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

Ada LETTERS

Volume XXXIV Number 2 August 2014

Table of Contents

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

Volume XXXIV Number 2, August 2014

Table of Contents

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

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

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

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

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

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

From the Editor's Desk

Alok Srivastava

Welcome to this issue of ACM Ada Letters. In this issue you will find a very interesting paper by Bo Wang, Yuichi Goto and Jingde Cheng on several Queue Operation Related Tasking Deadlocks in Ada 2012 Programs. Another remarkable paper is by our regular contributor Negin Mahani on Ada Implementation of Transaction Level Modeling Transport Channel. Specifies on several remarkable Ada Gems have also been included. In this issue you will find details on major Ada event, the 20th International Conference on Reliable Software Technologies Ada-Europe 2015 to be held from June 22-26, 2015 in Madrid, Spain.

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

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

We look forward to hearing from you!

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

Editorial Policy (from Alok Srivastava, Managing Editor)

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

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

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

Deadline	**Issue**	**Deadline**	**Issue**
June 1st, 2015	August, 2015	October 1st, 2015	December, 2015
February 1st, 2015	April, 2015	June 1st, 2015	August, 2015

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
Verocel
1849 Briland Street
Tarpon Springs, FL 34689, USA
Phone: +1-646-375-0734
Fax: +1-978-392-8501
Email: Gicca@Verocel.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

Queue Operation Related Tasking Deadlocks in Ada 2012 Programs

Bo Wang Yuichi Goto Jingde Cheng

Department of Information and Computer Sciences
Saitama University
Saitama 338-8570, Japan
{wangbo, gotoh, cheng}@aise.ics.saitama-u.ac.jp

Abstract. This article presents some queue operation related tasking deadlocks in Ada 2012 programs.

1 Introduction

A tasking deadlock in a concurrent Ada program is situation where some tasks form a circular waiting relation at some synchronization points that cannot be resolved by the program itself (including the behavior of other tasks), and hence these tasks can never proceed with their computation by themselves [3, 4]. Tasking deadlocks is a serious and complex issue in concurrent Ada programs [6].

To detect, avoid and resolve Ada tasking deadlocks, it is indispensable to identify all types of tasking deadlocks. A task synchronization waiting relation between tasks is a relation such that to synchronize with the other task or tasks, a task is blocked until the synchronization takes place, unless the synchronization waiting has a deadline or is breaked by another task. Cheng proposed a way to completely classify all types of tasking deadlocks by different combinations of various synchronization waiting relations between executing tasking objects [3, 4]. According to this way of classification, various combinations of synchronization waiting relations concerning synchronization waiting tasks may lead to various types of tasking deadlocks [3, 4, 5, 6, 7, 8]. And there is a method and a tool to detect tasking deadlocks at run-time in [6, 8].

Ada 2012 defined four new operations those can block a task, i.e., procedure Enqueue and procedure Dequeue in the package Synchronized_Queue_Interfaces in Queue container, procedure Wait_For_Release in the package Synchronous_Barriers and procedure Suspend_Until_True_And_Set_Deadlines in the package Ada.Synchronous_Task_Con

trol.EDF [1, 2, 12]. Moreover, [7] showed some types of tasking deadlocks concerning these operations. On the other hand, since the days of Ada 95 [10], a requeue statement has come available, which can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue [10, 11, 12]. The requeue statement are permitted using procedure renamed an entry, as same as Enqueue and Dequeue in Ada 2012 [12]. Therefore, though the requeue is different from Enqueue and Dequeue in the queue container, they are implemented essentially as same as entry [12]. They are together called queue operations. There may be some queue operation related tasking deadlocks.

The article is organized as follows: Section 2 presents synchronization waiting relations and queue operations in Ada 2012 programs; Section 3 gives examples of queue operation related tasking deadlocks in an Ada 2012 program; Section 4 shows concluding remarks.

2 Synchronization Waiting Relations and Queue Operations

2.1 Synchronization Waiting Relations

Ada 95 defines eight types of synchronization waiting relations, i.e., activation waiting relation, finalization waiting relation, completion waiting relation, acceptance waiting relation, entry-calling waiting relation, protection waiting relation, protected-entry calling waiting relation, and suspension waiting relation [5, 6, 10]. In Ada 2005, there is no new synchronization waiting relations [7, 11]. Since Ada 2012, four new operations are extended [1, 2, 12], such as procedure Enqueue and Dequeue defined in Annex A.18.27 [12], procedure wait_for_release defined in Annex D.10.1 [12], corresponding to enqueue waiting relation, dequeue waiting relation, barrier-release waiting relation [7], respectively, and procedure Suspend_Until_True_And_Set_Deadlines defined in Annex D.10 [12], which does not cause new synchronization waiting relation, because of temporary blocking, i.e., no continuous blocking [7]. So far, there are eleven types of synchronization waiting relation. We also proposed them in [4, 5, 6, 7, 8].

2.2 Queue Operations

The Enqueue and Dequeue in generic package Containers.Synchronized_Queue_Interfaces defined in Annex A.18.27 [12]. From it, four different queue containers are derived, such as A.C.Unbounded_Synchronized_Queues in A.18.28, A.C.Bounded_Synchronized_Queues in A.18.29, A.C.Unbounded_Priority_Queues in A.18.30, A.C.Bounded_Priority_Queues in A.18.31 [1, 2, 12].

A.18.28 representing task-safe queue is unbounded, whose Enqueue can never block, because the capacity of the queue is unbounded; A.18.29 is bounded, and thus Enqueue might cause blocking, if the capacity of the queue is full; Similarly, A.18.30

provides task-safe queue, and Enqueue can never block, while Enqueue in A.18.31 might cause blocking, because of bounded capacity of the queue. If the capacity of the queue is empty, the Dequeue of these four queue containers might cause blocking, because of a dequeue waiting relation. Note that A.18.30 and A.18.31 provide procedure Dequeue_Only_High_Priority, which is not blocking, returns immediately. Unlike Dequeue, it is not synchronization waiting relation [12]. Thus, in this article, we do not discuss the Dequeue_Only_High_Priority of queue operations in Ada 2012.

On the other hand, though Enqueue and Dequeue are defined in Ada 2012, there has been introduced a requeue statement and a requeue with abort statement since Ada 95 [10]. A requeue_statement presents the completion of an accept statement or entry body, while redirecting the corresponding entry call to a new (or the same) entry queue [10, 11, 12]. The statement's form is

```
requeue entry_name_or_procedure_name [with abort];
```

The entry body or accept statement, that has a requeue, is exited, as long as the entry call has been redirected to another entry queue, or the same entry queue. Requeue must is within the body of an accept statement, or within entry body, which is a protected entry. With abort is option, which allows to cancel the call, due to an abort or the expiration of a delay [10, 11, 12], if any.

Ada 83 allows renaming an entry as a procedure [9]. Like this,

```
procedure P(Para: in Item) renames T.E;
```

Ada 2012 provides a new feature [12] that we use a requeue statement using an entry renamed as a procedure as following:

```
requeue P;
```

As mentioned above, the requeue is just redirecting an entry call to a new (or the same) entry queue. If there are not any entry calls in this entry queue, the entry call requeued can reach the accept statement corresponding to it. The sequence of statements, if any, of the accept statement is executed by the called task, while the calling task remains suspended, called rendezvous [9, 10, 11, 12]. But the accept statement or entry body that has the requeue statement is completed. However, if there are another entry calls in this queue, the entry call requeued is queued to waiting until the accept statement corresponding to this entry removes other call in front of it from the queue, i.e., the call requeued must wait until the call in front of it rendezvous. The calls are processed in the order of arrival [9, 10, 11, 12]. And if the requeue statement redirects an entry call within a protected body, then the calling task must wait until protected entry call is finalized. Therefore, a requeue statement cannot cause a synchronization waiting relation, such that blocking a task directly. But it can indirectly cause tasking deadlock.

We consider that the Requeue is also a queue operation, though it is different from Enqueue and Dequeue in Queue Container, as above. Because procedure Enqueue and Dequeue in A.18.27 (The Generic Package Containers. Synchronized_Queue_Interfaces) are implemented by entry in essence [12], as same as Requeue. Three operations are called queue operations. This article is focused on queue operation related tasking deadlocks in Ada 2012 programs.

3 Examples of Queue Operation Related Tasking Deadlocks in Ada 2012 Programs

Seven example programs are presented in this section. We present each program by its Ada 2012 code and Task-Wait-For Graph corresponding to it. A program has a temporary circle with requeue statements of queue operations. Though it does not lead to a tasking deadlock, but an infinite requeue circle. All of others have queue operation related tasking deadlocks.

3.1 Task-Wait-For Graph

A Task-Wait-For Graph (TWFG) at time t (this time may be a physical time in an interleaved implementation of Ada or a virtual time in a distribed implementation of Ada) is a kind of arc-classified digraph to represent tasking waiting state in an execution of an Ada program [4]. The TWFG explicitly represents various types of waiting relations in an execution of an Ada program. The notion of TWFG was originally proposed for classification and detection of tasking deadlocks in Ada 83 programs [3, 4] and was extended to deal with tasking deadlocks in Ada 95 programs [5]. In a TWFG, vertices present tasking objects. An executing tasking object in an execution state of a concurrent Ada 2012 program is any of the following: a task whose activation has been initiated and whose state is not terminated, a block statement that is being executed by a task, a subprogram that is being called by a task, a protected subprogram that is being called by a task, a protected object on which a protected action is undergoing, and a suspension object that is being waited by a task. Arcs indicate synchronization waiting relations which are binary relations between tasking objects [3, 4]. In a TWFG of an Ada 2012 program, there are 11 types of arcs: activation waiting arc, finalization waiting arc, completion waiting arc, acceptance waiting arc, entry-calling waiting arc, protection waiting arc, protected-entry-calling waiting arc, suspension waiting arc, enqueue waiting arc, dequeue waiting arc, and barrier-release waiting arc, respectively, corresponding to an activation waiting relation, finalization waiting relation, completion waiting relation, acceptance waiting relation, entry-calling waiting relation, protection waiting relation, protected-entry-calling waiting relation, suspension waiting relation, enqueue waiting relation, dequeue waiting relation, and barrier-release waiting relation [3, 4, 5, 6, 7, 8].

Various types of arcs are shown by $\rightarrow_{Act}$, $\rightarrow_{Fin}$, $\rightarrow_{Com}$, $\rightarrow_{Acc}$, $\rightarrow_{EC}$, $\rightarrow_{Pro}$, $\rightarrow_{PEC}$, $\rightarrow_{Sus}$, $\rightarrow_{Enq}$, $\rightarrow_{Deq}$, and $\rightarrow_{BR}$, denoting activation waiting arc, finalization waiting

arc, completion waiting arc, acceptance waiting arc, entry-calling waiting arc, protection waiting arc, protected-entry-calling waiting arc, suspension waiting arc, Enqueue waiting arc, Dequeue waiting arc, and Barrier-release waiting arc, respectively [3, 4, 5, 6, 7, 8].

Though the TWFGs corresponding to enqueue and dequeue waiting relations have been proposed in [7], there is no representation of the requeue of queue operations. It needs to present it that might cause a tasking deadlock, even though it is not synchronization waiting relation. The arc $\rightarrow_{R_{eq}}$ represents the entry call requeued is waiting to rendezvous an accept statement corresponding to it in another task. Especially, if the requeue statement is within a protected entry body, the TWFG requires additional representation of tasking objects, i.e., entry body.

3.2 Example Program 1: Infinite Requeue Circle without Tasking Deadlocks

This program has no synchronization waiting relations. Between two entry bodies in the protected body, two requeue statements form a circle representing an infinite requeue action that cannot be resolved by the program itself. Nevertheless, there are no tasking deadlocks, because the circle in Figure 1 is not continuous but temporary, that is, it allows to have just one entry calling at time t, i.e.,

$$E_1 \rightarrow_{R_{eq}} E_2$$

$$E_2 \rightarrow_{R_{eq}} E_1$$

Example Program 1

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Infinite_Requeue_Circle is
  task T1;
  task T2;

  protected E2E is
    entry E1;
    entry E2;
  end E2E;

  protected body E2E is
    entry E1 when True is
    begin
      requeue E2;
    end E1;

    entry E2 when True is
    begin
      delay 1.0;
      put_line("!");
      requeue E1;
    end E2;
  end E2E;

  task body T1 is
```

```
  begin
    E2E.E1;
  end T1;

  task body T2 is
  begin
    E2E.E2;
  end T2;

begin
  null;
end Infinite_Requeuee_Circle;
```

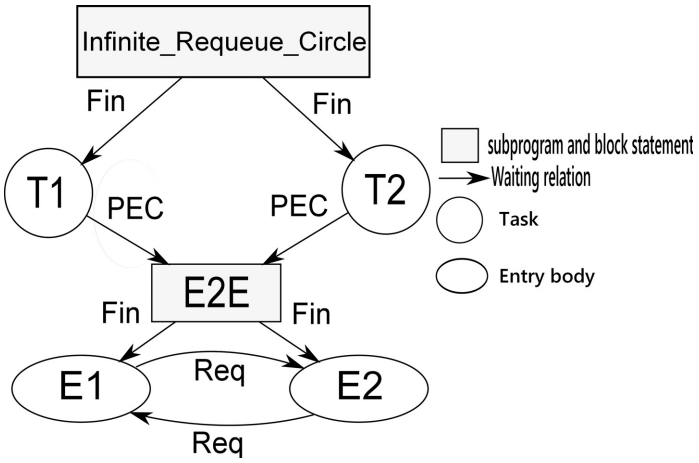


Fig. 1: TWFG of example program 1 for an infinite requeue circle without tasking deadlocks

3.3 Example Program 2: Tasking Deadlocks with Requeue Operations

Two requeue statements cause two synchronization waiting relations indirectly, i.e., there is a continuous circle that cannot be resolved by the program itself. The program might have a queue operation related tasking deadlock, which is circular entry calling during the activation of a task in Figure 2:

$$T_1 \rightarrow_{R_{eq}} T_2 \rightarrow_{R_{eq}} T_1$$

Example Program 2

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Req_Req is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
```

```
            Default_Capacity => 1);

   Q1 : String_Priority_Queues.Queue;
   Elem : Queue_Element;

   task T1 is
     entry E1;
   end T1;

   task T2 is
     entry E2;
   end T2;

   protected E2E is
     entry E1;
     entry E2;
   end E2E;

   protected body E2E is
     entry E1 when True is
     begin
       requeue T2.E2;
     end E1;

     entry E2 when True is
     begin
       requeue T1.E1;
     end E2;
   end E2E;

   procedure P1 renames E2E.E1;
   procedure P2 renames E2E.E2;

   function GET return Integer is
   begin
     P2;
     return 0;
   end GET;

   task body T1 is
   begin
     P1;
     accept E1;
   end T1;

   task body T2 is
     I :Integer :=GET;
   begin
     accept E2;
   end T2;

begin
   null;
end Req_Req;
```

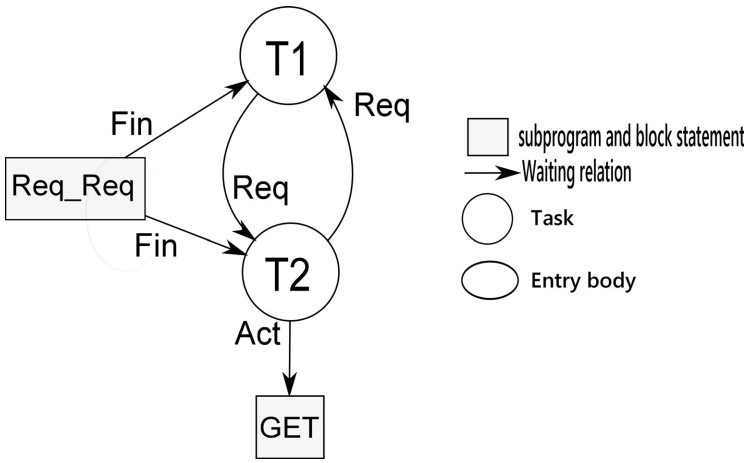


Fig. 2: TWFG of example program 2

3.4 Example Program 3: A Tasking Deadlock with Requeue and Enqueue Operations

In this program requeue and enqueue operations form a circle between two entry bodies, which indirectly result in a tasking deadlock between tasks, when P2 is requeued to entry-calling wait for the finalization of entry body E1 in protected body E2E, while E1 is blocking to wait to enqueue an element to the queue that is full.The synchronization waiting circle in Figure 3 is represented as follows:

$$E_2 \rightarrow_{Req} E_1 \rightarrow_{Enq} E_2$$

Example Program 3

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Req_Enq is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
           Default_Capacity => 1);

  Q : String_Priority_Queues.Queue;
  Elem : Queue_Element;

  protected E2E is
    entry E1;
    entry E2;
  end E2E;

  protected body E2E is
    entry E1 when True is
    begin
      Q.Enqueue(Elem);
```

```
    end E1;

    entry E2 when True is
    begin
      Q.Enqueue(Elem);
      requeue E2E.E1;
    end E2;
  end E2E;

  task T1;
  task T2;

  procedure P1 renames E2E.E1;
  procedure P2 renames E2E.E2;

  task body T1 is
  begin
    delay 0.1;
    P1;
  end T1;

  task body T2 is
  begin
    P2;
  end T2;

begin
  null;
end Req_Enq;
```

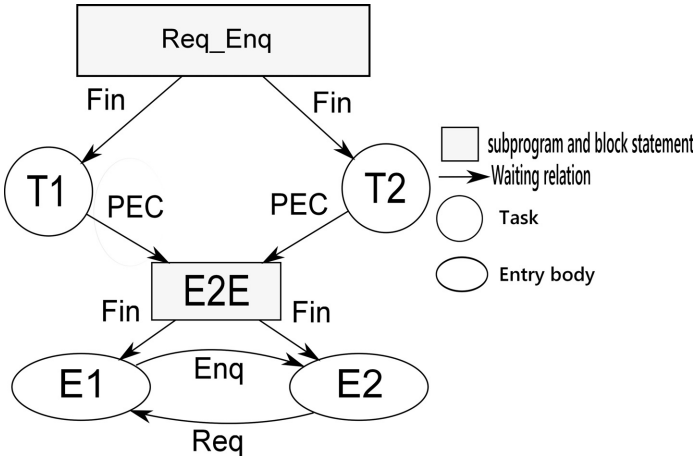


Fig. 3: TWFG of example program 3

3.5 Example Program 4: A Tasking Deadlock with Requeue and Dequeue Operations

Program Req_Deq forms a circle between two entry bodies by requeue and dequeue operations, which indirectly result in a tasking deadlock between tasks, when P2 is requeued to entry-calling wait for the finalization of entry body E1 in protected body E2E, the entry body E2 is completed by requeue statement. However, the Enqueue

below this requeue statement is unreachable code, namely the queue is still empty, resulting in that E1 is blocking to wait to dequeue an element. The synchronization waiting circle in Figure 4 is shown as follows:

$$E_2 \rightarrow_{R_{eq}} E_1 \rightarrow_{D_{eq}} E_2$$

Example Program 4

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Req_Deq is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
           Default_Capacity => 1);

  Q : String_Priority_Queues.Queue;
  Elem : Queue_Element;

  protected E2E is
    entry E1;
    entry E2;
  end E2E;

  protected body E2E is
    entry E1 when True is
    begin
      Q.Dequeue(Elem);
    end E1;

    entry E2 when True is
    begin
      requeue E2E.E1;
      Q.Enqueue(Elem);        --never reach,such that the queue is empty
    end E2;
  end E2E;

  task T1;
  task T2;

  procedure P1 renames E2E.E1;
  procedure P2 renames E2E.E2;

  task body T1 is
  begin
    delay 0.1;
    P1;
  end T1;

  task body T2 is
  begin
    P2;
  end T2;

begin
  null;
end Req_Deq;
```

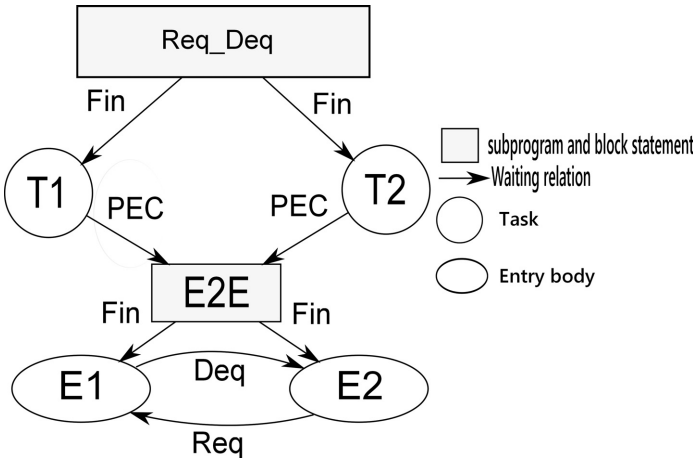


Fig. 4: TWFG of example program 4

3.6 Example Program 5: Tasking Deadlocks with Enqueue Operations

This program has enqueue waiting relations of the same queue. The two enqueue waiting relations form a circle at time t that cannot be resolved by the program itself, when the Enqueue of Q of T1 is enqueue waiting, while the Dequeue of Q of T2 is enqueue waiting, causing an enqueue operaiton related tasking deadlock. The synchronization waiting circle in Figure 5 is followed by:

$$T_1 \rightarrow_{E_{nq}} T_2 \rightarrow_{E_{nq}} T_1$$

Example Program 5

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Enq_Enq is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
           Default_Capacity => 1);

  Q: String_Priority_Queues.Queue;
  Elem : Queue_Element;

  task T1;
  task T2;

  task body T1 is
  begin
    loop
      Q.Dequeue(Elem);
      Q.Enqueue(Elem);
      Q.Dequeue(Elem);
    end loop;
  end T1;
```

```
   task body T2 is
   begin
      loop
         Q.Enqueue(Elem);
         Q.Enqueue(Elem);
         Q.Dequeue(Elem);
      end loop;
   end T2;

begin
   null;
end Enq_Enq;
```

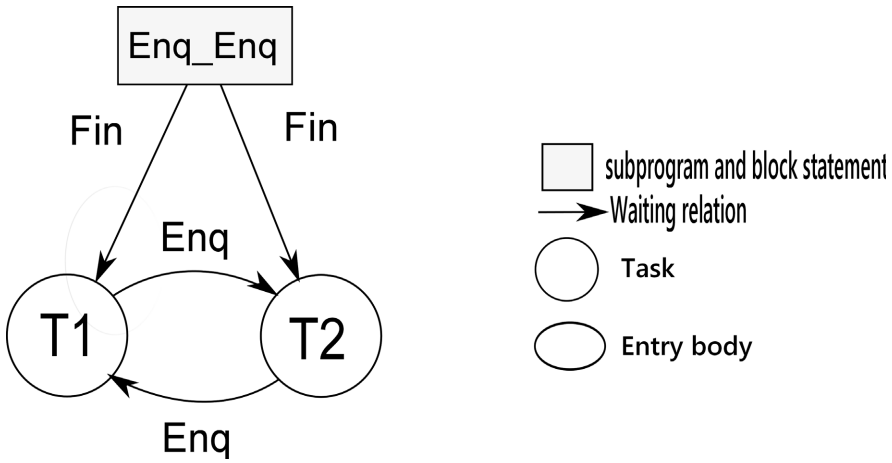


Fig. 5: TWFG of example program 5

3.7 Example Program 6: Tasking Deadlocks with Dequeue Operations

This program has dequeue waiting relations of different queues. The two dequeue waiting relations form a circle that cannot be resolved by the program itself, when the Dequeue of Q1 of T1 is dequeue waiting, while the Dequeue of Q2 of T2 is dequeue waiting, causing a dequeue operaiton related tasking deadlock. The synchronization waiting circle in Figure 6 is given as follows:

$$T_2 \rightarrow_{D_{eq}} T_1 \rightarrow_{D_{eq}} T_2$$

Example Program 6

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Deq_Deq is
   type Queue_Element is new String(1..0);
   package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
           (Element_Type => Queue_Element);
   package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
           (Queue_Interfaces => String_Queues,
```

```
            Default_Capacity => 1);

  Q1, Q2 : String_Priority_Queues.Queue;
  Elem : Queue_Element;

  task T1;
  task T2;

  task body T1 is
  begin
    loop
      Q2.Enqueue("");
      Q1.Dequeue(Elem);
    end loop;
  end T1;

  task body T2 is
  begin
    loop
      Q2.Dequeue(Elem);
      Q1.Enqueue("");
      Q2.Dequeue(Elem);
    end loop;
  end T2;

begin
  null;
end Deq_Deq;
```

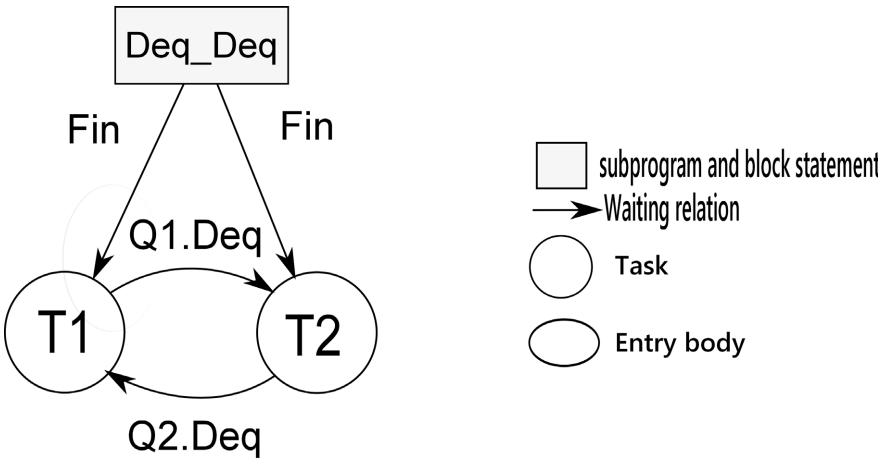


Fig. 6: TWFG of example program 6

3.8 Example Program 7: Tasking Deadlocks with Dequeue and Enqueue Operations

This program has dequeue waiting relations and enqueue waiting relations of different queues. The two dequeue waiting relations form a circle that cannot be resolved by the program itself, when the enqueue of Q1 of T2 is enqueue waiting, while the Dequeue of Q2 of T1 is dequeue waiting, causing a dequeue and enqueue operaitons

related tasking deadlock. The synchronization waiting circle in Figure 7 is expressed as follows:

$$T_2 \rightarrow_{D_{eq}} T_1 \rightarrow_{E_{nq}} T_2$$

Example Program 7

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;
procedure Deq_Enq is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
           Default_Capacity => 1);
  Q1, Q2 : String_Priority_Queues.Queue;
  Elem : Queue_Element;
  task T1;
  task T2;

  task body T1 is
  begin
    loop
      Q2.Dequeue(Elem);
      Q1.Dequeue(Elem);
    end loop;
  end T1;
  task body T2 is
  begin
    loop
      Q1.Enqueue("");
      Q1.Enqueue(Elem);
      Q2.Enqueue(Elem);
    end loop;
  end T2;
begin
  null;
end Deq_Enq;
```

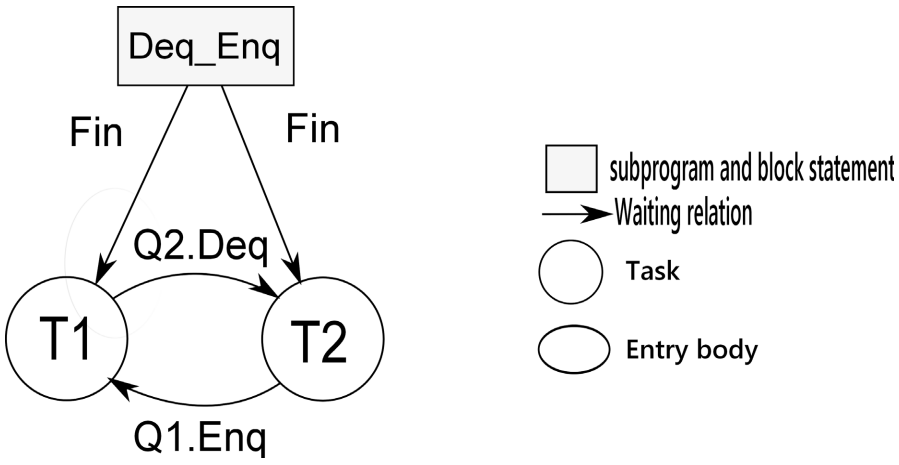


Fig. 7: TWFG of example program 7

3.9 Example Program 8: Tasking Deadlocks with Queue Operations

This program has requeue operation, dequeue and enqueue waiting relations of different queues. The three queue operations form a circle that cannot be resolved by the program itself, when the Enqueue of Q1 of T2 is enqueue waiting, and the Dequeue of Q2 of T3 is dequeue waiting, meanwhile the requeue operation of entry call E2 of T2, during the activation of the task T1, is waiting to rendezvous the finalization of the body of the accept statement of T2, causing queue operaiton related tasking deadlocks together. The synchronization waiting circles in Figure 8 are described as follows:

$$T_1 \rightarrow_{PEC} E2E \rightarrow_{F_{in}} E_1 \rightarrow_{R_{eq}} T_2 \rightarrow_{E_{nq}} T_3 \rightarrow_{D_{eq}} T_1$$

$$T_2 \rightarrow_{E_{nq}} T_3 \rightarrow_{D_{eq}} T_1 \rightarrow_{PEC} E2E \rightarrow_{F_{in}} E_1 \rightarrow_{R_{eq}} T_2$$

$$T_3 \rightarrow_{D_{eq}} T_1 \rightarrow_{PEC} E2E \rightarrow_{F_{in}} E_1 \rightarrow_{R_{eq}} T_2 \rightarrow_{E_{nq}} T_3$$

Example Program 8

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Bounded_Synchronized_Queues;

procedure Queue_Deadlock is
  type Queue_Element is new String(1..0);
  package String_Queues is new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Queue_Element);
  package String_Priority_Queues is new Ada.Containers.Bounded_Synchronized_Queues
          (Queue_Interfaces => String_Queues,
           Default_Capacity => 1);

  Q1, Q2 : String_Priority_Queues.Queue;
  Elem : Queue_Element;

  task T1 is
    entry E1;
  end T1;

  task T2 is
    entry E2;
  end T2;

  task T3;

  protected E2E is
    entry E1;
  end E2E;

  protected body E2E is
    entry E1 when True is
    begin
      requeue T2.E2;
    end E1;
  end E2E;

  procedure P1 renames E2E.E1;

  function GET return Integer is
  begin
    P1;
    return 0;
```

```
  end GET;

  task body T1 is
    I :Integer :=GET;
  begin
    accept E1;
    Q2.Enqueue(Elem);
  end T1;

  task body T2 is
  begin
    accept E2 do
      Q1.Enqueue(Elem);
      Q1.Enqueue(Elem);
    end E2;
    T1.E1;
  end T2;

  task body T3 is
  begin
    Q2.Dequeue(Elem);
    Q1.Dequeue(Elem);
    T2.E2;
  end T3;

begin
  null;
end Queue_Deadlock;
```

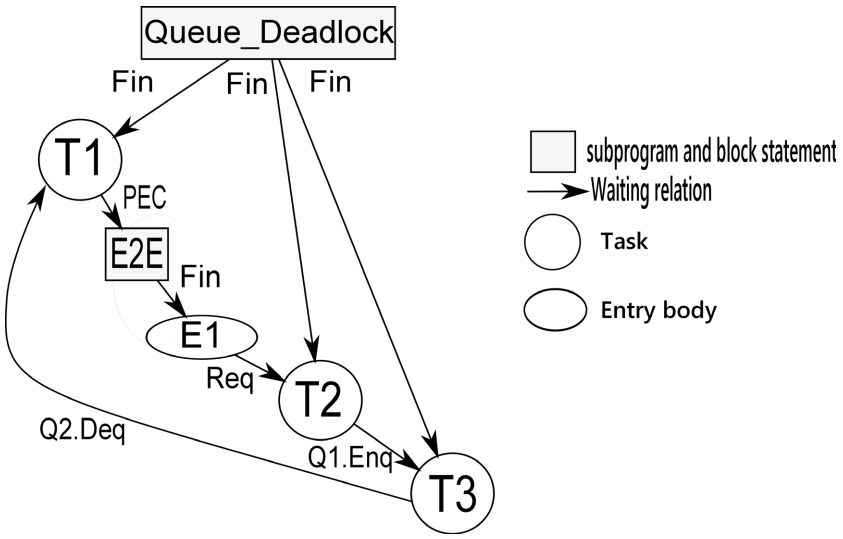


Fig. 8: TWFG of example program 8

4 Concluding Remarks

We have presented some examples of queue operation related tasking deadlocks in Ada 2012 programs, i.e., various types of tasking deadlocks concerning requeue, enqueue, and dequeue operations. The Requeue operation may cause a tasking deadlock indirectly, while the Enqueue operation and Dequeue operation may cause tasking

deadlocks directly. We are extending our run-time detection tool [6, 8] to detect queue operation related tasking deadlocks in Ada 2012 programs at run-time.

References

[1] Barnes, J.: Ada 2012 Rationale. Lecture Notes in Computer Science, Vol. 8338. Springer (2013)

[2] Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)

[3] Cheng, J.: A Classification of Tasking Deadlocks. In: ACM Ada Letters, Vol. 10, No. 5, pp. 110-127, ACM, New Year (1990)

[4] Cheng, J.: Task-Wait-For Graphs and Their Application to Handling Tasking Deadlocks. In: Proc. 3rd ACM Annual TRIAda Conference, pp. 376-390, ACM, New Year (1990)

[5] Cheng, J., Ushijima, K.: Tasking Deadlocks in Ada 95 Programs and Their Detection. In: A. Strohmeier (ed.) Reliable Software Technologies - Ada-Europe 1996. Ada-Europe In-ternational Conference on Reliable Software Technologies, Montreux, Switzerland, June 10-14, 1996, Proceedings, Lecture Notes in Computer Science, Vol. 1088, pp. 135-146, Springer, Heidelberg (1996)

[6] Cheng, J.: Run-Time Detection of Tasking Deadlocks in Real Time Systems with the Ada 95 Annex of Real-Time Systems. In: Reliable Software Technologies - Ada-Europe 2006. 11th International Conference on Reliable Software Technologies, Porto, Portugal, June 5-6, 2006, Proceedings, Lecture Notes in Computer Science, Vol. 4006, pp. 167-178, Springer, Heidelberg (2006)

[7] Ekiba, T., Goto, Y., Cheng, J.: New Types of Tasking Deadlocks in Ada 2012 Programs. In: ACM Ada Letters, Vol. 33, pp. 169-179, ACM, New Year (2013)

[8] Nonaka, Y., Cheng, J., Ushijima, K.: A Tasking Deadlock Detector for Ada 95 Programs. In: Ada User Journal, Vol. 20, No. 1, pp. 79-92 (1999)

[9] ISO/IEC: ISO/IEC 8652:1987 (E): Information Technology - Programming Language - Ada. (1987)

[10] ISO/IEC: ISO/IEC 8652:1995 (E): Information Technology - Programming Language - Ada. (1995)

[11] ISO/IEC: ISO/IEC 8652:2007 (E), Ed. 3: Information Technology - Programming Language - Ada. (2006)

[12] ISO/IEC: ISO/IEC 8652:2012 (E): Information Technology - Programming Language - Ada. (2012)

Ada Implementation of Transaction Level Modeling Transport Channel

Negin Mahani
Zarand High Education Center, Computer Engineering Department, Shahid Bahonar University, Kerman, Iran
Negin.Mahani @uk.ac.ir

Abstract

Transaction level modeling[1] as a new step toward system level modeling is the desired level for hardware descriptors now a days. In the previous researches we have developed TLM-FIFO and TLM-Request-Response channels -two basic cannels of TLM- and some primary master-slave architectures using those implemented channels in Ada programming language for getting some simulation time results.

In this research we have developed TLM-Transport channel specification and body in Ada programming language. As before we have done some simulation tests in Gnat compiler and made some comparisons to its equivalent SystemC TLM model. Several experiments have been conducted to compare the efficiency of SystemAda versus SystemC TLM in two different platforms. Experimental results show that there is no and in the worst case not much simulation time overhead when using SystemAda as compared to SystemC TLM.

1 Introduction

With more complex electronic systems, system design languages with a high level of abstraction for modeling systems are on the market today. Transaction level modeling is now commonly used to simplify the system design. This level of modeling allows designers to focus on design and performance as well as the register transfer level[2] details is released. In Transaction-level components are divided into two general categories computational and communicational components. The components of the communication channel layer are replaced by signals at the register transfer level.

In order to obtain the benefits of the transaction level modeling, use of system level hardware description languages have been widely accepted. To design such systems by hardware description languages a language (a system level language) which vary with the current languages (RT-level languages) should be used. This language should focus on the processing elements that are connected with communication channels (instead of registers and logic units that are linked together by signals at register transfer level). Furthermore designers must be able to communicate to these languages that are used to describe the system to be able to design complex hardware / software systems in the same environment.

With the technical features of the compiler -called Gnat compiler- and the simultaneity of Ada programming language, we can say that this language is a good choice for transaction-level hardware descriptions. This paper introduces SystemAda language -more than the previous papers- for describing hardware systems. According to the characteristics of the system languages, the development process has been divided into two parts TLM and RTL. The possible ways to connect Ada to RTL has been investigated in the previous works.

In TLM section the basic channels of TLM and its interfaces have been implemented based on their necessary characteristics -see our previous papers[1][2][3][4]-.

In previously published papers we have showed Ada examples of simple master-slave architectures using implemented channels and we have compared them to the equivalent SystemC models in terms of simulation time. Besides we have implemented a network on chip system to prove that our SystemAda can model complex systems based on these implemented basic TLM channels as well.

There is only one important channel left called TLM-Transport channel which we are going to implement it in this paper and then have a simulation time comparison with its SystemC equivalent model.

In the second section of this paper a brief introduction of TLM and its basic channels are presented. Section three is about our invented SystemAda and the implementation of the channels. Section four contains the simulation results of different tests. Finally section five is the conclusion.

[1] TLM
[2] RTL

2 What is TLM and TLM Basic Channels?

With the increasing complexity of the digital systems, digital design techniques move to a higher abstraction level. The examples are migration from Transistor Level design into Gate level and then RT level. When hardware design languages become more abstract, the design flow is going to use prefabricate modules and communicate between these modules in the chip level. This fact leads to income some concepts such as System on Chip[1] and Network on Chip[2]. At system level, interconnections become complex busses or switches and the components become complicated processing elements -often designed in RT-level systems-. Designing at the System Level involves defining the functionality of individual processing elements and defining their communications with abstract channels. Transaction Level Modeling (TLM) is regarded as the next step in the direction of system level design. TLM is a transaction-based modeling approach originally based on high-level programming languages such as C++, SystemC and SystemVerilog -and now in Ada called SystemAda-. It emphasizes on separating communications from computations within a system.

Since the channels are the basic entities in the primary TLM standard we should highlight the channels of the initial transaction to the Ada implementations. These channels are all of the components necessary to model communication in hardware systems. Their main characteristic is therefore coincidence. As a result, the units that implement concurrent Ada (tasks) must be used.
The primary channels for raising the level of abstraction used in hardware communication systems are semaphore and mutex channels. They have the same names that implement these concepts in TLM also. FIFO channel as another basic channel is a channel that has both the function and structure of a queue. Channels called Request – Response and Transport are implemented based on FIFO channel[1][2].

3 SystemAda

What language should be used for designing of system-level hardware? SystemC, SystemVerilog, UML, M, or maybe there are other options (Ada)? In terms of technical levels, transaction context is only implemented in object-oriented languages. But various languages such as C / C++, VHDL, and even Java for various levels of system analysis have been used -VHDL is not an object oriented language-. Criteria for choosing the proper system level language can be considered as linguistic structures and their related concepts such as foreign language communication, or include support tools, experts and so on. As it is mentioned in our previous researches we have chosen Ada to have transaction level modeling. The most important reasons for choosing Ada are listed as below:

- High readability
- Ability to describe large systems with high separation of implementation details
- High-speed debugging and having no problems -debugging in SystemC is sometimes confusing-
- The ability to link up with foreign languages
- Ada language is the parent language of VHDL language who has overcome its deficiencies in the field of object-orientation -so hardware designers are familiar with its syntax-
- Ability to communicate with register transfer level[2]
- Reduce the simulation time in some cases
- Ada language inherent concurrent constructs called tasks

Looking at the history of creating VHDL language and its extensions that most have been done based on Ada programming language were another reason for introducing SystemAda idea. A project called SUAVE, wanted to extend VHDL language by accepting common features of object-oriented Ada95 and adding forms of concurrency and communication in the language of the existing portfolio, and so it develop them. Finally the coverage of the world of modeling in system level VHDL has improved[2].

3.1 Describing TLM Channels using Ada

In this section we have described the process of implementation of TLM channels in SystemAda. The first two Channels are implemented in our pervious papers but the last one TLM-Transport channel is implemented for the first time in SystemAda and presented in this work.

[1] SoC
[2] NoC

3.1.1 TLM_FIFO Channel in SystemAda

TLM_FIFO has been implemented completely and its characteristics have been improved 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 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 containing 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[4]

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 the next section. In non-blocking mode, however, the request could be gathered in the request FIFO without paying attention to getting responses. The responses can be inserted in to response queue whenever they are created [4].

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.

3.2 TLM_Transport Channel in SystemAda

Transaction level modeling Transport channel is similar to Request-Response channel but only supports blocking communication. Thus the size of the request and response fifos for this channel is one. Package Transport is defined to implement this channel. It contains a Req_Res channel called Req_Res_Channel. A task called Tansport_Task is defined to implement the functionality of the channel. This task has five entries as Req_Res_Task. Since specification of this package is similar to the Req_Res package, we have not shown it here. Figure 4 shows the package body of transport channel. Transport_Task is similar to the Req_Res_Task except that "or" clauses between different accept statements are removed. This change results in sequential execution of different accept statements and blocking operation in other words. Figure 3 shows master-slave architecture that uses transport channel. In this figure, the messages that master and slave modules pass to the transport channel for blocking communication is shown.

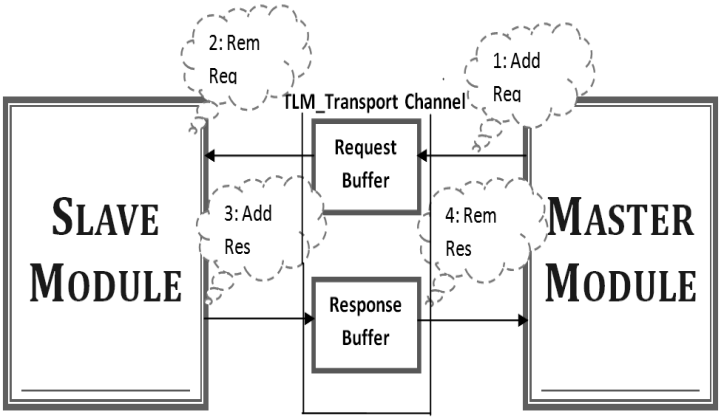


Figure 3. TLM-Master-Slave architecture using TLM-Transport channel

```
package body Transport is
  task body Transport_Task is
  begin
    loop
      select
        accept Add_Req do
          Req_Res_Channel.Req_Input := Req_Input;
          Req_Res_Channel.Tlm_Req_Res,Add_Req;
        end Add_Req;
--      or
        accept Rem_Req do
          if (Req_Res_Channel.Req_fifo.Empty_Flag = FALSE) then
            Req_Res_Channel.Tlm_Req_Res.Rem_Req;
            Req_Output := Req_Res_Channel.Req_Output;
          end if;
        end Rem_Req;
--      or
        accept Add_Res do
          Req_Res_Channel.Req_Input := Res_Input;
          Req_Res_Channel.Tlm_Req_Res.Add_Res;
```

```
          end Add_Res;
--        or
          accept Rem_Res do
            if (Req_Res_Channel.Res_Empty = FALSE) then
              Req_Res_Channel.Tlm_Req_Res.Rem_Res;
              Res_Output := Req_Res_Channel.Res_Output;
            end if;
          end Rem_Res;
          or
          accept Stop;
            Req_Res_Channel.Tlm_Req_Res.Stop;
            exit;
          end select;
      end loop;
    end Transport_Task;
end Transport;
```

Figure 4. Transport channel package body

4 Simulation Result

In previous work we have shown that 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_Transport channel like previous channels.

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_Transport 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 have read from and written to files. Figure 5 is based on the data in Table 1.

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

Packet Number	Ada	SystemC	Simulation Time Ratio	Optimization percent
10000	1.3194	2.8936	2.193118	54.40282
20000	2.632	5.8246	2.212994	54.81235
30000	4.0266	8.5216	2.116326	52.74831
40000	5.499	11.5886	2.107401	52.54819
50000	6.8032	13.9318	2.04783	51.16783
Average			2.135534	53.1359

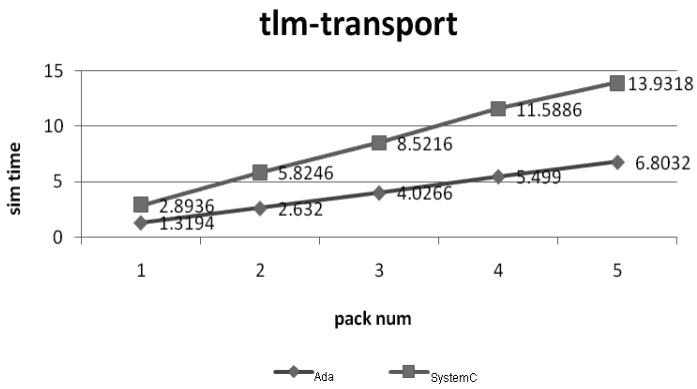


Figure 5. Ada and SystemC TLM_Transport channel simulation time (using I/O files)

Since the difference between SystemAda and SystemC models is much more than expected amount, like past experiments 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 and TLM_Request_Response channels these are I/O file commands[3][4].

So we omit these commands and repeat each of the experiments as before. Table 2 and Figure 6 –that is based on it- show the results of the experiments which have done without I/O file commands. The result shows that there is not much difference between these two models.

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

Packet Number	Ada	SystemC	Simulation Time Ratio	Optimization percent
10000	1.185	0.9126	0.770127	-29.8488
20000	2.3666	1.8588	0.785431	-27.3187
30000	3.5002	2.5432	0.726587	-37.6298
40000	4.7258	3.8102	0.806255	-24.0302
50000	5.9414	4.875	0.820514	-21.8749
Average			0.781783	-28.1405

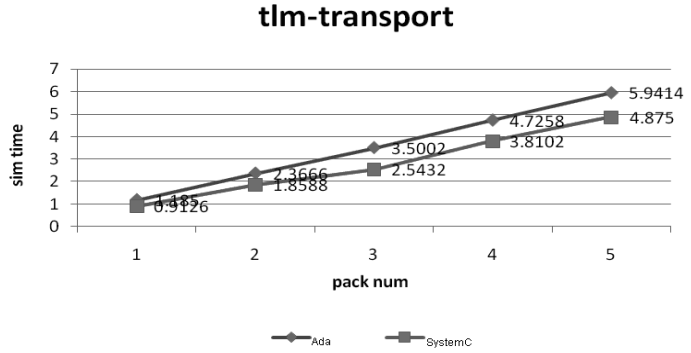


Figure 6. Ada and SystemC TLM_Transport channel simulation time (using no I/O files)

Since tasks are the key concepts in all of the hardware models implementation in Ada, we have implemented two master-slave test-benches for TLM-Transport channel with the same functionality. One has three tasks and the other has two tasks. The results show that the one with the fewer tasks was faster and the average speed ratio was 1.104 as it is shown in the following table.

This amazing result must be because of the number of the existing CPU cores and the process affinity feature. Of course in platform with three CPU cores which has made each task bound to one of its core, the model with three task is faster than the one with two task but in the current platform since the number of the cores do not match the number of tasks in the test-bench with three task and because of context switching cost, the one which implies two task is faster.

Table 3: TLM_Transport channel average simulation time in SystemAda and SystemC for variable packet numbers (the number of the tasks has been reduced)

Packet Number	Ada	SystemC	Simulation Time Ratio	Optimization percent
10000	1.0702	0.9126	0.852738	-17.2693
20000	2.215	1.8588	0.839187	-19.1629
30000	3.224	2.5432	0.788834	-26.7694
40000	4.1968	3.8102	0.907882	-10.1464
50000	5.2282	4.875	0.932443	-7.24513
Average			0.864217	-16.1186

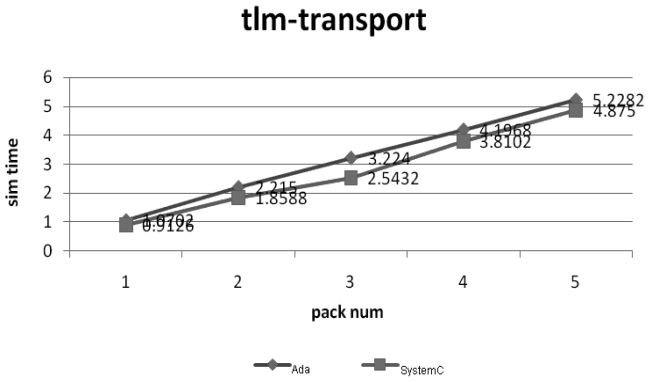


Figure 7. Ada and SystemC TLM_Transport channel simulatiobn time (the number of the tasks has been reduced)

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

As is obvious, an implementation with static memory allocation is faster than one using dynamic memory allocation. The results in Figure 8 based on Table 4 support this fact.

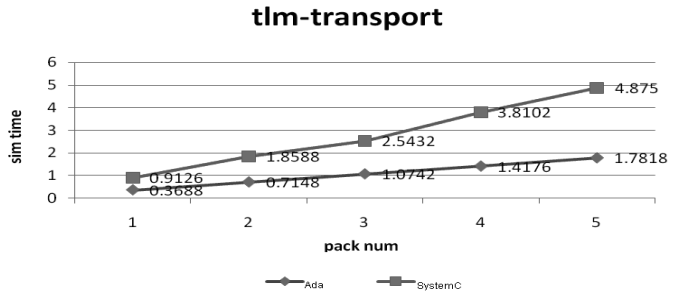


Figure 8. Ada and SystemC TLM_Transport channel simulatiobn time(using static memory allocation)

Table 4: TLM_Transport channel average simulation time in SystemAda and SystemC for variable packet numbers

(using static memory allocation)

Packet Number	Ada	SystemC	Simulation Time Ratio	Optimization percent
10000	0.3688	0.9126	2.474512	59.58799
20000	0.7148	1.8588	2.600448	61.54508
30000	1.0742	2.5432	2.367529	57.76187
40000	1.4176	3.8102	2.687782	62.7946
50000	1.7818	4.875	2.735997	63.45026
Average			2.573254	61.02796

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

In order to compare the simulation time of TLM channels implemented using SystemC and Ada, we have presented another appropriate platform, so we have changed some properties of the platform. The properties of the platform 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 VC++6.0 For SystemC

The obvious difference between the two introduced platforms is the compiler which has been 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 above. 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 5. Figure 9 is based on the data in Table 5.

As it is shown in Table 5 and Figure 9 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 6.33 percent simulation time optimization.

Table 5: TLM_Transport 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	0.456	0.437	0.958333	-4.34783
20000	0.814	0.828	1.017199	1.690821
30000	1.151	1.219	1.059079	5.578343
40000	1.489	1.672	1.122901	10.94498
50000	1.733	2.109	1.216965	17.82835
Average			1.074895	6.338934

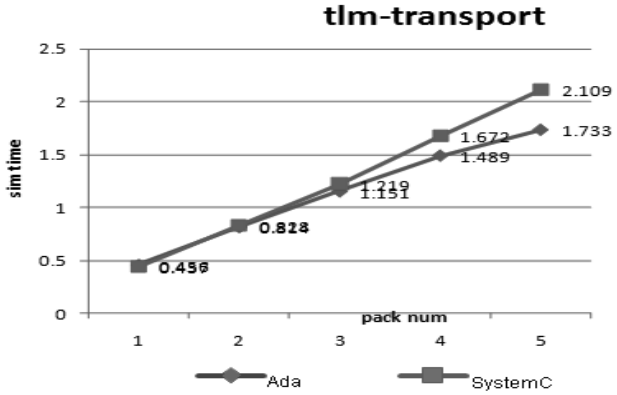


Figure 9. Ada and SystemC TLM_Transport channel simulation time (plat form two)

5 Conclusion

System level languages are the most popular languages for modeling hardware today. Each of these languages has special purpose for their appearance. Among these languages SystemC is more popular for modeling hardware at transaction level but this language has some shortcomings and since it is a C-based language, hardware designers do not feel comfortable while describing their models by SystemC. They prefer to work with a high level language which is similar to VHDL or Verilog in sysntax. This high level language should support inherent concurrency and special high level features such as object orientation and interfaces.

Looking at the VHDL appearance and extension history we encounter Ada programming language as a base language for VHDL. Besides Ada has the other necessary features for supporting TLM in itself (inherent concurrency, object orientation …) so we have chosen it to support TLM. We have modeled basic channels of TLM in our previous works and in this work we have modeled the last primary channel of the first standard of TLM called TLM-Transport channel. Besides like before we have done some simulation time experiments.

All in all, TLM is more than sufficiently fast for today's hardware modeling world. Therefore we do not worry about the speed. Also the results of the experiments show that we do not have significant simulation time penalty. In some cases when we have special platforms like complex compiles or when we use special command like I/O command Ada gains much better results.

In our future work we are going to develop other features of later TLM standards in Ada to show the strength of this language to model hardware at transaction level.

6 References

[1] Negin Mahani, ParniyanMokri, MahshidSedghi, Zainalabedinnavabi, “System Ada : An Ada based System-Level Hardware Description Language” ACM SIGADA AdaLetters, vol. 29, no. 2 , 2009, pp. 15-19.

[2] Negin Mahani, “Making Alive Register Transfer Level and Transaction Level Modeling In System-Ada”, ACM SIGADA AdaLetters, vol. , no. , 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, vol. 33, no. 1, 2013, pp. 157-168.

[4] Negin Mahani, “TLM_Request_Response Channel in SystemAda”, ACM SIGADA AdaLetters,. vol. 32, no. 1, 2012, pp.13-18.

Gem #120 : GDB Scripting -- Part 2

Author: Jean-Charles Delay, *AdaCore*

Let's get started…

Registers

GDB provides you with a number of predefined variables. We already saw some of them in the previous gem (`$argc`, `$arg0`, ...), but much more is available. Among them are variables to access the values of machine registers. See the list below for a full enumeration of them in the case of x86:

```
$eax
$ebx
$ecx
$edx
$eflags
$esi
$edi
$esp
$ebp
$eip
$cs
$ds
$es
$fs
$gs
$ss
```

Armed with those, and using the macro-coding language we saw in the previous Gem, we can easily write some functions to nicely display the variables. For example, let's write a function that displays the flag values in $eflags:

```
define eflags
    printf "     OF <%d>  DF <%d>  IF <%d>  TF <%d>",\
           (($eflags >> 0xB) & 1), (($eflags >> 0xA) & 1), \
           (($eflags >> 9) & 1), (($eflags >> 8) & 1)
    printf "  SF <%d>  ZF <%d>  AF <%d>  PF <%d>  CF <%d>\n",\
           (($eflags >> 7) & 1), (($eflags >> 6) & 1),\
           (($eflags >> 4) & 1), (($eflags >> 2) & 1), ($eflags & 1)
    printf "     ID <%d>  VIP <%d> VIF <%d> AC <%d>",\
           (($eflags >> 0x15) & 1), (($eflags >> 0x14) & 1), \
           (($eflags >> 0x13) & 1), (($eflags >> 0x12) & 1)
    printf "  VM <%d>  RF <%d>  NT <%d>  IOPL <%d>\n",\
           (($eflags >> 0x11) & 1), (($eflags >> 0x10) & 1),\
           (($eflags >> 0xE) & 1), (($eflags >> 0xC) & 3)
end
document eflags
Print eflags register.
end
```

With this macro, issuing an `eflags` command gives the following output:

```
gdb$ eflags
     OF <0>  DF <0>  IF <1>  TF <0>  SF <1>  ZF <0>  AF <0>  PF <0>  CF
<0>
     ID <1>  VIP <0> VIF <0> AC <0>  VM <0>  RF <0>  NT <0>  IOPL <0>
```

This outputs the value of each flag, but in a quite verbose way. Let's write a function to print the flag values by representing each of them by a letter, and print it in upper case if the flag is set, or in lower case if it is not.

```
define flags
    if (($eflags >> 0xB) & 1)
        printf "O "
    else
        printf "o "
    end
    if (($eflags >> 0xA) & 1)
        printf "D "
    else
        printf "d "
    end
    if (($eflags >> 9) & 1)
        printf "I "
    else
        printf "i "
    end
    if (($eflags >> 8) & 1)
        printf "T "
    else
        printf "t "
    end

    ...

    if ($eflags & 1)
        printf "C "
    else
        printf "c "
    end
    printf "\n"
end
document flags
Print flags register.
end
```

The output of this function is, for example:

```
o d I t S z a p c
```

You can of course create as many functions as you need, and customize them however you like.

Enhanced debugging

When you're debugging a running process, it's often desirable to get an overall view of the process context, and do this for each step in the program. The useful process-context commands you have created so far can be called automatically each time you break when debugging your program. For this, you can use a GDB-specific macro called `hook-stop`.

The hook function `hook-stop` is a special definition that GDB calls at every breakpoint event. This means that you can use it to call your user-defined functions each time GDB stops (after a breakpoint, after each `next`/`nexti`, etc.).

The only thing you have to do is define it, and make it do whatever you want.

For example:

```
define hook-stop
  eflags
end
document hook-stop
/!\ For internal use only - do not call manually /!\
end
```

Once this hook is defined, the function `eflags` -- which print the value of each register flag -- will be called each time GDB hits a breakpoint.

Ada-related hints when macro-coding

Beware that the GDB macro scripting syntax varies somewhat depending on the current language set. The language is usually set to "c" when debugging C programs, and is set to "ada" when debugging Ada programs. But the "c" syntax and the "ada" syntax differ. For example, an assignment in "c" mode is done with:

```
set var = 1
```

whereas in "ada" mode it looks like:

```
set var := 1
```

As you can see, GDB adapts its syntax to the current language, so that the user writes with the same convention whether he is developing or debugging. This feature has pros and cons, but this is not the topic of this Gem. The only thing you need to know is that, because of this, most of the macros we have defined so far won't work when debugging Ada programs.

There is, however, a workaround. Inside each macro definition you can manually set the current language. Therefore, the following macro:

```
define hexdump
  if $argc != 1
    help hexdump
  else
    Do the work ...
  end
end
```

can be rewritten as follows:

```
define hexdump
  set language c

  if $argc != 1
    help hexdump
  else
    Do the work ...
  end

  set language auto
end
```

This avoids any issues related to the current language in use, and since we set the language back to "auto", there won't be any side effects after a call to this macro.

Careful

GDB scripting is different from most programming languages, particularly because there is no notion of scoping: every command has a global effect on the GDB environment. Let's say, for example, that you have two macros **macroA** and **macroB**, and that the first one calls the second:

```
define macroA
  set language c              <= (1)
  ... do some stuff ...
  macroB
  ... do some stuff ...
  set language auto
end


define macroB
  set language c              <= (2)
  ... do some stuff ...
  set language auto           <= (3)
end
```

The above example will work like a charm in "c" mode, but look what happens in "ada" mode:

(1) You enter macroA, set the language to "c"

(2) You enter macroB, set the language to "c"

(3) You leave macroB, set the language back to "auto" (which is "ada").

Once back from macroB, the language is set to "ada". If the second part of macroA uses C-specific features, such as the comparison operator != (instead of /= for ada), the macro execution will fail.

For this reason, be sure to use the "c" language mode when executing macros, and always restore it to "auto" just before exiting a "scope".

Going further

Scripting is a powerful feature of the GNU debugger, and offers a significant aid when debugging programs using the command line. It can also be very useful when using GPS, since GPS sources the `.gdbinit` file, so that all predefined macros are available from the command-line interface of GPS. This allows one to combine the power of both textual and graphical debugging.

Related Source Code

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

Gem #121 Breakpoint Commands—Part 1

Author: Jerome Guitton, *AdaCore*

Let's get started…

Consider this basic model of a Bank:

```
package Bank is

   procedure Open_Account (Cash : Integer);

   procedure Deposit (Cash : Integer);

   procedure Withdraw (Cash : Integer);

end Bank;
```

This bank maintains only one account that has to be open before doing any transation. The only client of this bank would be a main subprogram P that opens an account and operates several transactions:

```
with Bank;

procedure P is
begin
   Bank.Open_Account (100);
   Bank.Deposit (45);
   Bank.Withdraw (55);
   Bank.Deposit (80);
end P;
```

The implementation of package Bank is straightforward; it also contains an additional transaction at elaboration time, that corresponds to a special offer of the bank (and one may already see that it will cause a problem):

```
package body Bank is

   Balance : Integer;

   procedure Open_Account (Cash : Integer) is

   begin
      Balance := Cash;
   end Open_Account;

   procedure Deposit (Cash : Integer) is
   begin
      Balance := Balance + Cash;
```

```
   end Deposit;

   procedure Withdraw (Cash : Integer) is

   begin
      Balance := Balance - Cash;
   end Withdraw;

   -- To celebrate the inauguration of the bank at elaboration time,
your
   -- account is credited with $100.

   Special_Offer : constant Integer := 100;

begin
   Deposit (Special_Offer);
end Bank;
```

There is a implicit property that the account should be opened only once, and before any transaction takes place; would this special offer break this property? This is a good opportunity to see how GDB can help us in checking properties of this form and break whenever the property is violated at run time. To do so, one may use the script language of GDB along with breakpoint commands.

The first way to do so would be set breakpoints on the bank operations:

```
(gdb) break open_account
Breakpoint 1 at 0x401592: file bank.adb, line 7.
(gdb) break deposit
Breakpoint 2 at 0x4015a2: file bank.adb, line 12.
(gdb) break withdraw
Breakpoint 3 at 0x4015b8: file bank.adb, line 17.
```

However, we would not like to stop on any transaction, but only on an invalid one. At each breakpoint, we would like to check if our invariants hold, continue if they do, stop with an error if they don't.

First, the properties that we want to check require to maintain a state: whether or not the account has been open. This may be represented by a GDB convenience variable that can be created as follows:

```
set variable $account_open := False
```

We can then attach a small program that will be executed each time the breakpoint is hit. e.g. for breakpoint 1 on open_account:

```
(gdb) commands 1
>if $account_open = True
 >echo error: account opened twice\n
 >else
 >set variable $account_open := True
 >printf "(info) account created with $%d\n", cash
 >continue
 >end
>end
```

This command will resume the program if the account has not been opened yet, and will stop with an error message if we are trying to open it for the second time. The same kind of program may be used for withdraw and deposit:

```
(gdb) commands 2
>if $account_open = False
 >echo error: someone tries to deposit before opening an account\n
 >else
 >printf "(info) deposit: $%d\n", cash
 >continue
 >end
>end
(gdb) commands 3
>if $account_open = False
 >echo error: someone tries to withdraw before opening an account\n
 >else
 >printf "(info) withdrawal: $%d\n", cash
 >continue
 >end
>end
```

Now let us run the program and look for invariant violations. One happens immediately:

```
(gdb) run
```

Breakpoint 2, bank.deposit (cash=100) at bank.adb:12 12 Balance := Balance + Cash; error: someone tries to deposit before opening an account (gdb)

Indeed, the special offer is credited before the account is opened:

```
(gdb) up
#1  0x004015da in  () at bank.adb:25
25         Deposit (Special_Offer);
```

Let us continue to find other violations. The program exits normally, meaning that no other violations were detected.

```
(gdb) c
Continuing.
```

Breakpoint 1, bank.open_account (cash=100) at bank.adb:7 7 Balance := Cash; (info) account created with $100

Breakpoint 2, bank.deposit (cash=45) at bank.adb:12 12 Balance := Balance + Cash; (info) deposit: $45

Breakpoint 3, bank.withdraw (cash=55) at bank.adb:17 17 Balance := Balance - Cash; (info) withdrawal: $55

Breakpoint 2, bank.deposit (cash=80) at bank.adb:12 12 Balance := Balance + Cash; (info) deposit: $80 [Inferior 1 (process 7472) exited normally]

The script language of GDB provides various flow-control commands; you may find their full documentation in the section "Command files" in the GDB user's guide. We have seen that this language of GDB, along with convenience variables and breakpoint commands, provides a simple way to check some simple dynamic properties on a run. In order to check more elaborated properties, the full power of expression of the python interface should be used. This will be the subject of a forthcoming Gem.

Related Source Code

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

Gem #122: Breakpoint Commands — Part 2

Author: Jerome Guitton, *AdaCore*

Let's get started…

We previously considered a bank that was managing only one account. Now suppose that this example has been extended to support several accounts and transactions between these accounts:

```
package Accounts is

   type Customer is (Alice, Bob);

   procedure Open_Account (C : Customer; Cash : Integer);

   procedure Pay (From, To : Customer; Cash : Integer);

end Accounts;
```

To check that no accounts are opened twice and that transactions only happen on opened accounts, we need to keep a record of the accounts that have been opened already. When we only had one account, the state was a simple Boolean variable; now it is a list of accounts, which is more complicated to express with only convenience variables. The Python interface of GDB can be used to bypass this limitation.

As a first step, let's describe the whole program. It contains a main procedure that opens two accounts and schedules a set of transactions between them:

```
with Accounts; use Accounts;

procedure P is
begin
   Open_Account (Alice, 50);
   Open_Account (Bob, 40);
   Pay (From => Bob,   To => Alice, Cash => 45);
   Pay (From => Alice, To => Bob,   Cash => 20);
   Pay (From => Bob,   To => Alice, Cash => 45);
   Pay (From => Alice, To => Bob,   Cash => 20);
end P;
```

As for the body of Accounts, it offers no difficulty. Oddly enough, it hides a transaction in its elaboration:

```
package body Accounts is

   type Account is limited record
      Balance : Integer;
   end record;

   Bank : array (Customer) of Account;
```

```
   procedure Open_Account (C : Customer; Cash : Integer) is
   begin
      Bank (C).Balance := Cash;
   end Open_Account;

   procedure Pay (From, To : Customer; Cash : Integer) is
   begin
      Bank (To).Balance := Bank (To).Balance + Cash;
      Bank (From).Balance := Bank (From).Balance - Cash;
   end Pay;

   --  At elaboration time, a suspicious operation is scheduled;
   --  Alice secretly pays a bribe to Bob:

begin
   Pay (From => Alice, To => Bob, Cash => 20);
end Accounts;
```

We will now implement, in Python, two hooks that should be executed whenever an operation on an account is scheduled. Create a new file named hooks.py with the following code:

```
current_customers = []

def open_account_hook():
    global current_customers
    c = str(gdb.parse_and_eval('c'))
    if c in current_customers:
        print "error: account initialized twice"
    else:
        print "(info) %s opens an account" % c
        current_customers.append(c)
        gdb.execute("continue")

def pay_hook():
    global current_customers
    f = str(gdb.parse_and_eval('from'))
    t = str(gdb.parse_and_eval('to'))
    if not f in current_customers:
        print "error: %s tries to pay before opening an account" % f
    elif not t in current_customers:
        print "error: %s cannot be paid before opening an account" % t
    else:
        cash = str(gdb.parse_and_eval('cash'))
        print "(info) %s gives %s $%s" % (f, t, cash)
        gdb.execute("continue")
```

This defines three entities: a global list that records the accounts that have been opened already, and two hooks to be executed when Open_Account and Pay are called.

Both get the value of the parameters using the function gdb.parse_and_eval (defined in the Python API) to check that the operation is valid. The documentation of this function,

as well as the documentation of any Python entities, can be accessed from GDB using Python's command-line help:

```
(gdb) python help(gdb.parse_and_eval)
Help on built-in function parse_and_eval in module gdb:
parse_and_eval(...)
    parse_and_eval (String) -> Value.
    Parse String as an expression, evaluate it, and return the result
as a Value.
```

Now, we just need to register these commands in GDB using the source command and attach them to their corresponding breakpoints.

```
(gdb) source hooks.py
(gdb) break open_account
Breakpoint 1 at 0x4015b4: file accounts.adb, line 11.
(gdb) commands
>python open_account_hook()
>end
(gdb) break pay
Breakpoint 2 at 0x4015f6: file accounts.adb, line 16.
(gdb) commands
>python pay_hook()
>end
```

This detects that the first transaction is indeed invalid, as it happens before the accounts are opened:

```
(gdb) run
Starting program: C:\home\guitton\GIT\GDB\builds\gems\3\p.exe
[New Thread 1872.0x638]
Breakpoint 2, accounts.pay (from=alice, to=bob, cash=20) at
accounts.adb:16
16            Bank (To).Balance := Bank (To).Balance + Cash;
error: alice tries to pay before opening an account
```

The Python API provides an advanced programming interface for GDB, with facilities for browsing through backtraces, threads, symbols, etc. For more information, we invite you to consult the dedicated section on those features in the GDB user's guide.

Related Source Code

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

Gem #123: Implicit Dereferencing in Ada 2012

Author: Christoph Grein

Let's get started…

In Gem #107, we presented an accessor for safely referencing objects stored in a container. The example concerned a reference-counted pointer, but such accessors can be defined on any kind of container.

An advantage of using accessors rather than simple access types is that the former cannot be used to deallocate the designated object. However, safety always comes with a cost, in this case in the form of awkward syntax. In this Gem, we show how some features of Ada 2012 can be used to simplify the syntax.

Consider a prototypical container as an example:

```
generic
  type Element is private;
  type Key     is private;
  with function Key_of (E: Element) return Key;
package Containers is

  type Container is private;

  type Accessor (Data: not null access Element) is limited private;

  procedure Put (C: in out Container; E: in Element);

  function  Get (C: Container; K: Key) return Accessor;

private

  ... implementation not shown
end Containers;
```

The container holds elements that can be retrieved via some kind of key. How elements are stored and retrieved is not of interest here. What is important is that Get grants direct access to the stored element (in other words, Get does not return a copy). This is crucial if you have a big object and only want to update a component:

```
Get (Cont, My_Key).Data.Component := Some_Value;
```

Note that if the discriminant were to be defined as

```
not null access constant Element
```

, then the accessor would allow only read access. Also, the null exclusion guarantees that an accessor will always reference an object.

This syntax is quite verbose, but Ada 2012 provides a new feature that helps simplify it, namely the Implicit_Dereference aspect.

Side Note: Ada 2012 has added a general mechanism called an *aspect specification* that allows defining various characteristics of declarations, called *aspects*, as part of the declaration itself. For example, representation attributes can now be specified by using an aspect specification rather than a separate attribute definition.

Here is how the Implicit_Deference aspect would be specified for our Accessor type:

```
type Accessor (Data: not null access Element) is limited private
    with Implicit_Dereference => Data;
```

A type defined with this aspect is called a reference type, and an object of such a type is a reference object. The use of this aspect allows us to reduce the statement to:

```
Get (Cont, My_Key).Component := Some_Value;
```

Note that the call Get (Cont, My_Key) is overloaded: its result can be interpreted as either an accessor value or the accessed object itself, and the compiler resolves this based on the context of the call. (This is the reason the Implicit_Dereference aspect cannot be used on the reference-counted pointer in Gem #107, where a type conversion is needed, because the argument of a type conversion must be resolved independently of the context.)

You might argue that this is not a significant simplification. However, this is not the end of the story. We're not interested so much in how to get an accessor value (the result of function Get) as we are in getting to the elements themselves. It happens that we can elide the call to Get by means of another aspect, called Variable_Indexing, that's applied to the Container type:

```
type Container is tagged private
    with Variable_Indexing => Get;
```

The result type of the Variable_Indexing function must be a reference type. It's worth noting that the name given in an aspect specification may denote something declared later. In this case it's a forward reference to Get, which is declared after the type.

Also, the type to which the Variable_Indexing attribute is applied must be tagged. Being a tagged type, this allows the Object.Operation notation, leading to:

```
Cont.Get (My_Key).Component := Some_Value;
```

Given the Variable_Indexing aspect that specifies Get, this can now be further reduced to simply:

```
Cont (My_Key).Component := Some_Value;
```

Effectively what we get is direct access to container elements, as though the container were a kind of array indexed by the key. In fact, it's possible to use the indexed name alone in a context requiring a variable of the element type, such as an assignment statement:

```
Cont (My_Key) := Some_Element_Value;
```

So, by combining the new aspects Implicit_Dereference and Variable_Indexing we get a concise and much more readable syntax for manipulating container elements.

Incidentally, there's also a companion aspect to Variable_Indexing called Constant_Indexing, that can be used to grant read-only access to element values. In that case, the associated function is not required to return a reference type, because all functions return a constant result.

Related Source Code

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

Gem #124 : Scripting GPS for Static Analysis

Authors: Yannick Moy and Nicolas Setton, *AdaCore*

Let's get started…

Being able to quickly query a code base for some simple properties is a valuable capability both during development and debugging. Most of the time one can get by with coarse grep-like queries, or maybe a simple script based on syntactic queries. For more complex queries that require semantic knowledge, one must resort to using existing tools, which might not always be a perfect fit.

This Gem provides a way to perform these queries using the Python scripting capabilities of GPS, the GNAT Programming Studio IDE.

As an example, consider a critical section whose API defines the procedures Enter and Leave:

```
package Critical_Section is

    procedure Enter;
    --  Enter the critical section

    procedure Leave;
    --  Leave the critical section.
    --  Important: every procedure calling Enter should also call
Leave.

end Critical_Section;
```

For purposes of illustration, their implementation can be as simple as:

```
package body Critical_Section is

    In_Critical : Boolean := False;

    procedure Enter is
    begin
       In_Critical := True;
    end Enter;

    procedure Leave is
    begin
       In_Critical := False;
    end Leave;

end Critical_Section;
```

These critical sections may be used in a main program, in ways that are more or less correct:

```
with Critical_Section; use Critical_Section;

procedure Main is

   procedure A is
   begin
      Enter; -- Do some processing Leave;
   end A;

   procedure B is
   begin
      Enter; -- Do some processing
   end B;

begin
   A;
   B;
end Main;
```

Ideally, we would like every subprogram that enters the critical section to also leave it. This requires knowledge of the call graph of the application, not something that a grep-like approach can solve, and you would be unlikely to have a tool that already computes this information for you.

Well, this is very easy to do in GPS!

Open GPS on the project and compile the code with GNAT. This generates cross-reference information, saved in the .ali files, containing in particular references to all entities in the code. The Python API in GPS allows one to query this information, know where each entity is referenced, determine an entity's kind, etc.

Open a Python shell with Window->Python, and enter the following lines in the shell (except comments introduced by #):

```
# get the entities corresponding to the functions
enter_entity = GPS.Entity("Enter", GPS.File("critical_section.ads"))
leave_entity = GPS.Entity("Leave", GPS.File("critical_section.ads"))

# get the list of calls for each function

enter_called_by = enter_entity.called_by()
leave_called_by = leave_entity.called_by()

# print the result
print ["Enter is called by %s, which does not call Leave" % fun for fun
in enter_called_by if not fun in leave_called_by]
```

This should print the following result:

```
['Enter is called by B:main.adb:12:14, which does not call Leave']
```

That's the information we were looking for!

Now, you may find this kind of query common enough that you would like to have a simple button to press in GPS to compute this result. The plug-in capabilities of GPS allow you to register the code above, so that the user will be prompted for the two subprograms Enter and Leave. The plug-in itself is simply:

```
""" Given the names of two subprograms Enter and Leave defining a
critical
     section, this plug-in detects subprograms which only enter the
critical
     section without leaving it.
"""

import GPS

def detect_critical_section_violation(self):
     enter_name, enter_file, leave_name, leave_file=
GPS.MDI.input_dialog("Enter the names of subprograms defining the
critical section:", "Enter", "File containing Enter", "Leave", "File
containing Leave")

     # get the entities corresponding to the functions
     enter_entity = GPS.Entity(enter_name, GPS.File(enter_file))
     leave_entity = GPS.Entity(leave_name, GPS.File(leave_file))

     # get the list of calls for each function

     enter_called_by = enter_entity.called_by()
     leave_called_by = leave_entity.called_by()

     # compute the violations
     violations = [fun for fun in enter_called_by if not fun in
leave_called_by]
     if violations == []:
         GPS.MDI.dialog ("No violation found")
     else:
         GPS.MDI.dialog(str(["Enter is called by %s, which does not
call Leave" % fun for fun in violations]))

def on_gps_started (hook_name):
     menu = GPS.Menu.create("/Tools/Browsers/Detect critical section
violation", detect_critical_section_violation)

GPS.Hook ("gps_started").add (on_gps_started)
```

Put this code in a file critical_section.py under either one of INSTALL/share/gps/plug-ins or HOME/.gps/plug-ins and the menu will be there the next time you open GPS.

For more information on how to use the Python API in GPS, we refer you to section 16 of the GPS User's Guide (Customizing and Extending GPS) and the Python API reachable under Help->Python extensions in GPS.

Related Source Code

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

Gem #125: Detecting infinite recursion with GDB's Python API

Author: AdaCore Jerome Guitton, *AdaCore*

Let's get started…

Suppose that you are printing a table of factorials from 9 down to 0, as follows:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Try is

   function Factorial (I : Integer) return Integer is
   begin
      if I = 0 then
         return 1;
      else
         return I * Factorial (I - 1);
      end if;
   end Factorial;

   N : Integer := 9;
   F : Integer;

begin
   loop
      F := Factorial (N);

      Put (Integer'Image (N));
      Put ("! = ");
      Put (Integer'Image (F));
      New_Line;

      exit when N < 0;
      N := N - 1;
   end loop;
end Try;
```

Now build the example:

```
gnatmake -g try.adb
```

When you execute this program, it will either crash with Storage_Error, or SIGSEGV, or hang, or corrupt the memory, depending on the target platform:

```
$ ./try
 9! =  362880
 8! =  40320
 7! =  5040
 6! =  720
 5! =  120
 4! =  24
 3! =  6
```

```
 2! =  2
 1! =  1
 0! =  1


raised STORAGE_ERROR : EXCEPTION_STACK_OVERFLOW
```

If you run the program in GDB, by using 'catch exception' you can detect infinite recursion in procedure Factorial:

```
(gdb) catch exception
Catchpoint 1: all Ada exceptions
(gdb) run
Starting program: C:\home\guitton\GIT\GDB\builds\gems\1\try.exe
[New Thread 12736.0x33a4]
 9! =  362880
 8! =  40320
 7! =  5040
 6! =  720
 5! =  120
 4! =  24
 3! =  6
 2! =  2
 1! =  1
 0! =  1

Program received signal SIGSEGV, Segmentation fault.
0x00401797 in try.factorial (i=-698787) at try.adb:9
9                return I * factorial (I - 1);
(gdb) backtrace 10
#0  0x00401797 in try.factorial (i=-698787) at try.adb:9
#1  0x0040179c in try.factorial (i=-698786) at try.adb:9
#2  0x0040179c in try.factorial (i=-698785) at try.adb:9
#3  0x0040179c in try.factorial (i=-698784) at try.adb:9
#4  0x0040179c in try.factorial (i=-698783) at try.adb:9
#5  0x0040179c in try.factorial (i=-698782) at try.adb:9
#6  0x0040179c in try.factorial (i=-698781) at try.adb:9
#7  0x0040179c in try.factorial (i=-698780) at try.adb:9
#8  0x0040179c in try.factorial (i=-698779) at try.adb:9
#9  0x0040179c in try.factorial (i=-698778) at try.adb:9
(More stack frames follow...)
```

But now you would like to know the context of the call to Factorial that causes the recursion. This can't easily be done at the exception point, as the debugger has too many frames (698787, presumably) to unwind to get to the original call.

If you add a breakpoint in Factorial, you would do several continue commands without finding anything suspicious:

```
(gdb) b factorial
Breakpoint 1 at 0x40177f: file try.adb, line 6.
(gdb) run
```

```
Starting program: C:\home\guitton\GIT\GDB\builds\gems\1\try.exe
[New Thread 11856.0x23b4]

Breakpoint 1, try.factorial (i=9) at try.adb:6
6              if I = 0 then

(gdb) cont
Continuing.

Breakpoint 1, try.factorial (i=8) at try.adb:6
6              if I = 0 then
(gdb) cont
Continuing.

Breakpoint 1, try.factorial (i=7) at try.adb:6
6              if I = 0 then
```

There are unfortunately a large number of valid calls to Factorial before the bogus one. What we would like to do is set an upper bound on the possible calls to Factorial, and stop only when that bound is reached.

Fortunately, the Python API of GDB can help: with this API, you can go through the debugger structures and get more information about the context at a point of execution. For example, here is a Python function that counts the number of frames in a backtrace:

```
def frame_count():
    """Count the number of frames in the backtrace"""
    c = 0
    f = gdb.newest_frame()
    while f is not None:
        c = c + 1
        f = f.older()
    return c
```

The function uses the class Frame from the Python API of GDB, and its method older()/newer() to browse through the backtrace.

After creating a file frame_count.py that contains the implementation of this command, it can then be added to the commands of GDB and used as follows:

```
(gdb) source frame_count.py
(gdb) python help(frame_count)
Help on function frame_count in module __main__:

frame_count()
    Count the number of frames in the backtrace

(gdb) python print frame_count()
4
```

Now you can set a breakpoint in Factorial that will stop only when frame_count reaches a maximum bound. Edit frame_count.py to add the following breakpoint hook:

```
def factorial_hook():
    """Called whenever Try.Factorial is reached"""
    max_number_of_frames = 100
    fc = frame_count()
    if fc <= max_number_of_frames:
        gdb.execute("continue")
    else:
        print ""

        print ("warning: more than %d frames in backtrace."
               % max_number_of_frames)
        print "         To ease the investigation, the selected frame
will be"
        print "         the first call to factorial."
        print ""

        f = gdb.newest_frame()
        while f is not None and f.name() == "try.factorial":
            gdb.Frame.select(f)
            f = f.older()
        # Finally, show the frame that this loop selected; it is the
first
        # call to factorial
        gdb.execute("frame")
```

Then attach this command to a breakpoint in Try.Factorial:

```
(gdb) source frame_count.py
(gdb) break factorial
Breakpoint 1 at 0x40177f: file try.adb, line 6.
(gdb) commands
>python factorial_hook()
>end
```

You can then run, and the program will stop when the maximum bound is reached. At that point you'll be able to investigate the infinite recursion much more easily:

```
(gdb) run
Breakpoint 1, try.factorial (i=-100) at try.adb:6
6             if I = 0 then

warning: more than 100 frames in backtrace.
         To ease the investigation, the selected frame will be
         the first call to factorial.

#99 0x0040179c in try.factorial (i=-1) at try.adb:9
9                return I * Factorial (I - 1);
```

We can now see that Try calls Factorial with -1; there is a bug in the exit condition in the loop.

```
(gdb) up
#100 0x004017c1 in try () at try.adb:17
17            F := Factorial (N);
(gdb) print N
$2 = -1
```

This is just one of the advanced debugging capabilities that the Python API of GDB provides. For more information, have a look at the corresponding section in the GDB user's guide.

Related Source Code

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

Gem #126 : Aggregate Library Projects

Author: Pascal Obry, *EDF R&D*

Let's get started…

A common scheme for building large Ada applications is to create multiple modules. Each module (or subsystem) comes with a project file, not only to support building it, but also to facilitate editing the code via GPS. This has the advantage of permitting developers to focus only on the code for the module of interest. On the whole, this decreases the complexity of an application by modularizing it.

Each module comes with a library project used for building a library (shared or static) containing the object code for the module. For an application, the final step is just linking together the libraries for all modules. So far, so good.

However, if the goal is not to build an application, but rather to deliver a library (containing API sources and objects), this approach is not really convenient. Indeed, all libraries have to be copied into the installation directory, and a project file for each library must be distributed for the final application to satisfy the required dependencies. This is cumbersome and somewhat tedious, and it is even more tedious when the library is to be used in an application built with C or C++, because we impose the complexity on the client of the API by requiring linking with multiple libraries.

In such cases it would be far more convenient to distribute a single library containing all the code.

Perhaps you're now thinking that it's a bad idea to create modules and develop big applications using a flat project structure? You're right, and that's where the aggregate library projects come into play.

Aggregate library projects enable the possibility of creating a single library (shared or static) out of a set of libraries or projects.

For example, let's assume the project contains two libraries:

```
library project MyLib1 is
   for Source_Dirs use ("src1");
   for Object_Dir use "obj1";
   for Library_Name use "mylib1";
   for Library_Kind use "static";
   for Library_Dir use "lib1";
end MyLib1;
```

```
library project MyLib2 is
   for Source_Dirs use ("src2");
   for Object_Dir use "obj2";
   for Library_Name use "mylib2";
   for Library_Kind use "static";
   for Library_Dir use "lib2";
end MyLib2;
```

Project mylib1.gpr and mylib2.gpr can be used separately to create mylib1.a and mylib2.a.

But by using an aggregate library it's possible to create a single library (say fulllib.a) combining both of the projects:

```
aggregate library project MyLib is
   for Project_Files use ("mylib1.gpr", "mylib2.gpr");
   for Library_Name use "fulllib";
   for Library_Kind use "static";
   for Library_Dir use ".";
end MyLib;
```

Note that an aggregate library project description is very close to a library project. The only new required attribute is Project_Files, which must list all of the projects that will be aggregated.

Note also that there's no need for an Object_Dir attribute, as aggregate libraries do not have object code by themselves.

The following command will build pck1.o and pck2.o (from src1/pck1.ads and src2/pck2.ads) using their respective projects, and will then aggregate the resulting object code in a single library named "libfulllib.a":

```
$ gprbuild -gnat05 -p mylib.gpr
```

The contents of the aggregate library can be verified with:

```
$ nm libfulllib.a
```

And, of course, it's also possible to do the same with shared libraries.

In a forthcoming installment we'll discuss encapsulated libraries, which are quite useful for multilanguage development.

Related Source Code

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

Gem #127: Iterators in Ada 2012 - Part 1

Author: Emmanuel Briot, *AdaCore*

Let's get started…

The following examples assume we have instantiated an Ada list such as:

```
with Ada.Containers.Doubly_Linked_Lists;

...

declare
   package Integer_Lists is
      new Ada.Containers.Doubly_Linked_Lists (Integer);
   use Integer_Lists;

   L : Integer_Lists.List;

begin
   L.Append (10);
   L.Append (20);
end;
```

In Ada 2005, an iteration over this list would look like:

```
declare
   C : Integer_Lists.Cursor;
begin
   C := First (L);
   while Has_Element (C) loop
      --  Print current value
      Put_Line (Integer'Image (Element (C)));

      --  Change the element in place in the list
      Replace_Element (L, C, Element (C) + 1);

      Next (C);
   end loop;
end;
```

If the list contains elements more complex than integers (controlled types for instance), the above code is not very efficient, since a call to function Element will return a copy of the element. To avoid a copy, one could use a nested subprogram and the procedures Query_Element and Update_Element, but that would make the code more complex and less readable.

Ada 2012 defines three forms of iterators. The first form is called a **generalized iterator**. The syntax and semantics for it is given in the Ada 2012 Reference Manual (5.5.2), but here is an example of its use:

```
   for C in L.Iterate loop
      Put_Line (Integer'Image (Element (C)));
      Replace_Element (L, C, Element (C) + 1);
   end loop;
```

In this code, C is also a cursor just like before, but it takes on values returned by the Iterate function defined in the list instantiation. This code is basically hiding the calls to First, Has_Element, and Next.

The second form of iterator, called a **container element iterator**, is even more user friendly and hides the use of cursors completely. This form of iterator gives direct access to the elements of a container (see Annex A of the Ada RM for specifications of the predefined containers):

```
   for E of L loop    -- Note "of" instead of "in" here
      Put_Line (Integer'Image (E));
      E := E + 1;
   end loop;
```

The third form of iterator, called an **array component iterator**, is similar to a container element iterator, but applies to array types. Here is an example of this form:

```
declare
   Arr : array (1 .. 2) of Integer := (1 => 10, 2 => 20);

begin
   for E of Arr loop
       Put_Line (Integer'Image (E));
       E := E + 1; -- Change in place
   end loop;
end;
```

As the example shows, we can even modify the iterator element E directly, and this modifies the value in the list itself. This is also efficient code when the list contains complex types, since E is not a copy of an element, but a reference to it.

The second part of this Gem series will explain how to write your own iterators and how the loops shown above are expanded by the compiler.

Related Source Code

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

Gem #128 : Iterators in Ada 2012 - Part 2

Author: Emmanuel Briot, *AdaCore*

Let's get started…

In Part 1, we discussed the basic forms of iterators in Ada 2012 and gave some simple examples. This part goes into greater detail, showing how to create iterators for your own data structures. We'll start by learning about two supporting features introduced in Ada 2012.

The first is the new generic package Ada.Iterator_Interfaces. This package defines two abstract types Forward_Iterator and Reverse_Iterator. The intent is that each container should declare extensions of these and provide concrete implementations for their primitive operations. Briefly, an iterator encapsulates a cursor and a container, and hides the First, Has_Element, and Next operations.

The second new feature is that of a *reference type*. A reference type is a record with an access discriminant that defines the "Implicit_Dereference" aspect. This is the actual type manipulated by the container element iterators, and the aspect eliminates the need to write ".all" every time an element is referenced.

Here is an example of such a declaration, taken from the standard package Ada.Containers.Doubly_Linked_Lists:

```
type Constant_Reference_Type
       (Element : not null access constant Element_Type)
          is private with Implicit_Dereference => Element;
```

Whenever we have such a reference, for example E of type Constant_Reference_Type, we can just use the name "E", and this is automatically interpreted as "E.Element.all". Another advantage of this type over a simple access to element is that it ensures the user cannot accidentally free the element.

Now that we understand what iterators and references are, we can start applying them to our own data structures.

Let's assume we are creating our own data structure (such as a graph, a queue, or anything that is not a direct instantiation of an Ada 2005 container). The following examples are framed as a "list", but this really applies to any data structure. Let's also assume that the container holds unconstrained elements of type "T'Class", giving us a more realistic and interesting example than the Part 1 example that just contained Integers.

To provide iterators for this data structure, we need to define a number of Ada 2012 aspects, described in more detail below.

```
type T is tagged null record;  -- any type

type T_List is ...  -- a structure of such types
   with Default_Iterator  => Iterate,
        Iterator_Element  => T'Class,
        Constant_Indexing => Element_Value;

type Cursor is private;
function Has_Element (Pos : Cursor) return Boolean;
-- As for Ada 2005 containers

package List_Iterators is
   new Ada.Iterator_Interfaces (Cursor, Has_Element);

function Iterate (Container : T_List)
   return List_Iterators.Forward_Iterator'Class;
-- Returns our own iterator, which in general will be defined in the
-- private part or the body.

function Element_Value (Container : T_List; Pos : Cursor)
   return T'Class;
-- Could also return a reference type as defined in the Part 1 Gem
```

For those unfamiliar with aspects in Ada 2012, it's worth noting that they can be forward references: in the case above, for instance, the aspect "Default_Iterator" is defined before Iterate is declared (and we could not declare it first in any case, since the function Iterate needs to know about T_List).

To understand the aspects, let's look at how the generalized iterators are expanded by the compiler.

This loop:

```
for C in List.Iterate loop    -- C is a cursor
   declare
      E : T'Class := Element (C);
   begin
      ...
   end;
end loop;
```

is expanded into:

```
 declare
    Iter : Forward_Iterator'Class :=
       List.Iterate;           -- Default_Iterator aspect
    C : Cursor := Iter.First;  -- Primitive operation of iterator
 begin
    while Has_Element (C) loop -- From Iterator_Interfaces instance
       declare
          E : T'Class := List.Element_Value (C);
                                -- Constant_Indexing aspect
       begin
```

```
            ...
         end;
         C := Iter.Next (C);    -- Primitive operation of iterator
      end loop;
   end;
```

The subprogram Iterate, referenced in the "Default_Iterator" aspect, creates and returns a new iterator. In general, it will also hold a reference to the container itself to ensure the container lives at least as long as the iterator.

The iterator is then used to get and manipulate a cursor. Retrieving an element from the cursor is done via the function defined in the "Constant_Indexing" aspect. (A similar aspect "Variable_Indexing" is used when the loop needs to write the element, but we will not demonstrate that here.)

The function Element_Value is written here in its simplest form: it directly returns a copy of the element contained in the data structure. We could choose instead to return a reference type as explained in the Part 1 Gem, to avoid copies of the elements. (Note that in the case of Variable_Indexing, the function's result type must be a reference type.)

The container element iterators are expanded similarly. The only difference is that the cursor C is not visible.

For the actual implementation of Iterate and Element_Value, we recommend looking at the implementation for the standard containers, such as Doubly_linked_Lists. All of the Ada 2005 containers were enhanced to support iterators, and these provide various examples of code that can be reused for your own applications.

Finally, let's look at a code pattern that might be useful. The test case is the following: we have implemented a complex data structure that contains elements of type T'Class. When we use the container element iterators, E is thus of type T'Class, which we can express with the following syntax:

```
for E : T'Class of List loop
   ...
end loop;
```

Now let's consider a type TChild that extends T. We can still store elements of type TChild in the data structure, but we then need explicit conversions in the loop above to cast E to TChild'Class. We would like to minimize the amount of code needed to create a container that holds TChild'Class elements. For instance:

```
type TChild_List is <see full type below>;

Child_List : TChild_List;
```

```
for E of Child_List loop
   -- E is of type TChild'Class, so no conversion is needed.
end loop;
```

Of course, one possibility is to make our container generic and instantiate it once for T'Class, once for TChild'Class, and so on. That's certainly a minimal amount of Ada source code, but it can still represent a significant amount of compiled code and will increase the size of the final executable. In fact, we can simply mirror the T / TChild hierarchy in the containers themselves and redefine only a minimal number of aspects to achieve the goal.

```
type TChild_List is new T_List with null record
   with Constant_Indexing => Child_Value,
        Default_Iterator  => Iterate,  -- inherited from T_List
        Iterator_Element  => TChild'Class;

function Child_Value (Self : TChild_List; Pos : Cursor'Class);
   return TChild'Class is
begin
   return TChild'Class (Element_Value (Self, Pos));
end Child_Value;
```

The amount of additional code is minimal (just one extra function, which is likely to be inlined), and now we can write the container element loop with no need for conversions. Since the containers themselves are now organized as a hierarchy, we can have subprograms that work on a T_List that also work on a TChild_List (the usual reuse of object-oriented code).

However, the new structure is not perfect. One caveat is that it's possible to insert an object of type T in a TChild_List (because the list contains T'Class elements). The consequence is that the iterator will raise Constraint_Error in the implicit call to Child_Value in the expanded code.

We hope that this Gem has helped to explain some of the "magic" behind the Ada 2012 iterators and containers, and will enable you to use them more effectively in your own code. Even though they do require quite a lot of boilerplate code, written once up front for a container, they definitely make code in clients of the container easier to read and understand.

One final note: the examples in this Gem require a fairly recent version of the compiler, which includes a number of adjustments to reflect recent clarifications in the Ada 2012 rules.

Related Source Code

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

Gem #129 : Type-Safe Database API - Part 1

Author: Emmanuel Briot, *AdaCore*

Let's get started…

SQL is the traditional language used to query a database management system (DBMS). This language is mostly standardized, even though each vendor provides their own extensions and restrictions. It is convenient when the data could be organized into tables and fields following the relational model. More recently, we have seen the emergence of so-called "noSQL" databases, which use a different paradigm altogether to provide enhanced performance in exchange for additional restrictions. We will not discuss noSQL databases in this Gem.

Despite all its advantages, there are lots of limitations with SQL as it is traditionally used.

* The query is written as a string.

With a string it isn't possible to check at compilation time whether the syntax is correct. It's also not possible to check whether the tables and fields referenced are valid. Furthermore, it isn't possible to have type checking (ensuring comparison of integers with integers, etc.).

* Sending the query to the DBMS is vendor specific.

Each vendor provides a C API, but they all differ from one another. The concepts are in general similar: preparing the query; sending it to the server; passing the parameters; fetching the results; and so on. A standard API exists (ODBC) which is implemented by most, but not all, vendors. However, there is a performance penalty compared to using the native API in general. The setup is also more complex.

Some vendors provide preprocessor-based solutions: you write the SQL query directly in your source code, which is then preprocessed before being passed to the compiler. However, this solution means that the compiler is not seeing exactly the code you have written, which for instance can result in error messages pointing to the wrong line in the file.

The GNAT Components Collection proposes two packages, GNATCOLL.SQL and GNATCOLL.SQL.Exec, which provide solutions to the problems highlighted above. One of the requirements is that these packages should be usable with existing code to provide an easier transition.

We'll begin by addressing the first limitation (SQL as strings). The package GNATCOLL.SQL provides an Ada API to write SQL queries. However, for it to provide type safety, we first need to describe the schema of our database. For this, GNATColl

relies on a command-line tool called gnatcoll_db2ada. If you already have an existing and running database, you can simply run the following command:

```
gnatcoll_db2ada -dbname <name> -dbhost <host> -dbuser <user>
```

This command will generate two Ada packages (by default, they are called Database and Database_Names). Only the former is meant to be used directly in your applications. It contains Ada packages that match the database schema, for instance:

```
package Database is
   type T_Table1 (...) is record
      Field1 : SQL_Field_Integer (...);
      Field2 : SQL_Field_Text (...);
   end record;

   type T_Table2 (...) is record
      Field3 : SQL_Field_Integer (...);
   end T_Table2;

   Table1 : T_Table1 (null);
   Table2 : T_Table2 (null);

   ...
end Database;
```

The actual discriminants are irrelevant in our discussion of the basics of GNATCOLL.SQL. They are used when you need to have multiple references to the same table (through different aliases) in the same query.

The names T_Table1, Field1, Field2, etc. are the actual names of the tables and fields in the schema, so should be familiar to the developers.

If you don't have an existing database, it's possible to describe the schema in a text file. This file can be created initially from an existing database. Here is an example of such a file. Full documentation for the format is available in the GNATColl documentation.

```
| TABLE  | Table1         |          | |
| Field1 | AUTOINCREMENT | PK       | | Documentation for this field
| Field2 | Text          | NOT NULL | | Documentation for field2

| TABLE  | Table2     | | |
| Field3 | FK(Table1) | | | This is a foreign key
```

Once the Ada packages have been generated, the user can write SQL queries by using the Ada API provided in GNATCOLL.SQL. Here is a small example:

```
with GNATCOLL.SQL;   use GNATCOLL.SQL;
with Database;       use Database;   --  the generated package

...
```

```
declare
   Q : SQL_Query;
begin
   Q := SQL_Select
      (Fields => Table1.Field2 & Table2.Field3,   --  line 8
       From   => Table1 & Table2,                 --  line 9
       Where  => Table1.Field1 = Table2.Field3);  --  line 10

   --  Converting Q to a string via To_String will now display:
   --     "SELECT table1.field2, table2.field3 FROM table1, table2
   --      WHERE table1.field1 = table2.field3);"
end;
```

Although it might appear to be longer to type than when using the string, this code already has several advantages:

* It is pure Ada.

There is no need for preprocessing, so any error message the compiler reports will point to the correct line of code.

* It guarantees that the SQL is syntactically correct.

By design, the API prevents you from entering a syntactically incorrect query (for instance, a typo like "SELCET", or missing spaces, or any other kind of error). The API makes sure you can only specify a table or a list of tables in the FROM clause, and there's a similar limitation for the FIELDS clause.

* It guarantees that only valid tables and fields are referenced.

Since the query is written using the generated API, this ensures that if you change the schema of your database, the generated Database package will also change, and the compiler will then flag queries that reference fields that no longer exist. When the query is written as a string, changing the schema requires searching in the source for possible impacts, rather than relying on the compiler to do this work on our behalf.

* It checks types at compilation time.

If we modify line 10 to compare Table1.Field2 and Table2.Field3, the compiler will complain that we are comparing an integer and a text field.

Because the queries are built as a tree which is then flattened into a string, GNATCOLL.SQL is also able to provide automatic completion on queries. For instance, we could omit line 9 above, and add:

```
  Auto_Complete (Q);
```

In this case there is no gain, and we are actually losing in readability and efficiency. But this auto-completion capability is especially useful for completing the GROUP BY clauses. For instance, when one of the fields returned by the query is the result of an aggregation function, and you also return ten other fields, the auto-completion removes the need to manually maintain the list of nine fields in GROUP BY.

Of course, there is a performance penalty implied by the creation of the query in memory, and then its serialization as a string. To alleviate that, GNATCOLL.SQL provides support for prepared queries (client or server side), as well as parameterized queries. The second Gem in this series goes into more detail on this, and also describes the package GNATCOLL.SQL.Exec, which abstracts communication with the DBMS engine to a vendor-agnostic API.

To be continued in part 2 ...

Related Source Code

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

Gem #130 : Type-Safe Database API - Part 2

Author: Emmanuel Briot, *AdaCore*

Let's get started…

The first Gem in this series discussed how to write syntactically correct and type-safe SQL queries. We now need to execute these queries on the DBMS of choice and retrieve the results. This Gem explains how to use the DBMS-agnostic API in GNATColl to do this.

Currently, GNATColl provides support for two DBMS: PostgreSQL and SQLite. It is extensible to other DBMS by overriding certain primitive operations (see the documentation in gnatcoll-sql-exec.ads for more information).

When you compile GNATColl, it automatically detects whether any of the supported DBMS are installed on your system and optionally compiles the support for them. But, of course, when you link your application with GNATColl you do not want to systematically depend on every DBMS that is recognized by GNATColl (for instance both PostgreSQL and SQLite), just the one you actually need. Using project files, you can let the linker know which DBMS you need. Here is an example of a project file that adds support for SQLite:

```
with "gnatcoll_sqlite";
project Default is
   for Main use ("main.adb");
end Default;
```

In the Ada code, we now need to connect to the actual DBMS. This is the only place in the code that will explicitly mention which DBMS we are using. So porting your application from SQLite to PostgreSQL is merely a matter of changing this code. The rest of the application is unaffected.

```
with GNATCOLL.SQL.Sqlite;   -- or PostgreSQL
declare
   DB_Descr : GNATCOLL.SQL.Exec.Database_Description :=
      new GNATCOLL.SQL.Sqlite.Setup ("dbname.db");
   DB : GNATCOLL.SQL.Exec.Database_Connection;
begin
   ...
end;
```

The parameters to Setup depend on the DBMS. For instance, PostgreSQL lets you specify the host, username, and password.

DB_Descr is just a description of the future connections. To get an actual connection, there are a number of possibilities:

1. If your application is not multi-tasking, you can use the following:

```
DB := DB_Descr.Build_Connection;
```

2. If, however, you are using multitasking, it might be better to always retrieve the same connection from within a given task. For instance, a web server generally uses one task per HTTP query, and we can retrieve the task-specific connection with:

```
DB := GNATCOLL.SQL.Exec.Get_Task_Connection (DB_Descr);
```

3. Finally, it is possible to have a pool of such connections, which will be explained in the third part of this Gem series.

At this point, we still haven't connected to the DBMS, but this is done automatically the first time we execute a query. If for some reason the connection was broken (say, due to a network error), GNATColl automatically tries to reconnect a number of times before giving up.

Let's now execute our first query:

```
declare
   Q : constant SQL_Query := ... ;   -- See first Gem in the series
   R : Forward_Cursor;
begin
   R.Fetch (Connection => DB, Query => Q);
end;
```

R will contain the actual result. There are two possible types for R: a Forward_Cursor can only be iterated one row at a time; when you move to the next row, the previous one is lost for good. In exchange for this limitation, GNATCOLL.SQL does not have to retrieve all results in memory at once, which can be more efficient if you intend to stop the iteration early. The second type is a Direct_Cursor, which retrieves all rows at once, keeps them in memory, and lets you traverse them forward or backward, or even jump to a specific row directly. This is more flexible, but requires more memory in your application.

Printing the results can be done using the familiar cursor-based loop idiom:

```
   while Has_Row (R) loop
      Put_Line (Integer'Image (Integer_Value (R, 0))  -- First field
                & ' ' & Value (R, 1));  -- Second field, as a string
      Next (R);
   end loop;
```

We need to know the index of each field in the query, and whether we need to retrieve it as an integer, a string, a Boolean, etc. The third Gem will show a solution that is safer and results in a compile-time error when we use incorrect types.

Here are two additional capabilities provided by GNATCOLL.SQL.Exec, which are fully described in the documentation:

* Automatic transactions

When the SQL command modifies the database (using an INSERT or an UPDATE, for instance), GNATColl will automatically start a transaction on the DBMS so that all such changes are grouped and either all executed or all discarded. For this it provides two primitive operations on a connection: Commit and Rollback.

* Prepared and parameterized queries

As we mentioned in the first Gem, creating the query in the first place is less efficient (but safer) than using a string. The solution is to *prepare* the query. Preparation can occur either on the client (so GNATColl converts it to a string just once, and then reuses it every time you send the same query), or directly on the DBMS server (which will then parse the string and prepare its execution so that multiple executions are much faster). Of course, it is not often that you want to execute the exact same query, so the latter is only useful when part of the query can be substituted via parameters. Here is an example of such a query:

```
declare
   Q : constant SQL_Query := SQL_Select
      (Fields => Table1.Field2 & Table2.Field3,
       From => Table1 & Table2,
       Where => Table1.Field1 = Table2.Field3
          and Table1.Field1 = Integer_Param (1));
   Prepared_Q : constant Prepared_Statement :=
      Prepare (Q, On_Server => True);
begin
   ...
end;
```

There is a lot going on here: Q itself indicates that it will need one parameter (index 1) that is an integer; Prepare_Q is then the same as Q, but will be prepared on the server. At the point of declaration, there is no connection, so obviously the query is not prepared on the server yet. What's more, GNATColl needs to prepare it for each connection to the database, not just once. So this preparation will take place the first time the query is executed for a specific connection.

When you execute Prepared_Q you need to provide the parameter value, as in:

```
R.Fetch (DB, Prepared_Q, Params => (1 => +23)); -- first execution
R.Fetch (DB, Prepared_Q, Params => (1 => +34)); -- second execution
```

The second execution will be much faster than the first. As a point of comparison: when passing the SQL_Query directly and using a Direct_Cursor, it took 4.05s to execute the same query 100_000 times with SQLite; the same query prepared on the client took 2.50s; finally, the same query prepared on the server, as above, took only 0.55s.

Preparing queries can thus have a very significant impact on your application's performance.

To be continued in part 3 ...

Related Source Code

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

Gem #131 : Type-Safe Database API - Part 3

Author: Emmanuel Briot, *AdaCore*

Let's get started…

A recent tendency in a lot of web frameworks, which is extending into other areas, is to hide SQL in the persistent layer of applications. Users want to manipulate objects, indicate that they should be made persistent somehow, and forget that this is done through SQL.

GNATCOLL.SQL.ORM provides such an API. It is fully compatible with GNATCOLL.SQL, so one can decide to mix explicit SQL queries, like the ones we discussed before, with persistent ORM objects.

To use the ORM, it's necessary to generate some additional code from a schema description, as we saw in the first part. For this, you need to pass an extra parameter "-orm" in the call to gnatcoll_db2ada:

```
gnatcoll_db2ada -dbname <name> -dbhost <host> -dbuser <user>
   -orm ORM -api Database
```

Access to the database in the context of the ORM is slightly different. The connections are encapsulated in objects called Sessions. A session is basically responsible for executing the SQL queries, caching the results when appropriate, and committing or rolling back the changes. In fact, GNATColl provides a pool of such sessions so that the whole process is lighter weight: when you need a new session, you request it from the pool. If one is available, it's returned to you and it will not have to connect to the database again (which can be slow); otherwise, a new session and database connection are created. When the session is no longer needed, it's automatically returned to the pool.

So we first need to initialize this pool of sessions:

```
declare
  Session : GNATCOLL.SQL.Sessions.Session_Type;
begin
  GNATCOLL.SQL.Sessions.Setup
     (Descr => DB_Descr,   -- See second part of this Gem series
      Max_Sessions => 2);  -- Maximum number of concurrent sessions

  --  Then, when you need a session:
  Session := Get_New_Session;
  ...
end;
```

Using the same schema we had in the first two parts, here is how we could return all elements of Table1 that have a specific value for Field2:

```
with ORM;  use ORM;    -- The generated package

...

declare
   L : Table1_List := All_Table1   -- Generate function
                      .Filter (Field2 => "some value")
                      .Get (Session);
begin
   while L.Has_Row loop
      Put_Line (Integer'Image (L.Element.Field1));
      L.Next;
   end loop;
end;
```

The ORM is based on managers. There is one type of manager per table for a given schema. A manager is responsible for building the queries. In fact, the generated ORM package has one global instance of these types per table.

Here, we are using the All_Table1 manager. By default, this will return all elements of the table. We then filter the results by specifying that they should have a specific value for Field2. We can call Filter multiple times; there are no queries executed until we call Get on the third line.

We then retrieve a list of rows from Table1. The list behaves exactly like the Forward_Cursor we saw in the second part of the Gem series. It provides additional subprograms, so that what we actually manipulate is not just a row with fields, but an actual tagged object that corresponds to the schema, and so that instead of retrieving the first field of the row, we can simply retrieve "Field1". This is of course safer and more readable.

It's possible to write the filter so that it also involves the second table of the schema. The GNATColl documentation provides more details on how to do this. (Note that this is still an evolving API, which currently requires writing the JOIN explicitly through GNATCOLL.SQL, although the intent is to also hide this completely.)

But the ORM can also be used to modify objects from the database. Here is an example that retrieves a specific row from Table1, modifies some information and writes it back to the database.

```
declare
   T : Table1 := Get_Table1 (Session, Field1 => 2);  -- Lookup by id
begin
   T.Set_Field2 ("New value");
   T.Set_Field4 (...);   -- If it was defined in the schema
   Session.Commit;
end;
```

This is much more readable than the equivalent SQL query, is type safe, and is about as efficient (since in fact only one query is executed when Commit is called, where none were executed before).

In summary, using an ORM often provides queries that are easier to read. More importantly, it provides more convenient code to retrieve the result of the query, so that the code manipulates objects instead of rows.

Related Source Code

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

Gem #132 : Erroneous Execution - Part 1

Author: Bob Duff, *AdaCore*

Let's get started…

Ada is pretty good about requiring compilers to detect errors at compile time, or failing that, at run time. However, there are some kinds of errors that are infeasible to detect. The Reference Manual calls such errors "erroneous execution".

RM-1.1.5(10) defines the term:

...[T]he implementation need not detect such errors either prior to or during run time. ...[T]here is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable.

An example of erroneous execution is suppressing a check that fails. For example:

```
pragma Suppress (All_Checks); -- Or use the -gnatp switch
...
A (X) := A (X) + 1;
```

If X is out of bounds, then the above results in erroneous execution. That means that the program can do anything at all. In explaining the meaning of erroneousness, people often like to talk of spectacular disasters: "It might erase your system disk!" "Your keyboard might catch on fire!" "Nasal demons!"

I think that's somewhat misleading. For one thing, if you're running under an operating system, with proper protections set up, erroneous execution will not erase your system disk. The point is that Ada doesn't ensure that, but the operating system does. Likewise, Ada doesn't prevent your keyboard from catching on fire, but we hope the computer manufacturer will.

One disaster that actually might happen is that the above code will overwrite some arbitrary memory location. Whatever variable was stored there might be destroyed. That's a disaster because it can take hours or even days to track down such bugs. If you're lucky, you'll get a segmentation fault right away, making the bug much easier to figure out.

But the worst thing of all is not keyboard fires, nor destroyed variables, nor anything else spectacular. The worst thing an erroneous execution can cause is for the program to behave exactly the way you wanted it to, perhaps because the destroyed memory location wasn't being used for anything important. So what's the problem? If the program works, why should we care if some pedantic language lawyer says it's being erroneous?

To answer that question, note this common debugging technique: You have a large program. You make a small change (to fix a bug, to add a new feature, or just to make the code cleaner). You run your regression tests, and something fails. You deduce that the

cause of the new bug is the small change you made. Because the change is small relative to the size of the whole program, it's easy to figure out what the problem is.

With erroneousness, that debugging technique doesn't work. Somebody wrote the above erroneous program (erroneous if X is out of bounds, that is). It worked just fine. Then a year later, you make some change totally unrelated to the "A (X) := A (X) + 1;" statement. This causes things to move around in memory, such that now important data is destroyed. You can no longer assume that your change caused the bug; you have to consider the entire program text.

The moral of the story is: Do not write erroneous programs.

Related Source Code

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

Gem #133 : Erroneous Execution - Part 2

Author: Bob Duff, *AdaCore*

Let's get started…

The moral of the story was: Do not write erroneous programs.

Strictly speaking, that's wrong usage. "Erroneous" refers to a particular execution of a program, not to the program itself. It's possible to write a program that has erroneous behavior for some input data, but not for some other input data. Nonetheless, it's reasonable to use "erroneous program" to refer to a program that *might* have erroneous behavior. Just remember that "erroneous" is not a property of the program text, but a property of the program text plus its input, and even its timing.

Never deliberately write code that can cause erroneous execution. For example, I've seen people suppress Overflow_Checks, because they "know" the hardware does wrap-around (modular) arithmetic, and that's what they want. That's wrong reasoning. The RM doesn't say that overflow, when suppressed, will do what the hardware does. The RM says anything at all can happen.

If you suppress Overflow_Checks, you are telling the compiler to assume to assume that overflow will not happen. If overflow *can* happen, you are telling the compiler to assume a falsehood. In any mathematical system, if you assume "false", anything at all can be proven true, causing the whole house of cards to tumble. Optimizers can and do prove all sorts of amazing things when told to assume "false".

Never try to guess that the optimizer isn't smart enough to cause trouble. Optimizers are so complicated that even their authors can't accurately predict what they will do. For example:

```
X : Natural := ...;
Y : Integer := X + 1;

if Y > 0 then
    Put_Line (Y'Img);
end if;
```

The above will print some positive number, unless X is Integer'Last, in which case Constraint_Error will be raised. The optimizer is therefore allowed to deduce that when we get to the 'if', Y must be positive, so it can remove the 'if', transforming it to:

```
X : Natural := ...;
Y : Integer := X + 1;

Put_Line (Y'Img);
```

That's good: removing the 'if' probably makes it run faster, which is the optimizer's goal. But if checks are suppressed, the optimizer can *still* do the above transformation. The reasoning is now: "when we get to the 'if', either Y must be positive, or we must be erroneous". If the former, the 'if' can be removed because it's True. If the latter, anything can happen (it's erroneous!), so the 'if' can be removed in that case, too. "X + 1" might produce -2**31 (or it might not).

We end up with a program that says:

```
if Y > 0 then
    Put_Line (Y'Img);
end if;
```

and prints a negative number, which is a surprise.

Another possible behavior of the above code (with checks suppressed) is to raise Constraint_Error. "Hey, I asked for the checks to be suppressed. Why didn't the compiler suppress them?" Well, if the execution is erroneous, anything can happen, and raising an exception is one possible "anything". The purpose of suppressing checks is to make the program faster. Do not use pragma Suppress to suppress checks (in the sense of relying on not getting the exception). Compilers do not remove suppressed checks if they are free -- for example, imagine a machine that automatically traps on overflow.

Perhaps pragma Suppress should have been called something like Assume_Checks_Will_Not_Fail, since it doesn't (necessarily) suppress the checks.

To be continued in part 3

Related Source Code

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

Gem #134 : Erroneous Execution - Part 3

Author: Bob Duff, *AdaCore*

Let's get started…

We showed how this:

```
X : Natural := ...;
Y : Integer := X + 1;

if Y > 0 then
    Put_Line (Y'Img);
end if;
```

can end up printing a negative number if checks are suppressed.

In fact, a good compiler will warn that "Y > 0" is necessarily True, so the code is silly, and you can fix it. But you can't count on that. Optimizers are capable of much more subtle reasoning, which might not produce a warning. For example, suppose we have a procedure:

```
procedure Print_If_Positive (Y : Integer) is
begin
    if Y > 0 then
        Put_Line (Y'Img);
    end if;
end Print_If_Positive;
```

It seems "obvious" that Print_If_Positive will never print a negative number. But in the presence of erroneousness, that reasoning doesn't work:

```
X : Natural := ...;
Y : Integer := X + 1;

Print_If_Positive (Y);
```

The optimizer might decide to inline the call, and then optimize as in the previous example.

Other language features that can cause erroneous execution include:

- Shared variables (erroneous if multiple tasks fail to synchronize)
- Address clauses
- Unchecked_Conversion
- Interface to other languages
- Machine-code insertions
- User-defined storage pools

- Unchecked_Deallocation

A complete list can be found by looking up "erroneous" in the RM index, or by searching the RM. Every case of erroneous execution is documented under the heading "Erroneous Execution".

You should try to minimize the use of such features. When you need to use them, try to encapsulate them so you can reason about them locally. And be careful.

As for suppressing checks: Don't suppress unless you need the added efficiency and you have confidence that the checks won't fail. If you do suppress checks, run regression tests on a regular basis in both modes (suppressed and not suppressed).

The final part of this Gem series will explain the rationale behind the concept of erroneous execution in Ada.

Related Source Code

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

Gem #135 : Erroneous Execution - Part 4

Author: Bob Duff, *AdaCore*

Let's get started…

Many programmers believe that "optimizers *should not* change the behavior of the program". Many also believe that "optimizers *do not* change the behavior of the program". Both beliefs are false in the presence of erroneousness.

So if erroneousness is so bad, why does the Ada language design have it? Certainly, a language designer should try to minimize the amount of erroneousness. Java is an example of a language that eschews erroneousness, but that comes at a cost. It means that lots of useful things are impossible or infeasible in Java: device drivers, for example. There is also an efficiency cost. C is an example of a language that has way too much erroneousness. Every single array-indexing operation is potentially erroneous in C. (C calls it "undefined behavior".)

Ada is somewhere in between Java and C in this regard. You *can* write device drivers in Ada, and user-defined storage pools, and other things that require low-level access to the machine.

But for the most part, things that can cause erroneousness can be isolated in packages -- you don't have to scatter them all over the program as in C.

For example, to prevent dangling pointers, try to keep the "new" and Unchecked_Deallocations together, so they can be reasoned about locally. A generic Doubly_Linked_List package might have dangling pointer bugs within itself, but it can be designed so that clients cannot cause dangling pointers.

Another way to prevent dangling pointers is to use user-defined storage pools that allow deallocation of the entire pool at once. Store heap objects with similar lifetimes in the same pool. It might seem that deallocating a whole bunch of objects is more likely to cause dangling pointers, but in fact just the opposite is true. For one thing, deallocating the whole pool is much simpler than walking complicated data structures deallocating individual records one by one. For another thing, deallocating en masse is likely to cause catastrophic failures that can be fixed sooner rather than later. Finally, a user-defined storage pool can be written to detect dangling pointers, for example by using operating system services to mark deallocated regions as inaccessible.

Note that Ada 2012 has "Subpools", which make user-defined storage pools more flexible.

A final point about erroneousness that might be surprising is that it can go backwards in time. For example:

```
if Count = 0 then
   Put_Line ("Zero");
end if;
Something := 1 / Count; -- could divide by zero
```

If checks are suppressed, the entire 'if' statement, including the Put_Line, can be removed by the optimizer. The reasoning is: If Count is nonzero, we don't want to print "Zero". If Count is zero, then it's erroneous, so anything can happen, including not printing "Zero".

Even if the Put_Line is not removed by the compiler, it can appear to be, because the "Zero" might be stored in a buffer that never gets flushed because some later erroneousness caused the program to crash.

Every statement about Ada must be understood to have ", unless execution is erroneous" after it. In this case, "Count = 0 returns True if Count is zero" is obviously true, but it really means "Count = 0 returns True if Count is zero, unless execution is erroneous, in which case anything can happen".

Moral: Take care to avoid writing erroneous programs.

Related Source Code

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

FCRC'15
Federated Computing Research Conference
Oregon Convention Center, June 12 - 20, 2015
Portland, Oregon
http://fcrc.acm.org/

FCRC 2015 will be held in Portland Oregon from June 12-20. Chaired by Rajiv Gupta of UC Riverside, the conference will assemble a spectrum of affiliated research conferences into a week-long coordinated meeting. The technical program for each affiliated conference will be independently administered, with each responsible for its own meeting's structure, content and proceedings. To the extent facilities allow attendees are free to attend technical sessions of other affiliated conferences being held at the same time as their "home" conference. Conferences include:

- CRA-W 2015: Career Mentoring Workshop
- EC 2015: The 16th ACM Conference on Economics and Computation
- HPDC 2015: The 24th International Symposium on High-Performance Parallel and Distributed Computing
- CCC: Computational Complexity Conference
- ISCA 2015: The 42nd International Symposium on Computer Architecture
- ISMM 2015: ACM SIGPLAN International Symposium on Memory Management
- IWQoS 2015: IEEE/ACM International Symposium on Quality of Service
- LCTES 2015: ACM SIGPLAN/SIGBED International Conference on Languages, Compilers and Tools for Embedded Systems
- PLDI 2015: 36th ACM SIGPLAN Conference on Programming Language Design and Implementation
- SIGMETRICS 2015: International Conference on Measurement and modeling of Computer Systems
- SPAA 2015: ACM Symposium on Parallelism in Algorithms and Architectures
- STOC 2015: 47th ACM Symposium on Theory of Computing

Call for Papers

20th International Conference on Reliable Software Technologies — Ada-Europe 2015

22–26 June 2015, Madrid, Spain

http://www.ada-europe.org/conference2015

Conference Chair

Alejandro Alonso
ETSIT-UPM
aalonso@dit.upm.es

Program co-Chairs

Juan A. de la Puente
ETSIT-UPM
jpuente@dit.upm.es

Tullio Vardanega
Università di Padova
tullio.vardanega@unipd.it

Tutorial Chair

Jorge Real
UPV
jorge@disca.upv.es

Exhibition Chair

Santiago Urueña
GMV
suruena@gmv.com

Industrial co-Chairs

Jørgen Bundgaard
Rambøll Danmark A/S
jogb@ramboll.dk

Ana Rodríguez
Silver Atena
ana.rodriguez@silver-atena.es

Publicity Chair

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

Local Chair

Juan Zamorano
ETSIINF-UPM
jzamora@fi.upm.es

"In cooperation" with ACM SIGAda, SIGBED, SIGPLAN, and with ARA

acm In-Cooperation

General Information

The 20th International Conference on Reliable Software Technologies – Ada-Europe 2015 will take place in Madrid, Spain. Following its traditional style, the conference will span a full week, including a three-day technical program and vendor exhibition from Tuesday to Thursday, along with parallel tutorials and workshops on Monday and Friday.

Schedule

11 January 2015	Submission of regular papers, tutorial and workshop proposals
25 January 2015	Submission of industrial presentation proposals
1 March 2015	Notification of acceptance to all authors
29 March 2015	Camera-ready version of regular papers required
12 April 2015	Industrial presentation abstracts required
17 May 2015	Tutorial and workshop materials required

Topics

The conference has over the years become a leading international forum for providers, practitioners and researchers in reliable software technologies. The conference presentations will illustrate current work in the theory and practice of the design, development and maintenance of long-lived, high-quality software systems for a challenging variety of application domains. The program will allow ample time for keynotes, Q&A sessions and discussions, and social events. Participants include practitioners and researchers representing industry, academia and government organizations active in the promotion and development of reliable software technologies.

Topics of interest to this edition of the conference include but are not limited to:

- **Multicore and Manycore Programming**: Predictable Programming Approaches for Multicore and Manycore Systems, Parallel Programming Models, Scheduling Analysis Techniques.
- **Real-Time and Embedded Systems**: Real-Time Scheduling, Design Methods and Techniques, Architecture Modelling, HW/SW Co-Design, Reliability and Performance Analysis.
- **Mixed-Criticality Systems:** Scheduling methods, Mixed-Criticality Architectures, Design Methods, Analysis Methods.
- **Theory and Practice of High-Integrity Systems**: Medium to Large-Scale Distribution, Fault Tolerance, Security, Reliability, Trust and Safety, Languages Vulnerabilities.
- **Software Architectures**: Design Patterns, Frameworks, Architecture-Centred Development, Component-based Design and Development.
- **Methods and Techniques for Software Development and Maintenance**: Requirements Engineering, Model-driven Architecture and Engineering, Formal Methods, Re-engineering and Reverse Engineering, Reuse, Software Management Issues, Compilers, Libraries, Support Tools.
- **Software Quality**: Quality Management and Assurance, Risk Analysis, Program Analysis, Verification, Validation, Testing of Software Systems.
- **Mainstream and Emerging Applications**: Manufacturing, Robotics, Avionics, Space, Health Care, Transportation, Cloud Environments, Smart Energy systems, Serious Games, etc.
- **Experience Reports in Reliable System Development**: Case Studies and Comparative Assessments, Management Approaches, Qualitative and Quantitative Metrics.
- **Experiences with Ada and its Future**: Reviews of the Ada 2012 new language features, implementation and use issues, positioning in the market and in the software engineering curriculum, lessons learned on Ada Education and Training Activities with bearing on any of the conference topics.

Program Committee

Mario Aldea, Universidad de Cantabria, Spain
Ted Baker, NSF, USA
Johann Blieberger, Technische Universität Wien, Austria
Bernd Burgstaller, Yonsei University, Korea
Alan Burns, University of York, UK
Maryline Chetto, University of Nantes, France
Juan A. de la Puente, Universidad Politécnica de Madrid, Spain
Laurent George, ECE Paris, France
Michael González Harbour, Universidad de Cantabria, Spain
J. Javier Gutiérrez, Universidad de Cantabria, Spain
Jérôme Hugues, ISAE, France
Hubert Keller, Institut für Angewandte Informatik, Germany
Albert Llemosí, Universitat de les Illes Balears, Spain
Franco Mazzanti, ISTI-CNR, Italy
Stephen Michell, Maurya Software, Canada
Jürgen Mottok, Regensburg University of Applied Sciences, Germany
Laurent Pautet, Telecom ParisTech, France
Luís Miguel Pinho, CISTER/ISEP, Portugal
Erhard Plödereder, Universität Stuttgart, Germany
Jorge Real, Universitat Politècnica de València, Spain
José Ruiz, AdaCore, France
Sergio Sáez, Universitat Politècnica de Valencia, Spain
Amund Skavhaug, NTNU, Norway
Tucker Taft, AdaCore, USA
Theodor Tempelmeier, University of Applied Sciences Rosenheim, Germany
Elena Troubitsyna, Åbo Akademi University, Finland
Santiago Urueña, GMV, Spain
Tullio Vardanega, Università di Padova, Italy

Industrial Committee

Jørgen Bundgaard, Rambøll Danmark A/S
Ana Rodríguez, Silver Atena, Spain
Dirk Craeynest, Ada-Europe & KU Leuven, Belgium
Jacob Sparre Andersen, JSA Consulting, Denmark
Jean-Loup Terraillon, ESA
Paolo Panaroni, Intecs, Italy
Paul Parkinson, Wind RIver, UK
Peter Dencker, ETAS GmbH, Germany
Rod White, MBDA, UK
Steen Palm, Terma, Denmark
Ahlan Marriott, White Elephant, Switzerland
Ian Broster, Rapita Systems, UK
Ismael Lafoz, Airbus Military, Spain
Jean-Pierre Rosen, Adalog, France
Robin Messer, Altran-Praxis, UK
Roger Brandt, Telia, Sweden
Claus Stellwag, Elektrobit AG, Germany
Quentin Ochem, Ada Core, France
Martyn Pike, Ada UK

Call for Regular Papers

Authors of regular papers which are to undergo peer review for acceptance are invited to submit original contributions. Paper submissions shall not exceed 14 LNCS-style pages in length. Authors shall submit their work via EasyChair following the relevant link on the conference web site. The format for submission is solely PDF.

Proceedings

The conference proceedings will be published in the Lecture Notes in Computer Science (LNCS) series by Springer, and will be available at the start of the conference. The authors of accepted regular papers shall prepare camera-ready submissions in full conformance with the LNCS style, not exceeding 14 pages and strictly by March 29, 2015. For format and style guidelines authors should refer to http://www.springer.de/comp/lncs/authors.html. Failure to comply and to register for the conference by that date will prevent the paper from appearing in the proceedings.

The CiteSeerX Venue Impact Factor has the Conference in the top quarter. Microsoft Academic Search has it in the top third for conferences on programming languages by number of citations in the last 10 years. The conference is listed in DBLP, SCOPUS and Web of Science Conference Proceedings Citation index, among others.

Awards

Ada-Europe will offer honorary awards for the best regular paper and the best presentation.

Call for Industrial Presentations

The conference seeks industrial presentations which deliver value and insight but may not fit the selection process for regular papers. Authors are invited to submit a presentation outline of exactly 1 page in length by January 25, 2015. Submissions shall be made via EasyChair following the relevant link on the conference web site. The Industrial Committee will review the submissions and make the selection. The authors of selected presentations shall prepare a final short abstract and submit it by April 12, 2015, aiming at a 20-minute talk. The authors of accepted presentations will be invited to submit corresponding articles for publication in the *Ada User Journal (http://www.ada-europe.org/auj/)*, which will host the proceedings of the Industrial Program of the Conference. For any further information please contact the Industrial Chair directly.

Call for Tutorials

Tutorials should address subjects that fall within the scope of the conference and may be proposed as either half- or full-day events. Proposals should include a title, an abstract, a description of the topic, a detailed outline of the presentation, a description of the presenter's lecturing expertise in general and with the proposed topic in particular, the proposed duration (half day or full day), the intended level of the tutorial (introductory, intermediate, or advanced), the recommended audience experience and background, and a statement of the reasons for attending. Proposals should be submitted by e-mail to the Tutorial Chair. The authors of accepted full-day tutorials will receive a complimentary conference registration as well as a fee for every paying participant in excess of 5; for half-day tutorials, these benefits will be accordingly halved. The *Ada User Journal* will offer space for the publication of summaries of the accepted tutorials.

Call for Workshops

Workshops on themes that fall within the conference scope may be proposed. Proposals may be submitted for half- or full-day events, to be scheduled at either end of the conference week. Workshop proposals should be submitted to the Conference Chair. The workshop organizer shall also commit to preparing proceedings for timely publication in the *Ada User Journal*.

Call for Exhibitors

The commercial exhibition will span the three days of the main conference. Vendors and providers of software products and services should contact the Exhibition Chair for information and for allowing suitable planning of the exhibition space and time.

Grants for Reduced Student Fees

A limited number of sponsored grants for reduced fees is expected to be available for students who would like to attend the conference or tutorials. Contact the Conference Chair for details.

ACM DL DIGITAL LIBRARY

Add the ACM Digital Library to your membership—or join ACM and get the DL at member rate

The ACM Digital Library is simply the world's largest, most respected online resource for computing professionals. The DL includes the full text of more than 88 ACM publications, including journal papers and magazine and SIG newsletter articles, plus proceedings from more than 500 annual conferences…all at your fingertips, from desktop or mobile platforms.

With a DL subscription, you can enjoy unlimited access to an integrated bibliography covering the entire computing field:

- Stay on top of the latest innovations from top thought leaders, researchers, entrepreneurs, and makers in cutting-edge technological fields
- Discover new ideas they shared at more than 170 SIG-sponsored events around the world—scan a paper, or watch a keynote talk
- Search across a comprehensive computing bibliography of more than 2.3 million records from over 5,000 publishers worldwide

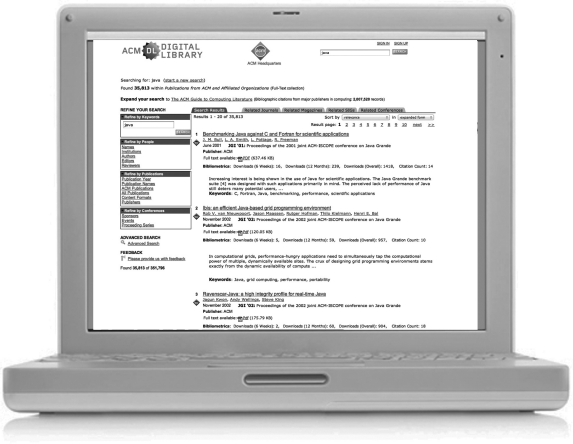
Available to ACM Members only

Current ACM Professional Members
can add the ACM Digital Library for only $99

Or join ACM now and include the DL at the member rate—$198
Join ACM online at www.acm.org/joinacm

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

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



SHAPE THE FUTURE OF COMPUTING.
JOIN ACM TODAY.

ACM is the world's largest computing society, offering benefits that can advance your career and enrich your knowledge with life-long learning resources. We dare to be the best we can be, believing what we do is a force for good, and in joining together to shape the future of computing.

SELECT ONE MEMBERSHIP OPTION

ACM PROFESSIONAL MEMBERSHIP:

- ❑ Professional Membership: $99 USD
- ❑ Professional Membership plus ACM Digital Library: $198 USD ($99 dues + $99 DL)
- ❑ ACM Digital Library: $99 USD (must be an ACM member)

ACM STUDENT MEMBERSHIP:

- ❑ Student Membership: $19 USD
- ❑ Student Membership plus ACM Digital Library: $42 USD
- ❑ Student Membership PLUS Print *CACM* Magazine: $42 USD
- ❑ ACM Student Membership w/Digital Library PLUS Print *CACM* Magazine: $62 USD

❑ **Join ACM-W:** ACM-W supports, celebrates, and advocates internationally for the full engagement of women in all aspects of the computing field. Available at no additional cost.

Priority Code: CAPP

Payment Information

Name

ACM Member #

Mailing Address

City/State/Province

ZIP/Postal Code/Country

Email

Payment must accompany application. If paying by check or money order, make payable to ACM, Inc, in U.S. dollars or equivalent in foreign currency.

❑ AMEX ❑ VISA/MasterCard ❑ Check/money order

Total Amount Due

Credit Card #

Exp. Date

Signature

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

Prices include surface delivery charge. Expedited Air Service, which is a partial air freight delivery service, is available outside North America. Contact ACM for more information.

Purposes of ACM

ACM is dedicated to:

1) Advancing the art, science, engineering, and application of information technology
2) Fostering the open interchange of information to serve both professionals and the public
3) Promoting the highest professional and ethics standards

Satisfaction Guaranteed!

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

Association for Computing Machinery

1-800-342-6626 (US & Canada)
1-212-626-0500 (Global)

Hours: 8:30AM - 4:30PM (US EST)
Fax: 212-944-1318

acmhelp@acm.org
acm.org/join/CAPP