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

# Ada LETTERS

Volume XXXII Number 1 April 2012

## Table of Contents





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

Volume XXXII Number 1, April 2012

**Table of Contents**



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

**CHAIR**
Ricky E. Sward, The MITRE Corporation, 1155 Academy Park Loop Colorado Springs, CO 80910 USA
Phone: +1 (719) 572-8263, RSward@Mitre.org

**VICE-CHAIR FOR MEETINGS AND CONFERENCES**
Alok Srivastava, TASC Inc., 475 School Street SW, Washington, DC 20024-2711 USA
Phone: +1 (202) 314-1419, Alok.Srivastava@TASC.Com

**VICE-CHAIR FOR LIAISON**
Greg Gicca, AdaCore, 1849 Briland Street, Tarpon Springs, FL 34689, USA
Phone: +1 (646) 375-0734, Gicca@AdaCore.Com

**SECRETARY**
Clyde Roby, Institute for Defense Analyses, 4850 Mark Center Drive, Alexandria, VA 22311 USA
Phone: +1 (703) 845-6666, Roby@ida.org

**TREASURER**
Geoff Smith, Lightfleet Corporation, PO Box 6256, Aloha, OR 97007, USA
Phone: +1 (360) 816-2821, GSmith@Lightfleet.Com

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

**PAST CHAIR**
John W. McCormick, Computer Science Department, University of Northern Iowa, Cedar Falls, IA 50614, USA
Phone: +1 (319) 273-6056, McCormick@cs.uni.edu

**ACM PROGRAM COORDINATOR SUPPORTING SIGAda**
Irene Frawley, 2 Penn Plaza, Suite 701, New York, NY 10121-0701
Phone: +1 (212) 626-0605, Frawley@ACM.Org



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

**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 written 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.

# From the Editor's Desk

*Alok Srivastava*

Welcome to this issue of ACM Ada Letters. In this issue you will find two very interesting papers, Updated MPHF Weights for Ada 2012 by John A. Trono and TLM Request Response Channel in SystemAda by Negin Mahani. A MPHF was created by John Trono for the 72 reserved words in Ada 2005, however, with the addition of a 73rd reserved word ("some"), the table size must be incremented, and new weights need to be determined. Author Negin Mahani has reiterated that Ada because of its intrinsic concurrency and object orientation is a good candidate to model hardware at transaction level modeling or TLM. In this paper she has implemented Request Response channel (TLM_Req_Res) as another basic channel of TLM based on their previously delimited TLM_FIFO channel.

The issue also provides details on AdaCore compiled **Ada Gems:**

- GPS - Smart Completion (Part 1 of 2) by *Quentin Ochem*
- Code Archetypes for Real-Time Programming - Part 1 by *Marco Panunzio*
- The Distributed Systems Annex, Part 4 - DSA and C by *Thomas Quinot*
- Smart Completion (Part 2 of 2) by *Quentin Ochem*
- Code Archetypes for Real-Time Programming - Part 2 by *Marco Panunzio*
- High Performance Multi-core Programming - Part 1 by *Pat Rogers*
- Code Archetypes for Real-Time Programming - Part 3 by *Marco Panunzio*
- Dynamic Stack Analysis in GNAT by Quentin Ochem

In this issue you will find the advance program of High-Integrity Language Technology SIGAda 2012 conference to be held from December 2-6, 2012 in Boston USA. Another major Ada event, the 18th International Conference on Reliable Software Technologies - Ada-Europe 2013 will take place in 2013 in Berlin, Germany, from June 10 to 14, 2013.

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 Ada Letters, I'd like to thank you for your continued support of ACM SIGAda, and encourage you to submit articles for publication. In addition, if there is some way we can make 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 2005 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 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 from quarterly 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, 12 | August, 2012 | October 1st, 12 | December, 2012 |
| February 1st, 13 | April, 2013 | June 1st, 13 | August, 2013 |

**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

# Updated MPHF Weights for Ada 2012

John A. Trono
Saint Michael's College
One Winooski Park
Colchester, VT 05439
jtrono@smcvt.edu

## Introduction

A minimal perfect hashing function (MPHF) allows software to directly verify set membership, in one array access, without any concern for collisions during this table lookup operation. A MPHF was created by this author for the 72 reserved words in Ada 2005 [3]; however, with the addition of a $73^{rd}$ reserved word ("some"), the table size must be incremented, and new weights need to be determined. This brief essay revisits the MPHF creation process, and outlines how those techniques [3] were reused as much as possible to shorten the time to derive a new MPFH.

## Updating Process

One basic structure for a traditional MPHF was published by Cichelli [1], but as explained in [3], this had to be modified to what appears just below that assignment – for several reasons.

h(word) = weights[1st letter in word] + weights[last letter in word] + length(word); //traditional

h(word) = (weights1[word[107 mod length(word)]] + weights2[word[108 mod length(word)]] [1]+
length(word)) mod 72;

First, this is a relatively large number of reserved words, but even so, it is possible to accommodate similar sets with the traditional equation. Unfortunately, two five letter reserved words in Ada ("raise" and "range") would always generate collisions in the traditional approach. Therefore, the introduction of two, distinct arrays, i.e. weights1[ ] and weights2[ ], was deemed necessary [3] to prevent such collisions, and given this updated strategy, two indices into the reserved words still needed to be discovered before appropriate weights could be determined. Secondly, with "accept" and "access" both beginning with the same first four letters, a lengthy process was undertaken to select two indices (107 and 108 – modulus the length of the reserved words) for the hybrid MPFH [3]. Lastly, a "mod tableSize" operation was added to ease the search for said weights without significantly decreasing the strategy's performance.

When determining these weights by hand [3], the most popular letters, in both selected positions within the reserved words, were assigned weights that attempted to distribute the reserved words (that have those popular letters in those selected positions) throughout the entire table. With weights1[ ], the letters {'t', 'n', 'd', 'e', 's'} were assigned the values {0, 14, 28, 42, 56}, and {3, 18, 32, 51, 68} were assigned to {'a', 't', 'i', 'r', 'e'} with respect to weights2[ ]. Unfortunately, these weights produced a few collisions, but by decrementing the value for weights2['t'], those

[1] `This notation assumes word is a character array that is accessed from [0] to [length-1], even though array indices in Ada begin with 1.`

were resolved. For the remaining, unplaced reserved words, active counts were deduced, and letters were assigned weights in decreasing order of collision likelihood until only the 16 letters that appear only once in the selected positions (referred to as wildcards) remained.

When assigning values to the wildcards, five reserved words were placed into the same table position as with the MPHF for Ada 2005, and overall, nineteen reserved words are linked with the same hash value as with the previous hash table [3]. In fact, 23 weights are identical, since that was one guideline followed when completing this weight assignment problem because it seemed reasonable that reusing the previous weights could make adding the new reserved word ("some") to the table much easier than starting from scratch. (As it turns out, it was easier by roughly a factor of five, and, twelve more weights were only +/- one, or two, for similar reasons.)

The original version of the Ada code below appears in [2], and it has been modified here to include the new weights. The original version was roughly three times faster than a similar binary search operation [3], and this version performs the searches just as quickly. Ironically, because 108 is evenly divisible by 2, 3, 4, 6, 9, and 12, reserved words of those lengths select the last and first letter – the opposite order of the Cichelli MPFH – in 47 of the 73 table entries.

```
Number_Of_Reserved_Words : constant := 73;

subtype Index_Range is Natural range 0 .. Number_Of_Reserved_Words - 1;
subtype Lowercase   is Character range 'a' .. 'z';
type    Table_Array is array (Lowercase) of Natural;

Table_1 : constant Table_Array :=
   (5, 34,  5, 28, 42, 16, 20, 1, 21,  0, 31, 32,  6,
   14, 13, 12,  3, 13, 56,  0, 0,  9, 34,  0,  0,  0);

Table_2 : constant Table_Array :=
   ( 3, 0, 27, 17, 68,  7, 27, 0, 32,  0, 43, 17, 5,
    39, 4,  2,  0, 51, 21, 17, 4, 46, 10,  9,  0, 0);

function Hash (Name: in String) return Index_Range is
-- Preconditions : Name contains only lowercase characters
--                 Name contains at least one character
begin
   return (Table_1 (Name(107 rem Name'Length + 1)) +
           Table_2 (Name(108 rem Name'Length + 1)) +
           Name'Length) rem Number_Of_Reserved_Words;
end Hash;
```

## Conclusion

A previously constructed minimal perfect hashing function (MPHF) was updated in a fairly straightforward manner, and that process was summarized here. Since the set of reserved words in Ada 2012 is only one larger than its predecessor, more than half of the reserved words remained in the same table positions as before, and only about 25% of them were farther than four places from their previous locations. Appendices A and B list the updated weights as well as the new table determined by this updated MPHF.

## References

[1] Cichelli, Minimal Perfect Hashing Functions Made Simple, *Communications of the ACM*,

Volume 23, #1, 1980.
[2] Dale and McCormick, *Ada 2005 Plus Data Structures: An Object-Oriented Approach*, Jones and Bartlett, 2nd edition, 2007.
[3] Trono, Optimal Table Lookup for Reserved Words in Ada, *Ada Letters*, Volume 26, #1, 2006.

Appendix A – weights assigned to letters.

| char | weights1[char] | weights2[char] |
|---|---|---|
| a | 5 | 3 |
| b | 34 | 0 |
| c | 5 | 27 |
| d | 28 | 17 |
| e | 42 | 68 |
| f | 16 | 7 |
| g | 20 | 27 |
| h | 1 | 0 |
| i | 21 | 32 |
| j | 0 | 0 |
| k | 31 | 43 |
| l | 32 | 17 |
| m | 6 | 5 |
| n | 14 | 39 |
| o | 13 | 4 |
| p | 12 | 2 |
| q | 3 | 0 |
| r | 13 | 51 |
| s | 56 | 21 |
| t | 0 | 17 |
| u | 0 | 4 |
| v | 9 | 46 |
| w | 34 | 10 |
| x | 0 | 9 |
| y | 0 | 0 |
| z | 0 | 0 |

Appendix B – Hash table placement for Ada 2012 reserved words.

The third column holds the index and letter selected by (107 mod length of the keyword), where L denotes the last letter in the keyword and zero the first: k[2nd] denotes the index and second letter selected; w1[1st] is the value for the letter at k[1st]; and w2[2nd] for k[2nd].

| [ ] | keyword | k[1st] | k[2nd] | w1[1st] | w2[2nd] | length |
|---|---|---|---|---|---|---|
| 0 | case | L-e | 0-c | 42 | 27 | 4 |
| 1 | private | 2-i | 3-v | 21 | 46 | 7 |
| 2 | null | L-l | 0-n | 32 | 39 | 4 |
| 3 | new | L-w | 0-n | 34 | 39 | 3 |
| 4 | body | L-y | 0-b | 0 | 0 | 4 |
| 5 | at | L-t | 0-a | 0 | 3 | 2 |
| 6 | digits | L-s | 0-d | 56 | 17 | 6 |
| 7 | out | L-t | 0-o | 0 | 4 | 3 |
| 8 | constant | 3-s | 4-t | 56 | 17 | 8 |
| 9 | accept | L-t | 0-a | 0 | 3 | 6 |
| 10 | interface | L-e | 0-i | 42 | 32 | 9 |
| 11 | reverse | 2-v | 3-e | 9 | 68 | 7 |
| 12 | record | L-d | 0-r | 28 | 51 | 6 |
| 13 | pragma | L-a | 0-p | 5 | 2 | 6 |
| 14 | requeue | 2-q | 3-u | 3 | 4 | 7 |
| 15 | with | L-h | 0-w | 1 | 10 | 4 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 16 | generic | 2-n | 3-e | 14 | 68 | 7 |
| 17 | is | L-s | 0-i | 56 | 32 | 2 |
| 18 | exception | L-n | 0-e | 14 | 68 | 9 |
| 19 | or | L-r | 0-o | 13 | 4 | 2 |
| 20 | elsif | 2-s | 3-i | 56 | 32 | 5 |
| 21 | array | 2-r | 3-a | 13 | 3 | 5 |
| 22 | of | L-f | 0-o | 16 | 4 | 2 |
| 23 | for | L-r | 0-f | 13 | 7 | 3 |
| 24 | renames | 2-n | 3-a | 14 | 3 | 7 |
| 25 | xor | L-r | 0-x | 13 | 9 | 3 |
| 26 | end | L-d | 0-e | 28 | 68 | 3 |
| 27 | select | L-t | 0-s | 0 | 21 | 6 |
| 28 | when | L-n | 0-w | 14 | 10 | 4 |
| 29 | declare | 2-c | 3-l | 5 | 17 | 7 |
| 30 | function | 3-c | 4-t | 5 | 17 | 8 |
| 31 | aliased | 2-i | 3-a | 21 | 3 | 7 |
| 32 | do | L-o | 0-d | 13 | 17 | 2 |
| 33 | loop | L-p | 0-l | 12 | 17 | 4 |
| 34 | and | L-d | 0-a | 28 | 3 | 3 |
| 35 | then | L-n | 0-t | 14 | 17 | 4 |
| 36 | mod | L-d | 0-m | 28 | 5 | 3 |
| 37 | until | 2-t | 3-i | 0 | 32 | 5 |
| 38 | all | L-l | 0-a | 32 | 3 | 3 |
| 39 | protected | L-d | 0-p | 28 | 2 | 9 |
| 40 | delay | 2-l | 3-a | 32 | 3 | 5 |
| 41 | else | L-e | 0-e | 42 | 68 | 4 |
| 42 | not | L-t | 0-n | 0 | 39 | 3 |
| 43 | while | 2-i | 3-l | 21 | 17 | 5 |
| 44 | goto | L-o | 0-g | 13 | 27 | 4 |
| 45 | limited | 2-m | 3-i | 6 | 32 | 7 |
| 46 | range | 2-n | 3-g | 14 | 27 | 5 |
| 47 | raise | 2-i | 3-s | 21 | 21 | 5 |
| 48 | in | L-n | 0-i | 14 | 32 | 2 |
| 49 | use | L-e | 0-u | 42 | 4 | 3 |
| 50 | if | L-f | 0-i | 16 | 32 | 2 |
| 51 | tagged | L-d | 0-t | 28 | 17 | 6 |
| 52 | task | L-k | 0-t | 31 | 17 | 4 |
| 53 | procedure | L-e | 0-p | 42 | 2 | 9 |
| 54 | delta | 2-l | 3-t | 32 | 17 | 5 |
| 55 | package | 2-c | 3-k | 5 | 43 | 7 |
| 56 | entry | 2-t | 3-r | 0 | 51 | 5 |
| 57 | begin | 2-g | 3-i | 20 | 32 | 5 |
| 58 | subtype | 2-b | 3-t | 34 | 17 | 7 |
| 59 | abstract | 3-t | 4-r | 0 | 51 | 8 |
| 60 | rem | L-m | 0-r | 6 | 51 | 3 |
| 61 | synchronized | L-d | 0-s | 28 | 21 | 12 |
| 62 | abs | L-s | 0-a | 56 | 3 | 3 |
| 63 | type | L-e | 0-t | 42 | 17 | 4 |
| 64 | separate | 3-a | 4-r | 5 | 51 | 8 |
| 65 | access | L-s | 0-a | 56 | 3 | 6 |
| 66 | others | L-s | 0-o | 56 | 4 | 6 |
| 67 | some | L-e | 0-s | 42 | 21 | 4 |
| 68 | terminate | L-e | 0-t | 42 | 17 | 9 |
| 69 | abort | 2-o | 3-r | 13 | 51 | 5 |
| 70 | overriding | 7-i | 8-n | 21 | 39 | 10 |
| 71 | return | L-n | 0-r | 14 | 51 | 6 |
| 72 | exit | L-t | 0-e | 0 | 68 | 4 |

# TLM Request Response Channel in SystemAda

Negin Mahani
*Zarand High Education Center, Computer Engineering Department, Shahid Bahonar University, Kerman, Iran*
*Negin @cad.ece.ut.ac.ir*

**Abstract**

Hardware description languages or HDLs have started their way from transistor level to transaction level modeling up to now. Ada because of its intrinsic concurrency and object orientation is a good candidate to model hardware at transaction level modeling or TLM. In our previous papers we have implemented some special and necessary features of gate level and also some fundamentals of TLM in Ada language[1][2][3]. In this paper we have implemented Request Response channel (TLM_Req_Res) as another basic channel of TLM based on our TLM_FIFO channel in our last work. Also we have done some simulation time comparisons to show that there is no significant simulation time penalty in SystemAda over SystemC like our previous implementations.

## 1 Introduction

Ada is an important programming language. It is one of the most popular languages in the whole world that many academic and nonacademic centers use it because of its unique features .The Ada programming language has got its strong structures from several languages. Its detailed evaluation process is unique among other programming languages.

Its special language features are composed of packages, exception handling, generic program units, parallel programming and object orientation. Ada also supports more flexible libraries, better control mechanisms for shared data and interfaces.

This programming language is used in fields like government (Department of Defense), banking systems, commercial aviation, communication systems, computer-aided design and Manufacturing [4] [5].

Ada is very similar to VHDL in syntax. So it is interesting for hardware designers. Besides as mentioned before it has concurrency, object orientation and interface structures which are not patchy. These are the exact features that are necessary for a TLM language.

As you may know the design of hardware has evolved from Transistor Level to Register Transfer Level (RTL), and now to Transaction Level. The inherent concurrency of Ada makes it a good candidate for describing register transfer level. Furthermore, its object oriented features along with inherent mechanisms for concurrency give it potentials for being used as a language for describing processing elements and communication channels of transaction level.

In this paper we follow our work on SystemAda and try to implement other TLM channels in this form of Ada language. This paper is organized as follows. Section 2 will be focus on the characteristic of Transaction Level Modeling (TLM) and a synopsis of TLM basic channels, while notable features of our SystemAda and our new channel implementation (Request Response channel) will be introduced in section 3. Next in section 4, simulation time in SystemAda and SystemC languages will be compared. Finally, conclusions are drawn in section 5.

## 2 TLM

Transaction level modeling is the last level of modeling for describing complex hardware systems today. In this level there are two basic components computational and communicational components. It means that TLM divides a system into computation parts, i.e. processing elements, and communication parts, i.e. channels. In this level of hardware modeling we divide our hardware systems into these two component types.

Transaction Level Modeling (TLM) enhances simulation performance of today's complex digital systems and also provides the ability of early design space exploration.

In the first standard of TLM there are some basic channels that play the role of communicational components in this level and try to exchange data between computational components.

Using these components, the level of abstraction is increased and the details of hardware system description like internal signals and gates are running away[7][8].

### 2.1 TLM Channels

There are three kind of basic channels in TLM: TLM_FIFO channel, TLM Request Response channel and TLM Transport channel. TLM_FIFO channel is the first channel of TLM that two other channels described based on it.

In the following there are brief descriptions of the TLM channels, like TLM_FIFO channel that we have implemented in our previous works, TLM Request Response channel that we want to implement here and also TLM Transport channel that we want to work on, in near future. Our focus in this paper is on TLM Request Response channel.

TLM _FIFO is a FIFO channel which is generic in size and type. It has some special characteristics which has mentioned in our previous work and also in section 3.1.1 briefly.

TLM Request Response channel is another channel in TLM. It composed of two TLM_FIFO channels. One is used for requests and the other is used for responses. And TLM Transport channel is a TLM Request Response channel with size one. Actually it is composed of two buffers with size one, one is for input data packets and the other one is for output data packets[9].

# 3 SystemAda

In previous researches, we developed packages in Ada to use this language as a system description language, like the way SystemC is used for description of systems. We refer to our form of Ada usage and its additional packages as SystemAda and we use a public Ada compiler (GNAT) to evaluate system descriptions written in Ada. SystemAda is meant for modeling system behavior and structure at the transaction level and we consider possible approaches for extending Ada to meet these requirements. Previous papers discussed the specification of our proposed SystemAda, its hardware description style, its RTL link, Ada description of Transaction Level Modeling (TLM) channels, presentation of TLM interfaces concept, and implementation of more complex models like a network on chip system in Ada[2][3].

A hardware description language at any level in addition to providing constructs for covering hardware at that level, it has to provide a minimum set of constructs for describing hardware at its immediate next lower level of abstraction. Therefore, for creating a transaction level hardware language from Ada, we have to cover some preliminary RTL level constructs as well. So our last works focus on TLM, while providing a sufficient link to RTL[1][2].

In [3] we have made simulation time comparisons between TLM_FIFO channel implemented in SystemAda-TLM and SystemC-TLM. As a result of these experiments we have shown that Ada TLM_FIFO channel is faster in simulation than one written in SystemC.

## 3.1 Describing TLM Channels using Ada

Since all of TLM channel structures can be described based on FIFO, we have started developing TLM channels from TLM_FIFO channel. All of these descriptions are based on functionality of the channel.

### 3.1.1 TLM_FIFO Channel in SystemAda

TLM_FIFO has been implemented completely and improved its characteristics in [3]. As it is mentioned there it has some special characteristics that all are covered in the Ada implementation version. These characteristics are as follows:

- Generic package
- Generic type
- Generic size
- Concurrency
- Blocking/non-blocking transport capability
- Export
- Multiple reader/writer
- Multiple instances

TLM_FIFO is implemented by dynamic and static memory allocation. The code of TLM_FIFO channel using dynamic memory allocation is depicted in the following figure (Figure 1). Each of its features and its static memory allocation form are explained completely in the previous paper[3].

```
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 1. FIFO package specification using dynamic memory allocation[3]**

### 3.1.2 TLM_Req_Res Channel in SystemAda

As mentioned earlier, Request Response channel is one of the transaction level channels which conceptually consist of two TLM_FIFOs, one for requests and the other one for responses.

```
generic package Req_Res is
  task type Req_Res_Task is
    entry Add_Req;
    entry Add_Res;
    entry Rem_Req;
    entry Rem_Res;
    entry Stop;
  end Req_Res_Task;

  …

  TLM_Req_Res : Req_Res_Task;
end Req_Res;
```

**Figure 2. Req_Res channel package specification**

```
package body Req_Res is
  task body Req_Res_Task is
  begin
    loop
      select
        accept Add_Req do
          Req_FIFO.Input := Req_Input;
          Req_FIFO.TLM_FIFO.Add;
        end Add_Req;
        or
        accept Rem_Req do
          IF  (Req_FIFO.Empty_Flag  =  FALSE)
then
            Req_FIFO.TLM_FIFO.Remove;
            Req_Output := Req_FIFO.Output;
          end IF;
        end Rem_Req;
        or
        accept Add_Res do
          Res_FIFO.Input := Res_Input;
          Res_FIFO.TLM_FIFO.Add;
        end Add_Res;
        or
        accept Rem_Res do
          if  (Res_FIFO.Empty_Flag  =  FALSE)
then
            Res_FIFO.TLM_FIFO.Remove;
            Res_Output := Res_FIFO.Output;
          end if;
        end Rem_Res;
        or
        accept Stop;
          Req_FIFO.TLM_FIFO.Stop;
          Res_FIFO.TLM_FIFO.Stop;
          exit;
        end select;
    end loop;
  end Req_Res_Task;
end Req_Res;
```

**Figure 3. Req_Res channel package body**

This channel supports both modes for transferring data, blocking and non-blocking. In blocking mode, each request is tied to a response and thus the size of each FIFO is limited to one. Such channel is called Transport channel which we will work on its implementation in near future. In non-blocking mode, however, the request could be gathered in the request FIFO without paying attention to getting responses[9].

TLM_Req_Res channel could also be implemented dynamically using linked list and statically using two cyclic unconstrained arrays (circular buffers). Figure 2 shows the specification of the package which implements Req_Res channel. To have concurrency in this channel, a task type called Req_Res_Task has been added to this package. A variable called TLM-Req_Res has been defined from this task type to be used later as a communication channel of this type. This task type has five entries that their name shows their functionality.

The body of the *Req_Res* package is shown in Figure 3. As shown in this figure, selective waiting has been used in order to model the non-blocking operation of the channel and *or* clause appears between different *accept* statements. Req_FIFO and Res_FIFO are FIFOs of type TLM_FIFO and are used for storing requests and responses respectively.

Figure 4 shows a master-slave architecture using our Req_Res channel. In this figure, the messages that Master and Slave modules pass to TLM_Req_Res channel for communication are shown.

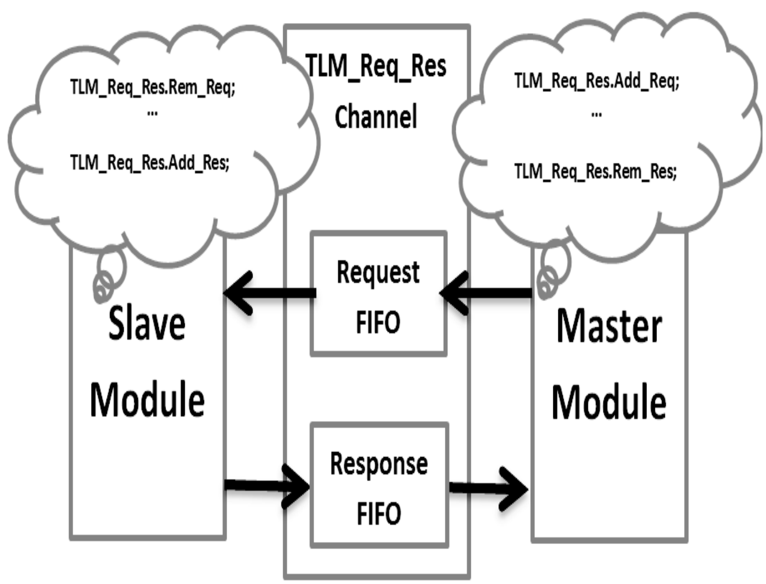



**Figure 4. TLM-Master-Slave architecture using TLM_Req_Res channel**

## 4 Simulation Result

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_Req_Res channel.

In order to compare the simulation time of the TLM channel implemented using SystemC and Ada, we need an appropriate platform. We have done several experiments on two platforms. The properties of the first platform which we have used are as follows:

- **Operating system**: Microsoft Windows XP Professional, 2002 version, Service pack2
- **System**: Intel Pentium 4, CPU 2GHz, RAM 1GB
- **Compiler:** Gnat GPL 2007 for Ada, Microsoft Visual Studio (.Net Frame Work 2005) For SystemC

### 4.1 Ada and SystemC simulation time comparison for TLM_Req_Res channel in platform one

For the first set of experiments we have chosen plat form one. We have done these experiments for different numbers of packets from 10000 to 50000 packets and we have recorded the simulation time lengths in **Table 1**. In this series of experiments we have used I/O file commands in the programs (i.e. we read from and write to files. **Figure 5** is based on the data in **Table 1**.

**Table 1: TLM_Req_Res channel average simulation time in SystemAda and SystemC for variable packet numbers (using I/O files)**

| Packet Number | Ada | System C | Simulation Time Ratio | Optimization percent |
|---|---|---|---|---|
| 10000 | 0.9458 | 2.7014 | 2.856206 | 64.98852 |
| 20000 | 1.967 | 5.1206 | 2.603254 | 61.58653 |
| 30000 | 2.882 | 7.95 | 2.758501 | 63.74843 |
| 40000 | 3.7576 | 10.5926 | 2.81898 | 64.52618 |
| 50000 | 4.732 | 13.075 | 2.763102 | 63.8088 |
| Average | | | 2.760009 | 63.73169 |

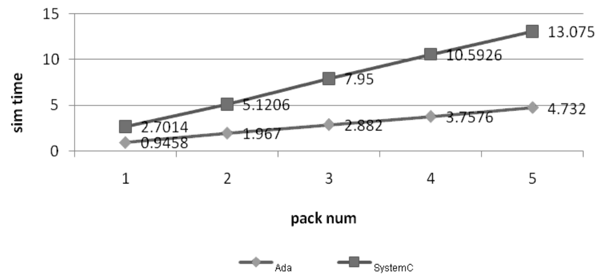



**Figure 5. Ada and SystemC TLM_Req_Res channel simulation time (using I/O files)**

Since the difference between SystemAda and SystemC models is much more than expected amount, it is concluded that it must be some special command that is costly in SystemC over SystemAda compilers. Based on previous experiments on TLM_FIFO these are I/O file commands. So we omit these commands and repeat each of the experiments as before. Table 2 and Figure 6 show the results of the experiments which have done without I/O file commands.

**Table 2: TLM_Req_Res channel average simulation time in SystemAda and SystemC for variable packet numbers (using no I/O files)**

| Packet Number | Ada | System C | Simulation Time Ratio | Optimization percent |
|---|---|---|---|---|
| 10000 | 0.8348 | 0.872 | 1.044562 | 4.266055 |
| 20000 | 1.6972 | 1.8062 | 1.064223 | 6.034769 |
| 30000 | 2.5244 | 2.5274 | 1.001188 | 0.118699 |
| 40000 | 3.3376 | 3.8466 | 1.152505 | 13.23247 |
| 50000 | 4.2272 | 4.5034 | 1.065339 | 6.133144 |
| Average | | | 1.065563 | 5.957026 |

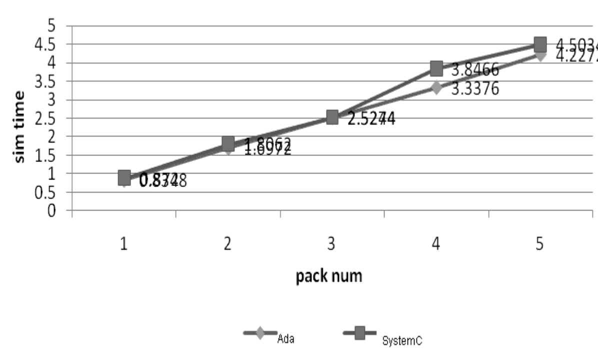



**Figure 6. Ada and SystemC TLM_Req_Res channel simulation time (using no I/O files)**

All of the above experiments are done using the source code with dynamic memory allocation. This time we repeat the experiments based on static memory allocation along with no I/O files.

As it is obvious static memory allocation is faster than dynamic memory allocation and the following results in Figure 7 based on Table 3 support this fact.

**Table 3: TLM_Req_Res channel average simulation time in SystemAda and SystemC for variable packet numbers (using static memory allocation)**

| Packet Number | Ada | System C | Simulation Time Ratio | Optimization percent |
|---|---|---|---|---|
| 10000 | 0.4704 | 0.872 | 1.853741 | 46.05505 |
| 20000 | 0.9144 | 1.8062 | 1.975284 | 49.37438 |
| 30000 | 1.3774 | 2.5274 | 1.834906 | 45.50131 |
| 40000 | 1.8098 | 3.8466 | 2.125428 | 52.95066 |
| 50000 | 2.2892 | 4.5034 | 1.967237 | 49.1673 |
| Average | | | 1.95132 | 48.60974 |

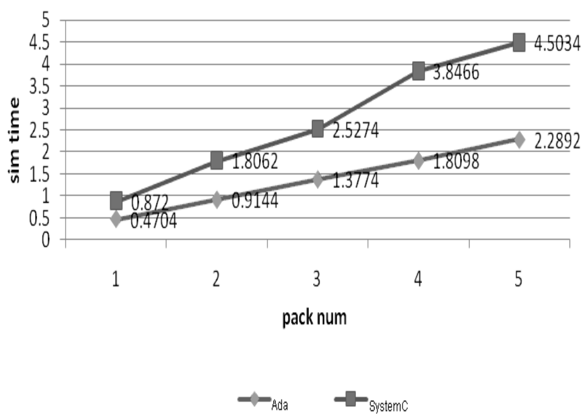



**Figure 7. Ada and SystemC TLM_Req_Res channel simulation time (using static memory allocation)**

## 4.2 Ada and SystemC simulation time comparison for TLM_Req_Res channel in platform two

The characteristics of the second plat form are as follow:

- **Operating system**: Microsoft Windows XP Professional, 2002 version, Service pack2
- **System**: Intel Pentium 4, CPU 2GHz, RAM 1GB
- **Compiler:** Gnat GPL 2007 for Ada, Microsoft VC++6.0 For SystemC

The obvious difference between the two introduced platforms is the compilers which are used for compiling SystemC source codes. We have used Microsoft Visual Studio (.Net Frame Work 2005) in the first platform and Microsoft VC++6.0 in the second platform For SystemC source codes.

The details of the simulations performed are presented in the following sections by using tables and charts.

For this set of experiments we have chosen plat form two. Again we have done these experiments for different numbers of packets from 10000 to 50000 packets and we have recorded the simulation time length in Table 4. Figure 8 is based on the data in Table 4.

**Table 4: TLM_Req_Res channel average simulation time in SystemAda and SystemC for variable packet numbers (plat form two)**

| Packet Number | Ada | SystemC | Simulation Time Ratio | Optimization percent |
|---|---|---|---|---|
| 10000 | 0.398 | 0.39 | 0.979899 | -2.05128 |
| 20000 | 0.748 | 0.797 | 1.065508 | 6.148055 |
| 30000 | 1.192 | 1.188 | 0.996644 | -0.3367 |
| 40000 | 1.479 | 1.469 | 0.993239 | -0.68074 |
| 50000 | 1.789 | 1.771 | 0.989939 | -1.01637 |
| Average | | | 1.005046 | 0.412593 |

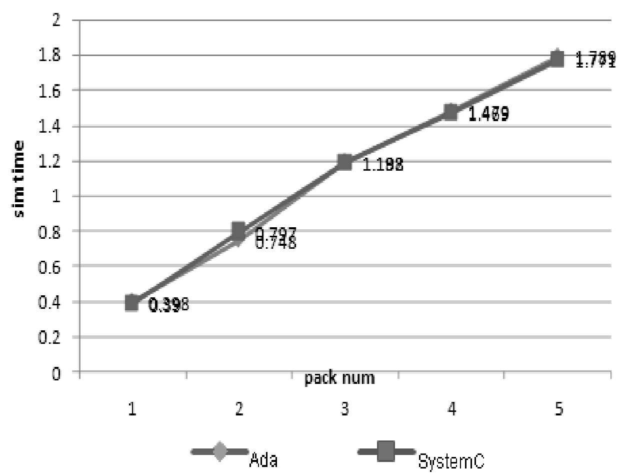



**Figure 8. Ada and SystemC TLM_Req_Res channel simulation time (plat form two)**

As it is shown in Table 4 and Figure 8 respectively, this time by the change in SystemC compiler from Microsoft Visual Studio (.Net Frame Work 2005) in the first platform and Microsoft VC++6.0 in the second platform there is only 0.4 percent simulation time optimization.

## 5 Conclusion

In our pervious works we have developed an RTL package in Ada and we have covered a link to RTL since every language that wants to cover TLM must have a link to RTL as well.

Also for proving TLM in Ada we have developed some basic structures of TLM in Ada like TLM_FIFO channel and some communicational interfaces which are unidirectional or bidirectional. Using this implemented TLM_FIFO channel we have developed a network on chip architecture to show it's applicable at Ada to model some more complex system.

In this work we have developed TLM Request Response channel in Ada based on our TLM_FIFO and we have done some simulation time comparison between SystemAda and SystemC equivalent models of TLM_Req_Res channel.

We have done these experiments in different circumstances like with using I/O files in source codes or not, using dynamic memory allocation or static memory allocation and also by a change in SystemC compilers. In some cases Ada models were much faster and somewhere else there was not mush difference. All in all in the worst case there was no significant simulation time penalty in SystemAda over SystemC.

In our future work we will implement TLM Transport channel in our SystemAda and do some more experiments on simulation time comparison between SystemAda and SystemC.

## 6 References


[1] 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.

[2] Negin Mahani, "Making Alive Register Transfer Level and Transaction Level Modeling In System-Ada", ACM SIGADA AdaLetters, Acm SIGAda Ada Letters, 2010, pp. 15-23.

[3] Negin Mahani, "Investigating SystemAda: TLM_FIFO Detailed Characteristics Proof, TLM2.0 Interfaces Implementation, Simulation Time Comparison to SystemC", ACM SIGADA AdaLetters, 2012.

[4] Sebesta, Robert T. (1996). "Concepts of Programming Languages", Addison-Wesley Publishing Company, Inc. Menlo Park, Ca.

[5] "Ada Programming Language", Available at : http://groups.engin.umd.umich.edu/CIS/course.des/cis400/ada/ada.html

[6] J. E. Sammet, "Why Ada is not Just another Programming Language", Communications of the ACM, vol. 29, no. 8, August 1986, pp. 722-732.

[7] "SystemC TLM1.0 Standard", Available at: http://www.systemc.org/home

[8] Stuart Swan, "A Tutorial Introduction to the SystemC TLM Standard", Cadence Design Systems, Inc, March 2006, Available at: http://www.ti.unituebingen.de/uploads/media/Presentation-13-OSCI_2_swan.pdf

[9] "TLM Modeling Techniques", Available at: http://www.ict.kth.se/courses/IL2452/Sept2009/TLM_modeling_techniques.pdf

## Gem #88 GPS - Smart Completion (Part 1 of 2)

**Author:** Quentin Ochem

### Let's get started…

In this Gem we're going to create a few files using the completion mechanism. All the semantic information required to provide this computation is done on the fly. In order to follow along, simply open GPS on any project.

Note that the completion mechanism relies on the ad hoc parser launched at GPS startup. You can see the progression of the parser on the bottom right corner of the screen. Completion may not be accurate before the parsing is finished.

### Simple completion of components

Create a new file main.adb:

```
procedure Main is
begin
   null;
end Main;
```

First we'll declare a few types and objects. Create a new record type in the declarative part, and then a variable of that type, for example:

```
type Rec is record
   A, B : Integer;
end record;

A_Variable : Rec;
```

Then, in the sequence of statements, type "A" and hit <control-space>. <control-space> is the shortcut for manually querying the completion. A popup is opened, showing all declarations and keywords starting with "A". At this stage, there are just too many of them, so let's narrow down the list by adding the character "_". The list is now much shorter. You can select the appropriate completion. A_Variable will be entered in the text.

<control-space> can be used any time to query a completion. When writing code, the completion can be triggered in certain cases. Enter a dot ('.') character in the code. The smart completion popup is automatically triggered, and offers to complete with the components of Rec, namely A and B.

This information is completely synchronized with the current contents of the editor. Adding a component, for example C, in the completion view, and then querying the completion again will show the three components.

**Completion of with and use clauses**

We're now going to add some with and use clauses to the program. In particular, say we want to add a dependency on some standard unit that provides mathematical operations.

Write "with" at the top of the main subprogram. Query the completion by entering <control-space>. All packages that can be "withed" are now listed here. You can narrow down the list of possibilities by writing, for example, "Ad", and selecting Ada. Add a dot. The completion mechanism will list all children of the predefined Ada package. You can select "Numerics". The interesting thing is that we may not know at this stage what's available in Numerics. Adding another dot will list all the child packages of Numerics. Scroll down the list. "Elementary_Functions" sounds like what we need. Select it.

**Completion of subprogram profiles**

Create a new Float variable, such as "X : Float;". We're going to use that variable in a mathematical computation. Since we still don't really know what's in the "Elementary_Functions" package, we'll start writing a fully prefixed call. Enter "X := Ada.Numerics.Elementary_Functions.". You can now see all the functions listed in the completion popup. Select, for example, Arctan and enter a left parenthesis. The completion mechanism will offer to complete with several profiles. The red diamonds provide complete profile completion with named notation. Select the first one, and give a value to X and Y.

**Completion and OOP**

Consider a simple package named "Base":

```
package Base is

   type Root is tagged private;

   procedure Prim (V : Root; I1, I2 : Integer);

   type Child is new Root with private;

   procedure Prim2 (V : Child);

private

   type Root is tagged record
      A : Integer;
   end record;

   type Child is new Root with record
      B : Integer;
```

```
   end record;

end Base;
```

In Main, create two variable, V1 and V2 of type Root and Child:

```
   V1 : Root;
   V2 : Child;
```

When completing, say, "V1.", the completion mechanism offers prefixed primitives, such as "Prim". When completing "V2.", the implicitly inherited primitive "Prim" and the newly declared "Prim2" are offered.

Completion is also sensitive to visibility context. Let's create a body for Base:

```
package body Base is

   procedure Prim (V : Root; I1, I2 : Integer) is
   begin
      null;
   end Prim;

   procedure Prim2 (V : Child) is
   begin
      null;
   end Prim2;

end Base;
```

Try writing "V." in the body of Prim2. You will now have access to all the visible components of Child, including the fields which are hidden from Main because of the private part.

### Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

**Gem #89**

**Author:** Marco Panunzio, *University of Padua*

**Let's get started…**

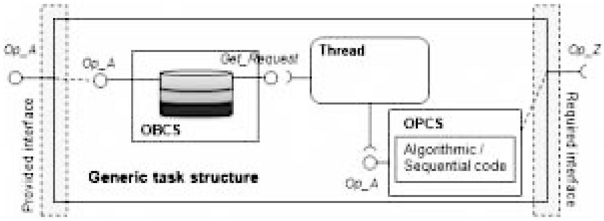
**Introduction** In this series of Ada Gems we propose a set of code archetypes for the development of real-time systems. The code archetypes comply with the restrictions of the Ravenscar Profile, a subset of the Ada language specifically suited for the development of high-integrity real-time systems. The Ravenscar Profile was devised to guarantee that programs written in accordance with it are amenable to static analysis in the time dimension. In fact, the profile excludes all Ada constructs that are exposed to nondeterministic or unbounded execution time. In the space dimension, the profile prohibits the use of constructs that implicitly perform dynamic memory allocation.

As an additional benefit, Ravenscar systems can be implemented on top of real-time kernels that can be very small in size and fast in time, which are attractive characteristics for applications that can afford little overhead and must undergo extensive qualification/certification.

The essential elements of the Ravenscar restrictions are: (i) *static existence model*. The system is composed of a fixed set of tasks and protected objects, defined at library level and with a statically assigned priority or ceiling priority. No task terminates, and abort statements are disallowed. (ii) *communication model*. Tasks are only allowed to communicate asynchronously, via protected objects. Task rendezvous is therefore disallowed. (iii) *deterministic execution model*. The profile excludes all constructs that introduce nondeterminism: relative delays, as they introduce nondeterminism in the suspension time of tasks; requeue statements; the use of the Ada.Calendar package, as all time-related operations rely on the high-precision Ada.Real_Time package; protected objects are restricted to having at most one entry on which only a single task can enqueue; guards of entries are composed of a simple Boolean condition to avoid side effects and nondeterminism in the evaluation time.

In this series of Ada Gems we illustrate a set of code archetypes that realize common programming patterns suited for the development of Ravenscar-compliant real-time systems. The use of the archetypes permits factorization and thus helps reduce the size of the code that implements the concurrent elements of the system.

An important additional goal of these archetypes is to achieve complete separation between the algorithmic/sequential code of the system and the code that manages concurrency and real-time aspects. This separation of concerns permits developing the algorithmic contents of the system (that is, the behavior of the system) independently of the management of tasking and real-time issues.




This goal is achieved by encapsulating the sequential code in a suitable task structure. The figure above depicts the generic structure of our task archetypes.

The sequential code is enclosed in a structure that we term the **Operational Control Structure** (OPCS). The code 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: as we shall see, we use it mainly for *sporadic* tasks. The OBCS consists of a protected object that stores incoming requests for services to be executed by the Thread. As multiple clients may independently require 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 (FIFO ordering is the default) and then executes the sequential code, stored in the OPCS, which corresponds with the request.

As we will illustrate in the third Gem of this series, the operations provided by the OBCS, which form the *provided interface* of the overall entity, match the signature of the sequential operations of the OPCS (Op_A in the figure above). Thanks to that, the callers need not be aware that they are in fact only posting execution requests in the OBCS, while the actual execution will be performed by the Thread.

The sequential code embedded in the OPCS may need to invoke services from other software entities (the operation Op_Z in the figure above). In the fifth Gem of this series we will describe how those functional needs can be fulfilled.

As a conclusion, our entities encapsulate their internal structure and expose to the external world just 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 are handled by the *OBCS*.

### 1.1.1 Structure of this Ada Gems Miniseries

1. Introduction and Cyclic Task
2. Simple Sporadic Task
3. Sporadic Task - System Types and Task Types
4. Sporadic Task - Sequential Code and OBCS
5. Intertask Communication

**Acknowledgments** The task structure we adopt is an evolution of the HRT-HOOD design methodology [1], from which we also inherit the terms OBCS and OPCS.

Early work on code generation from HRT-HOOD to Ada was described in [2] and [3].

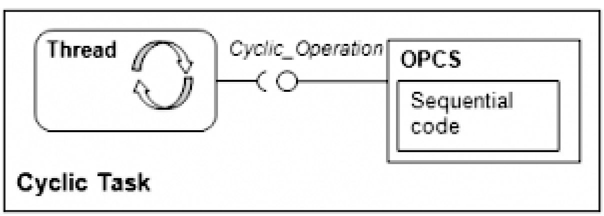
The code archetypes that we describe in this Ada Gems miniseries were used for the code generation in the HRT-UML track of the EU-funded ASSERT project [4]. In that project, Matteo Bordin, then at the University of Padua, was the main designer of the code generation strategy and code archetypes.

Finally we would like to thank Tullio Vardanega for his preliminary review of the contents of this miniseries, and Matteo Bordin, now with AdaCore, for his extensive review of the contents and his useful suggestions.

### 1.1.2 Let's get started…

### 1.1.3 Cyclic Task

In this section we illustrate our code archetype for cyclic tasks. It allows the developer to create a cyclic task by instantiating a generic package, passing the operation that needs to be executed periodically.




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 element of the archetype is:

```
with System; use System;

generic
   with procedure Cyclic_Operation;
package Cyclic_Task is

   task type Thread_T
      (Thread_Priority : Priority;
       Period : Positive) is
      pragma Priority (Thread_Priority);
   end Thread_T;

end Cyclic_Task;
```

The specification above 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 Ada body is specified as follows:

```
with Ada.Real_Time;
with System_Time;

package body Cyclic_Task is

   task body Thread_T is
      use Ada.Real_Time;
      Next_Time : Time := System_Time.System_Start_Time;
   begin
      loop
         delay until Next_Time;
         Cyclic_Operation;
         Next_Time := Next_Time + Milliseconds (Period);
      end loop;
   end Thread_T;

end Cyclic_Task;
```

The body of the task consists of an infinite loop. Just after activation, 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 their 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 in the instantiation of its generic package. Then it calculates the next time it needs to be released (*Next_Time*) and as first instruction of the subsequent loop, it issues a request for absolute suspension until the next multiple of its period.

The code archetype is simple to understand, yet a few comments are in order.

Firstly, we must stress that 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 from drifting.

Secondly, 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.

Finally, 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. In the next Ada Gem we will illustrate a simple code archetype to realize sporadic tasks.

**References**


[1] Alan Burns and Andy J. Wellings: "HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems". Elsevier Science, 1995. ISBN 978-0444821645.

[2] Juan Antonio de la Puente, Alejandro Alonso, and Angel Alvarez: "Mapping HRT-HOOD Designs to Ada 95 Hierarchical Libraries." Reliable Software Technologies - Ada-Europe, Springer Volume LNCS 1088, 1996.

[3] Matteo Bordin and Tullio Vardanega: "Automated Model-Based Generation of Ravenscar-Compliant Source Code". Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005. ISBN 0-7695-2400-1.

[4] ASSERT project (Automated proof-based System and Software Engineering for Real-Time Systems) http://www.assert-project.net


**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

**Gem #90: The Distributed Systems Annex, Part 4 - DSA and C**

**Author:** Thomas Quinot

**Let's get started…**

The previous DSA Gems showed how components in a pure Ada application can be spread across several partitions and use static or dynamic remote calls to interact. Wouldn't it be nice if other languages such as C could also benefit from these features?

Of course, you can embed C code in an Ada partition just as you would in any nondistributed application. Your C code can also call back to Ada code (as long as the Ada subprograms have the C convention). Remote (RCI) subprograms can thus be called from C. If the call occurs on the partition to which the RCI is assigned, nothing special happens, this is just a regular call. On other partitions, the compiler-generated calling stubs are used, and this is a transparent remote call, just as it would be if it occurred in Ada code: a remote subprogram has nothing special at the call point; all the magic is done in the generated stubs.

This is all well and good, but you still have to write your complete application in Ada, and in particular have the main subprogram of each partition declared in the GNATDIST configuration file.

What if you would like to incorporate DSA client or server code in an existing C application? This can be achieved by combining the DSA with GNAT's *stand-alone libraries*, a feature allowing an Ada partition to generate a loadable module rather than a full-fledged executable image. Here's how...

### 1.1.4 Rebuild PolyORB with -fPIC

The "-fPIC" switch instructs the compiler to generate so-called *Position Independent Code*, that is, code that can be dynamically loaded as a shared library.

In order to have a DSA partition in a stand-alone library, you need to set CFLAGS="-O2 -g -fPIC" in your environment when calling the PolyORB configure script. (The resulting PolyORB build can also be used for normal applications.)

### 1.1.5 Build your Ada partitions as usual, also with -fPIC

Let's assume for example that your application has a server partition that is fully written in Ada, and a client partition meant for embedding in a C/C++ application as a shared object. The server partition will be built using:

```
po_gnatdist -fPIC xxxx.cfg server_partition
```

### 1.1.6 Create a dummy main subprogram for the client side

You need to provide a dummy main subprogram for the client partition. You should make this a null library subprogram that has WITH clauses for any package (including RCIs) that you want to reference from the C side.

Also, it may be convenient to include in this closure an "Exports" package containing suitable subprogram declarations for those routines that you want to call from C, with C-compatible argument types, and using pragma Export to give them friendly C names. (Note that this is not specific to the Distributed Systems Annex: such an interface package is typically created any time you need to call Ada code from C code.)

```
with RCI_1;
...
with RCI_n;

with Exports;
procedure Client is
begin
   null;
end Client;
```

### 1.1.7 Build the client library

This is the crucial point. To build a partition as a stand-alone library instead of a regular executable, special arguments are passed to GNATDIST:

```
po_gnatdist -fPIC -g xxxx.cfg client_partition \
  -bargs rci_1.ali ... rci_n.ali polyorb-dsa_p-partitions.ali \
         -shared -LClientName \
  -largs -shared
```

In this command line, you need to list the ALI files for all RCI packages referenced in your client partition (rci_1.ali .. rci_n.ali), and also the one for the internal RCI polyorb-dsa_p-partitions.ali.

You can replace the name "ClientName" with an arbitrary prefix of your choosing (it is used for some automatically generated symbols, see below).

This will generate a file client_partition, which you can rename to client_partition.so.

### 1.1.8 Call client library from C code

Once you have your loadable object generated, you can load it from C code using the standard dlopen(3) function.

Symbols from the library can then be obtained using the dlsym(3) function. You first need to retrieve the symbols ClientNameinit and ClientNamefinal from the library.

ClientNameinit corresponds to the elaboration of all Ada units in the library, and should be called once upon module load. This starts the Ada PCS and connects to the DSA name server to retrieve the initial location of RCI units.

ClientNamefinal corresponds to the finalization, and should be called once, just before unloading the module or terminating the application (*ClientName* here is the prefix you passed on the GNATDIST command line above).

Finally, you can retrieve and call the symbols for RCI subprograms, or any subprogram exported by your Ada units, and call them as though they were normal C routines.

**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

## Gem #91: Smart Completion (Part 2 of 2)

**Author:** Quentin Ochem

### Let's get started…

In this Gem, we're going to fill in additional parts of the Ada source files created in Part 1 of this series on using GPS's smart completion mechanism. All the semantic information required to provide this computation is done on the fly. Note that the features described in this Gem are not yet available, and will be released as a part of the GPS version following 4.4.0. GNAT Pro supported customers can request early access to a GPS version with these capabilities.

The completion mechanism relies on a parser launched at GPS startup. The progress of the parser will appear on the bottom right corner of the GPS window. Be aware that completion may not be accurate before the parsing is finished.

### Completion of aggregates

GPS can automatically complete qualified aggregates. Instead of writing a series of assignments to individual components of a variable, let's try automatically initializing the value as a whole. Type "`A_Variable := Rec'(`". The left parenthesis is an automatic trigger for the completion mechanism. The first entry offers the choice of filling in all the fields for that record in a named aggregate -- only the component values need to be provided by the user. Other entries offer the choice of adding a new name in the list of values for the aggregate.

### Completion of pragmas and attributes

Let's say we want to add an assertion before a call to `Arctan`, checking that the argument is greater than or equal to zero.

Just before the statement containing the call, type "**`pragma`**". The smart completion feature will be triggered automatically. The list of all available pragmas is shown, with their associated documentation. Scroll down to `Assert`. You now have access to the documentation for the `Assert` pragma. Press enter once you've finished viewing the documentation, and complete it, for example: "`pragma Assert (X >= 0)`".

Attributes can be listed the same way. For example, by typing `X'`, smart completion is triggered, and all available attributes are listed.

### Completion of generic entities

Let's consider an instance of an Ada container in our application. Start with a simple main subprogram:

```
with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;

package Main is
   type Rec is record
      A, B, C : Integer;
   end record;
begin
   null;
end Main;
```

Then, in the declarations, enter: "`package R_List is new Doubly_Linked_Lists (`". Smart completion is triggered, and you can complete the formal part. Let's just choose `Element_Type` here, and provide `Rec` as the actual parameter. Add a use clause on that package. You should now have:

```
package R_List is new Ada.Containers.Doubly_Linked_Lists (Element_Type
=> Rec);
use R_List;
```

Declare a variable of that list kind, such as "`L : R_List.List;`". Then in the sequence of statements, after the begin, add a few elements, for instance "`L.Append (Rec'(0, 0, 0));`". Then try to access the first element. Type "`L.First_Element.`". The completion feature understands the generic instantiation and offers to complete the selected name with one of the three fields, `A`, `B`, or `C`.

This completes our small tutorial on using the GPS smart completion capability. As mentioned, the features presented in this Part 2 Gem will be available in the upcoming release of GPS (version 5.0.0).

**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

**Gem #92: Code Archetypes for Real-Time Programming - Part 2**

**Author:** Marco Panunzio**,** *University of Padua*

**Let's get started…**

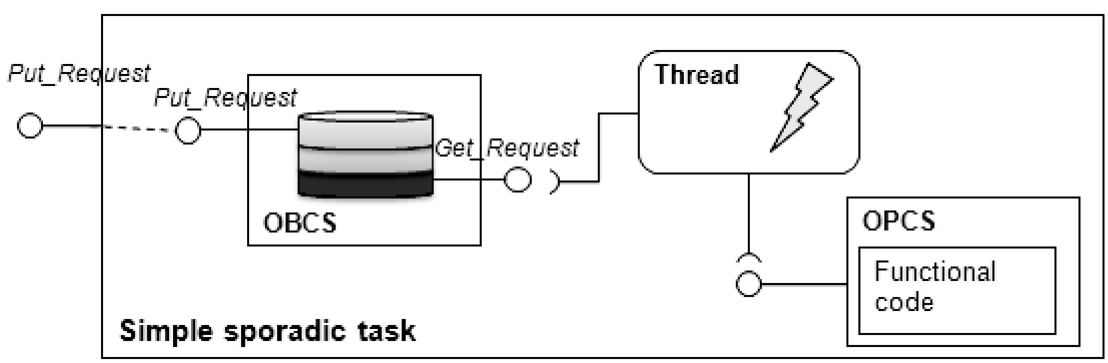
In the previous Gem in this series we introduced the key concepts underlying our code archetypes and described the simplest of our archetypes, which realizes a cyclic task. In this Gem we show how to realize an equally simple sporadic task. In the next Gem in this series, we will depart from this level of simplicity to realize a complete archetype that overcomes some of the limitations intrinsic in these initial solutions.

**Simple sporadic task**

A sporadic task is a task such that any two subsequent activations of it are always separated by no less than a minimum guaranteed time span. This minimum separation is typically called *minimum inter-arrival time* (MIAT). In this initial archetype, the task executes a single operation at each activation, and it does so in response to a request issued by an external client.

Much like the cyclic task, our sporadic task is composed of: (i) a protected object, shared with the outside world, that external clients invoke to post their requests for execution. In the previous Gem we termed this resource an OBCS; and (ii) a thread of control that waits for incoming requests, fetches the first of them from the protected object and executes the sporadic operation that corresponds to a specific operation as provided by the OPCS.

The task structure whose code we are about to show is sketched in the figure below.




Fortunately, the Ada language is well equipped to realize that structure, and we implement that in our archetype using a protected object under the *Ceiling_Locking* policy for (i) and a task type for (ii).

For the moment, we illustrate a base version of the archetype for a sporadic task. In the explanation we also illustrate some of its limitations that will not be present in a more complex version of the archetype.

The specification for the simple sporadic task follows:

```
with System;

generic
   with procedure Sporadic_Operation;
   Ceiling : System.Priority;
   OBCS_Size : Integer;
package Simple_Sporadic_Task is

   procedure Put_Request;

   task type Thread_T
     (Thread_Priority : System.Priority; Interval : Integer)
   is
      pragma Priority (Thread_Priority);
   end Thread_T;

end Simple_Sporadic_Task;
```

The specification defines (as for the Cyclic task that we presented in the previous Gem) a task type inside a generic package. When instantiating the package we specify the sporadic operation for the task, the Ceiling Priority for the OBCS protected object, and the size of the queue of requests of the OBCS.

Additionally, we create the procedure *Put_Request*, that is used by clients to post a request to the sporadic task.

The body for the package is instead:

```
with System_Time;
with System_Types;
with Ada.Real_Time;

package body Simple_Sporadic_Task is

   Protocol : System_Types.Simple_Sporadic_OBCS (Ceiling, OBCS_Size);

   procedure Put_Request is
   begin
      Protocol.Put_Request;
   end Put_Request;

   task body Thread_T is

      use Ada.Real_Time;
      Next_Time : Time := System_Time.System_Start_Time +
        System_Time.Task_Activation_Delay;
      MIAT : Time_Span := Milliseconds (Interval);
      Release : Time;

   begin
      loop
         delay until Next_Time;
```

```
         Protocol.Get_Request (Release);
         Next_Time := Release + MIAT;
         Sporadic_Operation;
      end loop;
   end Thread_T;

end Simple_Sporadic_Task;
```

Comparing this body to the body of the cyclic task, two major differences appear: (i) the presence of an OBCS; and (ii) a slightly modified loop structure.

As in the cyclic task, the sporadic task enters its infinite loop and suspends itself until the system-wide start time. After that: (i) it calls the *entry* Get_Request(Time) of the OBCS; (ii) after the execution of the entry (from which, as we show later on, it obtains a timestamp of when release actually occurred), the task executes the *Sporadic_Operation* (single, for now) specified at the instantiation of its generic package; (iii) it calculates the next earliest time of release (*Next_Time*) so as to respect the minimum separation between subsequent activations. Therefore, on the next iteration of the loop the task issues a request for absolute suspension until that time, and thus it won't probe the OBCS for execution requests until the required minimum separation has elapsed.

As a final note, when the procedure *Put_Request* is called, it just performs a simple indirection to an OBCS procedure with the same name. To appreciate that, we must take a look at the OBCS, which acts as the synchronization agent for the task.

The specification of the OBCS is as follows:

```
with System;
with Ada.Real_Time; use Ada.Real_Time;

package System_Types is

   protected type Simple_Sporadic_OBCS (C : System.Priority; Size :
Integer) is
      pragma Priority(C);
      procedure Put_Request;
      entry Get_Request (Release_Time : out Time);
   private
      Max_Pending : Integer := Size;
      START_Pending : Integer := 0;
      Barrier : Boolean := False;
   end Simple_Sporadic_OBCS;

end System_Types;
```

The OBCS declares a procedure *Put_Request* that is used to post requests in its queue, and a guarded entry *Get_Request(Time)* that is used by the thread to fetch the requests. In the private part of the declaration, the *Max_Pending* attribute is used to set the maximum number of pending requests that the OBCS can hold (obviously no greater than its size); the *START_Pending* attribute indicates the actual number of pending requests; finally the Boolean *Barrier* is used to control the guard of *Get_Request(Time)*.

```
package body System_Types is

   protected body Simple_Sporadic_OBCS is

      procedure Update_Barrier is
      begin
         Barrier := Start_Pending > 0;
      end Update_Barrier;

      procedure Put_Request is
      begin
         if Start_Pending < Max_Pending then
            Start_Pending := Start_Pending + 1;
         end if;
         Update_Barrier;
      end Put_Request;

      entry Get_Request (Release_Time : out Time) when Barrier is
      begin
         Release_Time := Ada.Real_Time.Clock;
         Start_Pending := Start_Pending - 1;
         Update_Barrier;
      end Get_Request;

   end Simple_Sporadic_OBCS;

end System_Types;
```

The body of the OBCS is quite easy to understand. When the procedure *Put_Request* is called, the number of pending requests (*START_Pending*) is increased unless the maximum number has already been reached. In that case the new request is just silently ignored.

The entry *Get_Request(Time)* is used by the task to probe the OBCS for pending requests. In the case where there are requests, the *Barrier* guard is open and the task: (i) saves the time stamp of the execution of the entry (which notionally coincides with the release of the task), that is later used to calculate the next release time; and (ii) decreases the number of pending requests.

At the end of *Put_Request* and *Get_Request*, the value of the *Barrier* guard is refreshed using *Update_Barrier*. In the event that there are no more pending requests, *Barrier* is set to false. For this reason, if the guard is closed when the task calls the entry, the call is blocked until a new request is posted.

The check for the request queue to be not empty is not directly used as the guard expression for the entry, so as to comply with the restriction of the Ravenscar Profile that requires guards to be simple Boolean conditions, and thus have deterministic evaluation. The OBCS has a single entry, as the profile requires, and the only task that can be enqueued on it is the task to which the OBCS belongs, thus ensuring full compliance with the Ravenscar Profile.

While the proposed structure achieves our goal of creating a sporadic task, we immediately notice two potential drawbacks: the *Sporadic_Operation* is parameterless, and the synchronization protocol is very, perhaps too, simple to capture real-life system needs.

For what concerns the first issue, clients of the sporadic task simply trigger new releases of the task, but cannot, for example, pass data to the task as parameters of the release request. Creating a nontrivial producer-consumer collaboration pattern with this task structure is impossible because the consumer task (our sporadic task) cannot receive any data to process.

For what concerns instead the OBCS, in this version it is a simple counter of pending requests.

In the next Gems in this series, we will illustrate how to support sporadic operations with parameters and start to realize more complex queuing policies for execution requests.

**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

## Gem #93: High Performance Multi-core Programming - Part 1

**Author:** Pat Rogers

**Let's get started…**

Chameneos-Redux is part of the Computer Language Shootout, a suite of benchmarks that compares the implementations of various programming languages across different kinds of applications and platforms. The program is required to perform a specified number of rendezvous between mythical "chameneos" creatures, where each creature is represented by a distinct thread. Each rendezvous is symmetric, in that the participating creatures can be either the caller or the called member of any given encounter.

Although both single-core and multi-core machines are used in the benchmarks, we focus only on the multi-core versions with our Ada implementation. The multi-core benchmark results for all implementations are available here: Chameneos-Redux.

As you will see on that web page, there are several implementations. (All implementations are supplied by volunteers, written against common requirements.) The fastest implementations are currently written in GNU C, GNU C++, SBCL Lisp, and GNAT Ada. After those, the other implementations, also written in C, Java, C++, Ada, and numerous other languages, are considerably slower. On the author's development machine, the C++ and Ada implementations have essentially the same performance results, with a slight advantage for C++, but on the official benchmark machine our Ada version is noticeably slower than the C++ version. The discrepancy is under investigation. The overall results occasionally fluctuate too, due presumably to unexpected perturbations in the benchmarking platform, such that the top-ranked versions can move up or down a little relative to each other.

The reason the other implementations are considerably slower is that they do not use the same fundamental design. That fact is our first point to be made about performance: design trumps tuning. When it comes to performance, no amount of tweaking can compensate for an inherently slow design. For example, an earlier Ada implementation (supplied by another volunteer) used an asynchronous select statement. In addition to the difficulties in using this statement correctly, the semantics are such that it is inherently expensive, and would be so in any programming language. Worse yet, the asynchronous select statement was the "else part" of a selective accept statement for a rendezvous, all within a loop. Effectively the program was polling in the most expensive manner imaginable. Worst of all, this code was in the worst possible place – the code implementing the primary behavior of the threads. As a vehicle for displaying Ada constructs the design was impressive, but as a demonstration for performance it was not competitive.

In contrast, the three fastest designs (including the current Ada implementation) all use a shared variable for this most critical aspect of the implementation. The shared variable is accessed using a specific machine instruction that locks the memory bus and then atomically reads and updates the value. By packing both the number of rendezvous completed and the identities of the creatures awaiting rendezvous into a single shared word, this single-instruction approach provides an extremely fast method of updating program state and passing data among the threads.

The machine instruction is a "compare-and-swap" (CAS) instruction that can be accessed as a GCC compiler "built-in". It can also be accessed by writing the machine-code insertion directly, but the built-in is more convenient. To import it into an Ada program, one declares the subprogram and then uses pragma Import for the completion as usual, but with a convention of "intrinsic" because it is an instruction sequence issued directly by the compiler.

```
function Sync_Val_Compare_And_Swap_32
  (Destination : access Unsigned_32;
   Comparand   : Unsigned_32;
   New_Value   : Unsigned_32)
   return Unsigned_32;

pragma Import (Intrinsic, Sync_Val_Compare_And_Swap_32,
"__sync_val_compare_and_swap_4");
```

The declaration for the built-in is that of a function, because the value prior to the swap is returned. Specifically, the CAS built-in compares Comparand to Destination.all, and if they have the same value, writes New_Value into Destination.all. The caller can then check the value returned to see whether the update actually took place, as well as using that value for other purposes. (There is also a version that returns a Boolean value indicating whether the swap occurred, instead of returning the prior value.)

The other requirement for importing an intrinsic built-in is to deal with overloading. There are several forms of the CAS instruction, depending on the size of the data in question, resulting in overloaded versions of the built-in. GNAT currently does not automatically resolve overloaded intrinsic operations, so the External_Name parameter to pragma Import must identify which version is intended. A suffix is appended to the name for this purpose. In the declaration above, we are using 32-bit quantities, so we specify the name as shown, in which the suffix "_4" indicates the number of bytes to manipulate and thus the version of the built-in desired.

In a future Gem we will explore additional steps used to increase performance, including a user-defined allocator that ensures all allocations are cache-aligned, tuning by specifying adherence to language restrictions, and assigning threads to cores so that threads execute with maximum performance.

### Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

**Gem #94: Code Archetypes for Real-Time Programming - Part 3**

**Author:** Marco Panunzio, *University of Padua*

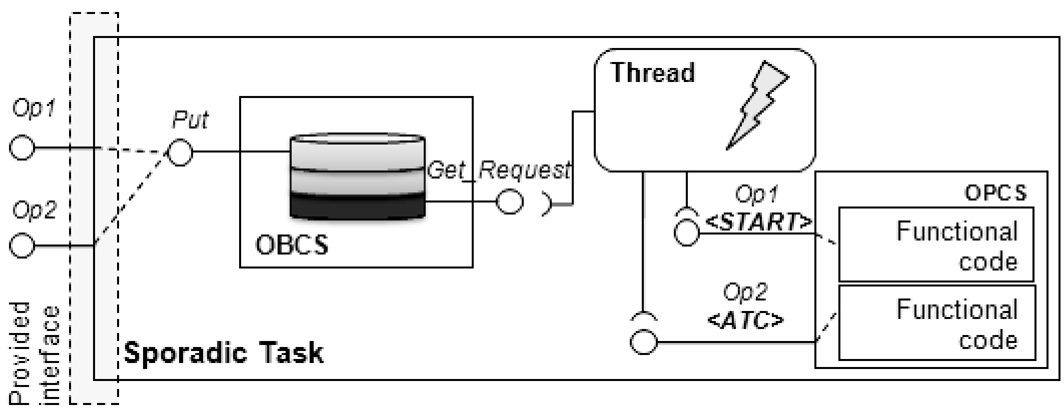
**Let's get started…**

In the previous Ada Gem we described a code archetype for a simple sporadic task. Nevertheless, we recognized that the archetype is not completely satisfactory for at least two reasons: (i) it is not possible to pass parameters to the sporadic operation; (ii) the synchronization agent (OBCS) is a simple counter of pending requests.

In this Ada Gem we illustrate a more complex archetype that supports the invocation of a sporadic operation with parameters. Additionally, we want to explore how it is possible to enrich the OBCS to support complex queueing policies for the incoming requests.

**Sporadic task**

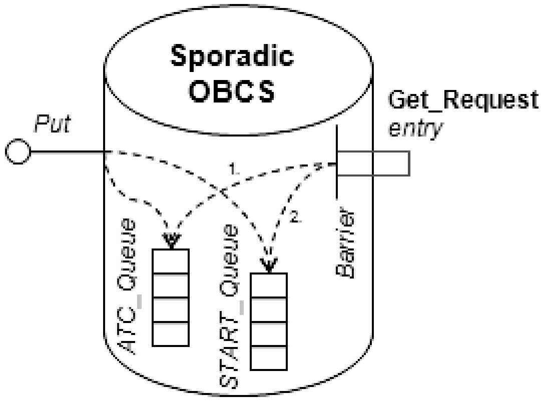
The archetype of a sporadic task that we wish to illustrate is depicted in the figure below.




Suppose that we want to create a sporadic task that at each release can execute either operation `Op1` or operation `Op2`, according to incoming requests by clients. Executing different operations with the same task is not an unusual need in real-time systems, especially when the execution platform has scarce computational power and memory resources, and the excessive proliferation of tasks may tax the system too much.

Furthermore, in this archetype we may also want to establish some relative ordering of importance between `Op1` and `Op2`. We consider `Op1` as the nominal operation of the sporadic task (the operation normally called by clients), and we call it the START operation; in contrast, we consider `Op2` to be a modifier operation, called the 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`. This choice implies that modifier operations are allowed to cause a one-time (as opposed to permanent) modification of the nominal execution behavior of the task.

**OBCS for a sporadic task**




We want to encapsulate the implementation of this policy and simply expose to clients of this sporadic task a set of procedures with the signatures of `Op1` and `Op2`; the role of these procedures is to reify 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.

**Sporadic Task -- System Types and Task Type**

Let us now have a look at the set of types we need to implement this archetype. They are declared in a modified version of the package `System_Types` that we also used in the preceding Ada Gem.

```
with System;
with Ada.Real_Time; use Ada.Real_Time;
with System_Time;
with Ada.Finalization; use Ada.Finalization;

package System_Types is

   -- Abstract parameter type --
   type Param_Type is abstract tagged record
      In_Use : Boolean := False;
   end record;

   -- Abstract functional procedure --
   procedure My_OPCS (Self : in out Param_Type) is abstract;

   type Param_Type_Ref is access all Param_Type'Class;
   type Param_Arr is array(Integer range <>) of Param_Type_Ref;
   type Param_Arr_Ref is access all Param_Arr;

   -- Request type --
   type Request_T is (NO_REQ, START_REQ, ATC_REQ);

   -- Request descriptor to reify an execution request
   type Request_Descriptor_T is
```

```
      record
         Request : Request_T;
         Params : Param_Type_Ref;
      end record;

   -- Parameter buffer
   type Param_Buffer_T(Size : Integer) is
      record
         Buffer : aliased Param_Arr(1..Size);
         Index : Integer := 1;
      end record;

   type Param_Buffer_Ref is access all Param_Buffer_T;

   procedure Increase_Index(Self : in out Param_Buffer_T);
```

We have declared a set of types to represent parameters, a type describing the kinds of requests (START_REQ, ATC_REQ, and an additional kind NO_REQ just for the sake of the explanation), and a request descriptor type to encapsulate the information about invocations of `Op1` and `Op2`. We also declare procedure `My_OPCS(..)`, which is an abstract procedure that represents all possible operations that can be invoked by the sporadic task.

We now continue on with the remainder of the specification of package `System_Types`:

```
   -- Abstract OBCS --
   type OBCS_T is abstract new Controlled with null record;
   type OBCS_T_Ref is access all OBCS_T'Class;

   procedure Put(Self : in out OBCS_T; Req : Request_T; P :
Param_Type_Ref)
      is abstract;

   procedure Get(Self : in out OBCS_T; R : out Request_Descriptor_T)
      is abstract;

   -- Sporadic OBCS --
   type Sporadic_OBCS(Size : Integer) is new OBCS_T with
      record
         START_Param_Buffer : Param_Arr(1..Size);
         START_Insert_Index : Integer;
         START_Extract_Index : Integer;
         START_Pending : Integer;
         ATC_Param_Buffer : Param_Arr(1..Size);
         ATC_Insert_Index : Integer;
         ATC_Extract_Index : Integer;
         ATC_Pending : Integer;
         Pending : Integer;
      end record;

   overriding
   procedure Initialize(Self : in out Sporadic_OBCS);

   overriding
```

```
   procedure Put(Self : in out Sporadic_OBCS; Req : Request_T; P :
Param_Type_Ref);

   overriding
   procedure Get(Self : in out Sporadic_OBCS; R : out
Request_Descriptor_T);

end System_Types;
```

Above, we declare a root type to represent an abstract OBCS (`OBCS_T`) and a `Sporadic_OBCS` type that implements the queueing policy we previously described. `START_Param_Buffer` and `ATC_Param_Buffer` are two distinct circular buffers that are used to store the invocations of the respective types of operation. In addition, we create a buffer for parameters.

The package body follows:

```
package body System_Types is

   -- Sporadic OBCS --
   procedure Initialize (Self : in out Sporadic_OBCS) is
   begin
      Self.START_Pending       := 0;
      Self.START_Insert_Index  := Self.START_Param_Buffer'First;
      Self.START_Extract_Index := Self.START_Param_Buffer'First;
      Self.ATC_Pending         := 0;
      Self.ATC_Insert_Index    := Self.ATC_Param_Buffer'First;
      Self.ATC_Extract_Index   := Self.ATC_Param_Buffer'First;
   end Initialize;

   procedure Put(Self : in out Sporadic_OBCS; Req : Request_T; P :
Param_Type_Ref) is
   begin
      case Req is
         when START_REQ =>
            Self.START_Param_Buffer (Self.START_Insert_Index) := P;
            Self.START_Insert_Index := Self.START_Insert_Index + 1;
            if Self.START_Insert_Index > Self.START_Param_Buffer'Last
then
               Self.START_Insert_Index :=
Self.START_Param_Buffer'First;
            end if;
            -- Increase the number of pending requests, but do not
overcome
            -- the number of buffered ones
            if Self.START_Pending < Self.START_Param_Buffer'Last then
               Self.START_Pending := Self.START_Pending + 1;
            end if;
         when ATC_REQ =>
            Self.ATC_Param_Buffer (Self.ATC_Insert_Index) := P;
            Self.ATC_Insert_Index := Self.ATC_Insert_Index + 1;
            if Self.ATC_Insert_Index > Self.ATC_Param_Buffer'Last then
               Self.ATC_Insert_Index := Self.ATC_Param_Buffer'First;
            end if;
```

```
            if Self.ATC_Pending < Self.ATC_Param_Buffer'Last then
               -- Increase the number of pending requests, but do not
overcome
               -- the number of buffered ones
               Self.ATC_Pending := Self.ATC_Pending + 1;
            end if;

         when others => null;
      end case;
      Self.Pending := Self.START_Pending + Self.ATC_Pending;
   end Put;

   procedure Get(Self : in out Sporadic_OBCS; R : out
Request_Descriptor_T) is
   begin
      if Self.ATC_Pending > 0 then
         R := (ATC_REQ, Self.ATC_Param_Buffer(Self.ATC_Extract_Index));
         Self.ATC_Extract_Index := Self.ATC_Extract_Index + 1;
         if Self.ATC_Extract_Index > Self.ATC_Param_Buffer'Last then
            Self.ATC_Extract_Index := Self.ATC_Param_Buffer'First;
         end if;
         Self.ATC_Pending := Self.ATC_Pending - 1;
         else
            if Self.START_Pending > 0 then
               R := (START_REQ,
Self.START_Param_Buffer(Self.START_Extract_Index));
               Self.START_Extract_Index := Self.START_Extract_Index +
1;
               if Self.START_Extract_Index >
Self.START_Param_Buffer'Last then
                  Self.START_Extract_Index :=
Self.START_Param_Buffer'First;
               end if;
               Self.START_Pending := Self.START_Pending - 1;
            end if;
      end if;
      R.Params.In_Use := True;
      Self.Pending := Self.START_Pending + Self.ATC_Pending;
   end Get;

   procedure Increase_Index(Self : in out Param_Buffer_T) is
   begin
      Self.Index := Self.Index + 1;
      if Self.Index > Self.Buffer'Last then
         Self.Index := Self.Buffer'First;
      end if;
   end Increase_Index;
end System_Types;
```

In the package body we implement the desired queuing policy. Procedure `Put(..)` simply inserts the representation of the incoming request in the queue of the requested operation kind (START_REQ or ATC_REQ). The ordering among requests of the same operation kind is FIFO.

Procedure `Get(..)` is used to extract a request descriptor. We can see that as long as there are pending ATC requests, they are selected based on their arrival order. When the ATC queue is empty, requests for START operations are fetched.

The task that uses this sporadic OBCS has a specification almost identical to the "simple sporadic task" we presented in the preceding Ada Gem. The only difference is that `Get_Request` now also fetches a request descriptor.

```
with System_Types; use System_Types;
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;

generic
   with procedure Get_Request(Req : out Request_Descriptor_T;
                              Release : out Time);
package Sporadic_Task is

   task type Thread_T (Thread_Priority : Any_Priority;
                      MIAT : Integer) is
      pragma Priority (Thread_Priority);
   end Thread_T;

end Sporadic_Task;
```

The body for the task type follows:

```
with System_Time; use System_Time;
package body Sporadic_Task is

   task body Thread_T is
      Req_Desc : Request_Descriptor_T;
      Release : Time;
      Next_Time : Time := System_Start_Time;
   begin
      loop
         delay until Next_Time;
         Get_Request (Req_Desc, Release);
         Next_Time := Release + Milliseconds (MIAT);
         case Req_Desc.Request is
            when NO_REQ =>
                    null;
            when START_REQ | ATC_REQ =>
                    My_OPCS (Req_Desc.Params.all);
            when others =>
                    null;
         end case;
      end loop;
   end Thread_T;

end Sporadic_Task;
```

Notice that the descriptor of the fetched request can be used to discriminate the action to perform according to the type of operation (this is done with the case statement). In our case, if we fetch a request of kind START_REQ or ATC_REQ, we simply execute My_OPCS, that will dynamically dispatch to the requested operation. This mechanism will be clear when, in a later Gem, we complete the picture with the declaration of `Op1` and `Op2` as seen by their clients.

**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

## Gem #95: Dynamic Stack Analysis in GNAT

**Author:** Quentin Ochem

### Let's get started…

Determining how much stack space should be allocated to tasks is a common memory-management problem. In the absence of tool support, often the only information that developers have is the output EXCEPTION_STACK_OVERFLOW when their program crashes. GNAT offers two basic ways for users to get information on a program's stack usage -- statically or dynamically. This Gem addresses how to obtain data on dynamic stack usage. Measurement of static stack usage will be covered in a later Gem.

### Computing the stack size at task termination

Let's start with a simple program that has a task whose stack size is determined at run time:

```
procedure Main is

   task T is
      entry E (Size : Integer);
   end T;

   task body T is
   begin
      accept E (Size : Integer) do
         declare
            V : array (1 .. Size) of Integer := (others => 0);
         begin
            null;
         end;
      end E;
   end T;

begin
   T.E (500_000);
end Main;
```

This program works fine, but what are its stack requirements? Is there a possibility that by adding new code which may consume additional stack, we'll hit the roof? Let's find out by compiling this with stack instrumentation:

gnatmake main.adb -bargs -u10

The "-bargs -u10" switch causes "-u10" to be passed to the GNAT binder, which will allow up to ten tasks to be instrumented and will output their stack usage upon program completion.

Compiled this way, the program outputs the following information:

```
  Index  | Task Name | Stack Size |      Stack usage
       1 | t         |    2097152 | 2008872 +/- 8188
```

This means that out of the 2,097,152 bytes that are available for the task's stack, 2,008,872 are currently used by the program.

**Adjusting the stack size**

Our stack seems quite full here, and it's probably reasonable to increase its size to be on the safe side and to avoid potential exceptions when the program is extended. This can be done easily by using a pragma Storage_Size:

```
task T is
   pragma Storage_Size (3_000_000);
   entry E (Size : Integer);
end T;
```

Compiling the same program with these changes results in these numbers:

```
  Index  | Task Name | Stack Size |      Stack usage
       1 | t         |    3000000 | 2008872 +/- 8188
```

This is much more reasonable.

**Computing the stack size at run time**

We're now going to create a new version of the task that can be called multiple times. Since this task is going to live longer, and do several things for different clients, we would like to be able to probe the task at different times, namely each time the entry is called. The run-time package GNAT.Task_Stack_Usage provides the means of instrumenting the task. Let's modify the task body as follows:

```
task T is
   pragma Storage_Size (3_000_000);
   entry E (Size : Integer; Name : String);
end T;

task body T is
begin
   loop
      accept E (Size : Integer; Name : String) do
         declare
            V : array (1 .. Size) of Integer := (others => 0);
         begin
            Put_Line ("MAX USAGE OF T AFTER " & Name & ":"
                        & Natural'Image
(GNAT.Task_Stack_Usage.Get_Current_Task_Usage.Value));
         end;
      end E;
```

```
   end loop;
end T;
```

Note the call to Get_Current_Task_Usage, which computes the amount of stack consumed so far after each call to E. Let's now call this entry several times:

```
   T3.E (5_000, "OP 1");
   T3.E (100_000, "OP 2");
   T3.E (20_000, "OP 3");
   T3.E (800_000, "OP 4");
```

This will output:

```
MAX USAGE OF T AFTER OP 1: 29392
MAX USAGE OF T AFTER OP 2: 409392
MAX USAGE OF T AFTER OP 3: 411204
raised STORAGE_ERROR : EXCEPTION_STACK_OVERFLOW
```

Observe that the size of the stack is computed after each call, except for the last call, which results in an exception. Also note an interesting side effect: "OP 3" should take less stack space than "OP 2", so we would normally expect the number to be the same. What's happening is that in "OP 2", the string "MAX USAGE OF T AFTER OP 2: 409392" is computed first, and then Put_Line is called, which itself consumes some stack, up to the level of 411204 bytes. So 411204 is actually the maximum amount of stack space used by OP 2, even though the value displayed is less.

**Related Source Code**

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

HILT 2012

Association for Computing Machinery

*Advancing Computing as a Science & Profession*

# HILT 2012: HIGH INTEGRITY LANGUAGE TECHNOLOGY

## ACM SIGAda's Annual International Conference

## December 2–6, 2012 / Boston, Massachusetts / Advance Program

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 2012 provides a forum for the leading experts from academia/research, industry, and government to present their latest findings in designing, implementing, and using language technology for high integrity software.

***Sponsored by SIGAda, ACM's Special Interest Group on the Ada Programming Language, in cooperation with SIGCSE, SIGPLAN, SIGSOFT, SIGBED, Ada-Europe, and the Ada Resource Association.***

## KEYNOTE TOPICS / FEATURED SPEAKERS

**High-Assurance Cyber Military Systems (HACMS): High-Assurance Vehicles**

KATHLEEN FISHER
DARPA Information Innovation Office

**Challenges for Safety-Critical Software**

NANCY LEVESON
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics
Engineering Systems Division

**Programming the Turing Machine**

BARBARA LISKOV
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

**Hardening Legacy C/C++ Code**

GREG MORRISETT
Harvard University
School of Engineering and Applied Sciences

**Programming Language Life Cycles**

GUY L. STEELE, JR.
Oracle Labs

## CORPORATE SPONSORS

PLATINUM LEVEL

AdaCore
The GNAT Pro Company

SILVER LEVEL

Ellidiss Software
TNI Europe Limited

LDRA

TASC

## PRE-CONFERENCE TUTORIALS / December 2–3

### SUNDAY
### Pre-Conference Tutorials

SF1— Full Day / 9:00 AM–5:30 PM
Bo I. Sandén /
Colorado Technical University
***Design of Multitask Software: The Entity-Life Modeling Approach***

SA1—Morning / 9:00 AM–12:30 PM
Jason Belt, Patrice Chalin, John Hatcliff, and Robby / Kansas State University
***Leading-Edge Ada Verification Technologies: Highly Automated Ada Contract Checking Using Bakar Kiasan***

SA2—Morning / 9:00 AM–12:30 PM
Ed Colbert / Absolute Software
***Ada 2012 Contracts and Aspects***

SP1—Afternoon / 2:00 PM–5:30 PM
Johannes Känig / AdaCore
***Leading-Edge Ada Verification Technologies: Combining Testing and Verification with GNATTest and GNATProve—The Hi-Lite Project***

SP2—Afternoon / 2:00 PM–5:30 PM
Ed Colbert / Absolute Software
***Object-Oriented Programming with Ada 2005 and 2012***

### MONDAY
### Pre-Conference Tutorials

MF1—Full Day / 9:00 AM–5:30 PM
Nancy Leveson, Cody Fleming, and John Thomas /
Massachusetts Institute of Technology
***Safety of Embedded Software***

MA1—Morning / 9:00 AM–12:30 PM
K. Rustan M. Leino / Microsoft Research
***Developing Verified Programs with Dafny***

MA2—Morning / 9:00 AM–12:30 PM
Ricky E. Sward / The MITRE Corporation
Jeff Boleng /
Software Engineering Institute
***Service-Oriented Architecture (SOA) Concepts and Implementations***

MP1—Afternoon / 2:00 PM–5:30 PM
Tucker Taft / AdaCore
***Multicore Programming using Divide-and-Conquer and Work Stealing***

MP2—Afternoon / 2:00 PM–5:30 PM
Kevin Nilsen / Atego
***Understanding Dynamic Memory Management in Safety Critical Java***

## TECHNICAL PROGRAM / December 4–6

### TUESDAY
### Analyzing and Proving Programs

**9:00 AM–10:30 AM**
**Greetings**
SIGAda and Conference Officers
**Keynote Address**
Barbara Liskov,
Massachusetts Institute of Technology
***Programming the Turing Machine***

10:30 AM–11:00 AM Break / Exhibits

**11:00 AM–12:30 PM**
**Program Verification at Compile-Time**

K. Rustan M. Leino
***Program Proving Using Intermediate Verification Languages (IVLs) like Boogie and Why3***

C. Dross, J. Känig, and E. Schonberg
***Hi-Lite: The Convergence of Compiler Technology and Program Verification***

***Industrial/Sponsor Presentation***

12:30 PM–2:00 PM Break / Exhibits

**2:00 PM–3:30 PM**
**Keynote Address**
Greg Morrisett, Harvard University
***Hardening Legacy C/C++ Code***

3:30 PM–4:00 PM Break / Exhibits

**4:00 PM–5:30 PM**
**Advancing Compilation Technology**

V. Pucci and E. Schonberg
***The Implementation of Compile-Time Dimensionality Checking***

H. Kirtchev
***A Robust Implementation of Ada's Finalizable Controlled Types***

5:30 PM–7:00 PM Break

**7:00 PM–10:00 PM**
**Social Event / Dinner**

### WEDNESDAY
### Security and Safety

**9:00 AM–10:30 AM**
**Announcements**
**SIGAda Awards**
Ricky E. Sward, SIGAda Chair
**Keynote Address**
Kathleen Fisher, DARPA
***HACMS: High-Assurance Vehicles***

10:30 AM–11:00 AM Break / Exhibits

**11:00 AM–12:30 PM**
**Languages and Security**

M. Norrish
***Formal Verification of the seL4 Microkernel***

D. S. Hardin
***DSL for Cross-Domain Security***

***Industrial/Sponsor Presentation***

12:30 PM–2:00 PM Break / Exhibits

**2:00 PM–3:30 PM**
**Keynote Address**
Nancy Leveson,
Massachusetts Institute of Technology
***Challenges for Safety-Critical Software***

3:30 PM–4:00 PM Break

**4:00 PM–5:30 PM**
**Languages and Safety**

**TRACK 1**
**Industrial Session on Safety**

K. Nilsen
***Real-Time Java in the Modernization of the Aegis Weapon System***

J. O'Leary
***Software for FAA's Automatic Data Comm Between Air Traffic Controller and Pilot***

***Industrial/Sponsor Presentation***

***Industrial/Sponsor Presentation***

**TRACK 2**
**Real Time Systems**

G. Bosch
***Synchronization Cannot Be a Library***

S. Li et al.
***Applicability of RT Schedulability Analysis on a Software Radio Protocol***

***Industrial/Sponsor Presentation***

5:30 PM–7:00 PM Break

**7:00 PM–10:00 PM**
**Workshops / Birds-of-a-Feather Sessions**

### THURSDAY
### Designing and Implementing Languages

**9:00 AM–10:30 AM**
**Announcements**
**Best Paper and Student Paper Awards**
Jeff Boleng, HILT 2012 Program Co-Chair
**Keynote Address**
Guy L. Steele, Jr., Oracle Labs
***Programming Language Life Cycles***

10:30 AM–11:00 AM Break

**11:00 AM–1:00 PM**
**Compiler Certification Issues**

D. Eilers and T. Koskinen
***Adapting ACATS for Use with Run-Time Checks Suppressed***

**Panel on Compiler Certification**
L. Berringer (CompCert), R. Brukardt (Ada), T. Plum (C, C++, Java)

**Announcements**
**(Ada-Europe 2013, SIGAda 2013)**

**Closing Remarks and Conference Adjournment**

**To register online, and for more information and updates, visit www.sigada.org/conf/hilt2012**

## VENUE / HOTEL

HILT 2012 will be held at the Hyatt Regency Boston, *www.hyattregencyboston.com,* conveniently located in downtown Boston with easy access from Logan Airport.

A block of rooms is reserved for the conference from Thursday night, November 29, through Wednesday night, December 12. The conference rate is $159 for single or double occupancy rooms, $183 for triple occupancy rooms, and $208 for quadruple occupancy rooms. All guest rooms include complimentary wireless Internet. Reservations must be guaranteed by credit card and received by November 3. After this date, the hotel is not obligated to honor conference rates. Please also visit *www.sigada.org/conf/hilt2012/hotel-rates.html* and *www.acm.org/sig_volunteer_info/whyhotel.htm* for additional details.

## SPONSORS / EXHIBITORS

HILT 2012 will include vendor participation, featuring presentations on their products and services during main sessions. For specific information, please contact the Exhibits Chair, Alok Srivastava, *alok.srivastava@tasc.com*.

## GRANTS TO EDUCATORS

As in past years, SIGAda is offering grants to educators to attend the conference. Grants cover the registration and tutorial fees; members of the GNAT Academic Program may be eligible for travel funds from AdaCore. Apply by e-mail, no later than November 16, 2012. Grant program details are available from the conference website or Professor Michael B. Feldman, *mfeldman@gwu.edu*.

## WORKSHOPS / BIRDS-OF-A-FEATHER

To propose a focused workshop or informal Birds-of-a-Feather session related to the conference theme, please contact the Workshops Chair, John W. McCormick, *mccormick@cs.uni.edu*.

## REGISTRATION FEES

**CONFERENCE (FULL)**
Member of ACM, SIGAda, or cooperating organization:
**$575 early** / $725 after Nov. 30
Non-members:
**$875 early** / $975 after Nov. 30
Full-time Student: $50

**CONFERENCE (ONE DAY)**
Member of ACM, SIGAda, or cooperating organization:
**$325 early** / $325 after Nov. 30
Non-members:
**$325 early** / $325 after Nov. 30
Full-time Student: $25

**TUTORIAL (FULL DAY)**
Member of ACM, SIGAda, or cooperating organization:
**$310 early** / $370 after Nov. 30
Non-members:
**$420 early** / $470 after Nov. 30
Full-time Student: $30

**TUTORIAL (HALF DAY)**
Member of ACM, SIGAda, or cooperating organization:
**$155 early** / $185 after Nov. 30
Non-members:
**$210 early** / $235 after Nov. 30
Full-time Student: $15

***For early registration rates, register online by November 30 at* http://sigada.org/conf/hilt2012/register/index.html**

## CONFERENCE TEAM

**Conference Chair / Local Arrangements Chair**
Ben Brosgol, AdaCore / *brosgol@adacore.com*

**Program Co-Chair / Proceedings Chair**
Jeff Boleng, US Air Force Academy / *jeff@boleng.com*

**Program Co-Chair**
Tucker Taft, AdaCore / *taft@adacore.com*

**Workshops Chair / Tutorials Chair**
John W. McCormick, University of Northern Iowa / *mccormick@cs.uni.edu*

**Treasurer**
Ricky E. Sward, The MITRE Corporation / *rsward@mitre.org*

**Webmaster**
Clyde Roby, Institute for Defense Analyses / *clyderoby@acm.org*

**Exhibits and Sponsorships Chair**
Alok Srivastava, TASC Inc. / *alok.srivastava@tasc.com*

**Registration Chair / Academic Community Liaison**
Michael B. Feldman, George Washington University (Ret.) / *mfeldman@gwu.edu*

**Publicity Chair**
Greg Gicca, AdaCore / *gicca@adacore.com*

**Logo Designer**
Weston Pan, Raytheon Space and Airborne Systems

### SIGAda Officers

**Chair**
Ricky E. Sward, The MITRE Corporation / *rsward@mitre.org*

**Vice Chair for Meetings and Conferences**
**Ada Letters Editor**
Alok Srivastava, TASC Inc. / *alok.srivastava@tasc.com*

**Vice Chair for Liaison**
Greg Gicca, AdaCore / *gicca@adacore.com*

**International Representative**
Dirk Craeynest, KU Leuven, Department of Computer Science / *dirk.craeynest@cs.kuleuven.be*

**Secretary**
Clyde Roby, Institute for Defense Analyses / *clyderoby@acm.org*

**Treasurer**
Geoff Smith, Lightfleet Corporation / *gsmith@lightfleet.com*

**Past Chair**
John W. McCormick, University of Northern Iowa / *mccormick@cs.uni.edu*

**Association for Computing Machinery**
2 Penn Plaza, Suite 701
New York, NY 10201-0701
USA