redirect the call to the OBCS
99         procedure Op1(a : in T1; b : in T2) is
100        begin
101           Protocol.Op1(a, b);
102        end Op1;
103
104        -- When a client calls Op2, redirect the call to the OBCS
105        procedure Op2(a : in T1) is
106        begin
107           Protocol.Op2(a);
108        end Op2;
109     end My_Sporadic_Factory;
110  end Op1_Op2_Sporadic_Consumer;
```

Revisiting Transactions in Ada

António Barros and Luís Miguel Pinho
CISTER/ISEP
Polytechnic Institute of Porto
Porto, Portugal
{amb,lmp}@isep.ipp.pt

Abstract

Classical lock-based concurrency control does not scale with current and foreseen multi-core architectures, opening space for alternative concurrency control mechanisms. The concept of transactions executing concurrently in isolation with an underlying mechanism maintaining a consistent system state was already explored in fault-tolerant and distributed systems, and is currently being explored by transactional memory, this time being used to manage concurrent memory access. In this paper we discuss the use of Software Transactional Memory (STM), and how Ada can provide support for it. Furthermore, we draft a general programming interface to transactional memory, supporting future implementations of STM oriented to real-time systems.

1 Introduction

Current architectures based on multiple processor cores in a single chip are becoming widespread, and are challenging our ability to develop concurrent and parallel software. The tendency to integrate even larger number of cores will further impact the way systems are developed, as software performance can no longer rely on faster processors but instead on efficient techniques to design and execute parallel software. It is also important to note that developing scalable and safe concurrent code for such architectures requires synchronisation control mechanisms that are able to cope with an increasing number of parallel tasks, providing the necessary building blocks for modular, concurrent, and composable software.

In uniprocessor systems, lock-based synchronisation became the *de facto* solution to avoid race conditions, despite the well-known pitfalls, such as complexity, lack of composability [1] or (bounded) priority inversion. In multiprocessor systems, lock-based synchronisation becomes more problematic. Coarse-grained locks serialise non-conflicting operations (which could actually progress in parallel) on disjoint parts of a shared resource, and may cause cascading or convoying blocks [2], wasting the parallel execution opportunities provided by such architectures. Fine-grained locks increase the complexity of system design, affecting composability seriously, and produce an increasing burden for the programmer. In multiprocessors, non-blocking approaches present strong conceptual advantages [3] and have been shown in several cases to perform better than lock-based ones [4].

Transactional memory is a concept that has been researched in parallel systems for nearly two decades. Although the first proposal of transactional memory was hardware-based [5], it was soon followed by a software-based adaptation [6]. Software Transactional Memory has the advantage of being easily reconfigurable and to enable transactional support on architectures that do not have native support for atomic operations. Currently, STM research is well ahead comparatively with hardware-support for transactions.

Under the transactional memory paradigm, critical sections are executed optimistically in parallel while an underlying mechanism maintains the consistency of shared data. Data is kept consistent by serialising critical sections, that is, the outcome is as if critical sections were executed atomically, in sequence. Concurrent critical sections may attempt to perform conflicting data accesses; such conflicts are usually solved by selecting the critical section that will conclude, and aborting or delaying the contenders. Due to this similarity with database transactions, in which an atomic sequence of operations can either commit or abort, critical sections are called *transactions*.

Transactional Memory promises to ease concurrent programming: the programmer must indicate which code belongs to a transaction and leaves the burden of synchronisation details on the underlying STM mechanism to conserve the consistency of shared transactional data. This approach has proved to scale well with multiprocessors [7], delivers higher throughput than coarse-grained locks and does not increase design complexity as fine-grained locks do [8].

The advantages of STM are counterbalanced with increased times in memory accesses, memory utilisation overhead and speculative execution of code. The latter is, in fact, a challenge that has to be addressed if STM is to be applied to real-time systems, because it is not admissible that one transaction can be unboundedly aborted and forced to repeat. Some work on the field of STM on multiprocessor real-time systems was already published [9, 10, 11, 12].

Several implementations of STM were implemented for programming languages such as java, C++, C# and Haskell, but currently there is no implementation for Ada. It is thus in this context that this paper revisits previous work on transactions support in Ada, for fault-tolerant systems, and proposes a programming interface to support Software Transactional Memory. A goal is for this interface to be independent of an eventual STM implementation, so programs can, for instance, change the contention mechanism (the algorithm that manages conflicting operations), according to application specific characteristics.

The paper is structured as follows. Section 2 revisits previous work on transaction support in Ada for fault-tolerant and distributed systems. Afterwards, an introduction to the STM essential issues is given in Section 3. In Section 4, we propose a general programming interface to access STM functionality. This paper terminates with conclusions and perspectives for further work in Section 5.

2 Previous work on transactions in Ada

Transaction support in Ada has been a subject of research in the field of fault-tolerant systems. The concept of a transaction grouping a set of operations that appear to be executed atomically (with respect to other concurrent transactions) if it had success, or having no effect whatsoever on the state of the system if aborted, is quite appealing as a concurrent control mechanism for fault-tolerant and/or distributed systems. The ability to abort a transaction due to data access contention or to an unexpected error and, consequently, rolling back any state modifications, automatically preserves the system in a safe and consistent state. Safe consistent states can be stored in a durable medium, so they might be available even after a crash.

The loose parallelism provided by the isolation property of the transactional paradigm is appealing for systems based on multiple processors (either clustered or distributed) and the inherent backward recoverability mechanisms suit fault-tolerant concerns.

Two paradigmatic implementations of transaction support in Ada are the Transactional Drago [13, 14] and the OPTIMA framework [15].

Both proposals share many common attributes, aiming to support competitive concurrency (between transactions) and cooperative concurrency (inside a transaction). The two provide the essential interface to start, commit and abort transactions. Transactions can be multithreaded, i.e. multiple tasks can work on behalf of a single transaction. Both implementations support nested transactions and exception handling.

Despite the similarities, both implementations take different approaches.

Transactional Drago is an extension to the Ada language, so it can only be used with compilers that include this extension. Transactions are defined using the transactional block, an extension that resembles an Ada block statement, but identified with the keyword `transaction` [14]. The transactional block creates a new scope in which data, tasks and nested transactions can be defined. Data declared inside a transactional block is volatile and subject to concurrency control. Tasks inside a transactional block work cooperatively on behalf of the transaction and their results will dictate the outcome of the transaction.

The transactional block provides a clean syntax, defining clearly the limits of the transaction, without the need for an explicit abort statement. Aborts are triggered by the raising of unhandled exceptions from within the transactional block. The following code sample illustrates a transactional block.

```
transaction
  declare
    -- data declared here is subject to concurrency control
  begin
```

```
    -- sequence of statements
    -- can include tasks that work on behalf of transaction
    -- can include nested transactions
  exception
    -- handle possible exceptions here...
end transaction;
```

The OPTIMA framework is provided as a library, which does not modify the language. In this framework, a transaction is created with the command `Begin_Transacion` and, depending on the result of the computation, ends with either `Commit_Transaction` or `Abort_Transaction`. The OPTIMA framework supports open multithreaded transactions, so the additional command `Join_Transaction` allows one task to join and cooperate on an ongoing transaction. When a task calls `Begin_Transacion` or `Join_Transaction`, it becomes linked to the transaction via the `Ada.Task_Attributes` standard library unit. From that moment on, the transaction support is able to determine the transaction context for the task. The following code sample illustrates a task that starts a transaction.

```
begin
  Begin_Transaction;
  -- perform work
  Commit_Transaction;
exception
  when ...
    -- handle recoverable exceptions here...
    Commit_Transaction;
  when others =>
    Abort_Transaction;
    raise;
end;
```

This example shows how the use of the exceptions mechanism in this framework permits to define handlers for foreseen exceptional cases, thus allowing forward recovery in such cases. However, unexpected exceptions will abort the transaction, like in Transactional Drago.

3 Lightweight memory transactions

STM shares the basic principles of transactions from databases and fault-tolerant systems: transactions can execute speculatively in parallel, but their effects should appear as if they executed atomically in sequence. However, the field of application has particular characteristics. Transactions are expected to be the size of typical critical sections, *i.e.* as short as possible. In the STM perspective, the transaction must conclude its work, so even if it aborts, the transaction should try again the necessary number of times until it is allowed to commit.

The STM mechanism keeps track of accesses to transactional objects that are exclusively memory locations (words or structures). Since memory accesses to transactional objects become indirect accesses, the STM mechanism must be light enough to avoid performance issues.

Unlike the previous work oriented to fault-tolerant systems, STM does not intend to store consistent states in a persistent medium, but simply to keep data in memory consistent. Furthermore, the multithreaded transaction concept does not apply: a transaction belongs exclusively to a task, *i.e.* it is part of the sequential code of the task.

In comparison with database transactions and fault-tolerant transactions, transactional memory transactions can be considered lightweight memory transactions [16].

Transactions can be divided in two classes, according to the transactional memory access pattern:

- **read-only transactions**, in which the transaction does not try to modify any transactional object, and
- **update transactions**, in which the transaction tries to modify at least one transactional object.

A conflict may occur when two or more transactions concurrently access one object and at least one of the accesses tries to modify the object: if the updating transaction commits before the contenders, the contenders will be working with outdated data and should abort.

Conflicts are typically solved by selecting one transaction that will commit and aborting the contenders. The performance of a STM is therefore dependent on the frequency of contention, which has direct effect on the transaction abort ratio. Thus, STM behaves very well in systems exhibiting a predominance of read-only transactions, short-running transactions and a low ratio of context switching during the execution of a transaction [17].

STM implementations may differ in multiple ways, namely [18]:

- **version management** – how tentative writes are performed;
- **conflict detection** – when conflicts are detected;
- **granularity of conflicts** – word, cache-line or object granularity.

Eager version management allows the object to be immediately modified, but requires that the object is unavailable to concurrent transactions until the updating transaction commits. If the updating transaction aborts, the value of the object is rolled-back to the previous version. *Lazy version management* defers the update of the object until the transaction commits, leaving the object available to other concurrent transactions. This implies that a transaction must work with an image of the object that will eventually replace the value of the transactional object.

Conflicts can be detected immediately, under *eager conflict detection*, or deferred until one transaction tries to commit under *lazy conflict detection*. Eager conflict detection inhibits a transaction to continue once it faces a conflict that will not win. Lazy conflict detection permits transactions to execute in total isolation and only when a transaction tries to commit, conflicts are detected and solved. Eager conflict detection better serves write-write conflicts, since one of the transactions will inevitably abort, but lazy conflict detection can be more efficient with read-write conflicts, as long as read-only transactions commit before update transactions [19].

The granularity of conflict detection determines the possibility of false conflict detection. Finer granularity means less false conflicts detected and lower transaction abort ratio, but at the expense of higher memory overheads.

Regardless which attributes are selected for an actual STM implementation, transactions will eventually be aborted, and some transactions may present characteristics (*e.g.* long running, low priority) that can potentially lead to starvation. In parallel systems literature, the main concern about STM is on system throughput, and the contention management policy has often the role to prevent livelock (a pair of transactions indefinitely aborting each other) and starvation (one transaction being constantly aborted by the contenders), so that each transaction will eventually conclude and the system will progress as a whole. In real-time systems, the guarantee that a transaction will *eventually* conclude is not sufficient to ensure the timing requirements that are critical to such type of systems: the *maximum time to commit* must be known. The verification of the schedulability of the task set requires that the WCET of each task is known, which can only be calculated if the maximum time used to commit the included transaction is known. As such, STM can be used in real-time systems as long as the employed contention management policy provides guarantees on the maximum number of retries associated with each transaction.

Recently, we have proposed new approaches to manage contention between conflicting transactions, using on-line information, with the purpose of reducing the overall number of retries, increasing responsiveness and reducing wasted processor utilization, while assuring deadlines are met [12]. With our proposed policy, conflicting transactions will commit according to the chronological order of arrival, except if an older contender is currently pre-empted. This approach is fair in the sense that no transaction will be chronically discriminated due to some innate characteristic. But most importantly, this approach is predictable, because the time overhead taken by a transaction until commit depends solely on the ongoing transactions at the moment the transaction arrives, being independent of future arrivals of other transactions, except for conflicting transactions executing in the same processor that arrive after the job being pre-empted by another job with higher urgency.

4 Providing STM support in Ada

Essential support to STM can be implemented in a library, without introducing modifications in the Ada programming language, easing the portability of an STM service, without the need to modify compilers and debuggers. The drawback is that the programmer must adhere to a somewhat less clean syntax.

The following code illustrates how a very simple transaction should be written.

```
-- we need a transaction identifier structure
My_Transaction : Transaction;

-- start an update transaction
My_Transaction.Start(Update);

loop
  -- read a value from a transactional object
  x := trans_object_1.Get(My_Transaction);

  -- write a value to a transactional object
  trans_object_2.Set(My_Transaction, y);

  -- try to commit transaction
  exit when My_Transaction.Commit;

exception
  -- handle possible exceptions here...
end loop;
```

This example shows how the initialisation of the transaction and the retry-until-commit loop have to be explicitly written.

Our STM perspective requires two key classes of objects: the *transactional object* and the *transaction identifier*.

The transactional object encapsulates a data structure with the transactional functionality. For instance, a write operation will not effectively modify the value of the object if the STM applies lazy version management.

The transaction identifier is a structure that stores the data required by the contention manager to apply the conflict solving policy chosen for the system.

4.1 Transactional object

A transactional object is a type of class that wraps a classical data structure with the transactional functionality. The interface provided is similar to the non-transactional version, but adds the operations required to maintain the consistency of the object, according to the implementation details of the STM.

Thus, for every access, the identification of the transaction is required, either to locate the transaction holding the object (case of eager version management) or track all transactions referring the object (case of lazy version management). In each case, the object must locate one or all accessing transactions, respectively, so under contention, transactions attributes are used to determine which transaction is allowed to proceed, according to the contention management policy.

Reading accesses can also be tailored for read-only transactions, if a multi-version STM is in use. Transparently, the transactional object can return the latest version to an update transaction, or a consistent previous version to a read-only transaction.

```
-- Transactional object
package Transactional_Objects is
  type Transactional_Object is tagged private;
  -- examples of transactional class methods
  procedure Set(T_Object: Transactional_Object;
                Transaction_ID : Transaction;
                Value : Object_Type);
  function Get(T_Object: Transactional_Object;
               Transaction_ID : Transaction)
           return Object_Type;
private
  type Transactional_Object is tagged
  record
```

```
    Current_Value : Object_Type;
    Accesses_Set : List_of_References_to Transaction_Identifiers;
    -- some other relevant fields...
  end record;
end Transactional_Objects;
```

4.2 Transaction identifier

The transaction identifier provides the transactional services to a task, uniquely identifying a transaction. The essential interface of this class should provide the `Start`, `Commit` and `Abort` operations, and keep track of the accessed objects.

```
type Transaction_Type is (Read_Only, Update);

-- Transaction identifier
package Transactions is
  type Transaction is tagged private;
  procedure Start(T : Transaction;
                  TRX_Type : Transaction_Type);
  procedure Abort(T : Transaction);
  procedure Terminate(T : Transaction);
  function Commit(T : Transaction)
           return Boolean;

private
  type Transaction is tagged
  record
    Data_Set : List_of_References_to_Transactional_Objects;
end record;
end Transactions;
```

The `Start` procedure initialises the transaction environment. Starting an already active transaction is not allowed, and an exception should be raised.

The `Abort` procedure erases any possible effects of the transaction, but the transaction remains active and is allowed to undertake further execution attempts. Aborting an inactive transaction is not allowed, and an exception should be raised.

The `Terminate` procedure cancels the transaction, leaving the transaction inactive. Terminating an inactive transaction is not allowed, and an exception should be raised.

The last operation provided by this interface is the `Commit` function that validates accessed data and resolves possible conflicts. If the transaction is allowed to commit its updates, then this function will return the `True` value. Commiting an inactive transaction is not allowed, and an exception should be raised.

This class also stores the references to the transactional objects that were accessed in the context of the transaction. These data are required when trying to commit, to validate read and modification locations.

Specific STM implementations will, most likely, require modified operation functionality and additional attributes. For example, some STM algorithms require to know the instant the transaction started, the current status of the transaction [12], or the instant the current execution of the transaction began [20, 21, 22].

These attributes can be included in extensions of this class, deriving a new class for each implementation, as the following example illustrates.

```
type Transaction_Status is (Active,
                            Preempted,
                            Zombie,
                            Validating,
                            Committed);
```

```
-- Transaction identifier
package Transactions_Foo_STM is
  type Transaction_Foo_STM is new Transaction with private;
  overriding
  function Commit(T : Transaction_Foo_STM) return Boolean;

private
  type Transaction_Foo_STM is new Transaction with
  record
    -- implementation specific elements
    -- some examples below
    Type_of_Accesses : Transaction_Type;
    Time_Started : STM_Time;
    Time_Current_Begin : STM_Time;
    Status : Transaction_Status;
    -- some other relevant fields...
  end record;
end Transactions_Foo_STM;
```

Our current approach to STM assumes that a transaction is not able to abort ongoing concurrent transactions, as this could be very costly for the implementation. However, we intend to evaluate this in future work, and if considered feasible we will also evaluate the usefulness of the Asynchronous Transfer of Control (ATC) feature of the language to detect the request to abort a transaction and execute the roll-back operations.

5 Conclusions

Current and foreseen multi-core architectures have raised performance issues to classical concurrency control based on locks: either coarse-grained locking impairs parallelism or fine-grained locking increases the difficulty to safely develop concurrent software.

Transactions were already considered as a concurrency control mechanism able to maintain the consistency of the system's state in which parallel transactions could abort due to data contention or hardware/software errors. Currently, the same concept is being applied to manage concurrent access to shared data, known as transactional memory.

In this paper, we discuss the use of software transactional memory and we draft a common programming interface to STM in Ada. This interface is independent of a particular STM implementation, so different implementations can address different utilization patterns. It is also the goal of this work to allow the research on contention mechanisms for software transactional memory in real-time systems. Future work will address new contention mechanisms for these systems and evolve the STM support in Ada (*e.g.* using generics or Ada 2012 aspects as wrappers, and reflecting the real-time issues in the Ada proposal).

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments to improve the paper.

This work was supported by the VIPCORE project, ref. FCOMP-01-0124-FEDER-015006, funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness') and by National Funds (PT) through FCT - Portuguese Foundation for Science and Technology, and the ARTISTDESIGN - Network of Excellence on Embedded Systems Design, grant ref. ICT-FP7-214373.

References

[1] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[2] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems - ICDCS 1993*, pages 264–273, May 1993.

[3] Philippas Tsigas and Yi Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications - RTCSA'99*, pages 247–254, December 1999.

[4] Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium - RTAS '08*, pages 342–353, April 2008.

[5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th anual International Symposium on Computer Architecture - ISCA '93*, pages 289–300, May 1993.

[6] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing - PODC '95*, pages 204–213, August 1995.

[7] Aleksandar Dragojevik, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[8] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, pages 47–56, January 2010.

[9] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. On Bounding Response Times under Software Transactional Memory in Distributed Multiprocessor Real-Time Systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition - DATE '09*, pages 688–693, April 2009.

[10] Toufik Sarni, Audrey Queudet, and Patrick Valduriez. Real-Time Support for Software Transactional Memory. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications - RTCSA '2009*, pages 477–485, August 2009.

[11] Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-Time Transactional Memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, pages 326–333, March 2010.

[12] António Barros and Luís Miguel Pinho. Software transactional memory as a building block for parallel embedded real-time systems. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications - SEAA 2011*, August 2011.

[13] Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Sergio Arévalo. Integrating groups and transactions: A fault-tolerant extension of Ada. In *Proceedings of the International Conference on Reliable Software Technologies - Ada-Europe '98*, pages 78–89, 1998.

[14] Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Sergio Arévalo. Implementing Transactions using Ada Exceptions: Which Features are Missing? *Ada Letters*, XXI(3):64–75, September 2001.

[15] Jörg Kienzle, Ricardo Jiménez-Peris, Alexander Romanovsky, and Marta Patiño-Martínez. Transaction Support for Ada. In *Proceedings of the 6th International Conference on Reliable Software Technologies - Ada-Europe 2001*, pages 290–304, 2001.

[16] Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, November 2003.

[17] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, pages 79–90, January 2010.

[18] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, December 2010.

[19] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, pages 179–193, 2006.

[20] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, pages 194–208, 2006.

[21] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, pages 284–298, 2006.

[22] Dmitri Perelman and Idit Keidar. SMV: Selective Multi-Versioning STM. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, April 2010.

Deferred Setting of Scheduling Attributes in Ada 2012 *

Sergio Sáez, Alfons Crespo
Instituto de Automática e Informática Industrial
Universitat Politècnica de València
{ssaez,acrespo}@disca.upv.es

Abstract

Some scheduling techniques, specially in multiprocessor systems, require changing several task attributes atomically to avoid scheduling errors and artifacts. This work proposes to incorporate the deferred attribute setting *mechanism to cope with this problem in the next Ada 2012.*

1 Introduction

A real-time system is usually composed by a set of concurrent task that collaborate to achieve a common goal. A real-time task has to produce its result within a temporal interval in order to be a valid result. To enforce this behaviour, the real-time system designer assigns a given priority to each system task. These priorities are used by the underlying real-time operating system (RTOS) to ensure a timeliness execution of the whole system.

In uniprocessor real-time systems using fixed priorities, the priority of a task tends to remain *fixed* during its entire lifespan[1]. However, in uniprocessor systems using dynamic priorities and in several scheduling approaches used in multiprocessor systems, the task has to update its scheduling attributes during the execution of the task's code. When several attributes have to be changed simultaneously some scheduling artifacts can arise if they are not updated atomically, giving rise to erroneous schedules.

In order to correctly implement these real-time systems with dynamic attribute setting, the underlying run-time system has to offer some support for atomically changing multiple attributes. This work proposes a flexible and scalable approach to incorporate this support to the Ada 2012 Run-Time System (RTS).

The rest of the paper is organised as follows: next section presents dynamic attribute changing in uniprocessor systems. Section 3 deals with multiprocessor scheduling and simultaneous setting of task attributes. Then, section 4 presents the proposed approach to solve the presented issues. Section 5 proposes to add CPU affinities to `Timing_Events` with deferred setting support. Finally, section 6 shows some conclusions.

2 Scheduling Attributes in Uniprocessor Systems

In Ada 2005 the priority of a task is represented by the task attribute `Priority` and, when the `EDF_Across_Priorities` policy is specified, by an absolute deadline. The Ada Run-Time System uses these *scheduling attributes* to choose which task or tasks have to be executed at a given instant. The initial value of these scheduling attributes can be specified within the task definition by means of the following pragmas:

```
pragma Priority (expression);                              -- See RM D.1
pragma Relative_Deadline (relative_deadline_expression);   -- See RM D.2.6
```

*This work was partially supported by the Vicerectorado de Investigación of the Universidad Politécnica de Valencia under grant PAID-06-10-2397 and European Project OVERSEE (ICT-2009 248333)

[1] Despite of priority changes due to priority inheritance protocols

Although on a uniprocessor system the priority of a task is not usually changed by the task itself, a periodic real-time task scheduled under the EDF policy has to change its absolute deadline on each activation. Ada 2005 offers two procedures to change the priority and the absolute deadline of a task that are shown bellow:

```
package Ada.Dynamic_Priorities is
   ...
   procedure Set_Priority(Priority : in System.Any_Priority;
                          T : in Ada.Task_Identification.Task_Id :=
                              Ada.Task_Identification.Current_Task);
end Ada.Dynamic_Priorities;
```

```
package Ada.Dispatching.EDF is
   ...
   procedure Set_Deadline (D : in Deadline;
                           T : in Ada.Task_Identification.Task_Id :=
                               Ada.Task_Identification.Current_Task);
end Ada.Dispatching.EDF;
```

However, as both procedures are dispatching points, when a task calls the `Set_Deadline` procedure to update its absolute deadline before getting suspended until its next release, it could be preempted by a task with a closer deadline, causing the scheduling artifacts shown in the Figure 1. It can be observed that when task `T0` changes its deadline to the next absolute deadline, the task `T1` becomes more urgent and does not allow task `T0` to execute its **`delay until`** statement until the task `T1` executes its own `Set_Deadline` procedure.

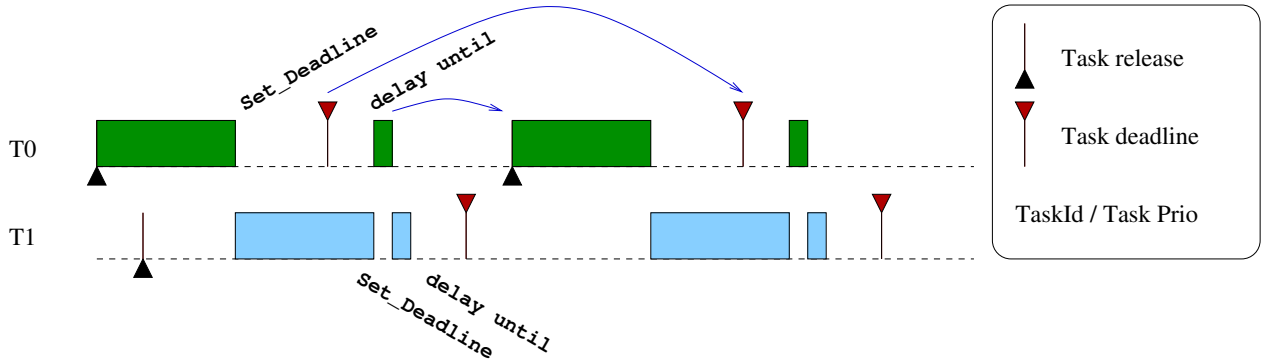


Figure 1. Scheduling artifact during `Set_Deadline` + `delay until` code sequence.

Although this anomalous behaviour does not cause a great damage in the scenario shown by Figure 1 (the only effect is that task `T0` executes its **`delay until`** after its deadline), Ada 2005 provides an additional `Delay_Until_And_Set_Deadline` procedure to avoid these artifacts. Its behaviour is defined as follows:

> The procedure `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again it will have deadline `Delay_Until_Time + Deadline_Offset`. (RM D.2.6 15/2)

Using this procedure, the main loop of a periodic task scheduled under EDF policy will be as follows:

```
loop -- Body of a periodic task under EDF policy
   -- Task code
   ...
   -- Preparation code with scheduling artifacts
   Next_Time := Next_Time + Period;
```

```
    Set_Deadline(Next_Time + Relative_Deadline);
    delay until Next_Time;
  end loop;
```

⇓

```
  loop -- Body of a periodic task under EDF policy
    -- Task code
    ...
    -- It performs atomically Set_Deadline + delay until
    Next_Time := Next_Time + Period;
    Delay_Until_And_Set_Deadline(Next_Time, Relative_Deadline);
  end loop;
```

3 Scheduling Attributes in Multiprocessor Systems

Real-Time and embedded systems are becoming more complex, and multiprocessor/multicore systems are becoming a common execution platform in these areas. In order to achieve a predictable schedule of a set of real-time tasks in a multiprocessor platform several approaches can be applied. Based on the capability of a task to migrate from one processor to another, the scheduling approach can be:

Global scheduling: All tasks can be executed on any processor and after a preemption the current job can be resumed in a different processor.

If the scheduling decisions are performed on-line, in a multiprocessor platform with M CPUs, the M active jobs with the highest priorities are the ones selected for execution. If the scheduling decisions are computed off-line, releases times, preemption instants and processors where tasks have to be executed are stored in a static scheduling plan.

Job partitioning: Each job activation of a given task can be executed on a different processor, but a given job cannot migrate during its execution.

The processor where each job is executed can be decided by an on-line global dispatcher upon the job activation, or it can be determined off-line by a scheduling analysis tool and stored in a processor plan for each task. The job execution order on each processor is determined on-line by its own scheduler using the scheduling attributes of each job.

Task partitioning: All job activations of a given task have to be executed in the same processor. No job migration is allowed.

The processor where a task is executed is part of the task's scheduling attributes. As in the previous approach, the order in which each job is executed on each processor is determined on-line by the scheduler of that processor.

In addition to these basic approaches, new techniques that mix task partitioning with task that migrate from one processor to another at specified times are already available in the literature. In this approach, known as *task splitting*, some works suggest to perform the processor migration of the split task at a given time after each job release [1] or when the job has performed a certain amount of execution [2]. It is worth noting that this approach normally requires the information about the processor migration instant to be somehow coded into the task behaviour.

To apply some of these scheduling approaches some specific support at kernel and user-space level is needed, e.g. system and CPU clock timers and dynamic scheduling attributes.

The forthcoming release of Ada 2012 is expected to offer explicit support for multiprocessor platforms through a comprehensive set of programming mechanisms shown in Listing 1 [3, 4]. Although these mechanisms have been shown adequate to apply task and job partitioning, and task splitting techniques [5, 6], they will suffer the same kind of artifact shown in Figure 1. Next subsection shows why the current proposal found in [3, 4] is still inadequate for real-time multiprocessor systems.

Listing 1. Ada 2012 facilities to set the target CPU

```
package System.Multiprocessors.Dispatching_Domains is
   ...
   type Dispatching_Domain (<>) is limited private;
   ...
   procedure Assign_Task(Domain : in out Dispatching_Domain;
                         CPU : in CPU_Range := Not_A_Specific_CPU;
                         T : in Task_Id := Current_Task);
   procedure Set_CPU(CPU : in CPU_Range; T : in Task_Id := Current_Task);
   function Get_CPU(T : in Task_Id := Current_Task) return CPU_Range;
   procedure Delay_Until_And_Set_CPU(Delay_Until_Time : in Ada.Real_Time.Time;
                                     CPU : in CPU_Range);
end System.Multiprocessors.Dispatching_Domains;
```

3.1 Multiprocessor scheduling requirements

Some multiprocessor scheduling approaches require to specify the target CPU for each real-time task. Additionally, in job partitioning and task splitting techniques the target CPU changes during the task lifespan. As each processor can have a different set of tasks or to use an EDF scheduling policy, moving a task from one CPU to another could also require to change its priority or its deadline. If all these task attributes are not changed atomically, some scheduling artifacts could arise giving rise to incorrect schedules.

Figures 2 and 3 shown how a task can miss its deadline trying to change simultaneously its priority and its target CPU. Both scenarios try to change task `T0` from one CPU to another, but using a different priority into the new CPU. In Figure 2 the task `T0` losses its deadline while executing the `Set_Priority + Set_CPU` sequence. After `T0` changes its priority, the task `T1` has a greater priority and avoids the task `T0` to complete the `Set_CPU` statement until it is too late. Figure 3 shows a different scenario where the incorrect sequence is `Set_CPU + Set_Priority`. The only solution to these situations is to provide a mechanism to simultaneously change the priority and the target CPU. Similar requirements are needed when using dynamic priorities.

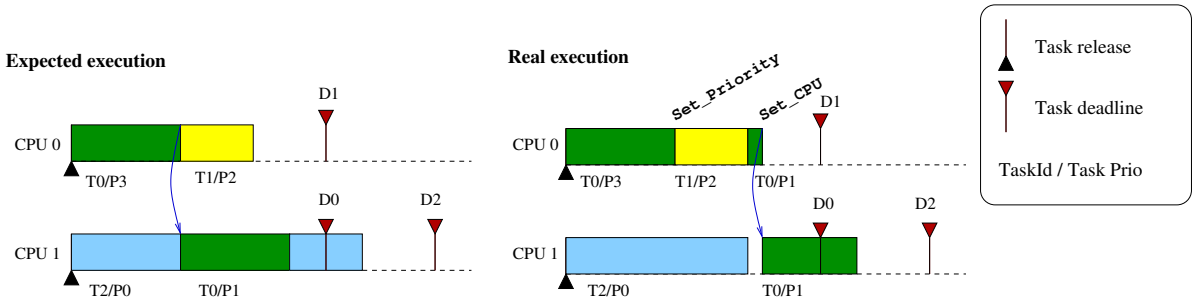


Figure 2. Expected and real execution for `Set_Priority + Set_CPU` code sequence.

Although the scenarios shown in Figures 2 and 3 can be solved by encapsulating both `Set_CPU` and `Set_Priority` inside a protected operation, this cannot be performed when the change of priority/deadline and target CPU is combined with a **`delay until`** statement. As shown in the following examples, no correct sequence of code is valid using the current multiprocessor support proposal.

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_Deadline(Next_Time + Relative_Deadline);
  Delay_Until_And_Set_CPU(Next_Time, Next_CPU);
  -- Similar to scenario with Set_Priority + Set_CPU
end loop;
```

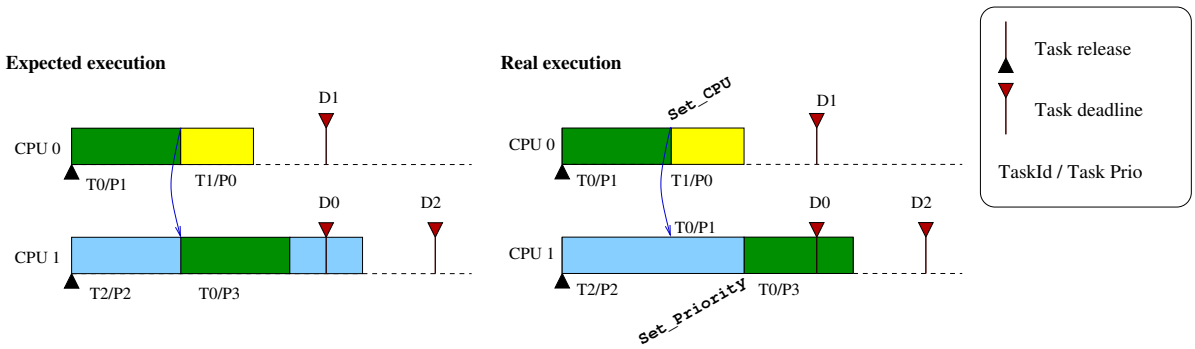


Figure 3. Expected and real execution for `Set_CPU + Set_Priority` code sequence.

```
loop
   -- Task code
   ...
   Next_Time := Next_Time + Period;
   Set_CPU(Next_CPU);
   Delay_Until_And_Set_Deadline(Next_Time, Relative_Deadline);
   -- Similar to scenario with Set_CPU + Set_Priority
end loop;
```

These sequences of code are common in job partitioning schemes, in order to set the CPU where the next job is going to be executed before the current job finishes, and in task splitting techniques, in order to reset the original CPU at the end of the job after one or more *splits*.

Next section presents a proposal to cope with these kind of scenarios using deferred attributes. This proposal was already briefly introduced in [6].

4 Deferred Attribute Setting

The Ada 2005 Reference Manual specifies the behaviour for a task that dynamically changes its priority or its deadline:

> On a system with a single processor, the setting of the base priority of a task T to the new value occurs immediately at the first point when T is outside the execution of a protected action. (RM D.5.1 10/2)
>
> On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor. (RM D.5.1 12.1/2)
>
> On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered on to the ready queue determined by the rules defined below. (RM 2.6 16/2)

However, although the current Ada Issue AI05-167/11 [4] does not specify anything about changing the CPU inside a protected objects it seems reasonable to apply similar restrictions to the setting of the CPU affinity. The text could be as follows:

> On a system with multiple processors, the setting of the target CPU of a task T to the new value occurs immediately at the first point when T is outside the execution of a protected action.

In order to follow the above mentioned restrictions, if a task invokes a procedure to change its attributes within a protected action, the Ada RTS has to defer the priority/deadline/CPU change until the task is outside the execution of the protected action. This deferred setting of task attributes can be used to solve the scheduling artifacts shown in the previous section.

This work proposes to add explicit support to perform a deferred setting of task attributes adding the following procedures:

```
package Ada.Dynamic_Priorities is
   ...
   -- Programs a deferred setting of the base priority
   procedure Set_Next_Priority(Priority : in System.Any_Priority;
                               T : in Task_Id := Current_Task);
   ...
end Ada.Dynamic_Priorities;
```

```
package Ada.Dispatching.EDF is
   ...
   -- Programs a deferred setting of the absolute deadline
   procedure Set_Next_Deadline (D : in Deadline; T : in Task_Id := Current_Task);
   ...
end Ada.Dispatching.EDF;
```

```
package System.Multiprocessors.Dispatching_Domains is
   ...
   -- Programs a deferred setting of the target CPU
   procedure Set_Next_CPU(CPU : in CPU_Range; T : in Task_Id := Current_Task);
   ...
end System.Multiprocessors.Dispatching_Domains;
```

The semantic of these procedures can sketched as:

> The deferred setting of a task attribute will delay the effective attribute setting until the next task dispatching point. If the task T is inside the execution of a protected action, the setting of the new value occurs immediately at the first point when T is outside the execution of the protected action.

With the introduction of these new procedures, the erroneous code presented in section 3.1 can be rewritten as follows:

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_Deadline(Next_Time + Relative_Deadline);     -- It changes the task deadline ...
  Delay_Until_And_Set_CPU(Next_Time, Next_CPU);    -- ... and then the CPU affinity
end loop;
```

or

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_CPU(Next_CPU);                               -- It changes the CPU affinity ...
  Delay_Until_And_Set_Deadline(Next_Time,          -- ... and then the task deadline
                               Relative_Deadline);
end loop;
```

⇓

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_Next_Deadline(Next_Time + Relative_Deadline); -- It specifies the next deadline
  Set_Next_CPU(Next_CPU)                            -- It specifies the next CPU
  delay until Next_Time;                            -- The next deadline and CPU are ...
                                                    -- ... changed atomically
end loop;
```

As **`delay until`** statement is a task dispatching point, the deferred attributes set by `Set_Next_Deadline` and `Set_Next_CPU` take effect at that point. Also the code described in the scenarios of Figures 2 and 3 can be easily solved with the new procedures, as shown in the right side of the next listings.

```
loop
  -- Task code
  ...
  Set_Priority(Next_Priority);
  Set_CPU(Next_CPU);
  ...
end loop;
```

```
loop
  -- Task code
  ...
  Set_Next_Priority(Next_Priority);
  Set_CPU(Next_CPU);
  ...
end loop;
```

⇒

```
loop
  -- Task code
  ...
  Set_CPU(Next_CPU);
  Set_Priority(Next_Priority);
  ...
end loop;
```

```
loop
  -- Task code
  ...
  Set_Next_CPU(Next_CPU);
  Set_Priority(Next_Priority);
  ...
end loop;
```

The proposed procedures give rise to a more orthogonal and cleaner task code. It can also be extended to cover other existing task attributes or new proposed ones, if required. On the other side, the procedure `Delay_Until_And_Set_CPU` will become unnecessary and the procedure `Delay_Until_And_Set_Deadline` deprecated.

5 CPU affinity of `Timing_Events`

Another feature of Ada 2005 highly required to support some of the multiprocessor scheduling approaches are the `Timing_Events`. They allow a task to program the execution of protected actions at specific time instants, e,g. to program future changes to its task attributes, required to perform a task split.

However, when `Timing_Events` are used to wake up a real-time task, it should be taken into account that the processor where the task has to be executed can be different from the one used to program the `TimingEvent`. This scenario can introduce a unnecessary scheduling overhead: a `TimingEvent` produces a clock interrupt in a processor `P1`, this processor executes the `TimingEvent` handler and wakes up a task that has to be executed in a different processor `P2`; after the handler execution, the processor `P1` has to send an Inter-Processor Interrupt (IPI) to force the scheduler execution in processor `P2`. It will be clearly more efficient, if the `TimingEvent` handler can be programmed to be executed directly in processor `P2`. However, implementation details have to be carefully studied in the case of using multiple timer queues. In this case, if the timer queue of the target processor `P2` is empty, the above scenario may require also an IPI to program the clock interrupt in `P2`.

Despite of these implementations details, this work proposes to add explicit support to control the CPU affinity of a `TimingEvent`. The proposal requires to modify the package `Ada.Real_Time.Timing_Events` to add the following procedures:

```
package Ada.Real_Time.Timing_Events is
   type Timing_Event is tagged limited private;
   ...
   -- Set the CPU where the current handler has to be executed
   procedure Set_CPU(Event : in out Timing_Event; CPU : in CPU_Range);
   -- Set the CPU where the next handler established with Set_Handler has to be executed
   procedure Set_Next_CPU(Event : in out Timing_Event; CPU : in CPU_Range);
   ...
end Ada.Real_Time.Timing_Events;
```

The procedure `Set_CPU` allows the programmer to specify where the handler that is already set has to be executed. It allows changing the target CPU of a programmed `TimingEvent` if the implicated task migrates from one CPU to another [2]. The procedure `Set_Next_CPU` establishes the target CPU to be used for the next handler when the procedure `Set_Handler` was used. If no handler is set, both procedures behave identical.

Depending on the available clock interrupt hardware, the Ada RTS can implement one or multiple queues to store the active `Timing_Events`. In the case of a global `Timing_Events` queue, the CPU information can be used to configure the interrupt controller and set the clock interrupt affinity to the target CPU of the closer event. In the case of a per-CPU clock interrupt hardware, multiple `Timing_Events` queues can be used. This could require to move `Timing_Events` from one to another when the `Set_CPU` is used. In both cases, the invocation of the procedure `Set_CPU` can require a reconfiguration of the clock interrupt hardware, while the invocation of the `Set_Next_CPU` is only used to configure the next event or to store it in a processor specific events queue.

6 Conclusions

A small set of modifications for the next Ada 2012 have been proposed to allow simultaneous setting of multiple task attributes. This behaviour is required to adequately support some of the multiprocessor scheduling approaches. The proposed solution is based on the introduction of *deferred attributes*.

The setting of deferred attributes allows the application to specify a set changes of task attributes that a task will apply in the next dispatching point. This gives rise to a simpler and scalable interface for simultaneously changing multiple task attributes. A similar interface has been proposed for the `Timing_Events` mechanism, allowing the Ada RTS to implement it efficiently in multiprocessor platforms.

References

[1] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *21st Euromicro Conference on Real-Time Systems, ECRTS 2009*, pp. 239–248, IEEE Computer Society, 2009.

[2] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *21st Euromicro Conference on Real-Time Systems, ECRTS 2009*, (Los Alamitos, CA, USA), pp. 249–258, IEEE Computer Society, 2009.

[3] A. Burns and A. J. Wellings, "Dispatching domains for multiprocessor platforms and their representation in Ada," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 41–53, 2010.

[4] Ada 2005 Issues. AI05-0167-1/11, *Managing affinities for programs executing on multiprocessors*, 2011. Version: 1.17. Status: Amendment 2012.

[5] B. Andersson and L. M. Pinho, "Implementing multicore real-time scheduling algorithms based on task splitting using Ada 2012," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 54–67, 2010.

[6] S. Sáez and A. Crespo, "Preliminary multiprocessor support of Ada 2012 in GNU/Linux systems," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 68–82, Springer, 2010.

[2] This action can produce a considerable overhead and its use must be carefully studied

Programming Language Vulnerabilities – Proposals to Include Concurrency Paradigms

Stephen Michell
Maurya Software Inc
Ottawa, Ontario, Canada

stephen.michell@maurya.on.ca

Abstract

The vulnerability methodology of the ISO/IEC/JTC 1/ SC 22/ WG 23 Programming Language Vulnerabilities Working Group is applied to the problem space of concurrency. A set of vulnerabilities is developed to capture the issues thread creation, thread termination, shared data access, resource hijacking and communication protocols..

1 Introduction

Software has always been at risk to failures due to mistakes in development, or due to attacks that exploit vulnerabilities in the code. A number of organisations, including the US Department of Homeland Security (DHS), the Open Web Application Security Project (OWASP), the Web Application Security Consortium (WASC) and the International Standards Organisation's Programming Language Vulnerabilities Working Group (ISO/IEC/JTC 1/SC 22/WG 23) are categorising ways that programs are being attacked and recommending approaches to counter threats to software. However, as noted by [BW 2009], the taxonomies being developed have largely left the issues of failures or attacks based upon the concurrent aspects of a program as future work.

The reasons for delaying the tackling of vulnerabilities associated with concurrency stem from the complexity of the overall issues being addressed, the complexity that concurrency adds, and the relatively small amount of real-life data available to consider concurrency vulnerabilities.

This paper will examine the problem space from the view of WG 23, and will propose a set of language-independent vulnerabilities that can be recommended to WG 23 for inclusion of the next version of their vulnerability technical report[TR 24772].

The paper is organised as follows. Section 2 discusses rationale behind the work. Section 3 discusses the general issue of concurrency vulnerabilities. Section 4 is a summary of the vulnerabilities examined. Section 5 is a detailed writeup of six vulnerabilities, each one following a standard pattern of TR 24772. Section 6 is conclusions and future work.

2 Vulnerability Work Undertaken

The Department of Homeland Security, through the Mitre Corp, has produced the Common Vulnerabilities Evaluation database[CVE], the Common Weakness Enumeration database[CWE] and the Common Attack Pattern Enumeration and Classification database[CAPEC]. All of these are based on "found in the wild" disruptions. CVE identifies individual instances of known weaknesses in applications and number in the thousands of identified items. CWE identifies more than 800 weaknesses that are a generalisation and aggregation of CVE vulnerabilities. CAPEC is a taxonomy of attack patterns and identifies almost 300 known attack patterns. [OWASP] and [WASC] use different categorisations, but arrive at similar levels of detail.

ISO/IEC/SC 22/WG 23 produces an international technical report, [TR 24772] that coalesces vulnerabilities at a higher level, and attempts to identify programming language features that make attacks and failures easier or more difficult to effect. The first report, published in 2010, contained 53 programming language vulnerabilities and 17 language-agnostic vulnerabilities. WG 23 is working on a second version of the document that will add more vulnerabilities and add annexes that map vulnerabilities into programming language specific guidance on avoiding vulnerabilities.

3 Concurrency Issues in Programming Language Vulnerabilities

Everyone working in the software security field has been aware that issues surrounding concurrency and vulnerabilities need to be addressed. There are a number of reasons that concurrency issues have not yet been addressed by WG 23:

1. The number and variety of attacks and threats that do not rely upon any aspect of concurrency are so pervasive that they have dominated the vulnerability analyses space.
2. Most of the analyses done to date has relied to a large part on attacks and failures that have occurred;
3. Most attacks and failures that have been catalogued have been found in threats to memory, input validation, command line processing, data weaknesses, but rarely to concurrent behaviours.

For example, in the CWE database, there are a few concurrency-related issues, such as ID 367 Time-of-Check-Time-of-Use, ID 362 "Concurrent Execution using Shared Resource with Improper Synchronization", ID 833 "Deadlock", ID 667 Improper Locking, and ID 414 "Missing Lock Check". It should be noted that the relative threat of these vulnerabilities has been seen to be significantly lower than that of others. In fact, in the SANS/CWE "Top 25" there is exactly one concurrency-related weakness identified, ID 362 "Concurrent Execution using Shared Resource with Improper Synchronization" which is last in the top 25.

WG 23 has attempted to take a higher level look at vulnerabilities than has been done by most of the other works. For example, [TR24772] attempts to discuss buffer overflows as a single problem, instead of address underflows, overflows, read access violations, write access violations, on stack and on heap violations as separate vulnerabilities. However, WG 23 has also struggled to capture concurrency vulnerabilities at the right level. Suggestions for inclusion have ranged from creating a single vulnerability that captures the most relevant concurrency misuses to documenting all 35 identified vulnerabilities in [BW 2009]. Neither solution is acceptable, so a different consideration of concurrency vulnerabilities is required.

4 Proposal for WG 23 Concurrency Vulnerabilities

Concurrency vulnerabilities are important to the communities served by WG 23 for a number of reasons. First, TR 24772 serves more than the security community by explicitly trying to address errors that happen by accident as well as by deliberate actions. Secondly, concurrent systems are becoming more prevalent as multiprocessor systems, cloud computing, client-server applications and large scale computing become commonplace. Thirdly, as more traditional vulnerabilities are being addressed by the operating systems and by analysis tools, attackers look towards leveraging other difficult to protect paradigms such as concurrency to gain footholds into systems. For these reasons, it is imperative that concurrency vulnerabilities be addressed in the next release of TR 24772.

There are many ways to address concurrency vulnerabilities. One could attempt to to roll all vulnerabilities into single writeup or break them into a set approximately as numerous as [BW 2009]. Our approach was to look at the major interactions of threads (and of processes) and attempt to organise around those. For the record, these major interactions would be: at the startup of a thread, at the termination of a thread; when threads share data; when threads share a communication protocol; and when threads interact with their environment. Some of these larger groupings create significant complexity when discussing the issues that arise and require further refinement, so the following vulnerabilities are proposed:

1. Thread creation and activation Issues (CGA[1]) – including static, dynamic creation, synchronised activation, unsynchronised activation, resource issues, and error propagation.

1 The three letter designation is a mechanism used by WG 23 to track and manage vunerabilities. In TR 24772 throughout its development, vulnerability writeups have been split, merged, reordered and moved between clause 6 and 7. The three letter code remains constant for a given vulnerability. The code itself is effectively random and should not be treated as an acronym.

2. Premature termination (CGS) – vulnerabilities that occur when a thread aborts or terminates before it is expected to finish.
3. Directed termination (CGT) – vulnerabilities that occur when a thread is directed to abort or terminate but does not finish to the expectations of the thread that directed the termination.
4. Shared data access (CGX) – vulnerabilities that can occur as a result of direct update to shared data.
5. Concurrent Data Access and Corruption (CGY) – vulnerabilities that can occur in concurrent systems when shared resources (such as files, or environment variables) are visible and available to other processes.
6. Protocol failures, including deadlock, livelock, monitors, synchronous messaging (CGM).
7. Clocks, scheduling, priorities (Real time vulnerabilities).

Proposed write ups for items 1-6 are attached in Annex A to this document. The final item, Clocks, scheduling and priorities, is still under development. It is proposed that these write ups be reviewed, possibly amended by IRTAW 15 and submitted to WG 23 as a recommended set of concurrency vulnerabilities for inclusion in the next release of [TR 24772]. Assistance in finishing number 7 would also be greatly appreciated[2].

5 Vulnerability Write ups

The write ups of each vulnerability follow the form adopted by WG 23, since the intention is that this material be directly usable by WG 23.

5.1 Concurrency – Activation [CGA]

5.1.0 Terminology[3]

Activation : The creation and setup of a thread up to the point where it begins execution. Threads may depend upon one or more other threads to define its existence for objects to be accessed and to determine the duration time of execution.

Activated thread: The thread that is created and begins execution as a result of the activation.

Activating thread: The thread that exists first and makes the library calls or contains the language syntax that causes new threads to be Activated. The Activating Thread may or may not wait for the Activated Thread to finish activation and may or may not check for errors if the activation fails. The Activating Thread may or may not be permitted to terminate until after the Activated Thread terminates.

Static Activation: The creation and initiation of a thread by program initiation, an operating system or runtime kernel, or by another thread as part of a declarative part of the thread before it begins execution. In static activation, a static analysis can determine exactly how many threads will be created and how much resource, in terms of memory, processors, cpu cycles, priority ranges and inter-thread communication structures, will be needed by the executing program before the program begins.

Dynamic Thread Activation: The creation and initiation of a thread by the another thread (including the main program) as an executable, repeatable command, statement or subprogram call.

Thread: A lightweight process that shares memory space with other threads.

---

2 Indeed, discussion of the real time vulnerabilities occurred at the workshop, and the results of this work will be considered by ISO/IEC/JTC 1/SC 22/WG 23 for inclusion in ISO IEC TR WD 24772.

3 This section and all other sections labelled 5.x.0 are terminology identified in each vulnerability and will be collated and placed into a general terminology section in the final document.

5.1.1 Description of Application Vulnerability

A thread is activated, but the lack of some resource or a programming error prevents the activation from completing. The activating thread may not have sufficient visibility or awareness into the execution of the activated thread to determine if it has indeed been activated at the point that it is needed. The unrecognised activation failure can cause a protocol failure in the activating thread or in other threads that rely upon some action by the unactivated thread. This may cause the other thread(s) to wait forever for some event from the unactivated thread, or may cause an unhandled event or exception in the other threads.

5.1.2 Cross References

Hoare A., "Communicating Sequential Processes", Prentice Hall, 1985

Holzmann G., "The SPIN Model Checker: Principles and Reference Manual"., Addison Wesley Professional. 2003

UPPAAL, available from www.uppaal.com,

Larsen, Peterson, Wang, "Model Checking for Real-Time Syetems"., Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

CWE 364 Signal Handler Race Condition

5.1.3 Mechanism of Failure

All threads except the main thread must be activated by program steps of another thread. The activation of each thread requires that dedicated resources be created for that thread, such as a thread stack, thread attributes, and communication ports. If insufficient resources remain when the activation attempt is made, the activation will fail. Similarly, if there is a program error in the activated thread or if the activated thread detects an error that causes it to terminate before beginning its main work, then it may appear to have failed during activation. When the activation is “static”, resources have been preallocated so activation failure because of a lack of resources will not occur, however errors may occur for reasons other than resource allocation and the results of an activation failure will be similar.

If the activating thread waits for each activated thread, then the activating thread will likely be notified of activation failures (if the particular construct or capability supports activation failure notification) and can be programmed to take alternate action. If notification occurs but alternate action is not programmed, then the program will execute erroneously. If the activating thread is loosely coupled with the activated threads, and the activating thread does not receive notification of a failure to activate, then it may wait indefinitely for the unactivated task to do its work, or may make wrong calculations because of incomplete data.

The single activation is a special case of activations of collections of threads simultaneously. This paradigm (activation of collections of threads) can be used in Ada with arrays of tasks that activate as part of their declaration, or in other languages that parallelise calculations and create anonymous threads to execute each slice of data. In such situations the activating thread is unlikely to individually monitor each activated thread, so a failure of some to activate without explicit notification to the activating thread can result in erroneous calculations.

If the rest of the application is unaware that an activation has failed, an incorrect execution of the application algorithm may occur, such as deadlock of threads waiting for the activated thread.

Activation failures usually result in deadlock of the application, or cause errors and incorrect calculations that may cause the user to lose trust in the application. It would be unlikely that an external attacker could take control of a system simply by causing activation failures; however when coupled with other vulnerabilities, activation failures could be used to change calculations or to further other attacks.

5.1.4 Applicable Language Characteristics

Languages that permit concurrency within the language, or that use support libraries and operating systems (such as POSIX or Windows) that provide concurrency control mechanisms. In essence all traditional languages on fully functional operating systems (such as POSIX-compliant OS or Windows) can access the OS-provided mechanisms.

5.1.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its effects in the following ways:

- Always check return codes on OS, library provided or language thread activation routines.
- Handle errors and exceptions that occur on activation.
- Create explicit synchronisation protocols, to ensure that all activations have occurred before beginning the parallel algorithm, if not provided by the language or by the threading subsystem.
- Use programming language provided features that couple the activated thread with the activating thread to detect activation errors so that errors can be reported and recovery made.
- Use static activation in preference to dynamic activation so that static analysis can guarantee correct activation of threads.

5.1.6 Implications for Standardisation

In future standardisation activities, the following items should be considered:

- Consider including automatic synchronisation of thread initiation as part of the concurrency model.

5.2 Concurrency – Directed Termination [CGT]

5.2.0 Terminology

Abort: The completion and shut down of a thread, where the thread is not permitted any execution after the command to abort has been received by the thread, or by the runtime services that control the thread. In particular, the thread will not be able to release any locks that it has explicitly acquired, and may not release any OS provided locks or data structures.

Abort deferred region: A section of code where a thread is permitted to ignore abort directives, usually because it is holding a system resource or the risk of corruption to the application is significant while the thread is manipulating certain resources.

Termination: The completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, return any heap-acquired storage, notify any dependent threads that it is terminating, and finalise system resources dedicated to the thread. There are a number of steps in the termination of a thread as listed below, but depending upon the multithreading model, some of these steps may be combined, may be explicitly programmed, or may be missing.

- The termination of programmed execution of the thread, including termination of any synchronous communication;
- The finalisation of the local objects of the thread;
- Waiting for any threads that may depend on the thread to terminate;
- Finalisation of any state associated with dependent threads;
- Notification of outer scopes that finalisation is complete, including possible notification of the activating task; and

- Removal and cleanup of thread control blocks and any state accessible by the thread by possibly threads in outer scopes.

Terminated Thread: The thread that is being halted from any further execution.

Termination Directing Thread: The thread (including the OS) that requests the abort or termination of one or more threads.

5.2.1 Description of Application Vulnerability

This discussion is associated with the effects of unmet termination directives. For a discussion of premature termination, see CGT Concurrency – Premature Termination.

When a thread is working cooperatively with other threads and is directed to terminate, there are a number of error situations that may occur that can lead to compromise of the system. The termination directing thread may request that one or more other threads abort or terminate, but the terminated thread(s) may not be in a state such that the termination can occur, may ignore the direction, or may take longer to abort or terminate then the application can tolerate.

Late termination may cause a failure to meet deadlines, implying incomplete calculation, leading the application to deliver no results or incorrect results. Non-termination may cause deadlock, livelock, failure to release resources, and corrupted data abstractions. All of these may lead to failure of the application.

5.2.2 Cross References

Hoare C.A.R., "Communicating Sequential Processes", Prentice Hall, 1985

Holzmann G., "The SPIN Model Checker: Principles and Reference Manual"., Addison Wesley Professional. 2003

Larsen, Peterson, Wang, "Model Checking for Real-Time Systems"., Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

The Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

CWE 364 Signal Handler Race Condition

5.2.3 Mechanism of Failure

The abort of a thread may not happen if a thread is in an abort-deferred region and does not leave that region (for whatever reason) after the abort directive is given. Similarly, if abort is implemented as an event sent to a thread and it is permitted to ignore such events, then the abort will not be obeyed.

The termination of a thread may not happen if the thread ignores the directive to terminate, or if the termination process raises exceptions that result in the thread not terminating.

The termination directing thread will be expecting the thread(s) to terminate (or abort) and may proceed on the expectation that the abort/termination occurred. If some of the threads that were directed to terminate do not terminate, then the overall application will not move on to the next phase of the application, or will produce erroneous results.

Livelock, the freeze-up of a system because some threads start cycling indefinitely, or deadlock, the freeze-up of the system because some threads are blocked waiting for other threads (such as waiting for them to terminate) are distinct possibilities resulting from this vulnerability. Arbitrary execution of random code is also a distinct possibility from some kinds of termination errors, but arbitrary execution of known code is not likely since it is hard to determine where nonterminating threads will be in their execution when the terminating thread notification is delivered.

5.2.4 Applicable Language Characteristics

Languages that permit concurrency within the language, or support libraries and operating systems (such as

POSIX-compliant OSs or Windows) that provide hooks for concurrency control.

5.2.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use a language that provides a complete concurrency mechanism.
- Use mechanisms of the language or system to determine that aborted threads or threads directed to terminate are indeed terminated. Such mechanisms may be direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress
- Program fall-back handlers to report or recover from failure to terminate situations.

5.2.6 Implications for Standardisation

In future standardisation activities, the following items should be considered:

- Provide a mechanism (either a language mechanism or a service call) to signal another thread (or an entity that can be queried by other threads) when a thread terminates.

5.3 Concurrency – Premature Termination [CGS]

5.3.0 Terminology

Abort: A request to immediately stop and shut down a thread. The request is asynchronous if from another thread, or synchronous if from the thread itself. The effect of the abort request (e.g. whether it is treated as an exception) and its immediacy (i.e., how long the thread may continue to execute before it is shut down) depend on language-specific rules. Immediate shutdown minimises latency but may leave shared data structures in a corrupted state.

Termination Directing Thread: The thread (including the OS) that requests the abort of one or more threads.

Termination: The completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, return any heap-acquired storage, and notify any dependent threads that it is terminating. There are a number of steps in the termination of a thread as listed below, but depending upon the multithreading model, some of these steps may be combined, may be explicitly programmed, or may be missing.

- The termination of programmed execution of the thread, including termination of any synchronous communication;
- The finalisation of the local objects of the thread;
- Waiting for any threads that may depend on the thread to terminate;
- Finalisation of any state associated with dependent threads;
- Notification of outer scopes that finalisation is complete, including possible notification of the activating task;
- Removal and cleanup of thread control blocks and any state accessible by the thread by possibly threads in outer scopes.

Terminated Thread: The thread that is being halted from any further execution.

Master of a Thread: Any thread which must wait for the terminated thread before it can take further execution steps (including termination of itself).

5.3.1 Description of Application Vulnerability

When a thread is working cooperatively with other threads and terminates prematurely for whatever reason

but unknown to other threads, then the portion of the interaction protocol between the terminating thread and other threads is damaged. This may result in: indefinite blocking of the other threads as they wait for the terminated thread if the interaction protocol was synchronous; other threads receiving wrong or incomplete results if the interaction was asynchronous; or deadlock if all other threads were depending upon the terminated thread for some aspect of their computation before continuing.

5.3.2 Cross References

Hoare C.A.R., "Communicating Sequential Processes", Prentice Hall, 1985

Holzmann G., "The SPIN Model Checker: Principles and Reference Manual"., Addison Wesley Professional. 2003

Larsen, Peterson, Wang, "Model Checking for Real-Time Systems"., Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

The Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

CWE 364 Signal Handler Race Condition

5.3.3 Mechanism of Failure

If a thread terminates prematurely, threads that depend upon services from the terminated thread (in the sense of waiting exclusively for a specific action before continuing) may wait forever since held locks may be left in a locked state resulting in waiting threads never being released.

If a thread depends on the terminating thread and receives notification of termination, but the dependent thread ignores the termination notification, then a protocol failure will occur in the dependent thread. For asynchronous termination events, an unexpected event may cause immediate transfer of control from the execution place of dependent thread to another (possible unknown), resulting in corrupted objects or resources; or may cause termination in the master thread, and an expected propagation of failures.

These conditions can result in

- premature shutdown of the system;
- corruption or arbitrary execution of code;
- livelock;
- deadlock;

depending upon how other threads handle the termination errors.

If the thread termination is the result of an abort and the abort is immediate, there is nothing that can be done within the aborted thread to prepare data for return to master tasks, except possibly the management thread or OS notifies others that the event occurred. If the aborted thread was holding resources or performing active updates when aborted, then any direct access by other threads to such locks, resources or memory may result in corruption of those threads or of the complete system, up to and including arbitrary code execution.

Arbitrary execution of random code is distinct possibility from some kinds of termination errors, but arbitrary execution of known code is not likely since it is hard to determine where nonterminating threads will be in their execution when the terminating thread notification is delivered.

5.3.4 Applicable Language Characteristics

Languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant OSs or Windows) that provide hooks for concurrency control.

5.3.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use a language that provides a complete concurrency mechanism.
- Use mechanisms of the language or system to determine that necessary threads are still operating. Such mechanisms may be direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress
- Handle events and exceptions from termination events
- Program fall-back handlers to report or recover from premature termination failures.
- Provide manager threads to monitor progress and to collect and recover from improper terminations or abortions of threads.

5.3.6 Implications for Standardisation

In future standardisation activities, the following items should be considered:

- Provide a mechanism (either a language mechanism or a service call) to preclude the abort of a thread from another thread during critical pieces of code. Some languages (eg Ada or real time Java) provide a notion of an abort-deferred region.
- Provide a mechanism (either a language mechanism or a service call) to signal another thread (or an entity that can be queried by other threads) when a thread terminates.
- Provide a structure within the concurrency service (either a language mechanism or a service call) that defers the delivery of asynchronous exceptions or asynchronous transfers of control.

5.4 Concurrency Data Access [CGX]

5.4.0 Terminology

Thread: A lightweight process that shares memory space with other threads.

5.4.1 Description of Application Vulnerability

Concurrency presents a significant challenge to program correctly, and has a large number of possible ways for failures to occur, quite a few known attack vectors, and many possible but undiscovered attack vectors. In particular, any resource that is visible from more than one thread and is not protected by a sequential access lock can be corrupted by out-of-order accesses. This corruption can lead to resource corruption, premature program termination, livelock, or system corruption.

5.4.2 Cross References

Burns A. and Wellings A., Language Vulnerabilities - Let's not forget Concurrency, IRTAW 14, 2009.

CWE 214 Information Exposure Through Process Environment

CWE 362 - Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

CWE 366 Race Condition Within a Thread

CWE 368 – Context Switching Race Conditions

CWE 413 Improper Resource Locking

CWE 764 Multiple Locks of a Critical Resource

CWE 765 Multiple Unlocks of a Critical Resource

CWE 821 Missing Synchronization

CWE 821 Incorrect Synchronization

5.4.3 Mechanism of Failure

All data that is openly visible to multiple threads is shared data and is open to being monitored or updated directly by a thread, whether or not that data has an access lock protocol in operation. Some concurrent programs do not use access lock mechanisms but rely upon other mechanisms such as timing or other program state to determine if shared data can be read or updated by a thread. Regardless, direct visibility to shared data permits direct access to that data concurrently. This can permit the following errors:

- Lost data or control signals by multiple updates by one thread without corresponding reads by another thread;
- Simultaneous updates of different portions of the data by different threads, resulting in corruption of the data;
- Simultaneous updates of different portions of the data by different threads, resulting in wrong data being passed;
- Missing or corrupt data;
- Precisely written (but wrong) data that changes the behaviour of the program to undertake new objectives;
- Livelock when necessary data is missed or never correctly read.

The above scenarios usually result in corruption, livelock, or corrupted applications. Results such as arbitrary code execution are usually not achievable because threads are programmed and built into the same application, but when combined with other attacks and vulnerabilities, arbitrary code execution may be possible.

5.4.4 Applicable Language Characteristics

The vulnerability is intended to be applicable to languages with the following characteristics:

- Languages that provide explicit concurrency in the language, such as tasks, threads, processes, and potentially share data between threads these entities.
- Languages that provide explicit concurrency and protected regions of sequential access and explicit concurrency control mechanism, such as suspend(thread), block(thread), resume(threads), enable(interrupt), and disable(interrupt)

5.4.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its effects in the following ways.

- Place all data in memory regions accessible to only one thread at a time.
- Use languages and those language features that provide a complete sequential protection paradigm to protect against data corruption. Ada's protected objects and Java's Protected class each provide a safe paradigm when accessing objects that are exclusive to a single program.
- Use operating system primitives, such as the POSIX pthread_xxx primitives for locking and synchronisation to develop a protocol equivalent to the Ada “protected” and Java “Protected” paradigm.

- Where state based mutual exclusion is important for correctness, implement blocking and releasing paradigms, or provide a test in the same protected region to check for correct order and generate errors if the test fails. For example, the following structure in Ada would implement an enforced order.

```
 package buffer_pkg is
   protected Buffer is
      entry Read (Data :    out Data_Type);
      entry Write (Data : in    Data_Type);
   private
      ...;
   end Buffer;
end Buffer_Pkg.
```

In this case, the writer must block until there is room to write a new record, and readers must block if there are no records available.

5.4.6 Implications for Standardisation

In future standardisation activities, the following items should be considered:

- Languages that do not presently implement concurrency but are considering the addition of thread-based concurrency should consider creating primitives that let applications specify regions of sequential access to data. Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.
- Provide the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see [CSP] and [Ravenscar].

5.5 Concurrency Data Corruption [CGY]

Note: This vulnerability should go into Clause 7 and not be included in Programming Language vulnerabilities. It has been written as a programming language vulnerability, but will need rewriting to fit into the clause 7 organization.

5.5.0 Terminology

Stateless Protocol: A communication or cooperation between threads where no state is preserved in the protocol itself (example HTTP or direct access to a shared resource). Since most interactions between processes require that state be preserved, the cooperating threads must use values of the resources(s) themselves or add additional communication exchanges to maintain state.

Stateless protocols require that the application provide explicit resource protection and locking mechanisms to guarantee the correct creation, view, access to, modification of, and destruction of the resource – i.e. the state needed for correct handling of the resource).

Process: A single execution of a program, or portion of an application. Processes often share a common processor, or network, or operating system or filing system or environment variables or other resources, but do not share a common memory space. Processes are usually started and stopped by an operating system and may or may not interact with other processes.

5.5.1 Description of Application Vulnerability

A resource that is directly visible from more than one process (at the same approximate time) and is not protected by access locks can be hijacked or used to corrupt, control or change the behaviour of other processes in the system. This corruption can lead to resource corruption, premature program termination, livelock, system corruption, or arbitrary execution of code.

Many vulnerabilities that are associated with concurrent access to files, shared memory or shared network resources fall under this vulnerability, including resources accessed via stateless protocols such as HTTP and remote file protocols.

5.5.2 Cross References

Burns A. and Wellings A., Language Vulnerabilities - Let's not forget Concurrency, IRTAW 14, 2009.

CWE 642 External Control of Critical State Data

CWE 15 External Control of System or Configuration Setting

5.5.3 Mechanism of Failure

Any time that a shared resource is open to general inspection, the resource can be monitored by a foreign process to determine usage patterns, timing patterns, and access patterns to determine ways that a planned attack can succeed.[4] Such monitoring could be, but are not limited to:

- Read resource values to obtain information of value to the applications
- Monitoring access time and access thread to determine when a resource can be accessed undetected by other threads (for example, Time-of-Check-Time-Of-Use attacks rely upon a determinable amount of time passage between the check on a resource and the use of the resource when the resource could be modified to bypass the check);
- Monitor a resource and modification patterns to help determine the protocols in use.
- Monitor access times and patterns to determine quiet times in the access to a resource that could be used to find successful attack vectors

This monitoring can then be used to construct a successful attack, usually in a later attack.

Any time that a resource is open to general update the resource can be changed by a foreign process to:

- Determine how such changes affect usage patterns, timing patterns, access patterns to determine ways that a planned attack can succeed.
- Determine how well application threads detect and respond to changed values

Any time that a shared resource is open to shared update by a thread, the resource can be changed in ways to further an attack once it is initiated. For example, a well known attack monitor's a certain change to a known file and then immediately replaces a virus free file with an infected file to bypass virus checking software.

With careful planning, the above scenarios occurrence can result in the foreign process determining a weakness of the attacked process and an exploit consisting of anything up to and including arbitrary code execution.

5.5.4 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its effects in the following ways.

- Place all shared resources in memory regions accessible to only one process at a time.
- Protect resources that must be visible with encryption or with checksums to detect unauthorised modifications.
- Protect access to shared resources using permissions, access control, or obfuscation.

---

4 Such monitoring is almost always possible by a process executing with system privilege, but even small slips in access controls and permissions lets such resources be seen from other processes. Indeed, even the existence of the resource, its size, or its access dates/times and history (such as "last accessed time") can give valuable information to an observer.

- Have and enforce very clear rules with respect to permissions to change shared resources.
- Detect attempts to alter shared resources and take immediate action

5.6 Protocol Lock Errors [CGM]

5.6.0 Terminology

Protocol: A set of rules and supporting structures for the interaction of threads or processes. A protocol can be tightly embedded and rely upon data in memory and hardware locks up to communications spanning networks and computer systems.

Thread: A lightweight process that shares memory space with other threads.

5.6.1 Description of Application Vulnerability

All concurrent programs, use protocols to control

- The way that threads interact with each other,
- How to schedule the relative rates of progress,
- How threads participate in the generation and consumption of data
- The allocation of threads to the various roles
- The preservation of data integrity, and
- The detection and correction of incorrect operations.

When protocols are incorrect, or when a vulnerability lets an exploit destroy a protocol, then the concurrent portions fail to work co-operatively and the system behaves incorrectly.

This vulnerability is related to [CGX] Shared Data Access and Corruption, which discusses situations where the protocol to control access to resources is explicitly visible to the participating partners and makes use of visible shared resources. In comparison, this vulnerability discusses scenarios where such resources are protected by protocols, and considers ways that the protocol itself may be misused.

5.6.2 Cross References

C.A.R Hoare, A model for communicating sequential processes, 1980

Larsen, K.G., Petterssen, P, Wang, Y, UPPAAL in a nutshell, 1997

Lundqvist, K and Asplund, L., “A Formal Model of a Run-Time Kernel for Ravenscar”, The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999,

CWE 667 Improper Locking

CWE 413 Improper Resource Locking

CWE 414 Missing Lock Check

CWE 833 Deadlock

CWE 609 Double Checked Locking

CWE 821 Incorrect Synchronization

5.6.3 Mechanism of Failure

All threads use locks and protocols to schedule their work, control access to resources, exchange data, and to effect communication with each other. Protocol errors occur when the expected rules for co-operation are

not followed, or when the order of lock acquisitions and release causes the threads to quit working together. These errors can be as a result of deliberate termination of one or more threads participating in the protocol, disruption of messages or interactions in the protocol, errors or exceptions raised in threads participating in the protocol, or errors in the programming of one or more threads participating in the protocol.

In such situations, there are a number of possible consequences. One common consequence is "deadlock", where every thread eventually quits computing as it waits for results from another thread. A second common consequence is "livelock", where one or more threads commandeer all of the computing resource and effectively lock out the other portions. In each case, no further progress in the system is made. A third common consequence is that data may be corrupted or lack currency (timeliness). A fourth common consequence is that one or more threads detect an error associated with the protocol and terminate prematurely, leaving the protocol in an unrecoverable state.

Concurrent protocols are very difficult for humans to correctly design and implement. Completely synchronous protocols, such as defined by CSP, by Petri nets or by the simple Ada rendezvous protocol can be statically shown to be free from protocol errors such as deadlock and livelock, and considerable progress has been made to verify that the complete system (data and concurrency protocols) is correct. Simple asynchronous protocols that exclusively use concurrent threads and protected regions can also be shown statically to have correct behaviour using model checking technologies, as shown by [LA 1999]. More complex and asynchronous protocols cannot be statically shown to have correct behaviours.

When static verification is not possible, the detection and recovery from protocol errors is necessary. Watchdog timers (in hardware or in an extremely high priority thread), or threads that monitor progress can be used to detect deadlock or livelock conditions and can be used to restart the system. Such recovery techniques also constitute a protocol, but the extreme simplicity of a detection and recovery protocol can usually be verified.

The potential damage from attacks on protocols depends upon the nature of the system using the protocol and the protocol itself. Self-contained systems using private protocols can be disrupted, but it is highly unlikely that predetermined executions (including arbitrary code execution) can be obtained. On the other extreme threads communicating openly between systems using well-documented protocols can be disrupted in any arbitrary fashion with effects such as the destruction of system resources (such as a database), The generation of wrong but plausible data, or arbitrary code execution. In fact, many documented client-server based attacks consist of some abuse of a protocol such as SQL transactions.

5.6.4 Applicable Language Characteristics

The vulnerability is intended to be applicable to languages with the following characteristics:

1. Languages that support concurrency directly:
2. Languages that permit calls to operating system primitives to obtain concurrent behaviours.
3. Languages that permit IO or other interaction with external devices or services.

5.6.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its effects in the following ways.

- Synchronise access to shared resources.
- Use high level synchronisation paradigms, for example monitors, rendezvous, or critical regions.
- Carefully design the architecture of the application to ensure that some threads or tasks never block, and can be available for detection of concurrency error conditions and for recovery initiation;
- Use model checkers to model the concurrent behaviour of the complete application and check for states where progress fails.Place all locks and releases in the same subprograms, and ensure that the order of calls and releases of multiple locks are correct.
- Use techniques such as digital signing (effective use of encryption and hashing) to protect data

being exchanged using open protocols.

- Implement simple detection and recovery protocols when verification of the complete protocol is not possible.

5.6.6 Implications for Standardization

In future standardisation activities, the following items should be considered:

- Raise the level of abstraction for concurrency services.
- Provide services or mechanisms to detect and recover from protocol failures such as deadlock.
- Design concurrency services that help to avoid typical failures such as deadlock.

6 Conclusions and Future Work

This paper shows that it is possible to discuss the major concurrency-related vulnerabilities in a relatively small set of identifiable vulnerabilities. The real time set of paradigms will be examined to produce additional concurrency vulnerability descriptions as appropriate.

Bibliography

[BW 2009] Burns A. and Wellings A., Language Vulnerabilities - Let's not forget Concurrency, IRTAW 14, 2009, ACM SIGAda Letters, Volume 30, Issue 1, April 2009

[CAPEC] Common Attack Pattern Enumeration and Classification database, available from cve.mitre.org

[CVE] The Common Vulnerabilities and Exposure database, available from cve.mitre.org

[CWE] The Common Weakness Enumeration database, available from cve.mitre.org

[LA 1999] [Lundqvist, K and Asplund, L., "A Formal Model of a Run-Time Kernel for Ravenscar", The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

[OWASP] The Open Web Application Security Project, available from www.owasp.org

[TR 24772:2010] ISO IEC TR 24772 "Information technology -- Programming languages -- Guidance to avoiding vulnerabilities in programming languages through language selection and use" , International Standards Organisation, 2010

[WG 23] ISO/IEC/JTC 1/SC 22/WG 23 Programming Language Vulnerabilities work products, available from www.aitcnet.org/isai

(other references included in Section 5.x.3 for each vulnerability as support for individual vulnerability discussions).

Adding Multiprocessor and Mode Change Support to the Ada Real-Time Framework

Sergio Sáez, Jorge Real, and Alfons Crespo
Departamento de Informática de Sistema y Computadores (DISCA)
Instituto de Automática e Informática Industrial (AI2)
Universitat Politècnica de València
Camino de Vera s/n. E46022, Valencia, Spain
Tlf: +34 96 387 7007 Ext. 75741
{ssaez,jorge,alfons}@disca.upv.es *

Abstract

Based on a previous proposal of an Ada 2005 framework of real-time utilities, this paper deals with the extension of that framework to include support for multiprocessor platforms and multiple operating modes and mode changes. The design of the proposed framework is also intended to be amenable to automatic code generation.

1. Introduction

One of the topics discussed at the 13th International Real-Time Ada Workshop was a proposal from Wellings and Burns [17, 18] to develop a framework of real-time utilities. The aim of that proposal was to provide a set of high-level abstractions to ease the development of real-time systems, taking advantage of facilities included in Ada 2005 such as timers and CPU timing mechanisms, and the integration of object oriented and concurrent programming. These new facilities of Ada 2005 are low-level abstractions like setting a timer, synchronizing concurrent tasks, or passing data among them. They are indeed adequate for a programming language, but not powerful enough themselves to abstract away much of the complexity of modern, large real-time systems. These systems:

- are typically formed by components with different levels of criticality that need sharing the available computing resources, e.g. by means of execution-time servers,
- perform activities subject to different release and scheduling mechanisms,
- require the management of timing faults if and when they occur at execution time,
- are increasingly being executed on multiprocessors,
- may execute in several modes of operation, characterized by performing different sets of activities under different timing requirements.

The proposal from Wellings and Burns (hereafter *the original framework*) was a first step in the direction of implementing an Ada 2005 library of real-time utilities. It addressed some of the features above, namely: the flexibility to choose the activation pattern of a task; the possibility to implement deadline and overrun handlers; and the implementation of execution-time servers. Other features were not covered but left as future work, as reported in [9].

*This work is partly funded by the Vicerrectorado de Investigación of Universitat Politècnica de València under grant PAID-06-10-2397 and the Europan Commission's OVERSEE project (FP7-ICT-2009-4, Project ID 248333).

This paper proposes extensions to the original framework in two aspects: (i) adapting the framework to multiprocessor platforms and (ii) defining modes of operation and providing the mechanisms to enable mode changes.

The paper is organized as follows: section 2 defines the context by summarizing the main features of the original framework. Section 3, and its subsections, enumerate the requirements we consider necessary for multiprocessor and multimode applications, and our idea about the design process to develop an application based on the framework. Section 4 describes the framework proposal. The flow of events and handlers under a deadline miss scenario is described in Section 5. Finally, Section 6 gives our conclusions and points out pending issues.

2. Original Framework

The framework proposed in [17, 18] is explained in those publications and, to a larger extent, in chapter 16 of [6]. Therefore, only a brief overview of the original framework is given here. The main goal of the original framework is to provide a reusable set of high-level abstractions for building real-time systems. These abstractions represent real-time, concurrent activities. In the original proposal, they allow to define:

- the nature of the activation mechanism: periodic, sporadic or aperiodic,
- mechanisms to manage deadline misses and execution-time overruns at run time, and
- the possibility to limit the amount of CPU time devoted to tasks by means of execution-time servers.

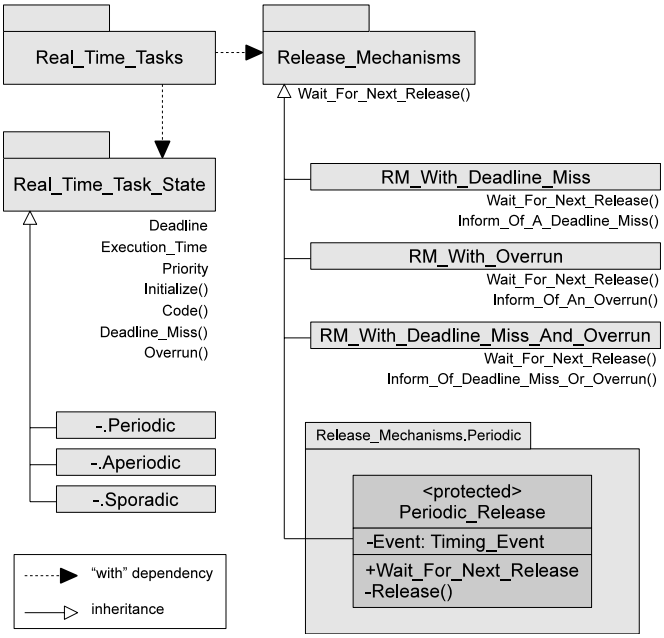


Figure 1. Top-level packages of the original framework

The top-level packages of the original framework are depicted[1] in figure 1. They give the support needed for implementing different kinds of real-time tasks. In particular,

`Real_Time_Tasks` provides idioms to specify how tasks must respond to deadline misses and execution-time overruns. Currently the package offers two types of tasks: a simple real-time task, not affected by these events, and a task type that will execute a proper handler just before being aborted.

[1] Also in the top level of the original framework, a package called `Execution_Servers` provides temporal firewalling mechanisms. This package is not represented in figure 1 for simplicity, since the use of execution-time servers is not considered in this paper.

`Real_Time_Task_State` encapsulates the functionality of the task. It allows defining its deadline, execution time and priority, and to implement procedures for the initialization, regular code, and handlers for both deadline miss and overrun (the latter two are predefined as null procedures). Three child packages extend and specialize this common interface to provide specific support for periodic, aperiodic and sporadic tasks.

`Release_Mechanisms` is a synchronized interface from which task activation mechanisms are derived. The original framework allows for activations with or without notification of events to the task (deadline miss and/or overrun). Child packages are provided for periodic, aperiodic and sporadic release mechanisms. Figure 1 shows in more detail the periodic release mechanism extension, implemented by means of the protected interface `Periodic_Release`. At the root of release mechanisms, an abstract procedure `Wait_For_Next_Release` is provided. This procedure is called by the task once per activation and is in charge of triggering the next release of the task at the proper time.

Since the detection and notification of relevant events is optional and orthogonal with respect to the kind of release mechanism selected for a task, the original framework needs to provide four different classes per release mechanism ($2^{\text{nr of events}} \times \text{nr of release mechanisms}$). Namely, the original framework provides the following release mechanisms for periodic tasks: `Periodic_Release`, `Periodic_Release_With_Deadline_Miss`, `Periodic_Release_With_Overrun`, and `Periodic_Release_With_Deadline_Miss_And_Overrun`. Then the same amount of variants for sporadic tasks. The implementation of multiprocessor scheduling approaches (see section 3.2) requires an exponentially increased number of variants to cope with all the possible events to be handled. We consider that this is an important drawback for extensions to the original framework.

We must point out that the original framework was not aimed at targeting multiprocessor platforms, nor was it prepared to support modes and mode changes. Although we shall enumerate the requirements for these two cases in the next section, we briefly note here that, in the original framework, jobs[2] are not allowed to migrate through CPUs. We also note that, in the original framework, any event such as a deadline miss or execution time overrun will cause the task to be aborted and execute the corresponding handler. While this behavior may be appropriate for these two particular events, it is not flexible enough to accommodate a number of mode change protocols [14, 13].

It is also important to remark that the class hierarchy of the original framework is not compatible with the use of a code generator tool, as proposed here. For example, Listing 1 shows how the periodic release mechanism `M` and the real-time task `T` have to be declared after the programmer defines and declares the final task state `P`. With this code structure, a code generation tool can only set the scheduling attributes of a task in its `Initialize` procedure, and therefore it has to provide a task-specific `Periodic_Task_State`with this procedure already implemented. When the programmer extends this new task state to implement the task behavior, its initialization code would collide with the one that should be produced by the code generator.

Listing 1. A simple example of a periodic task using the original framework

```
-- with and use clauses omitted
package Periodic_Test is
  type My_State is new Periodic_Task_State with
  record
    I : Integer;
  end record;

  procedure Initialize(S: in out My_State);
  procedure Code (S: in out My_State);

  P : aliased My_State;
  M : aliased Periodic_Release(P'Access);
  T : Simple_Real_Time_Task(P'Access, M'Access, 3);
end Periodic_Test;
```

3. Framework requirements and system model

In the following subsections we enumerate the requirements we have considered to extend the original framework to support efficient execution on multiprocessor platforms and to incorporate the concept of operating modes. We first describe

[2]The term *job* refers to a particular activation of a task. Note that this concept is not directly supported by the Ada language.

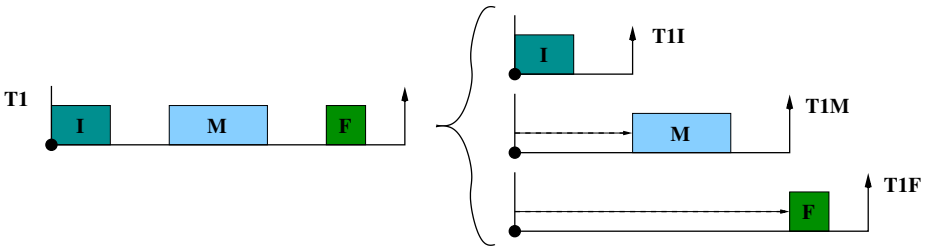


Figure 2. Decomposition of a task with multiple job steps.

the design context we are assuming, and then list the particular requirements for multiprocessor and multimode support we have considered. This set of requirements also serves the purpose of describing the system model we are assuming.

3.1. Design process requirements

The design and implementation of complex multiprocessor real-time systems requires using schedulability analysis techniques to ensure the system will meet its timing requirements at execution time. From the results of this analysis, the scheduling attributes are derived for each task in the system, across the different operating modes. The same applies to ceiling priorities of resources shared by means of protected operations under the ceiling locking protocol. These scheduling attributes may include multiple task priorities, relative deadlines, release offsets and task processor migrations at specified times. Additionally, the programmer may want to handle some special events, like deadline misses, mode changes and execution time overruns. Translating *manually* this set of attributes into the application code is error-prone, since it implies dealing with functional and non-functional properties at the same time in the code space. Hence this work proposes to use a specific development tool that will generate the scheduling task behavior and initialization code on top of the new real-time framework, leaving the purely functional behavior to the system programmer. However, the design and implementation of this tool is still work in progress and cannot be presented until the framework proposal reaches a sufficient degree of consolidation and agreement.

3.2. Multiprocessor requirements

A real-time system is composed of a set of *tasks* that concurrently collaborate to achieve a common goal. Each real-time task can be viewed as an infinite sequence of *job* executions. In many cases, a job performs its work in a single step without suspending itself during the execution. Therefore, a task is suspended only at the end of a job execution to wait for the next activation event. However, some systems organize the code as a sequence of *steps* that can be temporally spaced to achieve a given system goal. An example of such systems is the IMF model [3], oriented to control systems, where each job is divided into three steps or parts: an Initial part for data sampling, a Mandatory part for algorithm computation and a Final part to deliver actuation information. Although these steps usually share the job activation mechanism, different release offsets and priorities can be used for each step in order to reduce the input/output jitter of the sampling and actuation steps.

These job steps constitute the *code units* where the programmer will implement the behavior of each task. However, as pointed out in [18, 17], complex real-time systems could be composed by tasks that need to detect deadline misses, execution time overruns, minimum inter-arrival violations, etc. The system behavior when these situations are detected is task-specific and it has to be implemented in different code units in the form of task control handlers. An example of this task-specific behavior is a real-time control task with optional parts. These optional steps would help to improve control performance in case there is sufficient CPU time available, but they have to be cancelled if a deadline is in danger, in order to send the control action in time.

For the purpose of timing analysis, the steps are usually regarded as subtasks [3, 10]. The subtasks share the same release mechanism, typically periodic, and separate each job execution by a given release offset. Figure 2 shows the decomposition of a control task following the IMF model. The rest of the scheduling attributes of these notional subtasks (e.g. priority, or maximum CPU time allowed) are established according to a given goal, such as improving the overall control performance by reducing input/output jitter.

When and where a given code unit is executed is determined by the scheduler of the underlying operating system. The scheduler will use a set of *scheduling attributes* to determine which job is executed and, in multiprocessor platforms, on

which CPU. Examples of scheduling attributes are: release offset of the steps relative to the job activation, job priority, relative deadline, CPU affinity, worst-case execution time of the job, etc.

In a complex multiprocessor system, each job step can have a different set of scheduling attributes that could change during its execution depending on the selected scheduling approach. Based on the ability of a task to migrate from one processor to another, the scheduling approach can be:

Global scheduling: All tasks can be executed on any processor and after a preemption the current job can be resumed in a different processor. If the scheduling decisions are performed online, in a multiprocessor platform with M CPUs, the M active jobs with the highest priorities are the ones selected for execution. To ensure that online decisions will not jeopardize the real-time constraints of the system, different off-line schedulability tests can be applied [4, 5]. When the scheduling decisions are computed off-line, release times, preemption instants and processor affinities can be stored in a static scheduling plan.

Job partitioning: Each job activation of a given task can be executed on a different processor, but a given job cannot migrate during its execution. The processor where each job is executed can be decided by an online global dispatcher upon the job activation, or it can be determined off-line by a scheduling analysis tool and stored in a processor plan for each task. The job execution order on each processor is determined online by its own scheduler using the scheduling attributes of each job.

Task partitioning: All job activations of a given task are executed in the same processor. No job migration is allowed. The processor where a task is executed is part of the task's scheduling attributes. As in the previous approach, the order in which each job is executed on each processor is determined online by the scheduler of that processor.

Task splitting is a different technique that combines task partitioning with controlled task migration at specified times. Under this approach, some authors suggest to perform the processor migration of the split task at a given time after each job release [12] or when the job has performed a certain amount of execution [11]. It is worth noting that this approach normally requires the information about the processor migration instant to be somehow coded into the task behavior.

We note that, from the different techniques and approaches enumerated so far in this section, global scheduling imposes implementation requirements that need to be either fulfilled by the underlaying real-time operating system, or implemented at user level by means of some kind of application-defined scheduler [1]. But the rest of approaches (task and job partitioning, and task splitting) can be implemented by using features that are being considered for inclusion in Ada 2012. Some examples have recently being proposed in [7, 2, 16]. In particular, tasks will impose the following requirements to the framework for multiprocessor platforms:

- The ability to establish the tasks' scheduling attributes, including the CPU where each job will be executed. These scheduling attributes can be set at the task initialization phase to support task partitioning, but they can also be dynamically changed at the beginning of each job activation to provide support for job partitioning, or after a given amount of system or CPU time has elapsed, to provide support for task splitting techniques.
- The flexibility to program, and to be notified about, the occurrence of a wide set of runtime events. These events include: deadline miss, execution time overrun, mode change, timed events driven by the system clock or CPU clock to manage programed task migrations, etc. Some of these events may additionally require to terminate the current job.
- The possibility to specify time offsets in order to support the decomposition of tasks with multiple steps into several subtasks.

3.3. Mode change requirements

Multimoded real-time systems differ from single-mode systems in that the set of tasks to schedule changes with time. There exists one set of running tasks per mode of operation, each with the proper scheduling parameters (tasks periods, deadlines, priorities, etc.) Operating modes are decided at design time to accommodate the different situations to be faced during the system's mission.

A mode change request is a request for tasks to either:

- change their scheduling attributes (priority, deadline, period, affinity, etc.) to those of the new mode, or

- become active, if they were not active in the old mode, or
- become inactive, if they are not needed in the new mode, or
- preserve their scheduling attributes and activation pattern, if they remain unchanged from the old to the new mode.

When tasks share protected objects, it should also be possible to change the ceilings of those protected objects to accommodate the requirements of the new mode. Since Ada 2005, it is possible to dynamically change ceiling priorities of protected objects. Protocols for choosing the right time to change ceilings are proposed in [15]. Note that ceilings must be changed from a priority that is not above the current ceiling, to avoid violation of the ceiling locking protocol. Hence we cannot delegate the change of ceilings to a task with an arbitrarily high priority.

4. Framework proposal

Considering all the requirements described in Section 3, we propose now a redesign of the original framework. The following subsections describe the components of this new proposal.

4.1. Real-Time Task State, Real-Time Task Scheduling and Real-Time Task Attributes

The state variables of all tasks will be spread in three task state objects, containing different types of information.

Task_State Derives from the interface Task_State_Interface. It contains the task code in the form of four abstract or null procedures:

- Initialize: abstract procedure that must contain the user initialization code for the task;
- Code: abstract procedure that contains the task's functional code, to be executed at each new release of the task;
- Deadline_Miss: a null procedure to be redefined if the task needs to implement a deadline miss handler;
- Overrun: similarly to Deadline_Miss, this is the place to implement the execution-time overrun handler.

Task_Sched This type offers the following operations:

- Initialize. This is the abstract procedure where the user is required to set up the task's scheduling attributes (priority, deadline, period, CPU), define the events to be handled for that task and connect those events with their corresponding event handlers (deadline miss and overrun).
- Set_Task_Attributes. This procedure is defined as null in the Task_Sched type. It may be overriden by a not null procedure in order to reset the task attributes to the task's original values for the current operating mode. It is useful after a task split, or as the last action of a job when implementing job partitioning.
- Adjust_Job_Attributes. This procedure (null by default) is intended to dynamically change the current job attributes, for example for job migration or for supporting a dual priority scheme.
- Mode_Change_Handler. This is the user's handler for mode changes. A mode change may be handled in part by user's code, at the user's old-mode priority. This will, for example, allow the proper task to safely change the ceiling priorities to the new-mode values. This is also an appropriate place for the application code to perform device initialization for the new mode, if needed.

Task_Sched_Attributes This type offers the setter and getter subprograms for the individual task scheduling attributes, namely the execution time, relative deadline, priority, and offset. It also offers setters and getters for handling the *active* flag associated to each task. This flag informs the system about whether a task is currently running or waiting for activation.

4.2. Control Mechanisms

Adapting the original framework to support multiprocessor platforms and mode changes requires the framework to handle some *new* events. We propose to detach the event management from the release mechanisms, as proposed in the original framework. Hence the proposal of a new *control mechanism* abstraction.

A control mechanism is formed by a Control_Object and a Control_Event, that collaborate to implement the *Command* design pattern [8]. Control Objects will perform the *Invoker* role, that will ask to execute the *Command* implemented by the Control Event when some scheduling event occurs. The *Receiver* role is played by Task_State or Task_Sched types, while the *Client* role is performed by the initialization code that creates the event command and configures its receiver. Control objects allow triggering events at different times of the task's lifespan:

- *On job release*: This is useful, for example, to reset the deadline of a task scheduled under a deadline-based policy, such as EDF.
- *After a given amount of system time*: The use of Timing_Event allows, for example, to implement task splitting based on system time. The command executed by the control event will invoke the Adjust_Job_Attributes procedure of the Task_Sched object. Another example is to trigger the execution of a deadline miss handler.
- *After a given amount of CPU time*: Task splitting based on CPU time will use an execution time timer to trigger the appropriate event. Triggering a cost overrun handler is another example of an event controlled by a CPU timer.
- *On job completion*: Handling an event at the time of completion of a job allows to handle events whose handling must be deferred until the job completes its execution. This type of handler may change the task attributes before the next job activation occurs, e.g., the Mode_Change_Handler procedure could be used to change the priority and period of a task before reprogramming its next release event.

Each control object is complemented with a Control_Event. A control event only needs now to execute the designated event handler, provided either by the Task_State or the Task_Sched object, depending on the particular event to be handled (deadline miss, mode change, etc.). There are three classes of events: *immediate*, *abortive* and *deferred*, depending on when they will be handled. Immediate events are immediately dispatched when they are triggered. This is useful e.g. for implementing task splitting or dual priority. Abortive events may abort the execution of the task's code. Examples of use are deadline miss and cost overrun events. Finally, deferred events are handled only when the task is not running (in other words,when the task is waiting for the next release). This behaviour is adequate for handling mode changes because it lets the last old-mode job of the task to complete before adjusting the task parameters for the new mode.

4.3. Release Mechanisms

Release mechanisms are the abstractions used to enforce task activation at the proper times. We propose to use two kinds of release mechanisms:

- Release_Mechanism, as in the original framework but with some minor changes to support CPU affinities and release offsets. Specializations of this basic mechanism implement the periodic, sporadic and aperiodic release patterns.
- Release_Mechanism_With_Control_Object, almost identical to the former but invoking On_Release and On_Completion procedures of all registered control objects each time a job is released or completed. It also offers the notification operations Notify_Event, Pending_Event and Trigger_Event to add event management support. Listing 2 shows part of the specification and body of Periodic_Release_With_Control_Objects. The code for event handling procedures is shown separately in listing 3. As suggested in [16], the new Set_Next_CPU procedures of Timing_Event and Dispatching_Domains are used to avoid unnecessary context switches when a job finishes its execution in a different CPU than the next job release is going to use, e.g, due to the application of a job partitioning or task splitting scheme (see lines 38 and 62).

4.4. Task Templates

There are three types of tasks defined in the framework. One is the Simple_Real_Time_Task, that must be attached to a *simple* release mechanism (one with no control object associated) and whose body implements just an infinite loop with two sentences: a call to Wait_For_Next_Release and an invocation to the Code procedure defined in the task state. Not being attached to any control object, this type of task is insensitive to events other than its own releasing event.

The second type of task, shown in Listing 4, is the Real_Time_Task_With_Event_Termination. This type of task must be attached to a release mechanism with control object, and its body is more complex. During the time the task is waiting for its next release, it may be notified of a pending event (line 16) and handle it (line 17), going back to wait for the next release. In the absence of an event during the wait for the next release, the task enters an asynchronous transfer of control (ATC, a *select then abort* statement) where the task's code may be aborted in the event of a deadline miss or execution time overrun.

A third type of task is provided (Real_Time_Task_With_Deferred_Actions) that differs from tasks *with event termination* in that the task's code is never aborted. This is the type to use for tasks whose last job in a mode must not be aborted when a mode change request arrives.

Listing 2. Periodic Release Mechanism with Control Objects

```
1  -- spec
2    protected type Periodic_Release_With_Control_Objects
3         (A: Any_Periodic_Task_Attributes; NoC: Natural) is new Release_Mechanism_With_Control_Object with
4      entry Wait_For_Next_Release;
5      entry Notify_Event(E: out Any_Event);
6      entry Pending_Event(E: out Any_Event);
7      procedure Trigger_Event(E: in Any_Event);
8      procedure Set_Control(I: in Natural; C: in Any_Control_Interface);
9      procedure Cancel_Control(I: in Natural);
10     pragma Priority(System.Priority'Last);
11   private
12     entry Dispatch_Event(Postponed_Events)(E: out Any_Event);
13     procedure Release(TE: in out Timing_Event);
14     -- internal variables omitted
15   end Periodic_Release_With_Control_Objects;
16
17 -- body
18
19   protected body Periodic_Release_With_Control_Objects is
20
21     entry Wait_For_Next_Release when New_Release or not Completed is
22       Cancelled: Boolean;
23     begin
24       if First then          -- Release mechanism initialization
25         New_Release := False;
26
27         -- Initialize control objects
28         for I in 1 .. NoC loop
29           if Control_Objects(I) /= null then
30             Control_Objects(I).Initialize(Tid);
31           end if;
32         end loop;
33         if A.Task_Is_Active then
34           First := False;
35           Epoch_Support.Epoch.Get_Start_Time (Next);
36           Tid := Periodic_Release_With_Control_Objects.Wait_For_Next_Release'Caller;
37           Next_Release := Next + A.Get_Offset;
38           Event.Set_Next_CPU (A.Get_CPU);
39           Event.Set_Handler (Next_Release, Release'Access);
40         end if;
41         requeue Periodic_Release_With_Control_Objects.Wait_For_Next_Release;
42       elsif New_Release then  -- Job release
43         New_Release := False;
44         Completed := False;
45         -- On release control objects
46         for I in 1 .. NoC loop
47           if Control_Objects(I) /= null then
```

```
48              Control_Objects(I).On_Release(Next_Release);
49            end if;
50          end loop;
51        else                              -- Job completed or aborted
52          Completed := True;
53          for I in 1 .. NoC loop -- On completion control objects
54            if Control_Objects(I) /= null then
55              Control_Objects(I).On_Completion;
56            end if;
57          end loop;
58          if A.Task_Is_Active then
59            Next := Next + A.Get_Period;
60            Next_Release := Next + A.Get_Offset;
61            Event.Set_Next_CPU (A.Get_CPU);
62            Event.Set_Handler (Next_Release, Release'Access);
63          else
64            First := True;
65            Event.Cancel_Handler (Cancelled);
66          end if;
67          requeue Periodic_Release_With_Control_Objects.Wait_For_Next_Release;
68        end if;
69      end Wait_For_Next_Release;
70      ...
71      -- private
72
73      procedure Release (TE : in out Timing_Event) is
74      begin
75        New_Release := True;
76      end Release;
77
78    end Periodic_Release_With_Control_Objects;
```

Listing 3. Code of event handling subprograms

```
1   entry Notify_Event(E: out Any_Event) when Event_Occurred(Abortive) is
2   begin
3     requeue Dispatch_Event(Abortive);
4   end Notify_Event;
5
6   entry Pending_Event(E: out Any_Event) when Event_Occurred(Deferred) or Event_Occurred(Abortive) is
7   begin
8     if Event_Occurred(Abortive) then
9       requeue Dispatch_Event(Abortive);
10    else
11      requeue Dispatch_Event(Deferred);
12    end if;
13  end Pending_Event;
14
15  entry Dispatch_Event(for T in Postponed_Events)(E: out Any_Event) when True is
16  begin
17    E := Task_Event(T);
18    Event_Occurred(T) := False;
19    Completed := False; -- In case of abortion during WFNR
20  end Dispatch_Event;
21
22  procedure Trigger_Event(E: in Any_Event) is
23    ED : Event_Dispatching := E.Get_Event_Dispatching;
24  begin
25    case ED is
26      when Immediate =>
27        E.Dispatch;
28      when Postponed_Events =>
29        Task_Event(ED) := E;
30        Event_Occurred(ED) := True;
31    end case;
32  end Trigger_Event;
```

Listing 4. Task template for real-time tasks with event termination

```
--spec
  task type Real_Time_Task_With_Event_Termination (State: Any_Task_State_Interface;
                                                   Sched: Any_Task_Sched_Interface;
                                                   R: Any_Release_Mechanism_With_Control_Objects) is
  end Real_Time_Task_With_Event_Termination;

--body
  task body Real_Time_Task_With_Event_Termination is
    E : Any_Event;
    Released: Boolean;
  begin
    Sched.Initialize(State);
    State.Initialize;
    loop
      select
        R.Pending_Event(E);
        E.Dispatch;
        Released := False;
      then abort
        R.Wait_For_Next_Release;
        Released := True;
      end select;
      if Released then
        select
          R.Notify_Event(E);
          E.Dispatch;
        then abort
          State.Code;
        end select;
      end if;
    end loop;
  end Real_Time_Task_With_Event_Termination;
```

5. Propagation and handling of events: deadline miss example

Figure 3 shows the flow of events and actions under a deadline miss scenario. The framework entities involved in the process of handling the event (Task_State, Task_Sched, release mechanism, control object, timing event and control event) are shown in white-background rectangles. Vertical thin rectangles represent code executed in those entities. Everything starts on the top left corner of figure 3, when the Real_Time_Task starts and executes Initialize.

- During the task initialization, Task_Sched registers the control object DM_CO (of type Control_Object_With_System_Clock) with the task's release mechanism RM. The second call to Initialize refers to the initialization required by the task's logic, fully dependent on the application. In other words, this is user's code and not code related to any scheduling parameters of the task.
- Before the end of Wait_For_Next_Release, the On_Release procedure of the task's release mechanism RM is called. Within that procedure, the control object DM_CO programs a Timing_Event with the task's deadline before the task begins its execution.
- The task then enters the ATC with a call to Notify_Event, in the *select* part of the ATC, and an abortable call to State.Code. During the execution of State.Code, a deadline miss occurs (at the point marked as *Deadline timeout*, in figure 3) and the timing event executes the Event_Handler. The Event_Handler then triggers the event (via Trigger_Event) to the release mechanism, informing of the type of event (Deadline_Miss_Event).
- Since this is an *abortive* event, the barrier of Notify_Event becomes open, which aborts the call to State.Code and executes E.Disptach, which in turns ends up invoking the task's selected handler Task_State.Deadline_Miss.

The end of the loop makes the task go back to queue in Wait_For_Next_Release and the cycle starts over again.

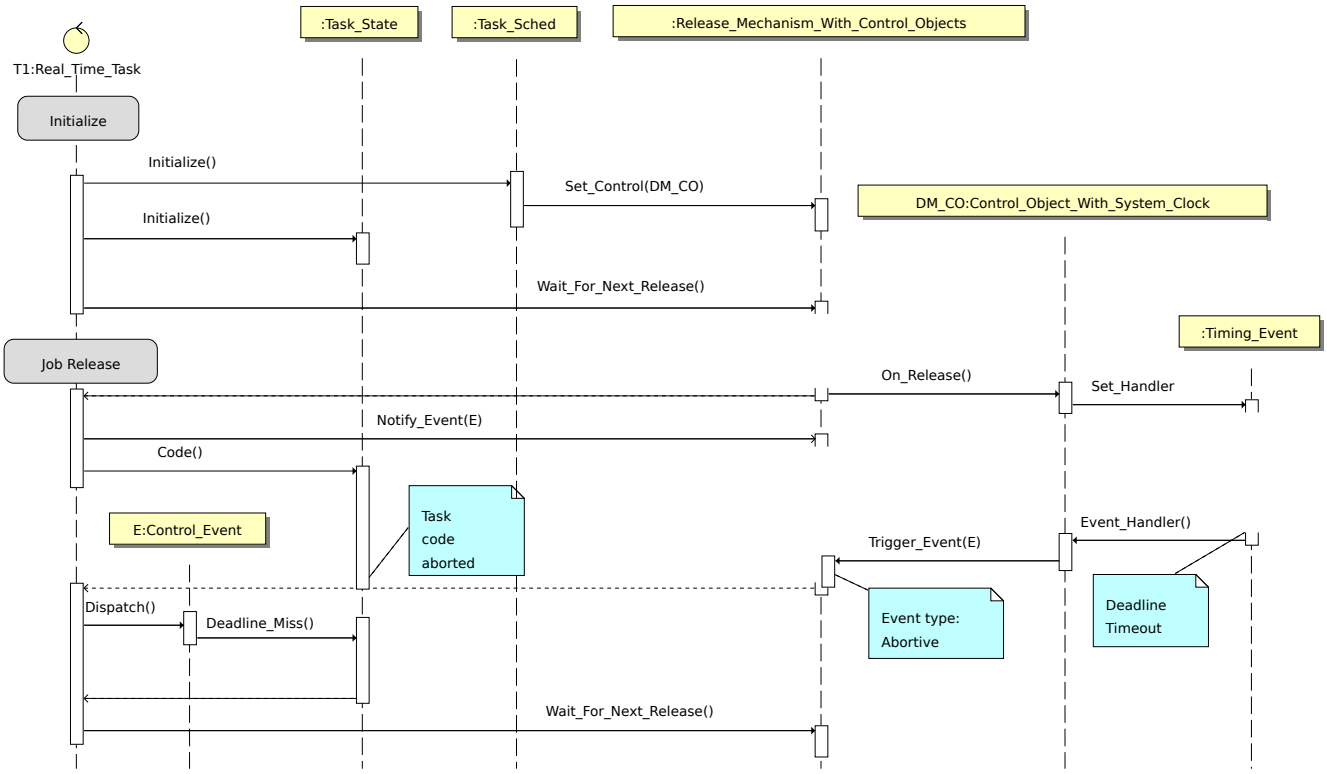


Figure 3. Control flow for a deadline miss scenario

6. Conclusions

This paper has given account of the main design principles for a new (actually a redesigned) framework of real-time utilities. Our goals were (i) to make the framework amenable to automatic code generation tools; (ii) to adapt the framework to support execution on multiprocessor platforms; and (iii) to incorporate the mechanisms to deal with modes and mode changes.

The use of control objects and the associated control events, and the range of possibilities of event handling (immediate, abortive, deferred), have contributed to the scalability of the framework with respect to the number of types of events to handle. There's no need anymore to implement an exponential number of different task patterns depending on the number of events and release mechanisms. Furthermore, the new design allows to handle events at different times during the lifespan of a job (on release, on completion, or after a certain amount of system or CPU time), which is useful for implementing different multiprocessor techniques (such as task splitting, or other *ad hoc* techniques) and mode changes.

We note however that this is work in progress and the proposal is not fully complete[3]. We need to complete the design with a mode manager (to receive mode change requests and redirect mode change events to all tasks) and we definitely need more testing and discussion around the proposed design.

References

[1] M. Aldea, J. Miranda, and M. González-Harbour. Implementing an Application-Defined Scheduling Framework for Ada Tasking. In A. Llamosí and A. Strohmeier, editors, *9th International Conference on Reliable Software Technologies – Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 283–296. Springer, 2004.

[3]The reader interested in the details of the code in their current state, may contact the authors to obtain the most updated version.

[2] B. Andersson and L. Pinho. Implementing Multicore Real-Time Scheduling Algorithms based on Task Splitting using Ada 2012. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 2010.

[3] P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo. A Task Model to Reduce Control Delays. *Real-Time Systems*, 27(3):215–236, September 2004.

[4] S. Baruah and T. P. Baker. Schedulability Analysis of Global EDF. *Real-Time Systems*, 38(3):223–235, April 2008.

[5] S. Baruah and N. Fisher. Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems. In *9th International Conference on Distributed Computing and Networking – ICDCN*, pages 215–226, 2008.

[6] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[7] A. Burns and A. Wellings. Dispatching Domains for Multiprocessor Platforms and their Representation in Ada. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2010.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Welsey, Reading, MA, 1995.

[9] M. González-Harbour and J. J. Gutiérrez. Session: Programming Patterns and Libraries. *Ada User Journal*, 29(1):44–46, March 2008.

[10] S. Hong, X. Hu, and M. Lemmon. Reducing Delay Jitter of Real-Time Control Tasks through Adaptive Deadline Adjustments. In IEEE Computer Society, editor, *22nd Euromicro Conference on Real-Time Systems – ECRTS*, pages 229–238, 2010.

[11] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In IEEE Computer Society, editor, *21st Euromicro Conference on Real-Time Systems - ECRTS*, pages 249–258, 2009.

[12] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In IEEE Computer Society, editor, *21st Euromicro Conference on Real-Time Systems - ECRTS*, pages 239–248, 2009.

[13] P. Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems*. Ph.D. thesis, University of York, Department of Computer Science, 1999.

[14] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

[15] J. Real, A. Crespo, A. Burns, and A. Wellings. Protected Ceiling Changes. *Ada Letters*, XXII(4):66–71, 2002.

[16] S. Sáez and A. Crespo. Preliminary Multiprocessor Support of Ada 2012 in GNU/Linux Systems. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2010.

[17] A. J. Wellings and A. Burns. A Framework for Real-Time Utilities for Ada 2005. *Ada Letters*, XXVII(2), August 2007.

[18] A. J. Wellings and A. Burns. A Framework for Real-Time Utilities for Ada 2005. *Ada User Journal*, 28(1):47–53, March 2008.

Ada Real-Time Services and Virtualization*

Juan Zamorano, Ángel Esquinas, Juan A. de la Puente
Universidad Politécnica de Madrid, Spain
jzamora,aesquina@datsi.fi.upm.es, jpuente@dit.upm.es

Abstract

Virtualization techniques have received increased attention in the field of embedded real-time systems. Such techniques provide a set of virtual machines that run on a single hardware platform, thus allowing several application programs to be executed as though they were running on separate machines, with isolated memory spaces and a fraction of the real processor time available to each of them.

This papers deals with some problems that arise when implementing real-time systems written in Ada on a virtual machine. The effects of virtualization on the performance of the Ada real-time services are analysed, and requirements for the virtualization layer are derived. Virtual-machine time services are also defined in order to properly support Ada real-time applications. The implementation of the ORK+ kernel on the XtratuM supervisor is used as an example.

1. Introduction

Virtualization techniques have raised significant interest in the embedded systems area. Virtualization enables a single hardware platform to be divided into a number of virtual machines, each of them providing a set of virtual resources that are mapped into the available physical resources. In this way, each virtual machine provides a partition for executing programs using a fraction of the physical processor time, memory capacity, and other devices. Since each partition is based on a virtual machine with a set of virtual devices, although with only a fraction of the capacity of the physical machine, it can host any kind of software organization, including different kinds operating systems or run-time kernels (figure 1).

Virtualization can provide temporal and spatial separation between partitions. Temporal separation means that each partition is guaranteed to have a fraction of the physical processor time, and no other partition can steal processor time from it. Spatial separation means that each partition is allocated a fraction of the global physical memory space in such a way that no other partition can access any part of it. In this way, applications running in different partitions are isolated from each other, and errors occurring in one of them cannot propagate to the others.

Virtualization has significant advantages for building complex embedded systems with high-integrity requirements, especially when there are subsystems with different levels of integrity. Isolation provides fault containment, and also simplifies the validation and verification process for high-integrity applications coexisting with lower integrity ones. It also enables more efficient fault detection and management techniques, and a better adaptation to the different system views that are commonplace in modern development methods. However, virtualization also creates new challenges, especially when real-time behaviour is considered. In addition to introducing some degree of execution-time overhead, multiplexing processor time among different partitions may undermine the temporal predictability of the applications. Both the implementation of the virtualization software layer and the application itself must be done in such a way that care is taken in order to keep temporal predictability, and to ensure that real-time applications running on virtual machines can be analysed for temporal correctness.

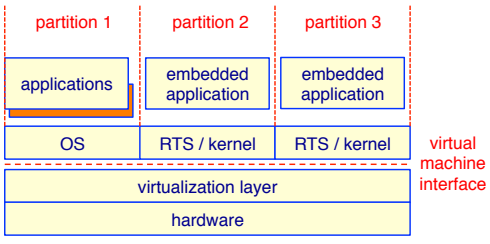


Figure 1. Virtualization and partitions.

*This work has been partially funded by MICINN, project TIN2008-06766-C03-01 RT-MODEL.

In the rest of the paper we analyse the effects of virtualization on real-time programs using the services defined in the Ada real-time annex [4, annex D]. Proposed Ada 2012 modifications are also discussed. Section 2 contains an overview of virtualization technology, with focus on bare machine supervisors. Ada real-time services are summarized in section 3, and the implications of implementing them on a virtual machine are analysed. Section 4 discusses the support needed from the virtualization layer in order to properly implement the Annex D real-time services. Finally, section 5 summarizes the porting of the ORK+ kernel to the XtratuM hypervisor as a case study. Conclusions and hints for future work are presented in section 6.

2. Overview of virtualization technology

2.1. Hypervisors

There are different approaches to virtualization, but not all of them are equally suitable for real-time systems, mostly for efficiency reasons. It is generally accepted that the best approach for embedded systems is based on the use of a hypervisor or virtual machine monitor [13]. A hypervisor is a layer of software that provides an execution environment in which several programs, including operating systems, can run in the same way as if they were executed on the real hardware. Type 1 or *bare-metal* hypervisors run directly on the physical hardware, usually in supervisor mode, whereas type 2 or *hosted* hypervisors run on top of an operating system. In order to get a predictable, efficient real-time behaviour, a bare-metal hypervisor is generally thought to be a better choice.

Hypervisors can work in two ways. When *full virtualization* is provided, the virtual machine interface is identical to the physical processor, and the code running in the partitions does not have to be modified. This requires hardware support which is not available on most embedded processors. *Paravirtualization* [6], on the other hand, is a technique in which the virtual machine is similar, but not identical, to the physical machine. This means that most machine instructions are executed on the real processor, but privileged instructions are replaced by system calls to the hypervisor. This technique requires changes in the guest operating system, but not in the application code.

2.2. Scheduling

In order to allow for real-time partitions to exhibit a predictable behaviour, as well as to ensure temporal isolation, processor time has to be multiplexed among the various partitions in a predictable way. A two-level hierarchical scheduling scheme is often used, where a *global scheduler* allocates processor time to partitions, and a *local scheduler* is used within each partition to choose a process or thread to run when the partition is active.

Different kinds of global and local scheduling policies can be used [12]. In the rest of the paper a static global scheduling policy is assumed, as specified in the ARINC 653 standard [3] and implemented in the current version of the XtratuM hypervisor [7]. The Ada real-time scheduling policies will be used at the local level for partitions running real-time Ada applications.

2.3. Interrupt management

One of the key elements of virtualization is interrupt management. Interrupts are handled by the virtualization layer, and virtual interrupts are dispatched to partitions in a similar way as conventional operating systems dispatch hardware events to processes (figure 2).

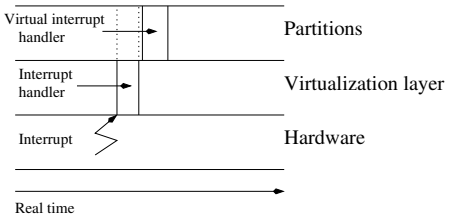


Figure 2. Immediate event notification.

It should be noticed that in this context the notification of events to the target partition may be delayed if the partition is inactive. In this case, the delivery of the virtual interrupt is delayed until the partition is active (figure 3).

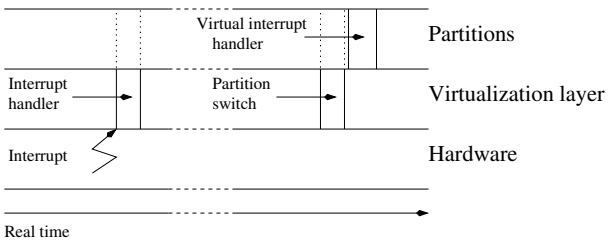


Figure 3. Delayed event notification.

It goes without saying that the dispatching of virtual interrupts has a direct impact on the performance of the real-time services provided by Ada partitions.

2.4. Virtualization interface

Hypervisors provide a virtualization interface that the guest operating systems or real-time kernels running in

partitions can use to replace the execution of privileged machine instructions. This interface usually takes the form of a set of system calls or *hypercalls*. Hypercalls give access to the hypervisor basic services: support for context switch, real-time clocks and timers, interrupt support, inter-partition communication, etc. In this way, the guest operating system can run in user mode, whereas the hypervisor is the only part of software running in supervisor mode.

In order to run Ada programs on a partition, the Ada run-time system has to be para-virtualized, i.e. it has to be modified so that the virtual hardware resources provided by hypercalls are used instead of the physical hardware.

3. Ada real-time services

3.1. Review of Ada real-time services

The Ada 2005 real-time services are specified in Appendix D of the ARM [4]. These services can be grouped into the following categories:

- Scheduling: priorities, dispatching policies, ceiling locking policy.
- Real time: real-time clock, delay until and timing events.
- Execution time: execution-time clocks and timers, group budgets.

The standard also defines the Ravenscar profile as a set of restrictions on tasking, including some restrictions on the above services.

The current Ada 2012 proposal [2] includes some additions and modifications, which can be summarized as follows:

- Support for multiple processors (AI05-0171-1); synchronous barriers (AI05-0174-1); group budgets for multiprocessors (AI05-0169-1); Ravenscar profile for multiprocessors (AI05-171-1).

  The analysis in this paper is restricted to monoprocessors, and therefore these real-time mechanisms will not be discussed.

- Improvements on real-time language features:
  - Scheduling: Yield for non-pre-emptive dispatching (AI-0166-1).
  - Execution time: monitoring the time spent in interrupt handlers (AI05-0170-1)
  - Fix for Ceiling_Locking with EDF (AI-055-1).
  - Extended suspension objects, usable with EDF (AI05-0168-1)

There are also some other minor fixes which will not be discussed here.

3.2. Impact of virtualization on Ada real-time services

Running Ada real-time programs on top of a virtualization layer raises some problems, which are derived from the differences between the virtual machine and the underlying physical machine. The various issues related to virtualization are discussed in the next paragraphs.

Scheduling. As explained in section 2 above, a two-level scheduling scheme is assumed. In order to be able to ensure the real-time behaviour of the applications, a global scheduling method with a predictable, bounded temporal behaviour must be used. Static cyclic scheduling, as specified by ARINC 653, provides such kind of behaviour and is thus a possible choice. More flexible approaches are also possible (see e.g. [12]).

The local scheduler is used to determine which task is dispatched to run in a partition. For a partition running an Ada program, any of the task dispatching policies defined in the real-time annex can be used. However, the implementation of context switches at the lowest level of the real-time kernel has to be modified as privileged instructions have to be replaced by hypervisor calls. This will generally result in longer context switch times.

A final remark is that the local scheduler can only dispatch a task for execution when the partition is active, which may significantly delay the response time of real-time tasks, as discussed in section 3.3 below.

Real-time clock, delays and timing events. The implementation of these services relies heavily on the underlying hardware [14], and therefore has to be modified when running on a virtual machine. In this case the physical hardware timers are handled by the hypervisor, and partition code has only access to them through hypervisor calls providing clock readings and virtual timers. Both virtual machine clocks and virtual interval timers refer to physical real-time, but virtual timer interrupts may be delayed if the timer expires when the partition is not active (see figure 3). As a result, tasks may suffer significant activation jitter, which has to be taken into account for temporal analysis.

Execution-time clocks and timers. CPU time is kept on physical processors by updating a per-task execution-time counter on every context switch according to the real-time clock value [14]. However, this implementation cannot be used when running on a virtual machine, as the time intervals during which the partition is not active should not be counted. Therefore, the execution-time clocks of all the tasks in a partition have to be stopped when the partition becomes inactive, and restarted when the partition becomes active again. This requires some support from the hypervisor. The best solution is to implement partition-time clocks in the hypervisor, on top of which the local real-time kernel can base the per-task execution-time clocks.

Monitoring the time spent in interrupt handlers. The way in which the execution time of interrupt handlers is kept is implementation-defined. A simple and legal implementation is to charge the time consumed by the interrupt handlers to the task that is running when the interrupt is generated. However, the Ada 2012 proposal allows an implementation to separately account for interrupt handling time. This case is considerably more complex, and for the moment this option is not recommended.

It should be noted that the execution time clock of any task can be charged with the execution time of any virtual interrupt delivered by the hypervisor to the partition, or even with the time spent in handling interrupts not related to the same partition. This may result in high inaccuracies in execution time measurement for any task, especially when there are non-real-time partitions with interrupts occurring at unpredictable times and unknown handling times.

Ravenscar profile. The Ravenscar profile is not affected by virtualization, except that the real-time behaviour of tasks may change due to the effects of running on a virtual machine.

3.3. Response-time analysis

Response time analysis for systems with hierarchical schedulers has been discussed by Almeida and Pereira [1], Davis and Burns [8], Pulido *et al.* [12], and Balbastre *et al.* [5], among others. Depending on the exact global and local scheduling methods that are used in a given system, a choice of techniques can be applied with various levels of accuracy. In any case, context switch and interrupt handling overheads, as well as other effects of virtualization on the temporal behaviour of the system, must be accounted for.

4. Required virtualization support

In this section we summarize the main features that the hypervisor implementing the virtualization layer must provide in order to support the execution of Ada real-time programs on one or more partitions, according to the discussion in section 3 above.

First of all, since we are assuming a paravirtualization approach, the hypervisor must provide hypercalls to replace all privileged instructions for the real processor architecture. This includes access to privileged registers, input/output instructions, interrupt support, and memory management, as well as any other processor-specific resources.

Real-time clocks and timers are basic resources for implementing the Ada real-time clock, delays and timing events. The hypervisor must provide a monotonic real-time clock base, and some timer mechanism based on it. Such basic mechanisms must be accessible by means of hypercalls, so that the Ada run-time system can use them to implement the Ada higher-level mechanisms.

Implementing execution-time clocks and timers requires a time base that only advances when a partition is active. The most efficient way to get it is that the hypervisor implements per-partition execution time clocks that measure the time spent in running each partition. Partition-time timers based on such clocks should also be implemented, and access to all of these mechanisms should be provided through appropriate hypercalls, as before.

If partition-time clocks are not provided at the hypercall level, the Ada run-time system must build an equivalent service based on lower-lever services provided by the hypervisor. For example, the hypervisor might deliver a virtual interrupt to a partition whenever it is activated. If the global scheduler is a static cyclic executive, knowledge of the minor and major cycle durations could then be used to compute the duration of the interval during which the partition is inactive. The Ada run-time system could then adjust a locally maintained partition-time clock by subtracting the duration of the inactive interval from the elapsed time.

Execution-time spent in interrupt handlers raises additional problems. The Ada run-time system can account for the time spent in virtual interrupt handlers that run in the partition, but not for the time used by lower-level interrupt handlers within the hypervisor. In order not to charge it to the active partition, the hypervisor should stop the partition-time clock during the execution of the interrupt handler. If there is no partition clock at the hypervisor level, the only possibility to prevent interrupt handling time from being erroneously attributed to the running task would be to notify the occurrence and

duration of interrupt handlers in such a way that the Ada run-time system could adjust its local partition clock.

Monitoring the time spent in interrupt handlers is even harder. At the lower level, partitions can be preempted by interrupts generated by timers or input/output services requested by other partitions. Moreover, some interrupts may be delivered to several partitions, e.g. those generated by hardware timers supporting timing events or delays. The only way time spent in interrupt handlers can be properly accounted for is to implement interrupt clocks within the hypervisor. To our knowledge this is not done by any current hypervisor implementation.

5. Case study

XtratuM [11] is an open-source bare-metal hypervisor for real-time embedded systems. It implements most of the above requirements, with the only exception of interrupt-time clocks. The global scheduler is a cyclic executive based on the ARINC 653 specification, supporting a variety of local operating systems at the partition level. We have ported the ORK+ kernel [14] to XtratuM using paravirtualization techniques on a LEON2 platform [10], an implementation of the SPARC V8 architecture. The ORK+/XtratuM kernel acts a guest partition operating system, on top of which a Ravenscar Ada application can run (figure 4).

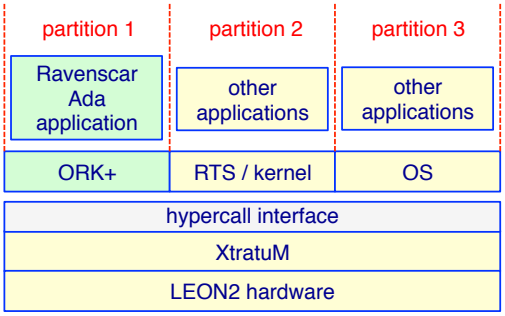


Figure 4. ORK+/XtratuM architecture.

The work is described in detail in another paper [9]. Paravirtualizing the kernel included adding a new Ada package for the hypercall interface, and modifying four more packages in order to replace privileged operations by XtratuM hypercalls. Overall, 1398 out of 7316 lines of code had to be modified, including the interface package and some low-level routines written in assembly language.

Evaluation experiments showed a low impact of virtualization on system performance. Although the total overhead for activating periodic tasks was found to be about 5 times the value for the original ORK+ running on a bare LEON2, the overall performance losses were found to be negligible for tasks with periods above 10 ms.

6. Conclusions and future work

The implementation of Ada real-time systems on virtual platforms has been analysed in the paper. The kind of virtualization kernel that has been taken as a reference is a bare-metal hypervisor with paravirtualization. Such kind of platform requires the Ada run-time system and the underlying real-time kernel to be modified so that it can run in user mode and access to physical devices is replaced by hypercalls giving access to virtual devices. The issues involved in such modifications have been analysed, and some requirements for the virtualization kernel have been derived. The approach has been applied to porting the ORK+ kernel to the XtratuM hypervisor. The porting has required only a moderate amount of effort and has given reasonable results in performance tests.

Planned future work includes doing a pilot implementation of interrupt time clocks on ORK+/XtratuM, and study more in depth the implications of virtualization on schedulability analysis. Another promising line is extending real-time virtualization concepts to multicore processor platforms.

Acknowledgments

The authors wish to acknowledge the fruitful collaboration with the XtratuM team at the Technical University of Valencia.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 95–103, New York, NY, USA, 2004. ACM Press.

[2] *ISO/IEC 8652:1995:AMD2(2007): Information Technology — Programming Languages — Ada — Amendment 2, draft 11*, 2011. Available on `http://www.ada-auth.org/amendment2.html`.

[3] ARINC. *Avionics Application Software Standard Interface — ARINC Specification 653-1*, October 2003.

[4] *ISO/IEC 8652:1995(E)/TC1(2000)/AMD1(2007): Information Technology — Programming Languages — Ada*.

[5] P. Balbastre, I. Ripoll, and A. Crespo. Exact response time analysis of hierarchical fixed-priority scheduling. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications — RTCSA 2009*, pages 315–320, 2009.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[7] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *Dependable Computing Conference (EDCC), 2010 European*, pages 67 –72, april 2010.

[8] R. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium — RTSS 2005*, 2005.

[9] A. Esquinas, J. Zamorano, J. A. de la Puente, M. Masmano, I. Ripoll, and A. Crespo. ORK+/XtratuM: An open partitioning platform for Ada. In A. Romanovsky and T. Vardanega, editors, *Reliable Software Technologies — Ada-Europe 2011*, number 6652 in LNCS, pages 160–173. Springer-Verlag, 2011.

[10] Gaisler Research. *LEON2 Processor User's Manual*, 2005.

[11] M. Masmano, I. Ripoll, A. Crespo, and J.-J. Metge. XtratuM: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, Dresden. Germany., 2009.

[12] J. A. Pulido, S. Urueña, J. Zamorano, T. Vardanega, and J. A. de la Puente. Hierarchical scheduling with Ada 2005. In L. M. Pinho and M. González-Harbour, editors, *Reliable Software Technologies — Ada-Europe 2006*, volume 4006 of *LNCS*. Springer Berlin/Heidelberg, 2006.

[13] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39 – 47, may 2005.

[14] S. Urueña, J. A. Pulido, J. Redondo, and J. Zamorano. Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Letters*, XXVII(2):61–66, August 2007. Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007).

Session Summary: Multiprocessor Issues, Part 1

Chair: Jorge Real
Rapporteur: José F. Ruiz

1 Introduction

The topic of multiprocessors was addressed by quite a number of position papers this year, so the whole first day of the IRTAW 15 workshop, while we were all fresh, was allocated to discussing multiprocessor issues. There was a discussion about dispatching domains and multiprocessor architectures during the morning, focusing on resource control protocols in the afternoon.

This summary addresses the morning session, whose goals were to review and evaluate the Ada 2012 support for multiprocessors, and think about possible additions to future (post 2012) language revisions. Specific issues discussed in this session were:

- The current definition of dispatching domains
- Per dispatching domain scheduling policies
- Dynamic dispatching domains
- Support for very large number of cores
- Non-SMP architectures
- Deferred attributes

The following sections will provide the details of the discussions around these subjects.

2 Dispatching domains in Ada 2012

Ada 2012 support for task affinities is provided by dispatching domains (Ada 2012 RM D.16.1), and there are already reference implementations [6] indicating that the current definition is appropriate and implementable on top of operating systems and kernels.

These early implementations showed an editorial error in the current definition of package *System.Multiprocessors.Dispatching_Domains*, which is intended to be *Preelaborate*, but it cannot be so because it depends on package *Ada.Real_Time* which is not *Preelaborate*. Alan Burns volunteered to submit this minor correction to ARG.

The definition of the *System_Dispatching_Domain* was found slightly misleading, because it is defined as constant (although it is implicitly modified by the creation of other dispatching domains), and it is not always a contiguous range as the other dispatching domains. Hence, for the *System_Dispatching_Domain*, *Get_First_CPU* and *Get_Last_CPU* do not make much sense, and *Assign_Task* cannot be used. However, these minor annoyances could probably be addressed better in the Rationale than by changes in the Reference Manual.

3 Dispatching domain's scheduling policy

The position paper submitted by Alan Burns and Andy Wellings [2] to last IRTAW 14 advocated for a more flexible definition of dispatching domains, where one could assign specific scheduling policies to each dispatching domains.

The discussion around this feature showed that this behavior can be achieved by carefully using the existing support for dispatching domains; if the priority range is partitioned into non-overlapping priority bands, each of these bands to be used only by tasks allocated to the same dispatching domain, we can set specific dispatching policies for the different ranges of priorities (and hence for each dispatching domain).

Therefore, there was not a strong motivation for trying to push forward this feature.

4 Dynamic dispatching domains

Ada 2012 defines dispatching domains as static entities. Therefore, mode changes involving migration of CPUs from one dispatching domain to another cannot be implemented. Reconfiguring dispatching domains when the underlying hardware changes (CPUs being added or removed to/from an Ada partition) is not supported either.

It would be interesting to support dynamic dispatching domains to address these situations. Note that the current Ada 2012 model assumes that the set of processors available to an Ada partition remains constant for its whole lifetime, so addressing changes to the hardware is outside the intention in the language. Still, the existing static support is not flexible enough for some kinds of mode changes which would be desirable.

The position papers discussed in the workshop did not address this functionality, so after discussing for a while its implications it was decided to encourage the submission of concrete proposals for the next workshop, motivating the need and detailing its design.

5 Very large number of cores

Luís Miguel Pinho started this discussion by stating that Ada is not fit for a large number of cores and we should start thinking about how to address this limitation. Tasks and protected objects are too heavyweight, in terms of time to create and destroy, context switch, synchronization protocol, etc.

When there are many CPUs available, what would be desirable is to have the notion of "parallel" activity, which could be functions (without side effects), blocks, loops, or others.

This notion of user-level fine-grained parallelism is already present in many programming languages and libraries. In ParaSail [8], language semantics are parallel by default, and subprograms, loops, statements and some other elements can be executed in parallel. Intel's Cilk [1] extends the C language with primitives to express parallelism, and the Cilk run-time system maps the expressed parallelism into a parallel execution framework. Cilk provides keywords to indicate how subprograms can be executed in parallel. Hence, there exist today different languages, libraries and APIs to express parallelism in a program. The field of automatic parallelization by compilers or tools has not yet been proved as very efficient, leaving user-level description of parallelism as the only means to achieve highly parallel applications effectively.

Therefore, the workshop identified this topic as a relevant one. Ada has been prominent in treating concurrent units (tasks) as first-class citizens in the language, and it would be good to address better support for parallelism. The submission of proposals about this subject is strongly encouraged for the next workshop.

6 Non-SMP architectures

Ada 2012 support for multiprocessor architectures focuses on Symmetric MultiProcessor (SMP) architectures, where all processors are identical and access to main memory is uniform (caching aside). In the Ada model a partition is a single address space.

At the last IRTAW 14 workshop there was a position paper by Andy Wellings et al. [9] addressing the difficulties to handle this kind of hardware architectures. The main issues explained there:

- Understanding the address map. The first requirement is to be able to represent the platform. The goal is to include the notion of locality and distance (in number of hops) of a processor from a memory bank servicing a particular address, so that tasks can allocate the objects they use in a "close" location. It would require to add the concepts of memory bank and distance to Ada. Looking at standard ways

to describe memory topologies, it was suggested to look at Portable Hardware Locality (hwloc) [4], which provides a portable abstraction of the hierarchical hardware topology. It would be useful to have an interface to a library to get the hardware representation.

- Using storage pools to partition the heap. The idea is to allow users to allocate objects at addresses where they can be accessed more efficiently, for example by choosing a local memory bank. Storage pools appear as the obviuos choice to do it, but the issue is that you cannot set the address of a storage pool; Ada lets you partition the heap into separate storage pools, but you cannot specify their address location. The workshop agreed it would be interesting to be able to specify the address attribute of a storage pool. POSIX has typed memory facilities [5] to control the allocation of memory pools.

Hence, it was considered appealing to add support to represent memory topologies, and to be able to use this information to allocate objects in an informed manner.

We considered as well whether it would be good to have the means to indicate that an object is only used by tasks executing on the same processor. This information could be used to indicate that all those tasks share the same cache, and hence enforcement of cache coherence would not be needed. However, we realized that this is not an issue because the hardware would enforce cache coherence only when needed, and if two tasks are in the same CPU, data can remain in cache without being written to memory (it would be written to memory if another core tries to read/write this memory area).

7 Deferred attributes

During this session, Sergio Sáez stated the existing limitations of the current model of setting attributes (priority, deadline and affinity). Possible ways to address this issue where discussed in the session about "Concurrency Issues" [3].

Priority, deadline and affinity can be changed dynamically (*Set_{Priority,Deadline,CPU}*) and all of these are dispatching points. Ada 2012 allows you to change deadline and affinity after a delay (*Delay_Until_And_Set_{Deadline,CPU}*), but it is not possible to set more than one of these attributes atomically. When these task attributes are not changed atomically, some scheduling artifacts could arise giving rise to incorrect schedules [7]. We agreed that we want to be able to atomically change task's attributes to avoid unintended effects.

Possible solutions using timing events or protected objects to perform these changes atomically were discussed. However, they could not meet the requirements when changing another task's attributes.

The addition of subprograms for setting attributes at the next dispatching point (such as *Set_Next_{Priority, Deadline,CPU}*) was proposed and discussed in more detail during the "Concurrency Issues" session.

8 Conclusion

There was a very interesting and constructive discussion during all the morning session (continued during the afternoon and the rest of the workshop) about the challenges raised by the new multiprocessor architectures.

The first important outcome to point out is that the current support for multiprocessors in Ada 2012 (dispatching domains) is considered very appropriate. In addition to that, it matches rather smoothly the support typically offered by multiprocessor operating systems.

Two mechanisms were explored to make dispatching domains more flexible. One was the possibility of setting dispatching domain's scheduling policies, which was deemed not very important since it can be achieved with the current support. The second was to migrate CPUs among dispatching domains, and was considered interesting for implementing mode changes. This workshop encouraged the submission of proposals addressing these dynamic dispatching domains.

Another subject considered very appealing for future workshops was the support for user-level fine-grained parallelism. As more and more CPUs are made available, the notion of task is perhaps not the right abstraction, with respect to controlling parallelism with smaller granularity.

It was also agreed that Ada provides limited expressive power for non-SMP architectures, and it would be good to add more control over the physical address map.

Finally, better support for deferred attributes was motivated during this morning session, but lunch time deferred the discussion of the required interface until a later session ...

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[2] Alan Burns and Andy J. Wellings. Supporting execution on multiprocessor platforms. *Ada Letters*, XXX(1):16–25, April 2010.

[3] Juan A. de la Puente and Stephen Michell. Session summary: Concurrency issues. *Ada Letters*, This issue, 2011.

[4] Portable hardware locality (hwloc). Available at `http://www.open-mpi.org/projects/hwloc`.

[5] IEEE. *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Advanced Realtime Extensions [C Language]*, 2000. 1003.1j-2000.

[6] José F. Ruiz. Going real-time with Ada 2012 and GNAT. *Ada Letters*, This issue, 2011.

[7] Sergio Sáez and Alfons Crespo. Deferred setting of scheduling attributes in ada 2012. *Ada Letters*, This issue, 2011.

[8] Tucker Taft. Multicore programming in ParaSail. In A. Romanovsky and T. Vardanega, editors, *Ada-Europe 2011, 16th International Conference on Reliable Software Technologies*, volume 6652 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011.

[9] Andy Wellings, Abdul H Malik, Neil Audsley, and Alan Burns. Ada and cc-NUMA architectures: What can be achieved with Ada 2005? *Ada Letters*, XXX(1):125–134, April 2010.

Session Summary: Multiprocessor Issues, part 2 (resource control protocols)

Chair: Andy Wellings

Rapporteur: Luís Miguel Pinho

1 Introduction

The second session on the topic of Multiprocessor Issues took place after a post-lunch relaxing walk on *Picos de Europa*, 800 meters above the workshop location and more than 1800 meters above the sea level. Indeed a stimulating activity to foster a productive session.

The goals of this session were:

A. To review and evaluate the efficacy of the Ada 2012 support in the area of multiprocessor resource control
   - The first topic being to analyze if the Reference Manual (RM) for Ada 2012 had been fully updated concerning priority inheritance issues, given that Ada 2012 allows tasks to be fully partitioned, globally scheduled or scheduled in disjoint domains;
   - A second topic was the wording of priority inheritance for the case of Earliest Deadline Scheduling in multiprocessors, where a potential issue had been put forward by [1].
   - The third topic was to evaluate whether Protected Objects should be used only for local resource access or if new access protocols were needed to allow it to be used globally without the need for spin-locks, following a position paper on the topic [1];

B. To look beyond Protected Objects and Rendezvous to other paradigms amenable to be used in multiprocessor platforms, for instance Software Transactional Memory (STM) or wait-free queues
   - The objective was to analyze to what extent the Ada primitives are suitable to implement these new paradigms and if secondary standards were the appropriate mechanism to introduce these paradigms into Ada;
   - In particular, STM would be analyzed in more detail as there was a position paper on the topic [2].

C. To review previous workshops proposals of new synchronization primitives to improve parallel execution of Ada programs.
   - Following a proposal from [3] the workshop decided to revisit broadcast primitives for calling PO subprograms in parallel [4, 5] and the use of parallel barriers in Protected Objects [4, 6].

2 Ada 2012 support in the area of multiprocessor resource control

2.1 Review of priority inheritance and task partitioning

In order to provide a basis for discussion, the Session Chair started by presenting an overview of the specification of priority inheritance (Ada 2012 RM D.1):

- *During activation, a task being activated inherits the active priority that its activator (see 9.2) had at the time the activation was initiated.*
- *During rendezvous, the task accepting the entry call inherits the priority of the entry call (see 9.5.3 and D.4).*

- *During a protected action on a protected object, a task inherits the ceiling priority of the protected object (see 9.5 and D.3).*

The chair then introduced into the discussion if the meaning of priority inheritance would still hold in a fully partitioned system (when the task inheriting the priority is on a different processor to the task which is waiting) or in a cluster-based system (when the task inheriting the priority is in a different dispatching domain to the task which is waiting). Would inheritance of priorities among domains be meaningful, if tasks were interacting in two different dispatching domains, using different priority ranges?

A note was nevertheless made that the current Ada model, albeit allowing multiple dispatching domains, imposes a single scheduling policy in all domains. Therefore, after several rounds of discussion, there was an agreement that priority assignments in all processors should be globally coherent. If that approach is followed, inheritance of a priority from one processor to the other is always correct.

2.2 EDF, priority inheritance and Multiprocessors

In this short topic, [1] noted a potential misleading wording of paragraph 26 in section D.2.6, that could allow programs to behave incorrectly:

> *For a task T to which policy EDF_Across_Priorities applies, the base priority is not a source of priority inheritance; the active priority when first activated or while it is blocked is defined as the maximum of the following:*
>
> *[…]*
>
> - *the highest priority P, if any, less than the base priority of T such that one or more tasks are executing within a protected object with ceiling priority P and task T has an earlier deadline than all such tasks; and furthermore T has an earlier deadline than all other tasks on ready queues with priorities in the given EDF_Across_Priorities range that are strictly less than P.*

The workshop discussed the issue, and although there was not an agreement that the sentence was incorrect, it was agreed that Alan Burns would further analyze the issue after the workshop to check if the wording would need to be clarified.

2.3 Access protocols for Protected Objects in multiprocessors

In the third topic of the session, the Chair started by providing an overview of the Reference Manual wordings concerning the access and control protocols for Protected Objects, noting that both the RM and the Annotated Reference Manual (ARM) do not fully define the access protocol for a protected object on a multiprocessor system.

For instance, Note 19, Section 9.5.1 states:

- *If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-defined ordering or queuing presumed for tasks competing to start a protected action on a multiprocessor such tasks might use busy-waiting; for monoprocessor considerations, see D.3, "Priority Ceiling Locking".*

  Discussion: The intended implementation on a multi-processor is in terms of "spin locks" – the waiting task will spin."

While in D.2.1, paragraph 3, it states:

- *It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.*

Thus it is implementation defined whether spinning occurs non-preemptively or, if not, at what priority it is performed. Furthermore, it is not defined whether there are queues (FIFO or priority) associated with the spin-lock. The workshop agreed that this could pose a potential problem to analyze maximum blocking times.

Afterwards, Andy Wellings performed a review of the most current used shared data protocols for multiprocessor systems, analyzing if these could be implemented in Ada. From the perform analysis, the conclusion was the

majority of these protocols could not be used, as they were based in suspending contending tasks in FIFO or Priority Queues, something that was disallowed in Ada.

At this point Michael Gonzalez noted that there was nothing in Ada preventing an implementation based on suspending on the lock. Andy noted that the RM states that access control in multiprocessors should be done with spin-locks. After a brief discussion, the general view was that although the RM promoted the spin-lock model, it allowed implementations to provide alternative implementations.

Alan Burns then proposed that if there was a recognized good solution for access control protocols based in suspension in the lock, the RM wording could be changed in the next revision process.

The session then went to analyze if it would be useful to allow the programmer the ability to change the underlying locking code (thus controlling the implementation). That would allow trying different protocols or to use the best fit for a particular applications.

Both Alan Burns and Tullio Vardanega noted that the current model in Ada is the result of many years of research which provided a sound access model for the monoprocessor case. The current work for the multiprocessor case was still far from that and there was no clear winner at the moment. The workshop came back to the issue of the suitability of the concurrency model of Ada for multiprocessors, but the general feeling was that there was yet no general agreement on what the model would be and what protocols to support.

A note was also made by José Ruiz that usually the access model of the Protected Objects uses mechanisms which are provided by the underlying Operating System. It would not be possible to give the programmer direct access to those. However, Andy noted that the idea being proposed was different, in that the approach would be to provide applications with an interface to specify application protocols that the runtime could used instead of the one provided by the underlying operating system.

Then Andy presented a proposal [1] of an API that allowed to control and extend the queue locks, and implement the access control protocols:

```
package System.Multiprocessors.Queue_Locks is
  type Queue_Order is (FIFO_Ordered, Priority_Ordered);
  type Spinning_Priority is (Active_Priority_Of_Task,
                                  Non_Preemptively);
  type Spin_Lock(
          Length : Positive := 1;
           Order : Queue_Order := FIFO_Ordered;
           At_Pri : Spinning_Priority := Non_Preemptively)
           is private;
  function Acquire(L : Spin_Lock) return Boolean
  procedure Release(L : Spin_Lock);

  type Suspension_Lock(
            Length : Positive := 1;
            Order : Queue_Order := FIFO_Ordered )
          is private;

  function Acquire(L : Suspension_Lock)  return Boolean;
  function Remove_Head(L : Suspension_Lock) return Task_Id;
  procedure Add(L : Suspension_Lock; T : Task_Id);
  function Highest_Queued_Priority(L : Suspension_Lock)
       return Any_Priority;
  procedure Release(L :  Suspension_Lock);

private
```

```
   ...
end System.Multiprocessors.Queue_Locks;


package System.Multiprocessors.Protected_Object_Access is
  type Lock_Type is (Read, Write);
  type Lock_Visibility is (Local, Global);
  type Protected_Controlled is new Limited_Controlled with private;
  overriding procedure Initialize (C : in out Protected_Controlled);
  overriding procedure Finalize (C : in out Protected_Controlled);
  procedure Lock (C : in out Protected_Controlled; L : Lock_Type;
                        V : Lock_Visibility;  Ceiling : Priority;
                        Tid :  Task_Id);
  procedure Unlock (C : in out Protected_Controlled; Tid :  Task_Id);
private
 ...
end System.Multiprocessors.Protected_Object_Access;
```

Simultaneously an Aspect in the Protected Objects would allow the programmer to specify that access protocol was user defined:

```
protected type X (PC : access Protected_Controlled) with
      Locking_Policy => (User_Protected,
                        Lock_Visibility => Global,
                        Locking_Algorithm => PC) is
  procedure A;
end X;
```

Several issues were still open, such as if the proposed functionality would be sufficient to build the current and future protocols or if spin-locks would still be predefined, but the advantage of this model was allowing to use better algorithms (and test new ones) and still be allowed to use Protected Objects as the data sharing mechanism. Furthermore, compilers can reject or ignore unsupported aspects/pragmas.

To summarize, two approaches could be made available to allow programmers to specify access protocols:

- One would be to use low-level abstractions (such as locks), being Protected Objects not used for multiprocessors;
- The second being Protected Objects augmented with user defined access protocols.

The general agreement of the workshop was that the latter would be the better approach. The workshop view is that this is a worthwhile idea that should be further exploited and presented in the next IRTAW.

3 Looking beyond Protected Objects

3.1 Software Transactional Memory

In the second part of the session, Miguel Pinho started by presenting an overview of Transactional Memory (TM), providing a quick overview of how in this approach atomic sections are executed concurrently and speculatively, in isolation. In particular, Miguel briefly explained how transactions worked with multiple versions of the data objects, allowing increasing the parallelization of execution.

Transactional Memory provides a higher-abstraction to programmers, which can focus on the algorithms, writing the sequential code and identifying the atomic sections. The underlying TM mechanism then controls the concurrent interaction of the transactions.

Studies show that TM is an alternative to locks for larger number of cores, particularly under low contention, when there is a predominance of read-only or short running transactions and there is a low ratio of pre-emptions during its execution. Nevertheless, TM suffers from extra overheads, such as the time needed to access data (it is not accessed directly and updates may be aborted) and the extra memory needed for having multiple versions of the data.

A proposal was being made to support Software TM, as, although less efficient that its hardware counterpart, it was more flexible and more implementations and research was being performed. The proposal was for a standard API which would allow programs to be independent of particular STM implementations and algorithms, and that would also allow to test new contention protocols.

Miguel then presented an example of a potential (conceptual) implementation from [2], currently only addressing non-nested transactions:

```
-- we need a transaction identifier structure
My_Transaction : Transaction;
-- start an update transaction
My_Transaction.Start(Update);
loop
  -- read a value from a transactional object
  x := trans_object_1.Get(My_Transaction);
  -- write a value to a transactional object
  trans_object_2.Set(My_Transaction, y);
  -- try to commit transaction
  exit when My_Transaction.Commit;
exception
  -- handle possible exceptions here...
end loop;


-- Transactional object
package Transactional_Objects is
  type Transactional_Object is tagged private;
  -- examples of transactional class methods
  procedure Set(T_Object: Transactional_Object;
                Transaction_ID : Transaction;
                Value : Object_Type);
  function Get(T_Object: Transactional_Object;
               Transaction_ID : Transaction)
           return Object_Type;
private
  type Transactional_Object is tagged
  record
    Current_Value : Object_Type;
```

```
    Accesses_Set : <list of pointers to transaction identifiers>
    -- some other relevant fields...
  end record;
end Transactional_Objects;

type Transaction_Type is (Read_Only, Update);
-- Transaction identifier
package Transactions is
  type Transaction is tagged private;
  procedure Start(T : Transaction;
                  TRX_Type : Transaction_Type);
  procedure Abort(T : Transaction);
  procedure Terminate(T : Transaction);
  function Commit(T : Transaction)
           return Boolean;
private
  type Transaction is tagged
  record
    Data_Set : List_Ref_Transactional_Objects;
  end record;
end Transactions;
```

In the following discussion, the workshop made a note that the initial read snapshots and the final commit operations would need to be actually performed in memory, so transactional data (only the original not the multi-versions) would need to be volatile.

A doubt was also raised if the implementation of STM would require changes to the language or if standard Ada provides all mechanisms to allow such implementation. It was concluded that the proposed API model is possible to implement, but other models (e.g. declaring transactional objects at the language level) could be interesting to address. It would be also important to incorporate in the proposal the capability for the user to implement its own contention control algorithm.

As a general conclusion, the work was encouraged by the workshop, and it was considered that a prototype implementation would be important to evaluate the proposal. Steve Michell then raised the issue that if a solution was achieved it could be released as a Technical Report or Technical Specification so that initial implementation and evaluations could be performed. Nevertheless, Joyce Tokar noted that it would not be possible if syntactic changes to the language were required – this can only be done through the RM.

4 Mechanisms for improved parallelism

4.1 Broadcast of operations to an array of POs

The first topic in the third part of the session concerned in proposal that was made at IRTAW 13 [4] to support a parallel broadcast of calls to an array of Protected Objects:

```
protected type po_type is
```

```
  procedure call(val : integer);
end po_type;

po_array : array (1..10) of po_type;
po_array(7).call(37);          -- single instance call
po_array.all.call(25);         -- broadcast to all elements of po array
po_array(1..10).call(25);      -- alternative broadcast to all
                               -- elements of po array
po_array(2..5).call(13);       -- broadcast to restricted range of elements
```

One issue that was debated was how the model could allow broadcasts with different data for each actual object. A proposal was also made to consider the use of synchronized interfaces to provide more flexibility to the broadcast.

Afterwards, the usefulness of this mechanism was largely discussed, particularly because the parallel calls would require some execution context (thus it related to the discussion on the previous session on the Ada parallel model). The initial proposal had been performed in the context of direct compilation of Ada code to hardware where these parallel calls could be directly mapped in the FPGA.

The final conclusion was that this was not considered to be currently needed.

4.2 Parallel barriers in functions on POs

The final topic of the session was to revisit the use of parallel barriers in functions within Protected Objects. In the previous workshop the incorporation of parallel barriers had been considerably discussed. The discussion had been separated in both defining a simple task parallel barrier mechanism similar to suspension objects, and allowing tasks in Protected Objects special entries to be released in parallel. Although the former has made it to Ada 2012 (Synchronous Barriers), the latter, although several rounds of discussion and multiple proposals being made (see [6]), had not reached a conclusion.

A note was made that one of the difficulties with implementing this type of barriers was the data passing issue. POSIX provides simple barriers, because some hardware platforms directly support them. Nevertheless, this does not include data sharing, so this would be difficult to implement.

Finally, the workshop agreed that it would be good to have a model similar to barriers, but more generic and allowing more complex interactions than Synchronous Barriers (and data passing), but that a suitable approach needs further investigation.

5 Conclusions

The session was mainly devoted to analyze how the ubiquitous multicore and multiprocessor platforms impacted the Ada mode for resource control among tasks:

- In the first topic analyzed (priority inheritance and task partitioning) the workshop concluded that the assignment of priorities in partitioned approaches had to be globally coherent and that in this case, priority inheritance between tasks in different domains was always correct.
- In the second topic (priority inheritance under EDF), the workshop felt that there was a possible misleading wording of the behaviour of priority inheritance under EDF, and it was decided to analyzed this further to propose clarification if required.
- In the third topic (access protocols for Protected Objects) the workshop considered important that users are given an interface to control and define different access protocols than simple spin-locks. Further work in this topic is encouraged.
- In the fourth topic (Software Transactional Memory), the workshop concluded that work on other paradigms for concurrency interaction with larger number of cores was important. Further work in this topic is encouraged.

- In the fifth topic (broadcast calls for Protected Object arrays), the workshop considered that currently there was not a need for this mechanism.
- Finally, in the last topic (parallel barriers in Protected Objects), the workshop concluded that this would be a good mechanism to have, but that a suitable approach needs further investigation.

References

[1] S. Lin, A.J. Wellings and A. Burns. *Ada 2012, Resource Sharing and Multiprocessors*. This issue.

[2] A. Barros and L. M. Pinho. *Revisiting Transactions in Ada*. This issue.

[3] A Burns and A. J. Wellings. *Support for Multiprocessor Platforms.* This issue.

[4] M. Ward and N. C. Audsley. *Suggestions for stream based parallel systems in Ada*. Ada Letters - Proceedings of the 13th International Real-Time Ada Workshop, XXVII(2), 2007.

[5] J. Real and S. Mitchell. *Beyond Ada 2005 session report*. Ada Letters - Proceedings of the 13th International Real-Time Ada Workshop, XXVII(2), 2007.

[6] T. Vardanega, M. González Harbour, L. M. Pinho, *Session Summary: Language and Distribution Issues.* Ada Letters – Proceedings of the 14th International Real-Time Ada Workshop, XXX(1), April 2010

Session Summary: Language Profile and Application Frameworks

Chair: Alan Burns

Rapporteur: Tullio Vardanega

1 Introduction

The Chair's introduction enumerates the issues raised by the papers assigned to the session:

- Beyond Ravenscar – other profiles
- Ravenscar and distribution
- Ravenscar and EDF
- Code archetypes for Ravenscar
- Real-time framework – dealing with multiprocessors and mode changes.

The initial group's perception is that the attention should focus first on the discussion of new language profiles, which is bound to require the largest fraction of the time duration of the session. The group agrees to this proposal, and the Chair presents the highlights of the profile proposal that Burns and Wellings made in [1].

2 Language profiles beyond Ravenscar

An element of the rationale for looking beyond Ravenscar is to avoid the feature creep phenomenon that may diminish the distinctive nature – and the measurable success – of the Ravenscar Profile (RP). The existence of the full language and its wealth of features should be considered to specify new profiles with an expressive power not necessarily close to the end of the RP.

The group understands that there exist two fundamental needs behind language profiles of interest to IRTAW, and to its constituency: (a) to have a coherent set of functionalities; (b) to warrant ease and efficiency of implementation, and, possibly, amenability to certification, although not necessarily to the highest level.

Andy Wellings illustrates the profile proposal made in [1]. The envisioned profile has a twofold motivation:

(1) To gain the ability to tolerate timing faults, which Ravenscar is poorly equipped for, since its fundamental strength is the assurance of absence of them.

(2) To address the greater uncertainty in timing analysis typical of multicore computing.

The direction taken by Burns and Wellings proposal is to incorporate in the profile sufficient means for software dynamic fault tolerance (as per Anderson and Lee's model in [2]): error detection, damage confinement and assessment, error recovery, fault treatment and continued service.

The discussion reviews some variants of fault-error-failure chains, all essentially based on the following, recurrent causal chain:

- Error: WCET overrun or blocking duration overrun (as a ramification of WCET underestimation).
- Error propagation: deadline miss.
- Failure: untimely delivery of service.

The detection means that can be deployed to counter the above chain range from the extreme of deadline miss detection – which however is not a necessary consequence of error propagation, perhaps because of slack capacity available due to less frequent arrival of sporadic tasks in the system – to budget time overrun detection, including blocking time, for better damage confinement, via monitoring the frequency of arrival of sporadic tasks.

The Ravenscar Profile does not allow the use of `Ada.Execution_Time.Timers` (D.14.1): as a consequence, one can only poll for overrun detection, which obviously is not satisfactory.

Frequency of sporadic arrivals cannot be controlled at language level, but can by the application, for example by forcing task suspension before waiting for the next release event.

Monitoring for overrun of blocking time is no standard provision for either operating system or Ada. The risk is that this fault may go undetected.

For damage confinement one could use budget servers, which the Ravenscar Profile does not support.

The error recovery strategies may include: stopping the overrunning task and then starting a new task to resume service; using the programmatic interface of the asynchronous task control package, D.11. Implementing them under the constraints of the Ravenscar Profile, which does not allow any of those features, calls for application-level solutions that may obfuscate the resulting code.

The group agrees that the above considerations provide sufficient motivation to define a new profile, which goes beyond Ravenscar and, quite possibly, includes all of its features.

Discussion ensues on the general characteristics of the profile. Joyce Tokar comments that the idea of a new profile is interesting and attractive but it is hard to tell how difficult it may be to implement. Michael González argues that we should pay attention to allowing features that at the time of Ravenscar were known to be extremely complex to implement. He observes that alternative models may exist, such as e.g., ARINC 653, which might be an interesting target to specify a language profile against. Joyce Tokar voices agreement to that consideration, but also reckons that discussing an ARINC 653 profile would stray the discussion away from the intended focus.

The discussion then moves to the need for dynamic priorities. The question before the group is what language features we really need to be able to suspend / resume individual tasks? The choice is between dynamic priorities (D.5) and asynchronous task control (D.11).

At the end of this first round of high-level discussion, there is full consensus from the group that we need a new profile, distinct from Ravenscar. What we want is a consistent, cohesive profile, and not a string of optional additions to Ravenscar. We should however pay attention to keeping the implementation and certification distance from Ravenscar affordable for language implementers. The group's consensus is also that the starting point for the definition of a new profile should be a clear application programming model, from which we can then determine the implementation requirements, language restrictions, and obstacles to certification. Burns and Wellings' paper [1] is a good starting point.

At this point the Chair invites the group to delve deeper into the discussion of specific features, with special attention at how they would work in a multicore environment.

The first feature on the list is `Set_CPU` (D.16.1). Including it in the profile supports error recovery strategies that use load balancing. In the envisioned model, tasks are statically allocated to groups, but they can be moved across cores. This feature provides key support to a task splitting approach to handling timing faults whereby overrun-work may be performed opportunistically, via load balancing, on a core different from the one in which the offending task was initially assigned.

Michael González voices his preference for a suspension-only model to one that also allows overrunning tasks to resume. An inconclusive discussion then follows on the implementation complexity that may be incurred by asynchronous task suspension. This issue, among others, needs to be further studied and should become a topic of investigation for IRTAW-16.

The second feature on the list is the restriction "one CPU Timer per task". The rationale for that restriction is that, on a single core, the timer resides in the same CPU as the task to which it belongs. It is natural to extend this notion to multicore. The problem with multicores however is that systems may exist where there is a single clock for all cores: in that case it may become complex to map time events to the CPU where the task resides. The reason to attach the event handler to the CPU where the task executes is that this assignment warrants that if the handler executes then certainly the task does not, and consequently we don't need complex asynchronous control to hold the task from running.

The issue of setting affinities for the handlers of timing events, CPU timers and "normal" interrupt handlers is discussed. The idea is to require the affinity for time-related handlers to be consistent (i.e., equal) to that of the task that causes handler invocation. The discussion then delves into the general problem of what affinities – also for

Ravenscar and full Ada – can be set to protected handlers in multicores. It rapidly becomes evident that this is a large, general problem which should be set aside for later and deeper analysis, for discussion at IRTAW-16.

The next feature submitted to discussion is the support for Group Budget: the concept is that fault containment strategies for collections may offer greater flexibility than for individual tasks (yet at the cost of not knowing what the offending task actually is). The group's concern is that a non-trivial implementation burden may be incurred in supporting that feature. To make things simpler we need to assume that the group budget must be per CPU. Intense discussion takes place on whether dynamic group management (i.e., adding and removing tasks from groups at run time) should be preferred to having static group membership only.

The discussion returns to the issue of dynamic priorities: there is majority sentiment that the current slant of the dynamic priority package is too general for our needs. We do not really want any generalized agent to be allowed to set task priorities: we rather want specialized / dedicated handlers (which perhaps could be identified by some restriction) to do that. One way around this problem is to prefer asynchronous task control (with hold only) to dynamic priorities. The group's sentiment on the alternative appears to be divided. The Chair calls for a straw poll, which shows 10 in favour of asynchronous task control, and 8 doubtful abstentions.

The final feature for discussion in the part of the session is entry queues. We still require single entry (for the same reasons why we had that restriction in Ravenscar), but we want to allow multiple calls to queue. We feel the guard should continue to be a Boolean only. `'Count` can however be used in the entry body. On that account, an intense yet inconclusive discussion takes place on the safeness, for our purposes, of the semantics of `Hold` being called when the task has a call in an entry queue. As part of this slot, the issue is also briefly discussed as to whether nested protected subprogram calls should be allowed in multicore processing: the group observes that the theory of real-time systems has not yet developed a convincing model for nested resources. It may therefore be wiser at this time to disallow nesting.

Before calling this discussion to an end, the Chair invites suggestions for further features of interest for the envisioned profile; the group response includes: barriers in multicores; relative delay statement and relative timing events. The intent is to record this need, invite the group to investigate it in the future, and then discuss the findings at IRTAW-16.

3 Ravenscar and distribution

The Chair then invites a short report on the progress of the University of Cantabria's (UC) group in the development of a Ravenscar- and SPARK-compliant implementation of the Distributed Systems Annex. Héctor Pérez, on behalf of UC, explains that two language features are needed by the current implementation but conflict with the SPARK restrictions: generics; and abstract types.

The group sentiment in that respect is that too strict adherence to SPARK may defeat the purpose of the UC project: "educated" generics and abstract types are useful abstractions for the project and they should be retained.

A side issue was raised before the discussion on this topic came to an end: language support for initialization-level configuration is desired (which comes handy for, e.g., end-point receivers) that does not resort to full-fledged programmatic interfaces, which could be exposed to erroneous usage. No conclusion is reached on this point, other than recording it for further investigation.

4 Code archetypes and programming frameworks

The subsequent slot of discussion focuses – in a joint fashion – on hearing a report on the progress of the work conducted at the University of Padova (UPD) for the finalization of Ravenscar code patterns for automated code generation [3], and at the Universitat Politècnica de València (UPV) for the extension to multicore of the real-time programming framework [4].

Both reports show good and interesting progress. Seeing some complementarity between the qualities of the two respective approaches, the group encourages the teams at UPV and UPD to investigate the possibility of integrating their results.

5 Profiles: Ravenscar and EDF

The final slot of the day's discussion is devoted to examining Alan Burns' proposal in [5] for an EDF version of the Ravenscar Profile. The author's rationale for the proposal is that Ada 2005 supports EDF and mixed dispatching policies via Baker's stack resource protocol [6]. However, the resulting protocol is complicated to understand as well as to implement.

In the author's vision, an interesting alternative to support resource sharing under EDF is to make protected execution non-preemptive. The consequences of that approach are: in the pro side, an easy implementation at language level since priorities are no longer needed and a single ready queue is required; on the cons side instead: longer blocking time for tasks owing to non-preemption during protected execution.

The group's sentiment on the proposal gets quickly divided. For some, Burns' model seems attractive and, in a way, conducive to a Ravenscar adoption of EDF symmetrical, for simplicity and theory support, to Fixed Priority Scheduling (FPS). For others instead, and for Michael González in particular, EDF alone should be considered insufficient for safely programming HRT systems: in Michael's view one would need additional features, either EDF+FPS (as in previous publications from our community) or budget control. Intense discussion takes place on this interesting subject, but it comes to no final conclusion owing to lack of time. The Chair invites the group to continue investigating this matter with a view to reporting progress at IRTAW-16.

References

[1] A. Burns, A.J. Wellings and A.H. Malik. *TTF-Ravenscar: A Profile to Support Reliable High-Integrity Multiprocessor Ada Applications.* This issue.

[2] T. Anderson and P.A. Lee. *Fault Tolerance Principles and Practice*. Prentice-Hall International, 2nd edition, 1990.

[3] M. Panunzio and T. Vardanega. *Charting the evolution of the Ada Ravenscar code archetypes*. This issue.

[4] S. Saez, J. Real, and A. Crespo. *Adding Multiprocessor and Mode Change Support to the Ada Real-Time Framework*. This issue.

[5] A. Burns. *An EDF Run-Time Profile based on Ravenscar*. This issue.

[6] T.P. Baker. *Stack-based scheduling of real-time processes*. Journal of Real-Time Systems, 3(1), March 1991.

Session Summary: Concurrency Issues

Chair: Juan Antonio de la Puente
Rapporteur: Stephen Michell

1 Introduction

This session was the final session of International Real Time Ada Workshop 15. It was chaired by Juan Antonio de la Puente. Stephen Michell was the rapporteur. Papers and issues discussed were

- Concurrency and real time vulnerabilities under consideration by ISO/IEC/JTC 1/SC 22/WG 23 Programming Language Vulnerabilities Working Group.
- Execution time accounting as being implemented by the Ada programming language and considerations for multiprocessor environments
- Discussion of Set_CPU and deferment of attribute setting in multiprocessor environments.

Section 2 discusses the concurrency vulnerabilities with 2.1 discussing the vulnerabilities in [Michell 2011] and 2.2 discussing real time vulnerability outlines presented by Stephen Michell at the workshop. Section 3 summaries the discussion of execution time accounting. Section 4 is Ada real time and virtualization. Section 6 is

2 Concurrency Vulnerabilities

2.1 Methodology

As discussed in [BW 209] and [Michell 2011], the work being done by WG 23 to date does not address the real issues of vulnerabilities presented by concurrent programs. [Michell 2011] addresses this with a proposal for six vulnerabilities for

- Thread activation,
- Thread termination – directed
- Thread termination - premature termination,
- Shared data access,
- Concurrent data corruption, and
- Concurrency protocol errors.

The workshop examined these proposals and then went further to consider three proposals to develop real time vulnerability writeups for

- Real time timing vulnerabilities,
- Real time thread control, and
- Real time scheduling.

2.2 General Vulnerability Discussion

Steve introduced the topic by first discussing vulnerabilities, the notion of exploits of vulnerabilities. For a general writeup on vulnerabilities, their effects, and the role that programming languages can play in creating and helping to avoid vulnerabilities, see section 5 of [TR 24772]. More discussions on weaknesses and vulnerabilities can be found at the Open Web Application Security Project [OWASP], the Common Weakness Evaluations [CWE], Common Attack Pattern Enumeration [CAPEC], and the Build Security In project [BSI].

There are two kinds of vulnerabilities discussed by TR 24772. The first is called “Language Vulnerabilities” and are documented in section 6 of that document. The second is “Application Vulnerabilities” in section 7 of this document. TR 24772 explains the difference as follows.

"This Technical Report focuses on a particular class of vulnerabilities, language vulnerabilities. These are properties of programming languages that can contribute to (or are strongly correlated with) application vulnerabilities—security weaknesses, safety hazards, or defects. An example may clarify the relationship. The programmer's use of a string copying function that does check length may be exploited by an attacker to place incorrect return values on the program stack, hence passing control of the execution to code provided by the attacker. The string copying function is the language vulnerability and the resulting weakness of the program in the face of the stack attack is the application vulnerability. The programming language vulnerability enables the application vulnerability. The language vulnerability can be avoided by using a string copying function that does set appropriate bounds on the length of the string to be copied. By using a bounded copy function the programmer improves the predictability of the code's execution.

The primary purpose of this Technical Report is to survey common programming language vulnerabilities; this is done in Clause 6. Each description explains how an application vulnerability can result. In Clause 7, a few additional application vulnerabilities are described. These are selected because they are associated with language weaknesses even if they do not directly result from language vulnerabilities. For example, a programmer might have stored a password in plaintext (see [XYM]) because the programming language did not provide a suitable library function for storing the password in a non-recoverable format."

The workshop spent most of the session discussing the real time issues that could become vulnerabilities. They are identified here as presented by Stephen on slides to lead the session.

It should be noted that TR 24772 uses the term "thread" where Ada uses "task" to designate entities that can execute concurrently. For the purpose of this section, the term thread will be used exclusively.

2.3 General Concurrency Vulnerabilities

2.3.1 Thread Activation [CGA]

Steve presented the general principals as presented in [Michell 2011] section 5.1 Thread Activation [CGA]. There was general agreement that there are language issues involved in the creation of threads such as resource exhaustion, undetected failure to activate of some threads, and the resulting system failures that can occur during creation. Therefore, this vulnerability belongs in section 6 Programming Language Vulnerabilities of ISO IEC TR 24772. There were no recommendations for further subdivision of the vulnerability or consolidation with other vulnerabilities. No further application workarounds or programming language extensions were discussed.

2.3.2 Thread Termination (on request) [CGT] (and premature) [CGS]

The workshop decided to discuss the issues of thread termination together. Both vulnerabilities were presented as being appropriate for section 6 of ISO IEC TR24772 as programming language vulnerabilities, since, even if the language does not have a concurrency component, the underlying environment as presented by its libraries have such a paradigm and can expose the issues. Languages that have concurrency as part of the language must consider all termination issues.There were no recommendations for further subdivision of the vulnerability or consolidation with other vulnerabilities.

Joyce raised the issue that finalization of data and of control space after the directed termination of a thread is an issue that must be discussed in [CGT]. Also, there must be a way to detect attempts to interact with threads that have been terminated or requested to terminate, both in the period of termination and after the thread has been terminated. These issues will be added to the submissions to WG 23.

2.3.3 Concurrent Data Access [CGX]

Stephen noted that this vulnerability was split from [CGY] so that issues of direct access to data in concurrent and real time data can be separated from shared resources that happen in external components of the system such as filing systems, environment variables, and databases.

The workshop reviewed the writeup, and agreed that this was a language vulnerability and as such belonged in section 6 of TR 24772. Andy noted that the issue of "volatile" of a shared variable was missing, in the sense that languages often reorder reads and assignments and may not be aware that other assignment operations exist in other threads. The "volatile" directive notifies the language processor not to perform such re-orderings, and to make fetches and assignments as atomic as possible.

It was noted that there is another recommendation to application developers to always specify data elements as "volatile" to

prevent language processors from performing such reordering.

2.3.4 Concurrent Data Corruption [CGY]

Steve proposed in his presentation to target this vulnerability to TR 24772 sect 7 as an application vulnerability, since external resources outside of the application itself, and the concurrent components accessing it may not be part of the same program, so language-specific mechanisms to control access are not feasible in general. There were no proposed additional issues, language vulnerabilities or application workarounds discussed.

2.3.5 Concurrent Protocol Errors [CGM]

The workshop discussed the vulnerability's target to TR 24772 sect 6 as a language or OS vulnerability and agreed that it was appropriate because a number of languages and most operating systems (through libraries) provide protocols for concurrency paradigms. No significant issues were identified as missing. No other application workarounds or language avoidance mechanisms were identified.

2.4 Real Time Vulnerability Discussion

Stephen led the discussion from slides prepared to present three real time vulnerabilities, with the plan to use the workshop discussions to generate writeups suitable for WG 23.

2.4.1 Real Time Timing [CGQ]

The application vulnerability was identified as an application vulnerability (as opposed to a programming language vulnerability) since almost all real time timing issues arise as a result of hardware issues, low level kernel issues and application issues. The main issues identified initially were

- Drift between clocks, such as real time and time of day clocks, or between clocks on different processors
- Failure to track time of day clock updates due to leap seconds, time zone changes or corrections; and
- mismatches between time accounting and notification requests, such as posted wakeup times or deadlines

The workshop added to this

- Rollover of bounded clocks can cause timer calculations to fail
- mismatches between the resolution of the clock and the expectations of the application can cause wakeups or deadlines to be too late or too early (usually too late)
- transfers between a time of day clock and real time clock can result in loss of precision and missed deadlines or missed wakeups

The effects of these errors can be

- Wrong calculations
- Missed deadlines
- Wakeups that are too early or too late causing portions of the concurrency structure to fail
- guardian code (that relies on being notified if deadlines are missed) may misbehave if timers drift

Recommended approaches for application developers are:

- Develop systems whose concurrency model is amenable to static analysis, such as model checking,
- Perform analysis of all aspects of timing with as much rigour as is possible;
- Choose a language or environment that provides the capabilities to express time, select the appropriate time paradigm, and clock management mechanisms
- Implement measures to monitor time accounting, and drift between clocks, and be prepared to take corrective action

- Identify mechanisms to identify misbehaving systems such as heartbeats or watchdog timers; and
- Include monitors with the ability to reset the complete system back to a known state before continuing.

The usual result of such errors is erroneous behaviours or misbehaving feedback systems. Systems that rely on on-time calculations and events and that experience such erroneous behaviour may experience catastrophic failures. Arbitrary code execution is unlikely.

2.4.2 Real Time Thread Control

Burns and Wellings [BW 2009] identified real time thread control issues as another real time potential vulnerability. The workshop agreed that this was already covered in the vulnerability Protocol Lock Errors [CGM].

2.4.3 Real Time Scheduling [CGP]

Real Time Scheduling was identified as a programming language vulnerability since there are a few languages and operating systems that provide scheduling protocols suitable for real time, and use of the high level paradigms, such as the Priority Ceiling Protocol can fix many of the issues in this domain. On the other hand, to make real time systems function to specification, even using these protocols requires applications designers that clearly understand the domain and take the necessary steps to allocate attributes to threads (such as deadlines and priorities) and to communications and protocol management code.

The main issues identified were

- Priority inversion when a thread of a lower urgency (lower priority or later deadline) is executing and preventing one of higher urgency (that is ready to execute) from executing.
- Scheduling mistakes resulting in threads not completing their work in the required time
- Missed thread deadlines due to priority inversions, scheduling mistakes and protocol errors
- Missed interrupts or events
- Excessive input or events
- Excessive use of a resource
- Lack of access policy or queuing policy resulting in indefinite waiting for some threads

These can result in

- Missed system deadlines
- Complete system failure
- System halting
- System ceasing to process input or deliver output
- Instability
- System not meeting specifications

It was noted that, in the real time realm, that even seemingly small changes, such as a change of a single priority to a single thread can have very large impacts.

Recommended approaches for application developers are:

- Develop systems whose concurrency model is amenable to static analysis, such as model checking,
- Perform this analysis with as much rigour as is possible;
- Choose a language or environment that provides the capabilities of priority inheritance and priority ceiling protocol, and to use those capabilities;
- If using multiprocessors, be aware of the issues with scheduling, thread interaction and data and cache coherency across multiple processors and be very conservative in the assignment of threads to processors;

- Identify mechanisms to identify misbehaving systems such as heartbeats or watchdog timers; and
- Include monitors with the ability to reset the complete system back to a known state before continuing.

Errors in this domain can be used to create covert channels, and can lead to complete system shutdown or misbehaviour, but arbitrary code execution as a result of this vulnerability alone is unlikely.

3 Execution-time accounting of interrupts

This session followed on from Session A, where the focus of the discussion was specialization to multiprocessor environments..

While the workshop was in progress, participants received an advanced copy of a paper by Kristoffer Gregersten on the implementation of execution-time accounting of interrupts. A paper submitted by the same author [Gregersten 2009] was instrumental in leading IRTAW 14 to recommend that Ada implement this accounting as part of Ada 2012, which is now being finalized and which includes this capability. In the paper to be submitted to the Ada Users Journal, Kristoffer summarizes an implementation of this new capability for the GNAT Ravenscar bare-board run-time library on a single-processor embedded microcontroller. In his paper he documents a number of tests that show negligible additional overhead to support this accounting, and he argues that this capability allows for protecting against interrupts arriving at a higher frequency than expected or handlers exceeding their allocated budget.

Jose Ruiz reported that AdaCore has similar functionality implemented for GNAT Ada on embedded Leon32 and PowerPC processors and suggested that the capability was very useful for their clients.

3.1 Deferred attribute setting mechanism

Sergio raised the issue that the current wording in the draft Ada 2012 reference manual says that System.Multiprocessors.Dispatching_Domains.Set_CPU can cause excessive context switches for tasks that are executing in a protected operation on CPU1 but make a call to Set_CPU to move itself to CPU 2. Clause D.16.1(27) of the ARM says

“A call of Set_CPU is a task dispatching point for task T. If T is the Current_Task the effect is immediate, otherwise the effect is as soon as practical.”

This could be interpreted to mean that Set_CPU calls into the kernel, effectively suspends the task while it moves it to “CPU”, possibly causing at least 2 context switches. It was noted that the same wording exists for System.Multiprocessors.Dispatching_Domains.Add_Task.

Here is a model of what can happen:

- T1 is executing on CPU1 with priority P1.
- T2 is executing on CPU2 with priority P2>P1.
- T1 calls a protected operation PO_A that has a ceiling priority of P3>P2>P1. Within PO_A, T1 (which is now executing at priority P3) executes a Set_CPU or a Add_Task call to assign itself to CPU2.
- T1 is immediately moved to CPU2 while still executing inside PO_A and preempts T2.
- T2 either is switched to a different CPU (depending upon T2's affininty) or is delayed until T1 completes execution of PO_A.
- T1 completes PO_A, and its priority drops to P1, causing another context switch to permit T2 to resume execution.

There was significant discussion on the various issues. It was noted that there is no significant issue if task T1 is blocked or delayed and another task changes its CPU or affinity. Changes to the running task, however, can have the difficulties described above. In a real time system where T2 has a deadline, such additional overhead of context switches and execution of protected operations in place of T2's execution could cause T2 to miss its deadline.

Miguel raised the issue that more deferred operations are required, such as Delay_Until_And_Set_Deadline (See ARM D.2.6), and noted that any combination of Set_CPU, Set_Deadline, and Set_Priority can cause excessive context switches and priority inversions (such as the one described above) unless we can explicitly defer the setting of these operations. Sergio's proposal contains the following subprograms [SC 2011]:

```
Ada.Dispatching.EDF.Set_Next_Deadline(
             D : in Deadline;
             T : in Ada.Task_Identification.Task_Id
               := Ada.Task_Identification.Current_Task);

      System.Multiprocessors.Dispatching_Domains.Set_Next_CPU(
             CPU : in CPU_Range;
             T   : in Task_Id := Current_Task);

      Ada.Dynamic_Priorities.Set_Next_Priority(
             Priority : in System.Any_Priority;
             T        : in Ada.Task_Identification.Task_Id
                      := Ada.Task_Identification.Current_Task);
```

Miguel proposed to add the following subprogram that set the deferred attributes immediately:

```
      Set_Attributes(
             Attr : Deferred_Attributes;
             T    : in Ada.Task_Identification.Task_ID := Current_Task);
```

Where

```
      type Deferred_Attributes is record
            CPU: CPU_Range;
            D: Deadline;
            ...
      end Deferred_Attributes;
```

The advantage of this approach would be that it provides a consistent Set_Attributes, whereas the calls

```
          Set_Next_Deadline,  Set_Next_CPU, ...,
```

permits different task calls to interleave, creating different "Next" attributes to be set, at the point of the Set_Attributes.

There was general support for this concept. Steve proposed that these be in a child package of each package, possibly called something like Ada.Dispatching.EDF.Deferred and Ada.Dynamic_Priorities.Deferred. It was not clear under such a scheme where Set_Attributes would be declared, so more work would be needed.

There was discussion as to what would happen if Set_Attributes was not called. The presumption was that the deferred settings would happen when

- Set_Attributes was called
- The task awoke from a delay, delay_until,
- The task was removed from a suspension object,
- The task was released from a protected entry, after completing the operation of that entry
- The task completed a protected operation, or the outermost protected operation if it was executing in a nested protected operation.

It was noted that one cannot rely upon a task being suspended, blocked or delayed for a significant period of time after having a deadline, priority or affinity set, hence there needs to be a subprogram call to set the attributes immediately.

Implementation approaches were discussed. The approach of having room for deferred attributes in the Task Control Block or task attributes was discussed. Such attributes would become effective when the task was released from a delay or blockage, or immediately upon a call to Set_Attributes.

It was agreed that these issues needed further investigation, modelling and trial implementations before a formal proposal could be made to WG9.

4 Ada Real-Time Services and Virtualization

There was a discussion of Ada real time services and virtualization. It was decided that this was not a language issue, hence discussion terminated.

5 Wrap up and Conclusions

This session was the final one of the workshop. The work on vulnerabilities gave Stephen material to feed back into the work of WG 23. He expects that the next publishing of TR 24772 will contain at 6-8 concurrency vulnerabilities based on the work done here.
The next meeting of the workshop is planned for York area, UK in the spring of 2013.

Bibliography

[BSI], the Build Security In website, www.BuildSecurityIn.us-cert.org

[BW 2009] Burns, Alan and Wellings, Andy, "Language Vulnerabilities - Lets Not Forget Concurrency", Proceedings of Real Time Ada Workshop 14, ACM SIGAda Letters, Volume 30 Issue 1, April 2010

[CAPEC], the Common Attack Pattern Enumeration and Classification project web site, www.capec.mitre.org

[CWE], the Common Weakness Enumeration web site, www.cwe.mitre.org

Gregertsen, Kristoffer, "Execution-time control for interrupt handling," Proceedings of IRTAW 14, ACM SIGAda Letters., vol. 30 Issue 1, April 2010.

[Michell 2011] Michell, Stephen, "Programming Language Vulnerabilities – Proposals to Include Concurrency Paradigms", transactions of IRTAW 15, ACM SIGADA Letters, Volume 31 Issue 4, November 2011

[OWASP] the Open Web Application Security Project, www.owasp.com(net, org?)

[SC 2011] Sergio Sáez and Alfons Crespo. Deferred setting of scheduling attributes in ada 2012. Ada Letters, This issue, 2011.

[TR24772] ISO IEC TR 24772 "Information technology -- Programming languages -- Guidance to avoiding vulnerabilities in programming languages through language selection and use" , International Standards Organisation, 2010

Investigating SystemAda: TLM_FIFO Detailed Characteristics Proof, TLM2.0 Interfaces Implementation, Simulation Time Comparison to SystemC

Negin Mahani
Zarand High Education Center, Computer Engineering Department, Shahid Bahonar University, Kerman, Iran
Negin @cad.ece.ut.ac.ir

Abstract
VHDL as a hardware description language has some short-comings for system level modeling. Since previous researches[10] tried to extend this language for high level modeling, using Ada structures, and also it has derived some of its basic structures from Ada at first, we have decided to extend Ada to a form called SystemAda that can model hardware at transaction level modeling. Ada because of its intrinsic features like concurrency and object orientation can be a good candidate for a high level hardware modeling language. In our previous works we have proved that Ada can have a link to Register Transfer Level (RTL) and Transaction Level Modeling (TLM) modeling[3]. Here we have proofed the detailed characteristics of our TLM_FIFO channel -- just like the real TLM_FIFO -- and a way to TLM2.0 interfaces. Finally by simulation time comparison between SystemAda and SystemC TLM equivalent models we have proved that there is no simulation time penalty in SystemAda over SystemC.

1 Introduction

There are lots of problems for hardware design that all are due to complexity such as the speed of the design for hardware/software co-design and time to market, the experience of the hardware designers, and the costs of the development tools. To solve these problems we should handle the complexity by raising the level of abstraction. Evolution of design abstraction levels show that every 15 to 18 years, design abstraction moves to a higher level. Back in early 1990's, design of digital systems changed from the old gate level design to register-transfer level design. New hardware design languages and new design methodologies were instrumental in this transition. At the RT level, designers have to focus on top-down partitioning, taking advantage of existing simulation and synthesis tools.

As systems are getting more complex and issues regarding hardware/software partitioning and various design constraints are becoming more important, another change is due to occur. This time, the present design at the RTL is giving way to the new transaction level design, TLM. At this level, software and hardware issues, use of embedded processors along with RT level components, combined hardware software verification and test, and relating higher level designs to low level manufacturing test have become issues that modern designers have to be dealing with [1].

According to the theory of design evolution [1], TLM is state of the art level in design of complex digital systems. In TLM, a system is divided into computation and communication parts. In TLM, processing elements play the role of computation parts which communicate with each other through channels that can be characterized as communication parts in TLM specification.

In this paper, a number of packages are defined using Ada to support modeling of TLM communication infrastructures. These packages are referred to as SystemAda. TLM channels as well as TLM interfaces are described using our SystemAda. This paper is organized as follows. In Section 2, TLM is introduced and its advantages are discussed. Section 3 compares system level languages with each other. In Section 4 and 5, TLM_FIFO channel detailed characteristics and TLM2.0 interfaces respectively are implemented using Ada. Section 6 presents the experimental results followed by conclusions in Section 7.

2 TLM as the First Step toward ESL

In the recent years, the Electronic System Level (ESL) industry has defined a level of abstraction in modeling and design of systems. This level of abstraction is generally higher than RTL in that the timing details are not considered and the communications between modules are modeled by high level channels instead of a wire or a group of wires. This level of abstraction, which is going to become the starting point in system level design, is called transaction level and the act of modeling a system in this level is referred to as Transaction Level Modeling (TLM)[6]. System level designers

and innovators utilize the transaction concept in this level of system modeling. This concept has been used in networks for many years.

As a matter of fact, there is no special and unique definition for transaction level modeling between system level designers. Each EDA vendor or system level designer defines TLM by one of its aspects. In a general definition of TLM, the system is divided into two parts: communication and computation parts. In this definition, TLM is considered as modeling the communication parts of a system at a high level of abstraction (e.g., by function calls). With this definition, the computation parts (modules) of a design can be at various levels of abstraction. It is obvious that the higher level the modules are designed, the faster their simulation process and the easier their connection with communication parts are[2].

Features and advantages of TLM over RTL modeling have made designers to move their design starting point from RTL models to transaction level models. Although it can be used for faster modeling, the main purpose that designers use TLM is its simulation speed. Since the communications between modules are modeled with efficient and high level functions and the computational modules are also modeled in a higher level of abstraction, designers can gain over 1000x simulation efficiency from TLM simulation over RTL simulation. The simulation efficiency leads to other TLM features including more efficient partitioning, faster and more precise functional verification, faster hardware/software co-verification, and faster hardware/software co-simulation.

Some of the above features produce other benefits for the designers. For example, by modeling a system with TLM, the designer has a more comprehensible view for the design partitioning. A mature partitioning in a design can even affect the power consumption of the manufactured SoC. Designing in transaction level provides designers and their costumers an early design prototype. This early prototype helps designers clarify whether the complete design is feasible to be developed and implemented. In this prototype, several critical decisions, improvements, and changes can be made to prevent the failure of an entire project. Without ESL, these critical decisions could only be made after spending a lot of time and money for developing an RTL prototype of the design. In addition to ESL, the software team can start its software development and verification by this early prototype. In traditional RTL designs, the software team has to wait for completion of the RTL design before starting the software development[2].

3 Comparison of System Level Languages

Since we introduced the benefits of modeling the designs at system level, the first question which any designer would encounter is that which language should be used for system level hardware design. There are possible choices: SystemC, SystemVerilog and our specialized form of Ada (SystemAda). The answer to this question is very dependent on the purpose of system level modeling but some tradeoffs from inside and outside of the language, such as language constructs and semantics, tool support, third-party IP availability and access to knowledgeable engineers are also involved in this choice[24].

The most popular languages which are used in hardware design nowadays are VHDL and Verilog which both are suitable for RTL and have substantial disabilities to cover system level. SystemVerilog is a system level language from Verilog family with verification purpose in mind from first advent. VHDL is used for high level design, but it lacks abstract timing and communication, genericity and Object Oriented modeling. Some groups like SUAVE have proposed some extension to VHDL to cover system level. Most of the extensions are added from Ada95 as the base language of VHDL from early development. It shows that Ada has potentials to be used as an HDL. Ada as an HDL has a long history which is out of scope of this paper[4][8][10].

Among all of the languages mentioned above, SystemC and SystemVerilog are more common. The base language of SystemVerilog is Verilog and its main focus is on RTL modeling like Verilog. The main purpose of this language is verification. The enhancements related to directed test generation, assertion definitions and coverage metrics are all very valuable capabilities, and all are closely tied to the RTL implementation domain.

SystemC is a class library in C++ which is patched on top of it and TLM is patched after on top of SystemC. Due to lots of patches, there are many problems with debugging which are the most common user problems with this language. Since Ada is the base language of VHDL and it has inherent concurrency as well as object oriented structures, we have chosen it for implementing our TLM platform.

3.1 Ada vs. C++ Family Languages

Ada has some advantages over C++ family languages as it is told in various references [5][11][12][13][23]. The most important advantages

of Ada over SystemC and C++ family languages are listed below:

- High readability and well descriptive language
- Faster and more convenient debugging
- Inherent concurrency and interface types (not being patchy)
- Having the link to RTL
- Capability to export and import to/from foreign languages
- Shorter simulation time

In order to give an idea of the difficulty of debugging in SystemC, we present two examples (Figure 1, Figure 2). In some cases, the error messages do not refer to the right point at the library and results in confusion of the designer. Here is a misleading error message that the real source of the error is in the design at TLM, but the error message itself refers to a predefined SystemC file that is in the lower layer rather than TLM (Figure 1).

```
Error: (E529) insert module failed: simulation
running
In file: C:\Program Files\Microsoft Visual
Studio\VC98\systemc-
2.1.v1\src\sysc\kernel\sc_module_registry.cpp:58
In process: WR1.run @ 1 ns
```

Figure 1. SystemC TLM confusing error message example one

This error is due to a misplaced instantiation. To solve this problem, changing the location of the instantiation to the main module is enough. Figure 2 shows another example of the confusing error message which refers to the port, relates to the error by its number and do not refer to its name to find out the name of the port and we should debug the code line by line.

```
Error: (E112) get interface failed: port is
not bound: port 'full1.port_2' (sc_in)
In file: C:\Program Files\Microsoft Visual
Studio\VC98\systemc-
2.1.v1\src\sysc\communication\sc_port.cpp:196
```

Figure 2. SystemC TLM confusing error message example two

4 TLM_FIFO Detailed Characteristics Mapping by Ada

Since all of TLM channel structures can be described based on FIFO, we have started developing TLM channels from TLM_FIFO channel as described in our previous works[3]. All of these descriptions are based on functionality of the channel.

4.1 TLM_FIFO Special Characteristics

TLM_FIFO has some special characteristics that all are covered in the Ada implementation version. These characteristics are as follows[7]:

- Generic package
- Generic type
- Generic size
- Concurrency
- Blocking/non-blocking transport capability
- Export
- Multiple reader/writer
- Multiple instances

Some of these characteristics which use specific Ada constructs are explained in the following sub-sections.

4.1.1 Generic Type

Generic in Ada is similar to template in C++. The data which is transferred via TLM channels could be of different types from basic data types to various record types (user defined types). There are different kinds of generics in Ada and each of them is appropriate to satisfy some specific requirements [16][14]. Figure 3 shows how we have generic type in TLM_FIFO element type. We have used one that has few limitations for the data type (for example limited types would not be allowed for a private generic formal parameter).

```
generic type FIFO_Element is  private;
package FIFO is…
```

Figure 3. Generic type FIFO

4.1.2 Generic Size

The size in TLM channels could be changed based on the requirements. To have arbitrary size TLM_FIFO channels, two solutions are to use dynamic and static memory allocation approach.
As it is obvious, using static memory allocation is faster and would lessen development cost. Using dynamic memory allocation implies access types in Ada language constructs which are the same as pointers in C++ (they are similar especially in purpose, but there are very significant differences by design). Implementing TLM_FIFO using access type referrers to link-list implementations and

modification operations. Figure 4 shows Ada access types used in implementing TLM_FIFO.

```
type FIFO_Node;
type FIFO_Pointer is access FIFO_Node;
type FIFO_Node is record
  Data : FIFO_Element;
  Link : FIFO_Pointer;
end record;
```

Figure 4. Unlimited size FIFO using dynamic memory allocation

With static memory allocation, we should put the size in generic part of the package such that it could be determined in instantiation step (Figure 5). Implementing TLM_FIFO with static memory allocation implies a cyclic buffer construct which is a static array type with round FIFO modification operations indeed[17].

```
generic Size: natural;
type FIFO_Element is  private;
package FIFO is…
```

Figure 5. Generic size FIFO

4.1.3 Concurrency

Hardware models have concurrency as their inherent feature. The basic concurrency unit in the Ada language is the task, as mentioned earlier [18][22]. We have used this language construct to have concurrent hardware channels at transaction level modeling such as TLM_FIFO. The task type specification which adds concurrency to the FIFO package is illustrated in the following Figure 6. As it can be seen the task type FIFO_Task has three entries named *Add*, *Remove* and *Stop* the name of which shows their functionalities. A variable called TLM_FIFO from FIFO_Task type has been defined to refer to as a TLM_FIFO channel in implementing communications.

```
task type FIFO_Task is
  entry Add;
  entry Remove;
  entry Stop;
end FIFO_Task;
TLM_FIFO : FIFO_Task;
```

Figure 6. FIFO_Task specification

4.1.4 Blocking/Non-blocking

One of the key concepts in TLM is supporting blocking and non-blocking modes of communication. In blocking, when the sender adds the input data packet to the communication channel the control of the channel is given to the slave to remove the desired data packet and give back the control of the communication channel to the master. Therefore, each *add* must be followed by a *remove* action and vice versa (Figure 7). In non-blocking mode of communication, each of the modules that are bound to a channel could call its method for the channel without keeping track of the other module actions. Therefore, any *add* and *remove* action could be done independently. The Ada language task construct have potentials to implement these concepts. As it is mentioned in our previous works[3], in the task body we could have select and accept blocks and each select block could contain several accept blocks. Each accept block could be separated by an *or* keyword from the other one. This kind of task implementation is called selective waiting. If we put *or* between different accepts (*Add* and *Remove* in Figure 8) the messages are processed in the order that they are received with no predefined priority [18].

```
task body FIFO_Task_B is
begin
  loop
    select
      accept Add do
        Add_FIFO;
      end Add;
    --or
      accept Remove do
        if Empty_FIFO=false then
          Rem_FIFO;
        end if;
      end Remove;
    end select;
  end loop;
end FIFO_Task_B;
```

Figure 7. Blocking communication in TLM_FIFO

```
task body FIFO_Task is
begin
  loop
    select
      accept Add do
        Add_FIFO;
      end Add;

    or

    when Empty_FIFO=false => --guard
      accept Remove do
        Rem_FIFO;
      end Remove;
    end select;
  end loop;
end FIFO_Task;
```

Figure 8. Non-blocking communication in TLM_FIFO

Figure 7 and Figure 8 respectively show blocking and non-blocking task bodies which implement these modes of communication in TLM_FIFO. Notice that in both tasks we have used loop statement[20].

4.1.5 FIFO Package Specification

It is obvious that a package in Ada has specification and body which recall entity and architectures of VHDL designs[15]. The specification of the FIFO package with dynamic and static memory allocation is depicted in Figure 9 and Figure 10 respectively. Actually these packages both have task types that discussed earlier in sections 4.1.3, 4.1.4 and various procedures that introduced in our previous papers[3][10].

```
generic type FIFO_Element is private;
package FIFO is
   input : FIFO_Element;
   output: FIFO_Element;
   type FIFO_Node is private;
   ...
   procedure Add_FIFO;
   ...
  private
   type FIFO_Channel is access FIFO_Node;
   type FIFO_Node is record
     Data : FIFO_Element;
     Link : FIFO_Channel;
   end record;
   Head : FIFO_Channel;
   ...
end FIFO;
```

Figure 9. FIFO package specification using dynamic memory allocation

```
generic Size: natural;
type FIFO_Element is private;
package FIFO is
   type FIFO_Type is array (natural range <>)
OF FIFO_Element;
  input : FIFO_Element;
   ...
  procedure Add_FIFO;
  ...
  private
  FIFO : FIFO_Type(0..Size);
  R, W : integer := 0;
  Count : integer:=0;
  ...
end FIFO;
```

Figure 10. FIFO package specification using static memory allocation

4.1.6 Instantiating a New FIFO Channel

Putting generic keyword at the beginning of the FIFO package in addition to have generic size and type would give FIFO package one more useful feature and it is the ability to have multiple instances of that channel as a generic package. Getting instance of the FIFO package which uses dynamic and static memory allocation is shown in Figure 11 and Figure 12 respectively. The only difference between the instantiation in Figure 12 with the previous one in Figure 11 is that we can determine the size of channel besides its element types. In Figure 12 FIFO_Channel1 is instantiated by name and FIFO_Channel2 is instantiated by position.

```
with FIFO;
package FIFO_Channel is new FIFO(Packet);
```

Figure 11. Instantiating a new FIFO channel with dynamic memory allocation

```
with FIFO;

package FIFO_Channel1 is new FIFO (Size=>8,
FIFO_Element=>Packet);
package   FIFO_Channel2   is   new   FIFO   (8,
Packet);
```

Figure 12. Instantiating a new FIFO channel with static memory allocation

4.1.7 Export Concept

TLM communication interfaces are implementable in channels or in target module using export concept[6][7]. Figure 13 shows the concept of exportability in an implemented example in Ada. Implementation of a master-slave architecture using exported TLM_FIFO channel is shown in section 4.2.2.

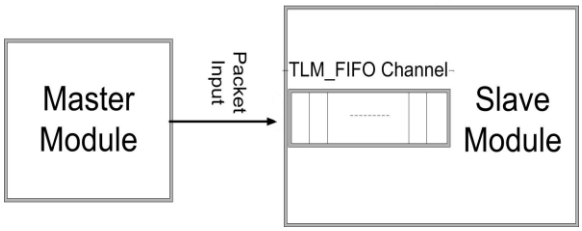


Figure 13. Export concept for TLM_FIFO channel

4.1.8 Multiple Reader/Writer

The Ada TLM_FIFO channel can have multiple reader/writer(s) like the standard SystemC TLM FIFO[7]. Figure 14 shows this concept.

Our TLM_FIFO can have this capability by defining various master and slave tasks that produce and consume data via one TLM_FIFO channel this scenario is illustrated in Figure 15.

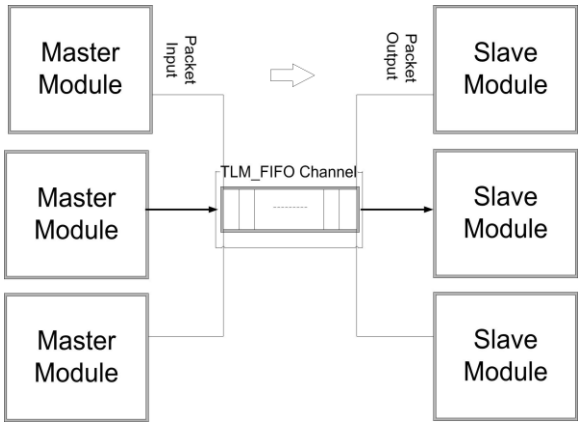


Figure 14. Multiple reader/writer for a TLM_FIFO channel

```
with FIFO_Channel;
procedure Multiple_Reader_Writer_TLM_FIFO is
   task Master1 is
     entry Start;
     entry Stop;
   end Master1;
   ...
   task Master2 is
   ...
   task  Slave1 is
   ...
   task  Slave2 is
   ...
   task Simulation is
   ...
   task body Simulation is
      begin
         accept Start;
         loop
            select
            accept Stop;
               exit;
            else
               Master1.Start;
               Master2.Start;
               ...
               Slave1.Start;
               ...
            end select;
            delay 0.0;
         end loop;
         --stops all the tasks
         Master1.Stop;
         Master2.Stop;
         ...
         Slave1.Stop;
         ...
      end Simulation;
   begin
      Simulation.Start;
      delay 1.0;
      Simulation.Stop;
   end Multiple_Reader_Writer_TLM_FIFO;
```

Figure 15. Multiplr_Reader_Writer_TLM_FIFO procedure

4.2 TLM Master-Slave Architectures Using TLM_FIFO

In order to show how TLM_FIFO channel is used for communication between modules, two different architectures for a master-slave system is implemented using SystemAda. In the first architecture, Master and Slave tasks communicate through an external TLM_FIFO channel, while in the second architecture the channel is defined inside the Slave task in order to show the concept of exportability for SystemAda channels.

4.2.1 Master-Slave Architecture Using External TLM_FIFO Channel

The master-slave architecture is shown in Figure 16. The two modules use the TLM_FIFO channel for their communication. They use method calls to the TLM_FIFO channel for their communication.

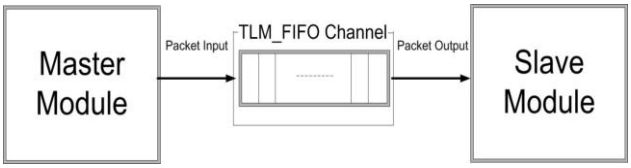


Figure 16. TLM_Master_Slave architecture using TLM_FIFO channel[3]

The body of the procedure which implements the functionality of this architecture is shown in Figure 17. This procedure consists of three tasks[9] called Master, Slave and Simulation. Each task has two entries, Start and Stop, which control the entrance and exit of the program flow to the corresponding task.

```
with FIFO_Channel;
procedure TLM_Master_Slave_FIFO is

  task  Master is
    entry Start;
    entry Stop;
  end Master;

  task  Slave is
    entry Start;
    entry Stop;
  end Slave;

  task Simulation is
    entry Start;
    entry Stop;
  end Simulation;
  ...
```

Figure 17. TLM_Master_Slave_FIFO procedure

When Start entry of the Master task is activated, a packet is read from an input file and is put into the TLM_FIFO. On the Slave side, on the other hand, a

packet is retrieved from the head of the TLM_FIFO when Start entry is activated for Slave task[3].

4.2.1.1 Simulation Task

As it is shown in the following figures (Figure 18, Figure 19), in the main block of the procedure the Simulation task starts its processing and stops from working after a predetermined delay of 1.0 second. This task has the control role for simulation process of the Master and Slave tasks.

```
begin
   Simulation.Start;
   delay 1.0;
   Simulation.Stop;
end TLM_Master_Slave_FIFO;
```

Figure 18. TLM_Master_Slave_FIFO procedure main part

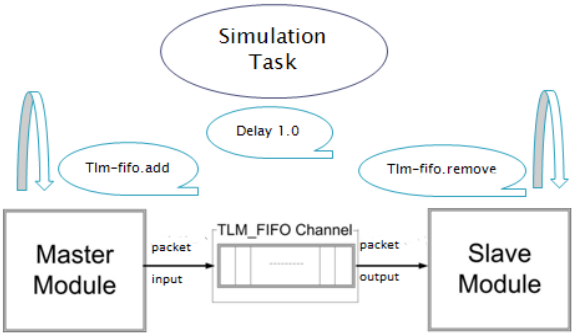


Figure 19. Simulation process of TLM_Master_Slave architecture using TLM_FIFO

```
1: task body Simulation is
2: begin
3:    accept Start;
4:      loop
5:        select
6:          accept Stop;
7:            exit;
8:          else
9:            Master.Start;
10:           Slave.Start;
11:       end select;
12:     end loop;
13:  Master.Stop;
14:  Slave.Stop;
15: end Simulation;
```

Figure 20. Simulation task body

Figure 20 shows the body of Simulation task. After accepting *Start* message, at line 3 while no *Stop* message is received, the *Start* message is sent to Master and Slave tasks (lines 9 and 10). As soon as accepting Stop message by Simulation task at line 6, the exit command executes and exits the loop. Afterwards, Stop messages are sent to Master and Slave modules (lines 13, 14).

4.2.2 Master-Slave Architecture Using Exported TLM_FIFO Channel

A master-slave architecture using exported TLM_FIFO channel is shown in Figure 13. In this architecture, the channel is defined inside the Slave to show the concept of exportability.

```
with FIFO_Channel;
procedure TLM_Master_Slave_FIFO_Ex is

   task  Master is
     entry Start;
     entry Stop;
   end Master;

   task  Slave is
     entry Send(Pack: in Packet);
     entry Stop;
   end Slave;

   task Simulation is
     entry Start;
     entry Stop;
   end Simulation;
    ...
begin
   Simulation.Start;
   delay 1.0;
   Simulation.Stop;
end  TLM_Master_Slave_FIFO_Ex;
```

Figure 21. TLM_Master_Slave_FIFO_Ex procedure for exported channel

Implementation of this architecture is similar to the previous one in that the functionality of the design is defined through a procedure which consists of the tasks Master, Slave and Simulation. However, the implementation of the tasks is slightly different from the previous one. The body of the procedure which implements the functionality of this architecture is shown in Figure 21. Each task has two entries, Start and Stop, except for the Slave task. Slave task has an entry called Send instead of Start entry. Send entry has an argument called Pack which is of type Packet. Type Packet defines the type of the elements of the TLM_FIFO.

The body of slave task is shown in Figure 22. Because in this architecture the TLM_FIFO is moved inside the Slave task, Master task needs to call an entry of Slave task in order to insert the input data into the TLM_FIFO. The *Send* entry is defined for this purpose (line 5). There is an infinite loop which contains a selective accept statement. Each time through this loop the task checks to see if either *Send* or *Stop* has been called (lines 5 and 8). If *Stop* has

been called the exit command is executed, terminating the loop. If *Send* has been called, input of *Send* entry, i.e. Pack, is passed as input to TLM_FIFO and then is put to the TLM_FIFO through calling *Add* entry of FIFO_Task. Then, if the FIFO is not empty, a packet is get from the TLM_FIFO by calling *Remove* entry of FIFO_Task.

```
1: task body Slave is
2: begin
3:  loop
4:   select
5:     accept Send(Pack: in Packet) do
        Pack_FIFO_Ch.Input := Pack;
        Pack_FIFO_Ch.TLM_FIFO.Add;
6:     end Send;
7:     or
8:     accept Stop;
        exit;
9:   end select;
10:    if Pack_FIFO_Ch.Empty_Flag= false
11:     Pack_FIFO_Ch.TLM_FIFO.Remove;
12:    end if;
13:  end loop;
14: end Slave;
```

Figure 22. Slave task body for exported channel

The body of Master task is shown in Figure 23. There is an infinite loop which contains a selective accept statement. Each time through this loop the task checks to see if either *Start* or *Stop* has been called (lines 8, 11). If *Stop* has been called the exit command is executed, terminating the loop. If *Start* has been called, a packet is read from input file to *Pack* variable and then *Send* entry of Slave task is called for this packet. For more information about files in Ada see [19].

```
1:  task body Master is
2:  Input_File: file_type;
3:  Pack: Packet;
4:  begin
5:    open (Input_File,in_file,"input");
6:    loop
7:         select
8:           accept Start do
             if not end_of_file(Input_File) then
               Get(Input_File,Pack);
               Slave.Send(Pack);
             end if;
9:           end Start;
10:          or
11:          accept Stop;
               exit;
12:        end select;
13:   end loop;
14:   close(Input_File);
15: end Master;
```

Figure 23. Master task body for exported channel

Figure 24 shows the body of Simulation task. After accepting *Start* entry (line 3), the task enters an infinite loop in which the *Stop* entry is checked first. If *Stop* has been called, the loop terminates (line 6). Otherwise, *Start* entry of Master task is called and Master task starts its simulation (line 9). In contrast to the previous architecture, there is no need to call *Send* entry for the Slave module since Master module is responsible for calling *Send* entry of Slave task whenever a new data packet is ready to be written to TLM_FIFO. Upon calling *Stop* entry, the loop terminates and *Stop* entries for Master, Slave and TLM_FIFO tasks are called and simulation ends.

```
1: task body Simulation is
2:  begin
3:   accept Start;
4:     loop
5:       select
6:         accept Stop;
7:           exit;
8:         else
9:           Master.Start;
10:      end select;
11:    end loop;
12:    Master.Stop;
13:    Slave.Stop;
14:    Pack_FIFO_ch.TLM_FIFO.Stop;
15:  end Simulation;
```

Figure 24. Simulation task body for exported channel

5 TLM Interfaces

Besides unidirectional/bidirectional, blocking/ non-blocking modes of transfer there are concepts in transaction level modeling called interfaces. These concepts are also patched in system level languages used today. But the interface concept is inherent in Ada2005 and we have used it to develop a basic communication interface. Since Ada is an object oriented language, the interface concept in it is a limited form of multiple inheritance (i.e. a type could inherit several interfaces)[21]. For example, in this case *Communication* interface has inherited from *Put* and *Get* interfaces. These interfaces could be implemented as user design methodology. They could be implemented as separated channels, i.e. TLM channels or they could be implemented in target modules which recall the export concept.

5.1 The Unidirectional Blocking/ Non-blocking Interfaces

The Put functionality in TLM is defined for the transaction which is sent from initiator to the target. The *Put_Interface* package has an interface type called *Put_If* that is passed to its abstract procedure called *Put*. Any package that inherits from this interface should implement this abstract procedure (Figure 25).

```
package Put_Interface is
   type Put_If is interface;
   procedure Put(Item:in Put_If) is abstract;
end Put_Interface;
```

Figure 25. TLM Put interface

The Get functionality is defined for the transaction which is sent from target to the initiator. The *Get_Interface* package contains an interface type called *Get_If*. This interface type is an argument of the *Get* abstract procedure which should be implemented by modules that inherit from this interface (Figure 26).

```
package Get_Interface is
   type Get_If is interface;
   procedure Get(Item:out Get_If) is abstract;
end Get_Interface;
```

Figure 26. TLM Get interface

In the next step, we have defined a communication interface that inherits from Put and Get interfaces. This interface implicitly inherits the *Put* and *Get* procedures and it is optional to put them in its package specification or not. A task interface is added to this package to be implemented where there is a need to synchronous interfaces and concurrency concepts which are the intrinsic features of hardware communications (Figure 27).

```
with Put_Interface, Get_Interface;
use Put_Interface, Get_Interface;

package Communication_Interface is

   type Communication_If is interface and
Put_If and Get_If;

   --procedure Put(Item: in Communication_If)
is abstract;
   --procedure Get ...

   type Communication_Task_If is task
interface;
end Communication_Interface;
```

Figure 27. Communication_Interface package specification

In Figure 28, the package specification of the implementation of this interface is shown. The type which is transferred during this communication is generic and is called Communication_Element (line3). The Communication_If type is specified here through declaring a new type called Communication_Comp. This type extends the Communication_If type with a record type (lines 7, 8, 9). This package should implement the abstract procedures which have inherited (Put and Get). The task interface inherited from above package is implemented here at line 12 (the task entries implements the functionality of the Put and Get procedures).

```
1: with Communication_Interface;
2: use Communication_Interface;

3: generic
4: Size : natural;
5: type Communication_Element is private;

6: package Communication_If_Im is
7:   type Communication_Comp is new
Communication_If with record
8:     Data: Communication_Element;
9:   end record;
10:  procedure Put(Item:in Communication_Comp);
11:  procedure Get(Item:out Communication_Comp);
12:  task type Communication_Task is new
Communication_Task_If with
13:    entry P(Item: in Communication_Comp);
14:    entry G(Item: out Communication_Comp);
15:  end Communication_Task;
16:  task type Communication_Task_B is new
Communication_Task_If with
17:    ...
18:    Com_Buffer_Array: array (0..Size) of
Communication_Comp;
19:    Input: Communication_Comp;
20:    Output: Communication_Comp;
21:    Com_Task: Communication_Task;
22:    Com_Task_B: Communication_Task_B;
23:    ...

24:end Communication_If_Im;
```

Figure 28. Communication_If_Im package specification

Implementation of this package defines two modes of communication: blocking and non-blocking. The method used here to implement these transfer modes is the same as before by modifying tasks bodies. Figure 29 shows instantiating this communication interface which specifies the communication element type.

```
with Communication_If_Im;

package Communication is new
Communication_If_Im(1,integer);
```

Figure 29. Instantiating a new Communication_If_Im

Figure 30 shows an example which uses this communication interface to have a simple unidirectional blocking interface.

```
with Communication;
use Communication;
...
procedure Master_Slave_Communication is
   ...
   task Slave is
     entry Start;
   end Slave;

   task body Slave is
   begin
     loop
       accept Start do
         Com_Task.G(Output);
         …
       end Start;
     end loop;
   end Slave;

   task Master is
   end Master;

   task body Master is
   begin
     For i in 1..10 loop
       Input.Data:=i;
       Com_Task.P(Input);
       Slave.Start;
     end loop;
   end Master;
   begin
   ...
end Master_Slave_Communication;
```

Figure 30. Master_Slave_Communication procedure

5.2 The Bidirectional Blocking/ Non-blocking Interfaces

Figure 31 shows the main members of Transport_Interface. The TLM bidirectional blocking/non-blocking interface consists of two one_directional interface type called *Req_If* and *Res_If*. As their names implies, one of them is for request transactions and the other one is for response transactions.

```
with Communication_Interface;
use Communication_Interface;

package Transport_Interface is
   type     Req_If     is     interface     AND
Communication_If;
   type     Res_If     is     interface     AND
Communication_If;
   type Transport_Task_if is task interface;
end Transport_Interface;
```

Figure 31. Transport_Interface package specification

The main members of the specification package of the implementation of Transport_Interface are shown in Figure 32.

```
with Transport_Interface;
use Transport_Interface;
generic
Size : natural;
type Req_Element is private;
type Res_Element is private;
package Transport_If_Im is
   type Req_Com is new Req_If with record
     Data: Req_Element;
   end record;
   type Res_Com is new Res_If with record
     …
   end record;
   procedure Put(Item : in Req_Com);
   procedure Get(Item : out Req_Com);
   procedure Put(Item : in Res_Com);
   procedure Get(Item : out Res_Com);
   task    type    Transport_Task    is    new
Transport_Task_If with
     entry Preq(Item : in Req_Com);
     entry Greq(Item : out Req_Com);
     entry Pres(Item : in Res_Com);
     entry Gres(Item : out Res_Com);
   end Transport_Task;
   task    type    Transport_Task_B    is    new
Transport_Task_If with
     …
   end Transport_Task_B;
   Req_Buffer: array (0..Size) of Req_Com;
   Res_Buffer: array (0..Size) of Res_Com;
   …
end Transport_If_Im;
```

Figure 32. Transport_If_Im package specification

The following figure (Figure 33) shows how to instantiate a new Transport_If_Im.

```
with Transport_If_Im;
package        Transport        is        new
Transport_If_Im(5,integer,integer);
```

Figure 33. Instantiating a new Transport_If_Im

6 Experimental Results

In order to compare the simulation time of TLM channels implemented using SystemC and Ada, we need an appropriate platform. The properties of the platform we have used are as follows:

- **Operating system**: Microsoft Windows XP Professional, 2002 version, Service pack3
- **System**: Intel Pentium 4, CPU 2GHz, RAM 1GB
- **Compiler:** Gnat GPL 2007 for Ada, Microsoft Visual Studio (.Net Frame Work 2005) For SystemC

The details of the simulations performed are presented in the following sections.

6.1 Ada and SystemC Simulation Time Comparison for TLM_FIFO Channel

Ada has some inherent structural advantages over SystemC. But to observe if Ada has simulation time penalty, we have done several simulation time comparisons between Ada and SystemC implementations of TLM_FIFO channel. According to the diagram shown in Figure 34 based on Table 1, the speed of TLM_FIFO in both models is not equivalent. All in all, we have 81.41 percent speed optimization in our Ada models compared to SystemC implementations, which is surprising and it shows that there should be a specific operation in Ada which is much faster than its equivalent SystemC model.

Table 1.TLM_FIFO average simulation time in SystemAda and SystemC for variable packet numbers (using I/O files)

Packet number	Ada	SystemC	Simulation time ratio	Optimization (%)
10000	0.5056	2.8236	5.584652	82.09378
20000	0.9966	5.34	5.358218	81.33708
30000	1.475	8.025	5.440678	81.61994
40000	2.051	10.8228	5.276841	81.04927
50000	2.4436	12.8328	5.251596	80.95817
average			5.382397	81.41165

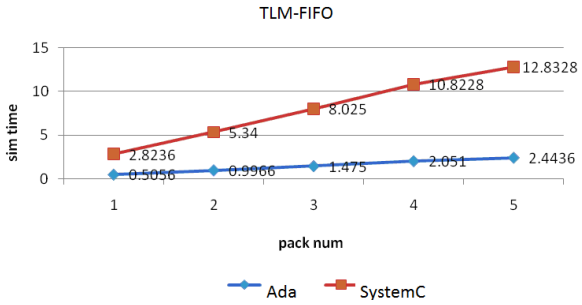


Figure 34. Ada and SystemC TLM_FIFO channel simulation time (using I/O files)

After doing the above experiments we have determined that the operation should be interacting to I/O files so we have omitted this operation and restarts the experiments again. As the following results shows (Table2 , Figure 35) there is also an optimization about 58.35 percent in simulation speed in contrast to SystemC models.

At this time we have no concern about simulation speed at TLM level in SystemAda but if by the passage of time the speed becomes our problem there is no worry again since we have no speed penalty in Ada compared to SystemC. Maybe this speed up is because of the platform or maybe this speed up would not be seen for the implementation of other channels so in our next work we are going to implement other TLM channels based on our FIFO channel and we also try to change the platform of the experiments to see if there would be a time penalty for Ada or not.

Table2 . TLM_FIFO average simulation time in SystemAda and SystemC for variable packet numbers (using no I/O files)

Packet number	Ada	SystemC	Simulation time ratio	Optimization percent
10000	0.3396	0.794	2.338045	57.22922
20000	0.7118	1.625	2.282945	56.19692
30000	1.0256	2.5314	2.468214	59.48487
40000	1.4386	3.614	2.512165	60.19369
50000	1.6988	4.1126	2.420885	58.6928
Average			2.404451	58.3595

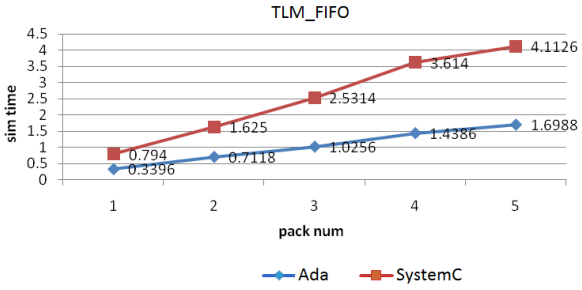


Figure 35. Ada and SystemC TLM TLM_FIFO channel simulation time (using no I/O files)

7 Conclusion

Digital systems and system level modelling of hardware systems are too complex today to the extent that Hardware Description Languages such as VHDL or Verilog are no longer efficient for modelling them. In order to satisfy the requirements of modelling such systems, new languages have emerged such as SystemC, HandelC and CatapultC which perform the design at higher levels of education. Ada is another language which has several advantages over

commonplace system level languages like SystemC. Therefore, we have examined Ada to model a complete system at transaction level. A number of packages have been developed using Ada for transaction level modelling. We refer to these packages and form of Ada language used here as SystemAda. In an earlier work, we have shown that Ada has a link to RTL and we have implemented a TLM_FIFO channel. Here we have proved the concept of TLM in Ada by improving our TLM_FIFO channels with its detailed characteristics proof and presenting TLM interfaces concepts and modelled a simple architecture with this new and detailed channel for our simulation speed experiments. Several experiments have been conducted to compare the efficiency of SystemAda versus SystemC TLM. Experimental results show that there is no simulation time overhead when using SystemAda as compared to SystemC TLM.

8 References

[1] H. Alemzadeh, S. Di Carlo, F. Refan, P. Prinetto, and Z. Navabi, "Plug & Test at System Level via Testable TLM Primitives," Proc. ITC, 2008, Paper 23.1, pp. 1-10.

[2] S. Mirkhani, and Z.Navabi, The VLSI Handbook, Chapter 86, CRC Press, 2ed Edition, 2006.

[3] Negin Mahani, "Making Alive Register Transfer Level and Transaction Level Modeling in Ada", ACM SIGAda AdaLetters, Volume 31, Issue 2, August 2011, pp.15-23.

[4] S. Wong, and Gertrude Levine, "Kernel Ada to Unify Hardware and Software Design", Proc. Annual ACM SIGAda International Conference on Ada, 1998, pp. 28-38.

[5] J. E. Sammet, "Why Ada is not Just another Programming Language", Communications of the ACM, vol. 29, no. 8, August 1986, pp. 722-732.

[6] S. Swan, "OSCI SystemC TLM", Available at: http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-13-OSCI_2_swan.pdf.

[7] "SystemC TLM1.0", Available at: http://www.systemc.org/home.

[8] M. Mills, and G Peterson, "Hardware/Software Codesign: VHDL and Ada 95 Code Migration and Integrated Analysis", Proc. Annual ACM SigAda international conference on Ada, 1998, pp. 18-27.

[9] D. A. Wheeler, "Lovelace tutorial", Section 13.2-Creating and Communicating with Tasks, Available at: http://www.dwheeler.com/ lovelace/ s13s2.htm

[10] Negin Mahani, Parniyan Mokri, Mahshid Sedghi, Zainalabedin navabi, "System Ada : An Ada based Syste- Level Hardware Description Language" ACM SIGADA AdaLetters, vol. XXIX , no. 2 , August 2009 , pp. 15-19.

[11] Stephen F. Zeigler, Comparing Development Costs of C and Ada, March 30, 1995 Ph.D Rational Software Corporation, Available at: http://www.adaic.com/whyada/ada-vs-c/cada_art.html

[12] Business Benefits: Available at: http://www.adacore.com/ home/ada_answers/ business_benefits/

[13] Why Ada? Available at: http://www.adaic.org/whyada/WhyAda.html

[14] Ada95 RM, Section3: Declaration and Types, Available at: http://www.adahome.com/rm95/

[15] Ada95 RM, Section7: packages, Available at: http://www.adahome.com/rm95/

[16] Ada95 RM, Section12: Generic Units, Available at: http://www.adahome.com/rm95/

[17] David A. Wheeler; ,Ada Lovelace Tutorial, section6.5 Arrays; Available at: www.dwheeler.com/lovelace.

[18] David A. Wheeler; Ada Lovelace Tutorial, section13.1 Tasking Basics ; Available at: www.dwheeler.com/lovelace.

[19] David A. Wheeler; ,Ada Lovelace Tutorial, section9.1-Simple Text File Input/Output; Available at: www.dwheeler.com/lovelace.

[20] Barnes, J.G.P. (1989). Programming in Ada, Chapter5 Section5.3 –loop Statement

[21] Ada Refrence Manual-legal information section 3.9.4 interface types, Available at: http://www.adaic.com/standards/05rm/html/rm-3-9-4.html

[22] R. K. Gedela, S. M. Shatz and Haiping Xu, "Visual Modeling of Synchronization Methods for Concurrent Objects in Ada 95", Available at: http://www.sigada.org/conf/sigada99/Paper22.pdf

[23] Comparison of Ada and C++ features, Available at: http://www.adahome.com/articles/1997-03/ada_vs_cpp.html

[24] System Level Design, Available at: http://chipdesignmag.com/sld/mcdonald/2009/02/19/esl-languages-which-one-is-right-for-your-needs/

New Types of Tasking Deadlocks in Ada 2012 Programs

Takeo Ekiba, Yuichi Goto, and Jingde Cheng

Department of Information and Computer Sciences, Saitama University,
Saitama, 338-8570, Japan
+81-48-858-3427
{ekiba, gotoh, cheng}@aise.ics.saitama-u.ac.jp

Abstract. This article presents some new types of tasking deadlocks concerning the new synchronization waiting relations defined in Ada 2012.

1 Introduction

A tasking deadlock in a concurrent Ada program is a situation where some tasks form a circle of synchronization waiting relations at some synchronization points that cannot be resolved by the program itself (including the behavior of other tasks), and hence can never proceed with their computation by themselves [2, 3]. To deal with Ada tasking deadlocks, it is indispensable to identify all types of tasking deadlocks.

A synchronization waiting relation between tasks is a such relation that to synchronize with other task or tasks, a task is blocked until the synchronization takes place, unless the synchronization waiting has a deadline. Cheng proposed a way to completely classify all tasking deadlocks by different combinations of various synchronization waiting relations between tasking objects [2, 3]. According to this way of classification, new combinations of synchronization waiting relations concerning new synchronization waiting relations may lead to new types of tasking deadlocks.

Ada 2012 defined four new operations that cause a situation where a task is blocked, i.e., procedure *Enqueue* and *Dequeue* in the package Synchronized_Queue_Interfaces, procedure *Wait_For_Release* in the package Synchronous_Barriers and procedure *Suspend_Until_True_And_Set_Deadlines* in the package Ada.Synchronous_Task_Control.EDF. Since the operations may involve synchronization waiting relations among tasks such that they form a tasking deadlock, an Ada 2012 program using the operations may have some tasking deadlocks that are not identified until now.

In this article, we present some examples of new tasking deadlocks concerning the new synchronization waiting relations defined in Ada 2012.

2 Synchronization Waiting Relations in Ada 2012 Programs

Ada 95 defines eight types of synchronization waiting relations, i.e., activation waiting relation, finalization waiting relation, completion waiting relation, acceptance waiting relation, entry-calling waiting relation, protection waiting relation, protected-entry-calling waiting relation, and suspension waiting [1, 4–6].

In Ada 2005, there is no new synchronization waiting relations. There are some changes for the tasking and real-time facilities in Ada 2005. For instance, one major extension is the ability to combine the interface feature with the tasking model. There are also many additional predefined packages in the Real-Time Systems annex concerning matters such as scheduling and timing [7]. The changes have no effect on synchronization waiting relations which have been defined. They also do not cause new synchronization waiting relations.

Ada 2012 defined four new operations that cause a situation where a task is blocked. The three of the four operations, i.e., the enqueue waiting relation and the dequeue waiting relation defined in Annex A.18.27 and the barrier-release waiting relation defined in Annex D.10.1, may cause the new synchronization waiting relations. The other, i.e., procedure *Suspend_Until_True_And_Set_Deadlines* defined in Annex D.10, do not cause new synchronization waiting relation.

Annex A.18.27 (The Generic Package Containers.Synchronized_Queue_Interfaces) provides interface type Queue and a set of operations for that types. A queue type that implements this interface is allowed to have a bounded capacity. If a queue object has a bounded capacity and the number of existing elements in the queue equals the capacity, then a procedure call for Enqueue will result in that the task calling Enqueue is blocked until storage becomes available. *Enqueue waiting* is that a task calling Enqueue is blocked when a queue in implemented Synchronized_Queue_Interfaces is full until another task calls Dequeue for this queue. If a queue is empty, then a procedure call for Dequeue will result in that the task calling Dequeue is blocked until an item becomes available. *Dequeue waiting* is that a task calling Dequeue is blocked when a queue in implemented Synchronized_Queue_Interfaces is empty until another task calls Enqueue for this queue. Enqueue waiting may be resolved by other task calling Dequeue. If no task has the possibility to resolve enqueue waiting, a task in enqueue waiting state will be blocked forever and may be involved with a tasking deadlock. Dequeue waiting may be resolved by other task calling Enqueue. If no task has the possibility to resolve dequeue waiting, a task in dequeue waiting state will be blocked forever and may be involved with a tasking deadlock.

Annex D.10.1 (Synchronous Barriers) provides a language-defined package to synchronously release a group of tasks after the number of blocked tasks reaches a specified count value. Each procedure calling *Wait_For_Release* results in that the tasks calling Wait_For_Release is blocked until the number of blocked tasks associated with the Synchronous_Barrier object is equal to threshold, when all blocked tasks are released. *Barrier-release waiting* is that tasks calling *Wait_For_Release* are blocked by other possible tasks calling it until the number of blocked tasks associated with the Synchronous_Barrier object is equal to threshold. If the needed number of all tasks which plan to call *Wait_For_Release* reaches to threshold, they can resolve barrier-release waiting. If any tasks can not have possibility to call *Wait_For_Release*, a task in barrier-release waiting state will be blocked forever and may be involved with a tasking deadlock.

Annex D.10 (Synchronous Task Control) defines procedure *Suspend_Until_True_And_Set_Deadline*. A procedure call for Suspend_Until_True_And_Set_Deadline may re-

sult in that the calling task is blocked temporally. Hence, the call does not cause any synchronization waiting relation.

3 Examples of Tasking Deadlocks in Ada 2012 Programs

This section shows six examples. Each example has an Ada 2012 code and its Task-Wait-For Graph. All of the examples have tasking deadlocks with the new synchronization waiting relations, however they are not all programs including tasking deadlocks. Four examples have tasking deadlocks with the new synchronization waiting relations. Other one example has two kinds of tasking deadlocks with new synchronization waiting relations, but each of the tasking deadlocks occurs exclusively. Another example may have three kinds of tasking deadlocks with new synchronization waiting, but they each of the tasking deadlocks occur exclusively, otherwise any of them do not occur.

3.1 Task-Wait-For Graph

A Task-Wait-For Graph (TWFG for short) is a kind of arc-classified digraph to represent tasking waiting state in an execution of an Ada program [3]. The TWFG explicitly represents various types of synchronization waiting relations in an execution of an Ada program. The notion of TWFG was originally proposed for classification and detection of tasking deadlocks in Ada 83 programs [2, 3] and was extended to deal with tasking deadlocks in Ada 95 programs [4]. In a TWFG, vertices indicate tasking objects. A tasking object in an execution state of a concurrent Ada 2012 program is any of the following: a task whose activation has been initiated and whose state is not terminated, a block statement that is being executed by a task, a subprogram that is being called by a task, a protected subprogram that is being called by a task, a protected object on which a protected action is undergoing, and a suspension object that is being waited by a task. Arcs indicate synchronization waiting relations which are binary relations between tasking objects. In a TWFG of a programs with Ada 2012, they are 11 types of arcs: activation waiting arc, finalization waiting arc, completion waiting arc, acceptance waiting arc, entry-calling waiting arc, protection waiting arc, protected-entry-calling waiting arc, suspension waiting arc, enqueue waiting arc, dequeue waiting arc, and barrier-release waiting arc, respectively, corresponding to an activation waiting relation, finalization waiting relation, completion waiting relation, acceptance waiting relation, entry-calling waiting relation, protection waiting relation, protected-entry-calling waiting relation, suspension waiting relation, enqueue waiting relation, dequeue waiting relation, and barrier-release waiting relation.

Here the affixing characters show types of arcs. $\rightarrow_{Act}$, $\rightarrow_{Fin}$, $\rightarrow_{Com}$, $\rightarrow_{Acc}$, $\rightarrow_{EC}$, $\rightarrow_{Pro}$, $\rightarrow_{PEC}$, $\rightarrow_{Sus}$, $\rightarrow_{Enq}$, $\rightarrow_{Deq}$, and $\rightarrow_{BR}$ denote activation waiting arc, finalization waiting arc, completion waiting arc, acceptance waiting arc, entry-calling waiting arc, protection waiting arc, protected-entry-calling waiting arc, suspension waiting arc, enqueue waiting arc, dequeue waiting arc, barrier-release waiting arc, respectively.

3.2 Example Program 1: Tasking Deadlocks with Enqueue Waiting Relations

This program has enqueue waiting relations. This two enqueue waiting relations form a cycle that cannot be resolved by the program itself. Enqueue waiting cycle in this program are as follows:

$$T_1 \rightarrow_{Enq} T_2 \rightarrow_{Enq} T_1$$

Example Program 1

```
with Ada.Containers.
          Synchronized_Queue_Interfaces;
with Ada.Containers.
          Bounded_Synchronized_Queues;
with Ada.Text_IO;
use  Ada.Containers;
use Ada.Text_IO;
procedure dl1 is
   subtype Queue_Element is String (1 .. 0);
   package String_Queues is
      new Synchronized_Queue_Interfaces (
      Element_Type => Queue_Element);
      package S_Queues is
      new Bounded_Synchronized_Queues (
      Queue_Interfaces => String_Queues,
      Default_Capacity => 1);
   Q1, Q2, Q3 : S_Queues.Queue;
   Element1   : Queue_Element
   task T1;
   task T2;
   task T3;
   task body T1 is
   begin
      delay 1.0;
      loop
         Q2.Enqueue ("");
         Q1.Dequeue (Element1);
      end loop;
   end T1;

   task body T2 is
   begin
      delay 1.0;
      loop
         Q2.Dequeue (Element1);
         Q1.Enqueue ("");
         Q2.Dequeue (Element1);
         Q3.Enqueue ("");
      end loop;
   end T2;

   task body T3 is
   begin
      delay 1.0;
      loop
         Q2.Enqueue ("");
         Q3.Dequeue (Element1);
      end loop;
   end T3;

begin
   Q1.Enqueue ("");
   Q2.Enqueue ("");
   Q3.Enqueue ("");
end dl1;
```

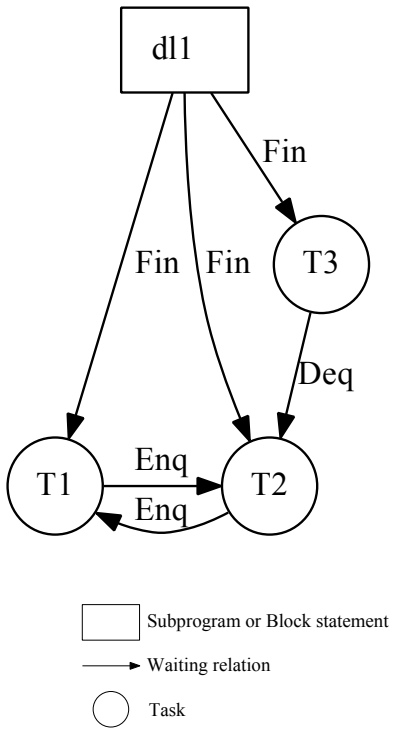


Fig. 1. TWFG of example program 1

3.3 Example Program 2: Tasking Deadlocks with Dequeue Waiting Relations

This program has dequeue waiting relations. This two dequeue waiting relations form a cycle that cannot be resolved by the program itself. Dequeue waiting cycle in this program are as follows:

$$T_1 \rightarrow_{Deq} T_2 \rightarrow_{Deq} T_1$$

Example Program 2

```
with Ada.Containers.
           Synchronized_Queue_Interfaces;
with Ada.Containers.
           Bounded_Synchronized_Queues;
with Ada.Text_IO;
use Ada.Containers;
use Ada.Text_IO;

procedure dl2 is
   type Queue_Element is new String(1..0);

   package String_Queues is
     new Synchronized_Queue_Interfaces
     (Element_Type => Queue_Element);

   package String_Priority_Queues is
     new Bounded_Synchronized_Queues
     (Queue_Interfaces => String_Queues,
      Default_Capacity => 1);

   Q1, Q2, Q3 : String_Priority_Queues.Queue;
   El1 : Queue_Element;

   task T1;
   task T2;
   task T3;
   task body T1 is
   begin
      loop
         Q2.Dequeue (El1);
         Q1.Enqueue ("");
      end loop;
   end T1;

   task body T2 is
   begin
      loop
         Q2.Enqueue ("");
         Q1.Dequeue (El1);
         Q2.Enqueue ("");
         Q3.Dequeue (El1);
      end loop;
   end T2;

   task body T3 is
   begin
      loop
         Q2.Dequeue (El1);
         Q3.Enqueue ("");
      end loop;
   end T3;

begin
null;
end dl2;
```

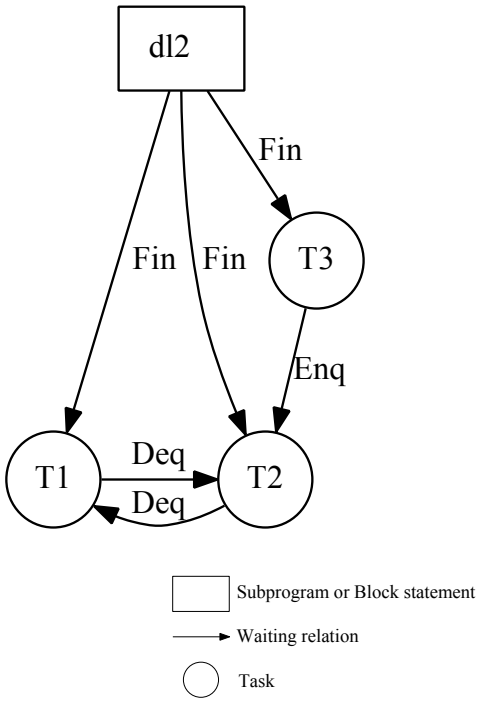


Fig. 2. TWFG of example program 2

3.4 Example Program 3: Tasking Deadlocks with Barrier-Release Waiting Relations

This program has barrier-release waiting relations. This two barrier-release waiting relations form a cycle that cannot be resolved by the program itself. Barrier-release waiting cycle in this program are as follows:

$$T_1 \rightarrow_{BR} T_2 \rightarrow_{BR} T_1$$

Example Program 3

```
pragma Task_Dispatching_Policy
  (Fifo_Within_Priorities);
with System;
with Ada.Synchronous_Barriers;
with Ada.Synchronous_Task_Control;
with Ada.Text_IO;
use  Ada.Synchronous_Barriers;
use Ada.Text_IO;

procedure dl3 is
   Number_Of_Tasks : constant := 2;
   BO1, BO2, BO3   :
   Ada.Synchronous_Barriers.Synchronous_Barrier
     (Number_Of_Tasks);
   Notif           : Boolean := False;

   task T1;
   task T2;
   task T3;

   task body T1 is
   begin
      loop
         Wait_For_Release (BO2, Notif);
         Wait_For_Release (BO1, Notif);
      end loop;
   end T1;
   task body T2 is
   begin
      loop
         Wait_For_Release (BO2, Notif);
         Wait_For_Release (BO1, Notif);
         Wait_For_Release (BO3, Notif);
      end loop;
   end T2;
   task body T3 is
   begin
      loop
         Wait_For_Release (BO2, Notif);
         Wait_For_Release (BO3, Notif);
      end loop;
   end T3;

begin
   null;
end dl3;
```

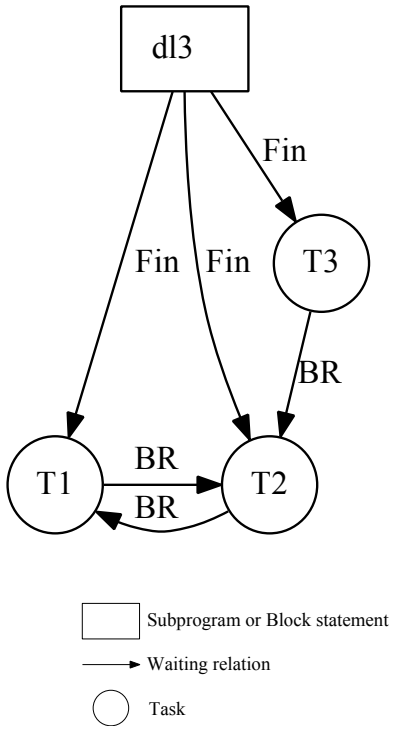


Fig. 3. TWFG of example program 3

3.5 Example Program 4: Tasking Deadlocks with All Synchronization Waiting Relations

This program has 11 types of synchronization waiting relations. The synchronization waiting relations form five cycles that cannot be resolved by the program itself. Therefore, five tasking deadlocks must occur in this program. The five cycles of synchronization waiting relations in this program are as follows:

$$T_1 \rightarrow_{Acc} T_4 \rightarrow_{PEC} V \rightarrow_{Fin} GET1 \rightarrow_{EC} T_2 \rightarrow_{Com} T_1$$

$$T_1 \rightarrow_{Acc} T_5 \rightarrow_{Fin} B \rightarrow_{Fin} T_7 \rightarrow_{Sus} T_2 \rightarrow_{Com} T_1$$

$$T_1 \rightarrow_{Acc} T_5 \rightarrow_{Fin} B \rightarrow_{Fin} T_8 \rightarrow_{Deq} T_6 \rightarrow_{Fin} T_9 \rightarrow_{Fin} GET2 \rightarrow_{BR} T_2 \rightarrow_{Com} T_1$$

$$T_1 \rightarrow_{Acc} T_5 \rightarrow_{Fin} B \rightarrow_{Fin} W \rightarrow_{Pro} V \rightarrow_{Fin} GET1 \rightarrow_{EC} T_2 \rightarrow_{Com} T_1$$

$$T_4 \rightarrow_{PEC} V \rightarrow_{Fin} GET1 \rightarrow_{EC} T_2 \rightarrow_{Com} T_3 \rightarrow_{Enq} T_4$$

Example Program 4

```
pragma Task_Dispatching_Policy
  (Fifo_Within_Priorities);
with System;
with Ada.Containers
          .Synchronized_Queue_Interfaces;
with Ada.Containers
          .Bounded_Synchronized_Queues;
with Ada.Synchronous_Task_Control;
with Ada.Synchronous_Barriers;
with Ada.Text_IO;
use Ada.Synchronous_Task_Control;
use Ada.Synchronous_Barriers;
use Ada.Containers;
use Ada.Text_IO;

procedure dl4 is
   type Queue_Element is new String(1..0);
   package String_Queues
    is new Synchronized_Queue_Interfaces
     (Element_Type => Queue_Element);
   package String_Priority_Queues
    is new Bounded_Synchronized_Queues
     (Queue_Interfaces => String_Queues,
      Default_Capacity => 1);
   Q1, Q2: String_Priority_Queues.Queue;
   El : Queue_Element;
   Number_Of_Tasks : constant := 2;
   BO   :
   Ada.Synchronous_Barriers
   .Synchronous_Barrier(Number_Of_Tasks);
   Notif           : Boolean := False;

   type ITEM is new Integer;
   task T1 is
      entry E1;
   end T1;
   task T2 is
      entry E2;
   end T2;
   task T3;
   task T4;
   S : Suspension_Object;
   function GET1 return ITEM is
   begin
      T2.E2;
      return 0;
   end GET1;
   function GET2 return ITEM is
   begin
      Wait_For_Release(BO, Notif);
      return 0;
   end GET2;
   protected V is
      procedure W (X : in ITEM);
      entry R (X : out ITEM);
   private
      Var : ITEM := 0;
   end V;
   protected body V is
      procedure W (X : in ITEM) is
      begin
         Var := X;
      end W;
      entry R (X : out ITEM) when True is
      begin
         X := GET1;
      end R;
   end V;
```

```
   task body T1 is
      task T5;
      task T6;
      task body T5 is
      begin
         B : declare
            task T7;
            task T8;
            task body T7 is
            begin
               Suspend_Until_True(S);
            end T7;
            task body T8 is
            begin
               Q2.Dequeue(El);
            end T8;
            Y : ITEM;
         begin
            V.W (Y);
         end B;
         T1.E1;
      end T5;
      task body T6 is
         task T9;
         task body T9 is
            I : ITEM := GET2;
         begin
            null;
         end T9;
      begin
         Q2.Enqueue("");
      end T6;
   begin
      accept E1;
   end T1;
   task body T2 is
   begin
      select when False =>
            accept E2;
      or
         terminate;
      end select;
      Set_True (S);
      Wait_For_Release(BO, Notif);
   end T2;
   task body T3 is
   begin
      Q1.Enqueue("");
      Q1.Enqueue("");
   end T3;
   task body T4 is
      Z : ITEM;
   begin
      V.R (Z);
      Q1.Dequeue(El);
      T1.E1;
   end T4;
begin
   null;
end dl4;
```

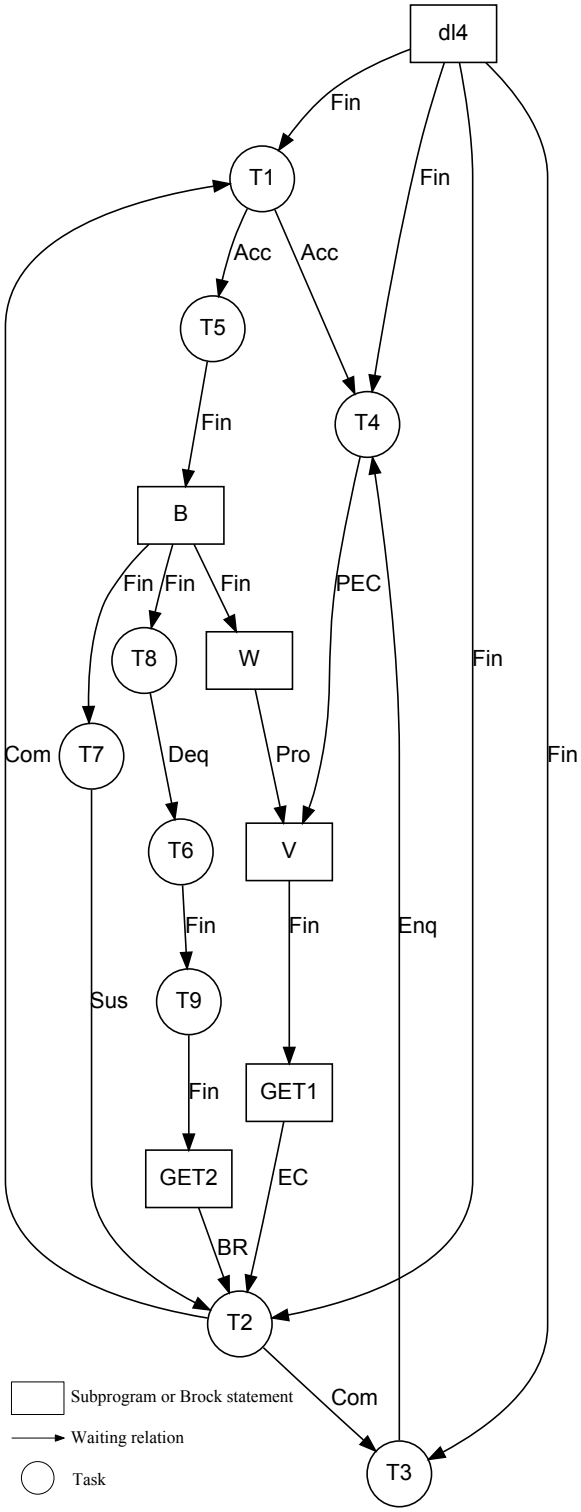


Fig. 4. TWFG of example program 4

3.6 Example Program 5: Two Tasking Deadlocks Each of Which Will Occurs Exclusively

This program has two kinds of tasking deadlocks each of which occurs exclusively. Which of the deadlocks occur depends on execution environment of the program. T1 and T2 may form dequeue waiting and barrier-release waiting cycle. Otherwise T3 and T4 may form enqueue waiting and barrier-release waiting cycle. If T3 calls Enqueue earlier than calling Enqueue of T1, then the cycle of synchronization waiting relations in this program is

$$T_1 \rightarrow_{Deq} T_2 \rightarrow_{BR} T_1.$$

Otherwise, the other cycle is

$$T_3 \rightarrow_{Enq} T_4 \rightarrow_{BR} T_3.$$

Example Program 5

```
pragma Task_Dispatching_Policy
  (Fifo_Within_Priorities);
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;
with Ada.Synchronous_Barriers;
with Ada.Synchronous_Task_Control;
with Ada.Text_IO;
use  Ada.Synchronous_Barriers;
use Ada.Text_IO;
procedure dl6 is
   type Queue_Element is new String(1..0);
   package String_Queues is
    new Ada.Containers.Synchronized_Queue_Interfaces
     (Element_Type => Queue_Element);
   package String_Priority_Queues is
    new Ada.Containers.Bounded_Synchronized_Queues
     (Queue_Interfaces => String_Queues,
      Default_Capacity => 1);
   Q1, Q2 : String_Priority_Queues.Queue;
   El : Queue_Element;
   Number_Of_Barrier_Tasks : constant := 2;
   BO1, BO2   :
   Ada.Synchronous_Barriers.Synchronous_Barrier
     (Number_Of_Barrier_Tasks);
   Notif           : Boolean := False;
   task T1;
   task T2;
   task T3;
   task T4;
   task body T1 is
   begin
      Q1.Dequeue(El);
      Wait_For_Release(BO1, Notif);
      Q2.Enqueue("");
   end T1;
   task body T2 is
   begin
      Q1.Enqueue("");
      Wait_For_Release(BO1, Notif);
      Q1.Enqueue("");
   end T2;
   task body T3 is
   begin
      Q1.Dequeue(El);
      Q2.Enqueue("");
      Wait_For_Release(BO2, Notif);
      end T3;
      task body T4 is
      begin
         Q2.Dequeue(El);
         Wait_For_Release(BO2, Notif);
         Q2.Dequeue(El);
      end T4;
   begin
      Q2.Enqueue("");
   end dl6;
```

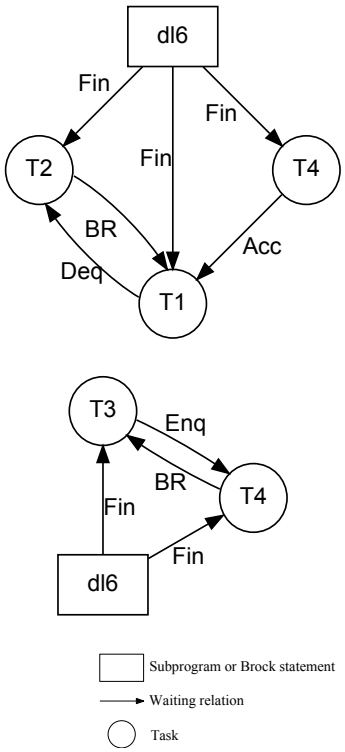


Fig. 5. TWFG of example program 5

3.7 Example Program 6: Three Tasking Deadlocks Each of Which May Occurs Exclusively

This program is just like Program 5. The barrier objects BO1 and BO2 change into BO as new barrier objects in this program. This is a program to exploit a characteristic in Synchronous barrier. This may have three kinds of tasking deadlocks with new synchronization waiting, but they each of the tasking deadlocks occur exclusively, or otherwise any of them do not occur. Each of three pairs of tasks may form a cycle of synchronization waiting relations in this program. The cycle may be

$$T_1 \rightarrow_{Deq} T_2 \rightarrow_{BR} T_1,\ \ T_2 \rightarrow_{BR} T_3 \rightarrow_{Deq} T_2,\ \ T_3 \rightarrow_{Enq} T_4 \rightarrow_{BR} T_3.$$

Example Program 6

```
pragma Task_Dispatching_Policy
  (Fifo_Within_Priorities);
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;
with Ada.Synchronous_Barriers;
with Ada.Synchronous_Task_Control;
with Ada.Text_IO;
use  Ada.Synchronous_Barriers;
use Ada.Text_IO;
procedure dl7 is
   type Queue_Element is new String(1..0);
   package String_Queues is
    new Ada.Containers.Synchronized_Queue_Interfaces
    (Element_Type => Queue_Element);
   package String_Priority_Queues is
     new Ada.Containers.Bounded_Synchronized_Queues
     (Queue_Interfaces => String_Queues,
      Default_Capacity => 1);
   Q1, Q2 : String_Priority_Queues.Queue;
   El : Queue_Element;
   Number_Of_Barrier_Tasks : constant := 2;
   BO   :
   Ada.Synchronous_Barriers.Synchronous_Barrier
     (Number_Of_Barrier_Tasks);
   Notif           : Boolean := False;
   task T1;
   task T2;
   task T3;
   task T4;
   task body T1 is
   begin
      Q1.Dequeue(El);
      Wait_For_Release(BO, Notif);
      Q2.Enqueue("");
   end T1;
   task body T2 is
   begin
      Q1.Enqueue("");
      Wait_For_Release(BO, Notif);
      Q1.Enqueue("");
   end T2;
   task body T3 is
   begin
      Q1.Dequeue(El);
      Q2.Enqueue("");
      Wait_For_Release(BO, Notif);
   end T3;
   task body T4 is
   begin
      Q2.Dequeue(El);
      Wait_For_Release(BO, Notif);
      Q2.Dequeue(El);
   end T4;
begin
   Q2.Enqueue("");
end dl7;
```

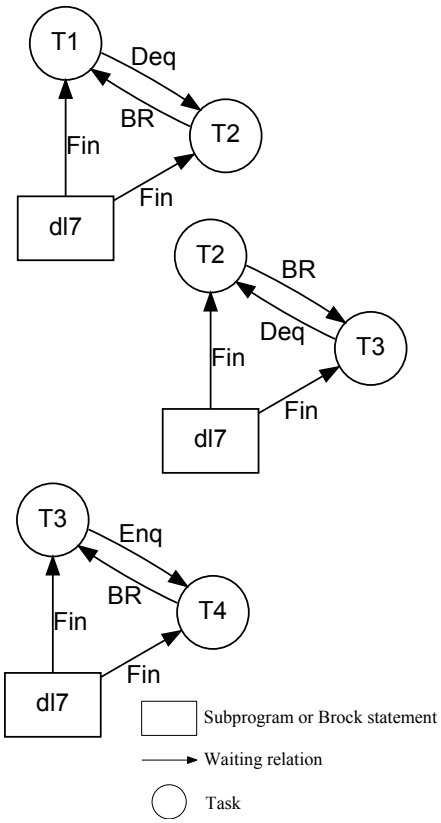


Fig. 6. TWFG of example program 6

4 Concluding Remarks

We have presented some new types of new tasking deadlocks concerning new synchronization waiting relations, i.e., enqueue waiting relation, dequeue waiting relation and barrier-release waiting relation, defined in Ada 2012. We are extending our run-time detection tool [5, 9] to detect the new tasking deadlocks in Ada 2012 programs.

References

1. Barnes, J.: Programming in Ada 2005, Addison-Wesley, 2006.
2. Cheng, J.: A Classification of Tasking Deadlocks, ACM Ada Letters, Vol. 10, No. 5, pp. 110–127, 1990.
3. Cheng, J.: Task-Wait-For Graphs and Their Application to Handling Tasking Deadlocks, Proc. 3rd ACM Annual TRI-Ada Conference, pp. 376–390, 1990.
4. Cheng, J., Ushijima, K.: Tasking Deadlocks in Ada 95 Programs and Their Detection, in A. Strohmeier (ed.), "Reliable Software Technologies - Ada-Europe '96, 1996 Ada-Europe International Conference on Reliable Software Technologies, Montreux, Switzerland, June 10-14, 1996, Proceedings," Lecture Notes in Computer Science, Vol. 1088, pp. 135-146, Springer-Verlag, 1996.
5. Cheng, J.; Run-Time Detection of Tasking Deadlocks in Real-Time Systems with the Ada 95 Annex of Real-Time Systems, "Reliable Sofware Technologies - Ada-Europe 2006, 11th International Conference on Reliable Software Technologies, Porto, Portugal, June 5-6, 2006, Proceedings," Lecture Notes in Computer Science, Vol. 4006, pp. 167–178, Springer-Verlag, 2006.
6. International Organization for Standardization Information Technology - Programming Language - Ada, ISO/IEC 8652:1995(E), 1995.
7. International Organization for Standardization Information Technology - Programming Language - Ada, ISO/IEC 8652:2007(E) Ed. 3, 2006.
8. International Organization for Standardization Information Technology - Programming Language - Ada, ISO/IEC 8652:2012, 2012.
9. Nonaka, Y., Cheng, J., Ushijima, K.: A Tasking Deadlock Detector for Ada 95 Programs, Ada User Journal, Vol. 20, No. 1 pp. 79–92, 1999.

Call for Participation

18th International Conference on
Reliable Software Technologies - Ada-Europe 2013

10-14 June 2013, Berlin, Germany

http://www.ada-europe.org/conference2013

Organized by Ada-Germany on behalf of Ada-Europe, in cooperation with ACM SIGAda, SIGBED, SIGPLAN

Conference AND hotel bookings ONLY via conference web site

The 18th International Conference on Reliable Software Technologies - Ada-Europe 2013 will take place in Berlin, Germany, from June 10 to 14, 2013. The conference will offer an outstanding technical program, invited talks, plus an exhibition from Tuesday to Thursday, and a series of tutorials on Monday and Friday.

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

The 16-page Preliminary Program (PP) brochure with extensive information is available on the conference web site. The PP contains the list of accepted papers and industrial presentations, as well as detailed descriptions of all tutorials, keynote presentations, and panel sessions. In the menu on the home page first select "Program", and then the PP download link. Also check the conference web site for registration, accommodation and travel information.

Quick overview
- Mon 10 & Fri 14: tutorials
- Tue 11 - Thu 13: core program

Proceedings
- published by Springer
- volume 7896 in Lecture Notes in Computer Science series (LNCS) will be available at conference

Program co-chairs
- Hubert B. Keller, Karlsruhe Institute of Technology, Germany; hubert.keller@kit.edu
- Erhard Plödereder, University of Stuttgart; Germany ploedere@iste.uni-stuttgart.de

Invited speakers
- Bruce Powel Douglass, Chief Evangelist IBM Rational, USA, "Model-based Ada Development for DO-178B/C and the Application of Agile Methods"
- Jack G. Ganssle, The Ganssle Group, USA, "The Way Ahead in Software Engineering: Replacing Artists with Disciplined Grownups"
- Giorgio C. Buttazzo, Scuola Superiore Sant'Anna of Pisa, Italy, "Research Challenges in Exploiting Multi-Core Platforms for Real-Time Applications"

Tutorials (half day)
- "Multicore Programming using Divide-and-Conquer and Work Stealing", S. Tucker Taft, AdaCore, USA
- "Designing and Checking Coding Standards for Ada", Jean-Pierre Rosen, Adalog, France
- "Effective Requirements Development Practices and their Role in Effective Design", William Bail, The MITRE Corporation, USA
- "Understanding Dynamic Memory Management in Safety Critical Java", Tom Grosman, Kelvin Nilsen, Atego Systems, Inc., USA

- "Developing Code Analysis Applications with ASIS", Jean-Pierre Rosen, Adalog, France
- "Verification and Validation Techniques for Dependable Systems", William Bail, The MITRE Corporation, USA
- "Testing Real-Time Software", Ian Broster, Rapita Systems, UK
- "Service-Oriented Architecture and Enterprise Service Bus", Rick Sward, The MITRE Corporation, USA, Jeff Boleng, The Software Engineering Institute, Pittsburgh, USA
- "Developing High-Integrity Systems with GNAT GPL and the Ravenscar Profile", Juan de la Puente and Juan Zamorano, Universidad Politécnica de Madrid, Spain
- "Maximize your Application Potential", David Sauvage, AdaLabs Ltd, Republic of Mauritius

Tutorials (full day)
- "Design of Multitask Software: the Entity-Life Modeling Approach", Bo Sandén, Colorado Technical University, USA

Panel session "How to Use the Heap in Real-Time Systems":
- "On Region-Based Storage Management for Parallel Programming", S. Tucker Taft, AdaCore, USA
- "Extending the Java Type System to Enforce Disciplined Use of Scope-Allocated Objects", Tom Grosman, Atego Systems, USA
- "On Dynamic Memory Management in Real-Time, Safety-Critical System", James Hunt, aicas GmbH, Germany

Papers and Presentations
- 11 refereed technical papers in sessions on Multicore and Distributed Systems, Ada and Spark, Dependability, Real-Time Systems
- 5 industrial presentations in 2 sessions
- 1 special session "Introducing Ada 2012"
- Submissions by authors from 17 countries, and accepted contributions from Canada, Denmark, France, Germany, Italy, Mauritius, Portugal, Spain, Switzerland, UK and USA.

Vendor exhibition
- 6 exhibitors already committed: AdaCore, Atego, Ellidiss Software, ETAS, Rapita, and Vector Software
- Vendor presentation sessions in core program

Social events
- each day: coffee breaks and sit-down lunches offer ample time for interaction and networking
- Monday evening: Get Together
- Tuesday evening: Welcome Reception
- Wednesday evening: Conference Banquet Dinner at the Botanic Garden of the Freie Universität Berlin, one of the largest and most diverse botanical gardens in the world, within walking distance to the hotel

Registration
- early registration discount up to Wednesday May 8, 2013
- additional discount for academia, Ada-Europe, ACM SIGAda, SIGBED and SIGPLAN members
- a limited number of student grants is available
- registration includes copy of printed proceedings at event
- includes coffee breaks and lunches
- three day conference registration includes all social events
- payment possible by bank transfer or credit card

Please make sure you book accommodation as soon as possible. A block of rooms at reduced prices has been reserved at the conference hotel until May 8. Note that those rooms are ONLY available for reservations made through the conference web site!

For more info and latest updates see the conference web site at <http://www.ada-europe.org/conference2013>.

ACM SIGAda Annual International Conference
High Integrity Language Technology
HILT 2013
Call for Technical Contributions

Developing and Certifying Critical Software

Pittsburgh, Pennsylvania, USA
November 10-14, 2013

Sponsored by ACM SIGAda
Contact: SIGAda.HILT2013 at acm.org www.sigada.org/conf/hilt2013

Confirmed Conference Keynote Speakers: Edmund M. Clarke (Carnegie Mellon University, 2007 ACM Turing Award), Jeannette Wing (Microsoft Research) and John Goodenough (Software Engineering Institute)

SUMMARY

High integrity software must not only meet correctness and performance criteria but also satisfy stringent safety and/or security demands, typically entailing certification against a relevant standard. A significant factor affecting whether and how such requirements are met is the chosen language technology and its supporting tools: not just the programming language(s) but also languages for expressing specifications, program properties, domain models, and other attributes of the software or overall system.

HILT 2013 will provide a forum for experts from academia/research, industry, and government to present the latest findings in designing, implementing, and using language technology for high integrity software. To this end we are soliciting technical papers, experience reports (including experience in teaching), and tutorial proposals on a broad range of relevant topics.

POSSIBLE TOPICS INCLUDE BUT ARE NOT LIMITED TO:

- New developments in formal methods
- Multicore and high integrity systems
- Object-Oriented Programming in high integrity systems
- High-integrity languages (e.g., SPARK)
- Use of high reliability profiles such as Ravenscar
- Use of language subsets (e.g., MISRA C, MISRA C++)
- Software safety standards (e.g., DO-178B and DO-178C)
- Typed/Proof-Carrying Intermediate Languages
- Contract-based programming (e.g., Ada 2012)
- Model-based development for critical systems
- Specification languages (e.g., Z)
- Annotation languages (e.g., JML)
- Teaching high integrity development
- Case studies of high integrity systems
- Real-time networking/quality of service guarantees
- Analysis, testing, and validation
- Static and dynamic analysis of code
- System Architecture and Design including Service-Oriented Architecture and Agile Development
- Information Assurance
- Security and the Common Criteria / Common Evaluation Methodology
- Architecture design languages (e.g., AADL)
- Fault tolerance and recovery

KINDS OF TECHNICAL CONTRIBUTIONS

TECHNICAL ARTICLES present significant results in research, practice, or education. Articles are typically 10-20 pages in length. These papers will be double-blind refereed and published in the Conference Proceedings and in ACM *Ada Letters*. The Proceedings will be entered into the widely consulted ACM Digital Library accessible online to university campuses, ACM's 100,000 members, and the software community.

EXTENDED ABSTRACTS discuss current work for which early submission of a full paper may be premature. If your abstract is accepted, a full paper is required and will appear in the proceedings. Extended abstracts will be double-blind refereed. In 5 pages or less, clearly state the work's contribution, its relationship with previous work by you and others (with bibliographic references), results to date, and future directions.

EXPERIENCE REPORTS present timely results and "lessons learned". Submit a 1-2 page description of the project and the key points of interest. Descriptions will be published in the final program or proceedings, but a paper will not be required.

PANEL SESSIONS gather groups of experts on particular topics. Panelists present their views and then exchange views with each other and the audience. Panel proposals should be 1-2 pages in length, identifying the topic, coordinator, and potential panelists.

INDUSTRIAL PRESENTATIONS Authors of industrial presentations are invited to submit a short overview (at least 1 page in size) of the proposed presentation and, if selected, a subsequent abstract for a 30-minute talk. The authors of accepted presentations will be invited to submit corresponding articles for ACM *Ada Letters*.

WORKSHOPS are focused sessions that allow knowledgeable professionals to explore issues, exchange views, and perhaps produce a report on a particular subject. Workshop proposals, up to 5 pages in length, will be selected based on their applicability to the conference and potential for attracting participants.

TUTORIALS can address a broad spectrum of topics relevant to the conference theme. Submissions will be evaluated based on applicability, suitability for presentation in tutorial format, and presenter's expertise. Tutorial proposals should include the expected level of experience of participants, an abstract or outline, the qualifications of the instructor(s), and the length of the tutorial (half day or full day).

HOW TO SUBMIT**: Except for Tutorial proposals use www.easychair.org/conferences/?conf=hilt2013**

Submission	***Deadline***	***Use Easy Chair Link Above***
Technical articles, extended abstracts, experience reports, panel session proposals, or workshop proposals	**June 29, 2013**	For more info contact: **Tucker Taft,** Program Chair taft@adacore.com
Industrial presentation proposals	**August 1, 2013** (overview) **September 30, 2013** (abstract)	
Send **Tutorial** proposals to	**June 29, 2013**	**John McCormick**, Tutorials Chair mccormick@cs.uni.edu

At least one author is required to register and make a presentation at the conference.

FURTHER INFORMATION

CONFERENCE GRANTS FOR EDUCATORS**:** The ACM SIGAda Conference Grants program is designed to help educators introduce, strengthen, and expand the use of Ada and related technologies in school, college, and university curricula. The Conference welcomes a grant application from anyone whose goals meet this description. The benefits include full conference registration with proceedings and registration costs for 2 days of conference tutorials/workshops. Partial travel funding is also available from AdaCore to faculty and students from GNAT Academic Program member institutions, which can be combined with conference grants. For more details visit the conference web site or contact **Prof. Michael B. Feldman** (MFeldman@gwu.edu)

OUTSTANDING STUDENT PAPER AWARD**:** An award will be given to the student author(s) of the paper selected by the program committee as the outstanding student contribution to the conference.

SPONSORS AND EXHIBITORS**:** Please contact **Greg Gicca** (gicca@adacore.com) to learn the benefits of becoming a sponsor and/or exhibitor at HILT 2013.

IMPORTANT INFORMATION FOR NON-US SUBMITTERS**:** International registrants should be particularly aware and careful about visa requirements, and should plan travel well in advance. Visit the conference website for detailed information pertaining to visas.

ANY QUESTIONS?

Please send email to SIGAda.HILT2013@acm.org, or contact the Conference Chair (**Jeff Boleng**, jlboleng@SEI.CMU.EDU), SIGAda's Vice-Chair for Meetings and Conferences (**Alok Srivastava**, alok.srivastava@tasc.com), or SIGAda's Chair (**Ricky E. Sward**, rsward@mitre.org).