ADA USER JOURNAL

Volume 26
Number 2
June 2005

Contents

Editorial Policy for *Ada User Journal*

Publication

Ada User Journal – The Journal for the international Ada Community – is published by Ada-Europe. It appears four times a year, on the last days of March, June, September and December. Copy date is the first of the month of publication.

Aims

Ada User Journal aims to inform readers of developments in the Ada programming language and its use, general Ada-related software engineering issues and Ada-related activities in Europe and other parts of the world. The language of the journal is English.

Although the title of the Journal refers to the Ada language, any related topics are welcome. In particular papers in any of the areas related to reliable software technologies.

The Journal publishes the following types of material:

- Refereed original articles on technical matters concerning Ada and related topics.
- News and miscellany of interest to the Ada community.
- Reprints of articles published elsewhere that deserve a wider audience.
- Commentaries on matters relating to Ada and software engineering.
- Announcements and reports of conferences and workshops.
- Reviews of publications in the field of software engineering.
- Announcements regarding standards concerning Ada.

Further details on our approach to these are given below.

Original Papers

Manuscripts should be submitted in accordance with the submission guidelines (below).

All original technical contributions are submitted to refereeing by at least two people. Names of referees will be kept confidential, but their comments will be relayed to the authors at the discretion of the Editor.

The first named author will receive a complimentary copy of the issue of the Journal in which their paper appears.

By submitting a manuscript, authors grant Ada-Europe an unlimited license to publish (and, if appropriate, republish) it, if and when the article is accepted for publication. We do not require that authors assign copyright to the Journal.

Unless the authors state explicitly otherwise, submission of an article is taken to imply that it represents original, unpublished work, not under consideration for publication elsewhere.

News and Product Announcements

Ada User Journal is one of the ways in which people find out what is going on in the Ada community. Since not all of our readers have access to resources such as the World Wide Web and Usenet, or have enough time to search through the information that can be found in those resources, we reprint or report on items that may be of interest to them.

Reprinted Articles

While original material is our first priority, we are willing to reprint (with the permission of the copyright holder) material previously submitted elsewhere if it is appropriate to give it a wider audience. This includes papers published in North America that are not easily available in Europe.
We have a reciprocal approach in granting permission for other publications to reprint papers originally published in *Ada User Journal.*

Commentaries

We publish commentaries on Ada and software engineering topics. These may represent the views either of individuals or of organisations. Such articles can be of any length – inclusion is at the discretion of the Editor.

Opinions expressed within the *Ada User Journal* do not necessarily represent the views of the Editor, Ada-Europe or its directors.

Announcements and Reports

We are happy to publicise and report on events that may be of interest to our readers.

Reviews

Inclusion of any review in the Journal is at the discretion of the Editor.
A reviewer will be selected by the Editor to review any book or other publication sent to us. We are also prepared to print reviews submitted from elsewhere at the discretion of the Editor.

Submission Guidelines

All material for publication should be sent to the Editor, preferably in electronic format. The Editor will only accept typed manuscripts by prior arrangement.
Prospective authors are encouraged to contact the Editor by email to determine the best format for submission. Contact details can be found near the front of each edition.
Example papers conforming to formatting requirements as well as some word processor templates are available from the editor. There is no limitation on the length of papers, though a paper longer than 10,000 words would be regarded as exceptional.

Editorial

As our readership know, June is a very busy month for Ada. Every June in fact for over two decades now sees the celebration of the Ada-Europe annual conference, which important meetings are often attached to. This year, an important meeting of the WG9, the ISO body in charge of the maintenance of the Ada language standard, was convened to decide among other subjects on the vernacular name to be given to the new revision of the language. This decision was of course less technically momentous than the approval of the wealth of features that the revision process had defined, yet it was felt as genuinely important by lots of people, especially those who linger outside the revision process itself. Since it is our great pleasure and honour to host in this journal since issue 25-4 the advance version of the Rationale for the revised language standard, we withheld the closing of this particular issue until we would hear final word from WG9 about the "official" vernacular name of the language, so that we would align to it. WG9 have spoken and deliberated that the vernacular name be "Ada 2005". So please go and use and spread this name, since the meaning it aims to convey is that the language revision will definitely be technically complete within the year 2005. Which is a jolly good news, I'd say.

This issue is especially rich. First and foremost, it carries two successive instalments of the Rationale for Ada 2005 edited by John Barnes: one illustrates the very important improvements made by the revision process on the access types; the other discusses the various improvements operated in the areas of program structure and visibility control. Lots of good stuff to read and educate oneself about! Then we are pleased to host a paper by Muthu Ramachandran, from the Leeds Metropolitan University, which illustrates a development that facilitates the production of reusable Ada components. The rest of the issue contains the usual wealth of Ada-related news and of significant conference events worldwide. It is often the case that some threads captured by the News section discuss the health of Ada as seen from various angles. From the editor's standpoint, I can most definitely assure you that Ada is more alive and kicking than ever. So long live Ada, especially now that it has to compete in open field with very powerful contenders without (at long last if I am allowed to say!) the very important financial backup that accompanied it until 1995. And of course enjoy the reading.

Tullio Vardanega
Padova
June 2005
Email: tullio.vardanega@math.unipd.it

News

Santiago Urueña

Technical University of Madrid. Email: suruena@datsi.fi.upm.es

Contents

Ada-related Events

[To give an idea about the many Ada-related events organized by local groups, some information is included here. If you are organizing such an event feel free to inform us as soon as possible. If you attended one please consider writing a small report for the Ada User Journal.
-- su]

Apr 11 - Ada-Belgium General Assembly

From: Dirk Craeynest <dirk@heli.cs.kuleuven.ac.be>
Subject: Ada Web Services & Eclipse Plug-in, Mon 11 Apr 20:00, Ada-Belgium
Date: 6 Apr 2005 22:35:35
Organization: Ada-Belgium, c/o Dept. of Computer Science, K.U.Leuven

Ada-Belgium will hold its 12th annual General Assembly on Monday, April 11, 2005, at the U.L.B., Department of Computer Science, Boulevard du Triomphe / Triomflaan, B-1050 Brussels, at 19:00. The official convocation is distributed separately to members and is also available on the Ada-Belgium web-server.

There will be refreshments and pizza for Ada-Belgium members at 18:15. Please notify us if you are a current or new member and intend to participate at this informal "pre-meeting".

At 20:00 the General Assembly will be followed by a short product announcement of an "Eclipse plug-in for Ada (ObjectAda or GNAT)", by Patricia Langle from Aonix France, and at 20:15 by a technical presentation on "Web-enabling Ada Applications with AWS", by Jean-Pierre Rosen from AdaLog, France.

Everyone interested is welcome: you don't have to be a member to attend.

20:00-20:15 - Eclipse plug-in for Ada (ObjectAda or GNAT)

Abstract
The Eclipse platform is a generic and open architecture for building integrated development environments (IDEs). Written in Java and available on a wide range of OS, it permits to manage user's workspace, providing navigation view, text editor, outline view, ... It's built on mechanisms for discovering, integrating, and running modules called plug-ins. A tool provider integrates tools in Eclipse by writing separate plug-ins that operate on files in the workspace.

The Eclipse platform, by itself is not dedicated to any language. The Java Development Tooling (JDT) is a set of plug-ins which add Java program development capabilities to the Platform. A C Development Tooling (CDT) does the same for C program development. Aonix is working today on an ADT (Ada Development Tooling) that will permit Ada developers to access that full-featured IDE.

Speaker
Patricia Langle, South Europe Presales Manager, Aonix France

More information
See the press release at http://www.aonix.com/pr_07.26.04.html and the article in ADT Magazine at http://www.adtmag.com/article.asp?id=9583.

20:15-21:45 - Web-enabling Ada Applications with AWS

Abstract
This presentation describes AWS, the Ada Web Server, and how to use it for the development of web applications. It describes the principles of AWS, from the most basic functionalities to the more advanced ones (Authentication, SOAP interface, session management, hot plugs, multi-server applications, etc.) The talk emphasizes practical usage of AWS, and presents design patterns that have proved effective for developing existing applications. It compares the development process with AWS to other techniques. The presentation provides attendees with the information needed to assess whether AWS is appropriate to their needs, and the necessary knowledge to start writing full-scale Web applications. Attendees should have some knowledge of Ada programming. No previous knowledge of Web programming or HTML is required.

AWS is a free (GMGPL) software component written by Pascal Obry and Dmitriy Anisimkov that allows developing Web applications in Ada. Unlike other methods that require a dedicated server (like Apache), AWS provides services to develop applications that act as autonomous Web servers, using the Ada language for the semantic part of the application instead of scripting languages like Perl or Python. This allows AWS to be used for regular Web servers as well as for writing applications that offer a Web interface to control more traditional processing functions. AWS is a mature product that has been used in many professional applications.

Speaker
J-P. Rosen graduated from ENST (Ecole Nationale Supérieure des Télécommunications) in 1975, and obtained PhD in 1986. He started as a software engineer at the computing centre of ENST. After a Sabbatical at New York University on the Ada/ED Project, he worked as Professor at ENST, where he was responsible for the teaching of Operating Systems, Software Engineering, Compilation and Ada. He created ENST's master's degree in Software Engineering. He has now formed Adalog, a company specialized in high level training, consultancy, and software development in the fields of Ada, OOD, and associated technologies.

J-P. Rosen has written "HOOD: an Industrial Approach for Software Design", the tutorial book for the HOOD 4 method. This was undertaken on behalf and under control of the HOOD User Group. This book is currently the only official tutorial book for the HOOD method; details can be obtained from http://www.adalog.fr/hoodbook.htm Other book publications by J-P. Rosen include the translation in French of Booch's "Software Engineering with Ada", and a book called "Méthodes de Génie Logiciel avec Ada 95" (Software Engineering Methods with Ada 95). He can be reached via email at rosen at adalog.fr

More information
AWS, a complete Web development framework, is available on the Libre Site for Free Software Developers at http://libre.adacore.com/aws/

Participation
Everyone interested is welcome at either or both parts of this meeting. As usual, the event is free and presentations are in English.

If you plan to attend the General Assembly or the technical presentation, we would appreciate it if you could inform us by e-mail at the address below (please also specify if you intend to participate at the informal "pre-meeting"). Although no formal registration is required, this helps our preparations.

For more information and directions see the web page mentioned above.

Looking forward to meet many of you in Brussels!

From: Dirk Craeynest <dirk@heli.cs.kuleuven.ac.be>
Date: 16 Apr 2005 13:26:47
Organization: Ada-Belgium, c/o Dept. of Computer Science, K.U.Leuven
Subject: Ada-Belgium updates: Ada Web Services, Eclipse Plug-in, Ada 2005

The Ada-Belgium evening event earlier this week featured a short product announcement of an "Eclipse plug-in for Ada (ObjectAda or GNAT)", by Patricia Langle from Aonix France, and a technical presentation on "Web-enabling Ada Applications with AWS", by Jean-Pierre Rosen from Adalog, France.

We are pleased to announce that the slides of these presentations are now available on-line on the Ada-Belgium web pages.

Finally, a long overdue update. A special "Ada 2005 Panel" was organized at the SIGAda'2004 conference in Atlanta. Several members of the ISO Ada Rapporteur Group presented a number of mini-briefings on the improvements that were already approved for inclusion in the Ada 2005 Amendment. By agreement with the ARG chairman, and as follow-up of his previous presentations at Ada-Belgium and Ada-Europe events, the slides of those mini-briefings are now available on-line on our site as well.

For easy access, check out "What's new on the Ada-Belgium web-pages?" at URL <http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/whatsnew.html> if you're interested.

Apr 25 - XIII Technical Day of Ada-Spain

From: José Javier Gutiérrez <gutierjj@unican.es>
Date: Wed, 15 Jun 2005 13:52:24
Organization: Ada-Spain
Subject: Meeting Report of the XIII Technical Day of Ada-Spain

The Technical Day of Ada-Spain is a yearly meeting devoted to presenting and discussing the results of research and development projects related to the Ada language and reliable software technologies.

This year, the meeting was held on April 25th at the Telecommunications Engineering School of the Technical University of Madrid, and it consisted of two invited talks and seven technical presentations.

The invited talks were:

* Moving Forward with Ada 2005 - New Real-Time Features and other Goodies, by Alan Burns, from the Real-Time Systems Research Group, Department of Computer Science, University of York (UK), and member of the ARG (Ada Rapporteur Group) of the standardization committee ISO/IEC JTC1/SC22/WG9. Alan gave a very interesting talk that revealed the current state of the new real-time services being added to the Ada language. The audience was pleased to hear that many services have been approved for the new standard, including the Ravenscar profile, execution time budgeting, timing events, dynamic priority ceilings, and new scheduling policies (round robin, EDF, non-preemptive) that can be used by themselves or in a mixed fashion.

* Current State of the Ada 2005 Implementation in Gnat, by Javier Miranda, from the Instituto Universitario de Microelectrónica Aplicada, University of Las Palmas de Gran Canaria, and member of the Gnat development team. Javier collaborates with AdaCore, one of the main developers of Ada compilers, in the implementation of the changes being introduced for the new version of the language. He gave a clear view of these changes and their implementation state and plans. The audience was very interested in particular with the important changes related to the object-oriented features. As these changes are implemented, they will be made available to interested users under the GNAT Academic Program.

The technical programme had the following presentations:

* "Integration of application schedulers with the new scheduling policies defined for Ada tasks", by Mario Aldea, from the University of Cantabria

* "Efficient techniques for reducing context switches in the implementation of real-time schedulers", by Sergio Sáez, from the Technical University of Valencia

* "ORK and Ada 2005", by Juan Antonio de la Puente, from the Technical University of Madrid

* "Interchangeable scheduling policies in RT-GLADE", by Juan López Campos, from the University of Cantabria

* "Distributing Criticality Across Ada Partitions", by Miguel Masmano, from the Technical University of Valencia

* "Ada and GNAT for high-integrity systems", by José Ruiz, from AdaCore, Paris

* "GNAT Academic Program", by Louise Arkwright, from AdaCore, Paris

Right after the finalization of the Technical Day, the General Assembly of Ada-Spain was celebrated. During the assembly, the winners of the yearly Ada-Spain Award to the best academic project developed in Ada were announced. This year, the project awarded with the first prize was entitled "Distribution, Real-Time and Ada" and was authored by Juan López Campos from the University of Cantabria. The second prize was awarded to the project "Robot force control: design of an experimental platform and comparative analysis of different techniques changing the sampling rate", authored by Ángel Llosá Guillén from the Technical University of Valencia.

[Cf. the "Call for Contributions" of the XIII Technical Day of Ada-Spain in AUJ 26-1 (Mar 2005), p.29. -- su]

Jun 20-24 - Ada-Europe 2005 Conference

From: Dirk Craeynest <dirk@heli.cs.kuleuven.ac.be>
Date: 12 Jun 2005 12:11:23
Subject: Press Release - Reliable Software Technologies, Ada-Europe 2005
Organization: Ada-Europe, c/o Dept. of Computer Science, K.U.Leuven

Final Call for Participation - UPDATED Program Summary

10th International Conference on Reliable Software Technologies - Ada-Europe 2005 20 - 24 June 2005, York, UK

http://www.ada-europe.org/conference2005.html

Full Program available on conference web site - Check out the tutorial program! - Printed proceedings available.

Press release
Conference on Reliable Software Technologies in York

York, UK (12 June 2005 12:00) - The University of York, sponsored by Ada-Europe and in cooperation with ACM's Special Interest Group in Ada, organizes this year the "10th International Conference on Reliable Software Technologies - Ada-Europe 2005" from 20 to 24 June in York.

The conference offers nine tutorials, including a look at Ada 2005, a full technical program of refereed papers, a collection of industrial presentations reflecting current practice and challenges, three eminent invited speakers, an exhibition, and a social program.

The 9 excellent tutorials cover a broad range of topics, including: developing web-aware applications in Ada, correctness by construction, real-time Java, architecture analysis and design, Ravenscar and SPARK, containers in Ada 2005, software fault tolerance, requirements engineering for dependable Systems, and a half day tutorial (at a reduced rate) on the new features of Ada 2005, presented by four of its designers:

John Barnes, Alan Burns, Pascal Leroy and Tucker Taft.

Technical Program. 21 fully refereed and carefully selected papers on the latest research on Ada-related issues, including new tools, applications and industrial practice and experience. A collection of 10 industrial presentations reflecting current practice and challenges. Springer Verlag publishes the proceedings of the conference, as LNCS Vol. 3555.

Keynote Speakers. John McDermid discusses model-based development of safety-critical software. Martyn Thomas presents "Extreme Hubris" in which the principles of Extreme Programming are examined and shown to be misguided and dangerous, and in which an alternative Manifesto for Reliable Software is proposed. Bev Littlewood talks about assessing the dependability of software-based systems.

The *exhibition* opens in the mid-morning break on Tuesday and runs continuously until the end of the afternoon break on Thursday. The exhibitors include the following vendors: AdaCore, Aonix, ARTiSAN Software, Esterel Technologies, Green Hills Software, I-Logix, LDRA Software Technology, PolySpace Technologies, Praxis High Integrity Systems, Silver Software, TNI Europe.

York is a beautiful and historical (small) city in the north of the UK. It has a first class university with one of the best Computer Science departments in the world. The Department has been involved with the development of programming languages for a number of years (indeed it ran the first series of technical meetings on Ada in the 1970s). It is pleased to host this meeting on reliable software technology.

York can be reached easily by train from London (approximately 2.3 hours), Manchester airport (2 hours), Leeds/Bradford Airport (1 hour). The conference is held at the Royal York Hotel which is adjacent to the York train station a few minutes from the centre of York and the Minster (Cathedral).

The conference's social program includes a wine and buffet reception on Tuesday evening at Bedern Hall, a 14th century hall which was used as a refectory of the vicars of York Minster, and the conference banquet on Wednesday evening at the National Railway Museum. This York-based Museum is the largest railway museum in the world, responsible for the conservation and interpretation of the British national collection of historically significant railway vehicles and other artifacts. The Museum contains an unrivalled collection of locomotives, rolling stock, railway equipment, documents and records.

Latest updates:

- The full "Advance Program" is available on the conference web site <http://www.ada-europe.org/conference2005.html> and directly at <http://www.cs.york.ac.uk/rts/adaeurope/advprogram.pdf> (pdf, 1.7M).

- Check out the 9 tutorials in the advance program and at <http://www.cs.york.ac.uk/rts/adaeurope/tutorials.html>.

- The proceedings, published by Springer Verlag as Lecture Notes in Computer Science Vol. 3555, are ready and will be distributed at the conference. More info is available at <http://www.springeronline.com/3-540-26286-5>.
Abstracts can be checked out at <http://springerlink.metapress.com/link.asp?id=xpm9f7atnwaw>

[…]

- For the latest information consult the conference web site. <http://www.ada-europe.org/conference2005.html>

[Cf. same topic in AUJ 26-1 (Mar 2005), p.5. -- su]

Ada-related Organizations

PolySpace Technologies Joins the ARA

URL: http://www.adaic.org/news/polyspc.html

Specialist in Embedded Analysis Tools Looks for "Unique Platform" in ARA

BELMONT, Mass. [April 21, 2005] - The Ada Resource Association (ARA) announced today the addition of a new member, PolySpace Technologies. Headquartered in Woburn, Mass., and maintaining over a dozen international offices and distributors, PolySpace specializes in tools that statically analyze the internal dynamics of embedded applications.

"The ARA is pleased to welcome PolySpace as a new member," said Ben Brosgol, ARA President. "One of our goals is to have Ada widely understood as the best choice for high-reliability applications. PolySpace is a well-known provider of static analysis tools in this domain, and their joining the ARA will help us get our message across."

PolySpace, Inc., President Chris Hote explained his decision to join the ARA: "The ARA offers a unique platform through which to support the ongoing development of the Ada standard, to help shape the Ada market, and to contribute to the Ada community," he said. "We want to be part of the future of Ada, and know we can do that best through the ARA."

PolySpace Technologies provides a variety of bug-detection products that help increase software development productivity and quality. Its static analysis tools have been chosen by more than 250 customers from the defense, airborne, space, automotive, ground transportation, and medical devices industries.

The Ada Resource Association (http://www.adaresource.com) is an international trade group comprising the principal vendors of Ada-related technology. The ARA promotes and publicizes Ada technology usage (http://www.adaic.org), and sponsors the ongoing development and maintenance of the Ada language standard and associated infrastructure.

The ARA's current members are AdaCore, Aonix, IBM Rational, Polyspace Technologies, Praxis Critical Systems, and SofCheck.

ARA Survey Launched

URL: http://www.adaic.org/news/survey-05.html

Ada Trade Group to Present Data at Ada Europe 2005

BELMONT, Mass. [May 3, 2005] - The Ada Resource Association (ARA), an international trade group comprising key vendors of Ada development environments and tools, today announced the availability of an on-line survey of Ada language usage. The survey is designed to quantify the global Ada software market.

Ben Brosgol, ARA president, encourages Ada developers to complete the survey. He addressed the community's possible concerns over privacy. "Data will be reported in aggregate and not associated with specific users who fill out the survey," Brosgol said. "The ARA member companies will not have access to any individual surveys."

The survey asks about individual Ada projects: their number of lines of Ada and other computer programming languages, whether they are in development or being fielded, and how they will be used. The results of the survey will be presented at Ada Europe this June in York, England, and subsequently published on the Ada News website.

The ARA members, Ada software engineers, and Ada users are all interested in seeing a "big picture" of the Ada market, according to Brosgol. "People I've met at conferences have been asking for this for years," he said. "The ARA survey should help answer some of their questions."

The Ada Resource Association promotes and publicizes Ada technology usage through the Ada News website, and sponsors the ongoing development and maintenance of the Ada language standard and associated infrastructure.

The ARA's current members are AdaCore, Aonix, IBM Rational, Polyspace Technologies, Praxis Critical Systems, and SofCheck.

Ada Semantic Interface Specification (ASIS)

ASIS for GCC 4.0

From: Martin Krischik <martin@krischik.com>
Date: Fri, 04 Feb 2005 11:23:43
Subject: ASIS for gcc 4.0.0 20050203 released
Newsgroups: comp.lang.ada

I have prepared an ASIS release.

This time I had a try with 4.0.0 - HEAD release - but remember that the source release contains the needed files for gnat-3.4.0, gnat-3.4.1, gnat-3.4.3, gnat-3.4.4, gnat-3.5.0 and gnat-4.0.0 - so the version number is only important for the binary releases.

Binary releases are available for SuSE Linux9.2i686 and Linuxx86_64.

For convenience AdaBrowse 1.02 and the semtools 1.3 are included in both binary and the source releases.

See http://gnat-asis.sourceforge.net for details.

From: Martin Krischik <martin@krischik.com>
Date: Mon, 07 Feb 2005 16:25:58
Subject: ASIS for GNAT homepage updated
Newsgroups: comp.lang.ada

I have updated the ASIS for GNAT homepage: http://gnat-asis.sourceforge.net

I hope the installation procedure is now easier to understand.

[Cf. "ASIS for GNAT: New Project and First Versions" in AUJ 25-2 (Jun 2004), p.56. -- su]

ASIS for MinGW

From: Fionn mac Cuimhaill
Date: Sat, 12 Feb 2005 04:08:14 GMT
Subject: ASIS for MinGW GNAT
Newsgroups: comp.lang.ada

Has anybody successfully installed Martin Krischik's ASIS for use by the latest MinGW GNAT (3.4.2)?

I presume that this is a non-trivial project. Otherwise, it could reasonable expected to be already be a part of MinGW.

Also, could somebody explain why ASIS is so sensitive to compiler versions?

From: Stephen Leake <stephen_leake@acm.org>
Date: 12 Feb 2005 03:58:25 -0500
Subject: Re: ASIS for MinGW GNAT
Newsgroups: comp.lang.ada

Strictly speaking, "ASIS" is a standard that doesn't change (well, it changes when the Ada standard changes). What is sensitive to compiler versions is the application library commonly called "ASIS-for-GNAT", which implements ASIS for the GNAT compiler.

In short, ASIS works by querying the compiler's internal data structures. In the case of GNAT (and probably most compilers), those data structures change (I hope only slightly !) for each release of the compiler.

The input to ASIS-for-GNAT is the "tree" file dumped by the compiler. That tree file is a serialization of the compiler's internal data structures. Since the data structures change, the format of the tree file changes, and ASIS-for-GNAT must change.

From: Fionn mac Cuimhaill
Date: Sat, 12 Feb 2005 18:05:43 GMT
Subject: Re: ASIS for MinGW GNAT
Newsgroups: comp.lang.ada

Martin Krischik wrote:

> Fionn mac Cuimhaill wrote:

>> I presume that this is a non-trivial project. Otherwise, it could reasonable expected to be already be a part of MinGW.

> I am unsure if the MinGW maintainers are interested in ASIS. However, if you are successful I make you a maintainer and you can make binary releases at sourceforge

It turned out to be simpler than I expected. I built ASIS on my Windows XP development computer.

I downloaded the MinGW Ada source, (which is v 3.4.2,) and your newest ASIS. After extracting both, I found that ASIS already had the relevant parts of the GNAT Ada compiler extracted into various subdirectories, one for each of several versions of the compiler. 3.4.2 was missing. I created the appropriate subdirectory, and, using one of the other directories as a model, I copied all of the appropriate GNAT source files into the ASIS 3.4.2 subdirectory . [...]

SPARK Training

[This information is included as examples of public Ada training courses: many are being organized regularly. For more, see also pointers in several previous AUJ issues. -- su]

URL: http://www.praxis-his.com/sparkada/training.asp

March 2005 - New dates for public SPARK courses

Dates for the next public Black Belt and UML to SPARK courses are available on the training page

Public Course Dates for 2005 - UK

Course 1 - "Software Engineering with SPARK" - 12th-15th September 2005, to be held at our offices in Bath.

The UML and RavenSPARK courses will be running on the day after this SPARK course. Both courses may be taken together in the same week.

Course 2 - "Black-Belt SPARK"

20th-22nd September 2005, to be held at our offices in Bath.

Course 3 - "High-Integrity Concurrent Software Design with RavenSPARK"

16th September 2005, to be held at our offices in Bath.

Note that this course directly follows Course 1 above. Both courses may be taken together in the same week.

Course 4 - "UML to SPARK" - Course Flyer (PDF).

16th September 2005, to be held at our offices in Bath.

Note that this course directly follows Course 1 above. Both courses may be taken together in the same week.

Courses in the USA

Praxis Critical Systems can run training courses at a customer's facilities as required. Training in the USA is also available from our partner company Pyrrhus Software.

Ada-related Tools

Most Up to Date AI302 Implementation

From: Alex R. Mosteo <alejandro@mosteo.com>
Date: Wed, 16 Feb 2005 18:50:10
Subject: Most up to date AI302 implementation?
Newsgroups: comp.lang.ada

Hello, I'm trying to get the subject thing and I'm a bit confused. At charles.tigris.org I'm directed to Mr Heaney website. There's a zip file almost a year old.

Files inside are named ai302.blah... but I detect some differences with the ones used in, for example, AWS, which I have around.

In AWS: AI302.Indefinite_Hashed_Maps;

In Heaney's website: AI302.Hashed_Indefinite_Maps;

Finally, in the CVS of tigris I see the files have the name: Ada.Containers.Indefinite_Hashed_Maps;

So I'm a bit puzzled. My first intention was to use the zip at Mr Heaney site, but now I don't know.

Or maybe the best option is to export the CVS version?

From: Matthew Heaney <mheaney@on2.com>
Date: 16 Feb 2005 09:54:54 -0800
Subject: Re: Most up to date AI302 implementation?
Newsgroups: comp.lang.ada

The latest version is always here: http://charles.tigris.org/source/browse/charles/src/ai302/

This is a CVS repository, so I recommend using your favourite front end to get the latest sources.

From: Martin Dowie <martin.dowie@btopenworld.com>
Date: Thu, 17 Feb 2005 17:26:18
Subject: Re: Most up to date AI302 implementation?
Newsgroups: comp.lang.ada

Preben Randhol wrote:

> Which would you recommend one use AI302 or charles? How is the status of AI302? Keep up the excellent work

AI302 - is going forward for approval to WG9 for Ada 2005. If you have an Ada 2005 compiler, then uses Matt's version at http://charles.tigris.org/

If you have a plain old Ada 95, then there is an upward compatible version at http://www.martin.dowie.btinternet.co.uk/

Simple Components

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Mon, 21 Feb 2005 21:22:41
Subject: Simple components v 1.9
Newsgroups: comp.lang.ada

Version 1.9 is here: http://www.dmitry-kazakov.de/ada/components.htm

1. Minor changes made in Generic_Set to support handling classes of equivalence;
2. A minor bug fix in the Ada expression parser example;
3. Changes in the documentation.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 16 Apr 2005 18:29:52
Subject: Simple components v 1.10
Newsgroups: comp.lang.ada

The new version is here: http://www.dmitry-kazakov.de/ada/components.htm

Changes:
Support for implied infix operators as in 2x + 3y;
Get_Text procedure is added to help creation of simple recursively descending parsers;
Ada expression parser bug fix (in numeric literals parsing).

[Cf. same topic in AUJ 26-1 (Mar 2005), pp.9-10. -- su]

Strings Edit

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sun, 13 Mar 2005 20:45:04
Subject: Strings edit for Ada, UTF-8 support.
Newsgroups: comp.lang.ada

The new 1.6 version of the library provides:
1. Generic UTF-8 support;
2. Conversions between Ada and UTF-8 strings;
3. Sub- and superscript integer I/O in UTF-8.

http://www.dmitry-kazakov.de/ada/strings_edit.htm

Booch Components

From: Simon Wright <simon@pushface.org>
Date: 21 Apr 2005 20:48:07
Subject: Booch Components move, new release
Newsgroups: comp.lang.ada

The Booch Components have moved to SourceForge, <http://booch95.sourceforge.net/>.

The new 20050420 release has the following (fairly minor) features:

Interface changes
  BC.Containers.Trees.AVL supports Container iteration.
  BC.Containers.Trees.Multiway.Append reworked.

A new package BC.Support.Synchronization.Debug reports the use of semaphores and monitors.

Implementation changes
The support hash table packages on which Bags, Maps and Sets rely used default subprogram parameters, which would fail if compiled in the presence of (for example) an enumeration type named Location.

BC.Support.High_Resolution_Time now includes support for PowerPC G4 (e.g., Apple PowerBook).

[Cf. same topic in AUJ 25-1 (Mar 2004), p.7. -- su]

AdaControl

From: Jean-Pierre Rosen <rosen@adalog.fr>
Date: Wed, 13 Apr 2005 16:26:31
Organization: Adalog
Subject: AdaRC: the Ada Rule Checker
Newsgroups: comp.lang.ada

Adalog is pleased to announce the availability of AdaRC, a tool that checks style and programming rules in Ada programs.

But AdaRC is more than a set of rules: it is a complete framework, intended to be easily extensible. If your favorite rule is not here, you can add it yourself! And of course, we intend to extend the number of rules in the future.

The development of AdaRC was funded by Eurocontrol. It is a mature tool that has been used to check Eurocontrol's Ada software, over 1_100_000 SLOCs.

For a complete description of AdaRC, and download, please go to Adalog's components page at http://www.adalog.fr/compo2.htm

AdaRC is distributed under the GMGPL: you are free to use it for any purpose, or to reuse any part of it in any free or proprietary software.

From: Jean-Pierre Rosen <rosen@adalog.fr>
Date: Wed, 13 Apr 2005 18:20:29
Organization: Adalog
Subject: AdaRC: the Ada Rule Checker
Newsgroups: comp.lang.ada

> Is it related to the tool of the same name made by RainCode?

Not at all, and if it is the same name, this is unfortunate.

It seems from their web site that Raincode's is called Ada Checker. If you are sure that their product is called adarc, let me know and I'll change the name.

From: Jean-Pierre Rosen <rosen@adalog.fr>
Date: Fri, 15 Apr 2005 11:25:18
Organization: Adalog
Subject: AdaRC renamed to AdaControl
Newsgroups: comp.lang.ada

Due to the clash in names with the tool provided by RainCode, I changed the name of the program to AdaControl (adactl for the command name).

BTW, it is a reminder that it has been developed by Adalog and Eurocontrol :-)

Please discard any version that you have downloaded, and get the fresh one from http://www.adalog.fr/compo2.htm

Sorry for the inconvenience.

GCC 4.0

From: Martin Krischik <krischik@users.sourceforge.net>
Date: Sat, 23 Apr 2005 12:53:07
Subject: gcc 4.0.0 released
Newsgroups: comp.lang.ada

gcc 4.0.0 has been released and here the good news for Ada users:

```
=== ACATS support ===
Generating support files... done.
Compiling support files... done.

=== ACATS tests ===
Running chapter a ...
Running chapter c2 ...
[…]
Running chapter c9 ...
Running chapter ca ...
[---]
```

```
Running chapter ce ...
Running chapter CVS ...
Running chapter cxa ...
[…]
Running chapter cxh ...
Running chapter cz ...
Running chapter d ...
Running chapter e ...
Running chapter gcc ...
Running chapter l ...

=== ACATS Summary ===
# of expected passes            2320
# of unexpected failures        0
```

From: Jeff C <jcreem@yahoo.com>
Date: Sat, 23 Apr 2005 16:43:46 -0400
Subject: Re: gcc 4.0.0 released
Newsgroups: comp.lang.ada

Steve wrote:

> Now if there were just pre-built binaries for various platforms at a nice central place... kind of like the old AdaCore public distributions.

While I agree it would be nice, things are starting to settle down. Ada is now included in the distribution media of most of the major Linux distributions...

Solaris (sparc at least) users can visit blastwave.org which has prebuilt gcc's that include Ada support.

Windows users can go the mingw.org route..

In some ways this is not as nice as the old public GNAT releases (one stop Ada shopping). The biggest problem for Windows users has been the lack of a good gdb that worked with modern Ada and mingw.

There were slightly painful ways around it but things are getting better.

Maybe all we need is a nice index/wiki write-up with links for each of the recommended solutions.

Mathpaqs

From: Gautier de Montmollin <gdemont@hotmail.com>
Date: Thu, 03 Feb 2005 22:20:58
Subject: mathpaqs re(up)loaded
Newsgroups: comp.lang.ada

Just a word to announce a severe dust-removing of my freeware Ada math toolbox (contains: algebra, matrices, probabilities, ODEs, PDEs, finite elements, fractals, multi-precision integers, ...). Description below.

http://www.mysunrise.ch/users/gdm/gsoft.htm#mathpaqs

(A) Mathematical, 100% portable, packages in Ada.

If you see "Ada 83" it also naturally compiles on Ada (95+) compilers. The list of packages below is sorted by topics. Each topic has a subdirectory. Of course there are interactions, like between Multi-precision numbers and Algebra (Euclidean_Ring_Tools).

>> Algebra <<

Euclidean_Ring_Tools: Generic package: given a type with the properties of an euclidean ring (like integers or polynomials, with 0,1,+,* and integral division), it gives the Greatest Common Divisor and the Bezout factors.

Frac: Generic package: given a type with the properties of a ring, it gives the fractions of it, with operators.

Frac.Order: Supplement of generic package 'Frac': provides an order relation from one of the ring

Frac_Euclid: Generic package: given a type with the properties of an *Euclidean* ring (with division), it gives the fractions field of it, with operators *and* reduction.

Frac_Euclid.Order: Supplement of generic package 'Frac_Euclid': provides an order relation from one of the ring

Polynomials: Generic package, creates polynomials on a field.

>Float_Polynomials: = Polynomials(float, 0.0,1.0, "-",...);

>Rationals.Polynomials: = Polynomials(rational, frac_0, frac_1, "-",...);

Rationals: Package for manipulation of rational numbers. = Frac_Euclid(integer, 0,1, "-","+","-","*","/");

>Rationals_Order: = Rationals.Order ("<");

>> Numerics <<

G_Matrices: Generic simple matrix package, with matrix-matrix, matrix-vector, vector operations

G_FEK: Parts of the Finite Element Kernel from M. Bercovier (original in Fortran);

Computes values and derivatives of elementary functions given the element's geometry.

Available so far:

1D: L2 (linear)
2D: Q4 (linear), Q9 (quadratic)
3D: B27 (quadratic)

Generic -> can be instantiated for any precision!

ConjGrad: Fast (Bi)Conjugate Gradient iterative methods for solving Ax=b. Generic -> applies to the matrix storage of your choice (e.g. sparse) !

>> Multi-precision integers <<

Multi_precision_integers: Multiple precision integers package

Pure Ada 83 (compiles on both DEC Ada and GNAT)

Multi_precision_integers_IO: Text_IO, for multi-precision integers

>> (Pseudo-) Random number generation <<

Box_Muller: Pseudo-random number generation with normal distribution, by the Box-Muller method

F_Random: Simple random generator package

(Pure Ada [.adb] and DEC/Compaq/HP [.dec] Ada 83 bodies)

Finite_distributed_random: random generation on any discrete, finite, type

U_Rand: Standalone random generator (Pure Ada 83)

>> Sparse matrices <<

Sparse: Sparse matrix package (Pure Ada 83); uses SparseB

SparseB: Low-level vector operations for Sparse package

(Pure Ada 83 [.adb] body and a DEC/Compaq/HP [.dec] body mapped to BLAS)

(B): Programs, tests, demos using the previously cited packages

----

______Linear Algebra:

Test_Sparse: Test of Sparse and ConjGrad packages.

_________________ODE:

Phases.adb: Phase diagram and vector fields for oscillators (2)

Champ_Vt.adb: Vector fields drawing (1)

_________________PDE:

Diffchal.adb: Solution of heat equation by finite differences ( )

____________Fractals:

Biomorph.adb: <<biomorphe>> fractals (cow skin) (1)

FracDesi.adb: Fractal lines design (2)

Koch_Haar.adb: Koch flake, Haar wavelet and others 1D fractals (1)

Henon.adb: Hénon orbitals (fractal strange attractors) (1)

__Random simulations:

Sim_Alea.adb: Simulation of random variables ( )

Porscher.adb: Probability "paradox" (in French) ( )

EDS_1.adb: Simulation of solutions of stochastic (1)

differential equations (SDE)

TestURan.adb: Test of U_Rand standalone random generator ( )

_____Multi-precision:

Test_Int.adb: Test of Multi_precision_integers package ( )

( ) Yes, it's standalone Ada !
(1) uses Graph package (WINGRAPH.ZIP or DOSGRAPH.ZIP) See page: http://www.mysunrise.ch/users/gdm/graph.htm

(2) uses DOS-Graph, Mouse packages (DOSGRAPH.ZIP, DOS_PAQS.ZIP)

See list: http://www.mysunrise.ch/users/gdm/locaux.htm

FFT Ada implementations

From: Jeff C <jcreem@yahoo.com>
Date: Wed, 13 Apr 2005 07:22:29 -0400
Subject: Re: FFT
Newsgroups: comp.lang.ada

> Do we have a good FFT (Fast Fourier Transform) implementation in Ada ?

I don't know of any great publicly available ones. There is a binding to FFTW http://privatewww.essex.ac.uk/~sjs/fftw_ada/fftwa.html but the binding is released as pure GPL (not GMGPL) which would be somewhat ok since I think FFTW itself is also pure GPL. However, I believe you can buy a proprietary license for the FFTW library from MIT.

I don't think you can buy a non-pure GPL license for the Ada binding.

Documentation claims it is Copyright Stephen J. Sangwine but I suspect this is one of those things that is encumbered by some sort of university interest in it as well. If this is the same university as PNG_IO then I'd look elsewhere if you need something non GPL. Several years ago we tried to purchase a license for PNG_IO and after several months our purchasing department still was unable to get the office that dealt with this stuff to complete the deal. Eventually the schedule moved far enough along that we had to come up with an alternate approach.

The FFTW library itself is pretty nice and complete so if you have no issues with the use of a GPL library/binding take a look.

From: Tom Moran <tmoran@acm.org>
Date: Wed, 13 Apr 2005 13:16:52 -0500
Subject: Re: FFT
Newsgroups: comp.lang.ada

In addition to FFTW, there's also http://cr.yp.to/djbfft.html which doesn't appear to have any restrictions. It appears to be for very fast computation of powers of two FFTs. It's in very nasty C, though. I have here a package fft_pack with Glassman's algorithm for arbitrary N, but I can't seem to track its provenance. If speed isn't an issue, it's quite convenient. Numerical Recipes in Fortran (etc) has source and discussion of various FFT situations.

Units of Measurement

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 19 Mar 2005 11:23:38
Subject: Units of measurement for Ada v1.7
Newsgroups: comp.lang.ada

The new version is here:

http://www.dmitry-kazakov.de/ada/units.htm

Changes:
1. An unshifted dimensionless measure can now be mixed with a shifted one in * and /;
2. UTF-8 encoding support;
3. Minor bug fix in units converter examples: Unit_Error is now caught;
4. The package Units has the new child Units.Edit. The function Image is moved there;
5. Units converter for GTK is now statically linked for i686 target;
6. Units converter for Windows supports Unicode.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 23 Apr 2005 21:19:16
Subject: Units of measurement v 1.8
Newsgroups: comp.lang.ada

Here it is: http://www.dmitry-kazakov.de/ada/units.htm

Changes made:
Get_Unit procedure was added for input pure measurement units (expression terms) rather than measures (arbitrary dimensioned expression), which can be useful for building customized parsers; To_Measure was added for explicit number to measure conversion; Some documentation bugs were fixed.

[Cf. "Physical Units Checking in Ada" in AUJ 25-2 (Jun 2004), pp.47-48. -- su]

Fuzzy Sets for Ada

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Sat, 16 Apr 2005 18:33:59
Subject: Fuzzy sets for Ada v 3.9
Newsgroups: comp.lang.ada

The new version is here: http://www.dmitry-kazakov.de/ada/fuzzy.htm

Changes made:
1. Linguistic variable conversions to number and interval were added;
2. Bug fix in implementation of binary operations on fuzzy variables;
3. Improved documentation;
4. Based on the version 1.10 of simple components.

[Cf. same topic in AUJ 25.4 (Dec 2004), p.186. -- su]

OpenALada

From: Aurele <aurele.vitali@gmail.com>
Date: 4 Mar 2005 06:56:35 -0800
Subject: OpenALada
Newsgroups: comp.lang.ada

OpenALada and OpenALdemo v1.1 have been updated and tested with ObjectAda v7.2.2 and GNAT v3.15p. www.OpenALada.com

[Cf. "Bindings for OpenAL (Open Audio Library)" in AUJ 26-1 (Mar 2005), p.11. -- su]

asound - Ada Sound Environment

From: Adrian Knoth <adi@thur.de>
Date: 12 Mar 2005 21:46:49 GMT
Subject: Announce: asound - Ada sound environment
Newsgroups: comp.lang.ada

I'm glad to release the first version of asound, the generic Ada environment for audio-related tasks.

The idea of asound arose some years ago and perhaps Preben Randhol did something on his own, I don't know...

Currently, asound isn't a generic audio library at all, but it could be one ;)

The original intention was to write a binding to Ogg/Vorbis and Preben proposed to start with libao. I haven't thought about it for years, but last week I decided to implement it.

Actually asound is only a binding to libao, and not even a complete one. It can play samples and this is more than I need, because I don't need it ;)

Perhaps someone is interested in extending it. I do not think that I'll have the time for great improvements within the next N*six months, so whoever wants to go on coding, please do it ;)

You can download it here: http://adi.thur.de/?show=asound

Drawplex

From: Marius Amado Alves <amado.alves@netcabo.pt>
Date: Mon, 14 Mar 2005 15:52:44
Subject: Announce: Drawplex
Newsgroups: comp.lang.ada

I'm pleased to announce the first release of Drawplex, a 100% Ada library for drawing on the complex plane. http://softdevelcoop.org/software/drawplex

AdaGPGME 1.0.2 - Binding to "GNUPG Made Easy"

From: Andreas Almroth <andreas@almroth.com>
Newsgroups: comp.lang.ada
Subject: Update to AdaGPGME and libgpg-error
Date: Tue, 29 Mar 2005 19:25:19

For those of you interested, I have updated the Ada 95 bindings to:
* GPGME 1.0.2 (GnuPG Made Easy C API)
* libgpg-error 1.0 (common error message library for GnuPG components)

A few of the test programs have been ported to Ada 95 to verify that the bindings work, or at least partly. Change the makefile to correspond to your environment.

The bindings are tested with GCC 3.4.3 on Solaris, but it should be possible to compile them on other platforms.

You will, obviously, need to install GnuPG 1.2.2+, libgpg-error 1.0 and gpgme 1.0.2 first.

The bindings can be found at:
AdaGPGME - http://www.almroth.com/gpgme/index.html

libgpg-error - http://www.almroth.com/libgpgerror.html

Any suggestions, comments and bugs are welcome and should be sent to andreas at almroth dot com.

For more information on GPGME and libgpg-error, please visit: http://www.gnupg.org/

[Cf. "AdaGPGME 0.4.1 - Binding to GNUPG Made Easy" in AUJ 24-3 (Sep 2003), p.143. -- su]

From: Andreas Almroth <andreas@almroth.com>
Date: Wed, 30 Mar 2005 08:13:22
Subject: Re: Update to AdaGPGME and libgpg-error
Newsgroups: comp.lang.ada

> One note. While I am a big fan of the GMGPL approach, it is not really clear that it is entirely helpful in this case since GNUPG itself appears to be GPL without exception...Not suggesting you need to change the license binding but people using it (as always) need to understand all of the license issues that are involved.

Regarding the license, yes, it may not be entirely clear, I agree fully. GnuPG is GPL only, GPGME is LGPL as its design is not limited to GnuPG, and in the future may include other backends that may use other licenses. I believe that could have been a reason why they choose LGPL. I use GMGPL for most of my work that I publish. I like the GMGPL, it is an approved license, and is based on GPL with the exception that any code instantiating generics or using parts does not necessarily make the final product GPL/GMGPL. However copyrights are still in place. GMGPL differs from LGPL, but to my understanding, not so much in reality.

In this specific scenario, it is hard to say where to draw the line, as GPGME, AFAIK, does not link to GnuPG, but merely calls the executable with the necessary arguments. AdaGPGME is then linking to GPGME and any resulting products would be based on LGPL, which means they can have other (even non-free) licenses. The GMGPL would not be in the way really.

Well, I'm not a legal eagle, but I don't see that GMGPL in any way is limiting/infringing LGPL.

Perhaps I should add a note to the README file...

Player/Stage Ada binding

From: Alex R. Mosteo <alejandro@mosteo.com>
Newsgroups: comp.lang.ada
Subject: Ada binding for Player
Date: Wed, 30 Mar 2005 11:34:26

I'm developing a partial binding to the libplayerc library for use in Ada programs. At present it includes interfacing to the connection, position, laser, localize, planner and blobfinder facilities.

It's available for download at http://ada-player.sf.net

Player/Stage/Gazebo is a control / simulation platform for robotics. It allows you to develop and test control algorithms over simulated and real robots using the same interface. Find more about it at http://playerstage.sf.net

AdaSQLBase

From: Andreas Almroth <andreas@almroth.com>
Date: Tue, 01 Feb 2005 21:45:55
Subject: AdaSQLBase binding
Newsgroups: comp.lang.ada

AdaSQLBase is a thin binding to the C API for Gupta's SQLBase database engine. SQLBase is a relational database that can be embedded with applications on Windows and Linux* platforms.

The binding is released under GMGPL.

The binding package can be found at: http://www.almroth.com/adasqlbase.html

As this is only a thin binding, please find more information on SQLBase at: http://www.guptaworldwide.com/Products/SQLBase.aspx

A simple test suite program is provided to test functionality, but also in a basic way show how to use the binding.

* This binding has been tested with GNAT 3.15p and GCC 3.4.2 (MINGW) on the Windows platform. Linux platform should theoretically be easy to port to, as it most likely only necessary to change the pragma import from STDCALL to C.

Packages for Text Filtering

From: Martin Krischik <martin@krischik.com>
Date: Fri, 11 Feb 2005 14:21:57
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

> I need to filter a resultfile produced by a program (only executable available) so I can get the different results I'm interested in. I need to do it in Ada (so no hints about python, perl etc ;-)) but I thought that there must be some packages (besides the GNAT packages) available to ease the filtering.

AdaCL (http://adacl.sourceforge.net) has a powerfully - yet easy to use - text filter library. And it's a class library - if what you need is missing you can extend it.

Look at the sarDO source to see how it works:
http://adacl.sourceforge.net/html/sarDo-CommandLine__adb.htm

From: Martin Krischik <martin@krischik.com>
Date: Fri, 11 Feb 2005 15:25:08
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

I forgot to mention: the I/O modules are also classes and can be replaced with specialised versions.

Currently available:
Textfile_1 => Textfile_2 where both files can be the same.

Textfile => Standart_Output as part of the CGI package.

From: Dmitry A. Kazakov <mailbox@dmitry-kazakov.de>
Date: Fri, 11 Feb 2005 14:25:41
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

OK, besides GNAT's spitbol:
http://www.dmitry-kazakov.de/ada/components.htm

- this project contains parsers for elaborated infix expressions of any kind (in case your output has something like 2+5*(3-6))
http://www.dmitry-kazakov.de/ada/strings_edit.htm

- this is a set of simple tools for parsing and formatting strings
http://www.dmitry-kazakov.de/match/match.htm

- this is pattern matching, though written in K&R/ANSI C, it has Ada interface.

From: Marius Amado Alves <amado.alves@netcabo.pt>
Date: Fri, 11 Feb 2005 11:56:38
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

GNAT includes nice string pattern matching packages. The best for me is GNAT.Spitbol.

There is also Open_Token out there.

(I should start collecting referral commissions :-)

From: Jeffrey Carter <jrcarter@acm.org>
Date: Sat, 12 Feb 2005 00:51:45 GMT
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

The PragmAda Reusable Components include regular expression matching. There's an example program, strmsub, that's a stream editor and was pretty easy to create.

http://home.earthlink.net/~jrcarter010/pragmarc.htm

From: Steve <steved94@comcast.net>
Date: Sat, 12 Feb 2005 17:35:20 -0800
Subject: Re: Package for text filtering?
Newsgroups: comp.lang.ada

I have used Aflex and Ayacc. I had an old Ada 83 version I had to doctor up to work with GNAT (several years ago). It looks like there are versions available on AdaPower:
http://www.adapower.com/index.php?Command=Class&ClassID=Utilities&Title=Ada+Utilities

There is a little bit of a learning curve, but they are powerful tools.

Ada-related Products

AdaCore - GPS 3.0

URL: http://www.adacore.com/pressroom_19.php

AdaCore Revs Up IDE

Powerful, Simple-to-use GNAT Programming Studio Streamlines Software Development, Supports Wide Range of Operating Systems

New York - May 23, 2005

AdaCore today introduced GPS 3.0, a highly upgraded version of the company's advanced Integrated Development Environment (IDE) that is already the IDE of choice for GNAT Pro and Ada developers. GPS (GNAT Programming Studio) 3.0 is aimed at streamlining Ada and multi-language software development from the initial coding stage through testing, debugging, system integration, and maintenance.

GPS 3.0 offers advanced features, such as multi-language support (including Ada, C, and C++), and support on a wide range of host environments for both native and cross-development platforms, including UNIX, Windows and GNU/Linux. An intuitive, unified visual interface, identical across all platforms, serves as a control panel to access tools from AdaCore's GNAT Pro Ada development environment as well as from third parties, easing both development and maintenance. As a result, GPS 3.0 is particularly suited for large, complex systems requiring tool chain integration, ease of use, user customization, and code navigation/analysis tools.

"Most embedded IDEs are targeted to one operating system platform, which limits both extensibility and adaptability," said Robert Dewar, president of AdaCore. "GPS 3.0 not only supports a wide variety of commercial platforms, but also is adaptable enough to be used with proprietary operating systems. Our platform-independent visual interface is also very easy to learn and use, which increases programmer productivity and ultimately speeds time-to-market."

GPS 3.0 provides many new improvements from previous releases, including:

* Automatic documentation generation from Ada sources
* Support for remote debugging / compilation
* Support for inter-process communication between GPS and external tools
* New visual comparison tool
* Visualization of Ada metrics
* Outline view, dynamically showing the code structure in the current editor
* Improved project editing, including support for library projects

As with all GNAT Pro components, GPS 3.0 is distributed with full source code and is backed by AdaCore's rapid and expert online support.

About GPS
GPS is a powerful IDE written in Ada, based on the GtkAda toolkit. GPS's extensive source-code navigation and analysis tools can generate a broad range of useful information, including call graphs, source dependencies, project organization, and complexity metrics. It also provides support for configuration management through an interface to third-party Version Control Systems, and supports a variety of platforms, including Alpha Tru64, Altix Linux, MIPS-IRIX, PA-RISC HP-UX, SPARC Solaris, x86 GNU Linux, x86 Solaris, and x86 Windows. GPS is highly extensible; a simple scripting approach enables additional tool integration. It is also tailorable, allowing programmers to specialize various aspects of the program's appearance in the editor for a user-specified look and feel.

Pricing and Availability
GPS 3.0 is part of the GNAT Pro toolset available today from AdaCore. Please contact AdaCore for the latest information on pricing and supported configurations. (sales@adacore.com)

About AdaCore
Founded in 1994, AdaCore is the leading provider of commercial, open-source software solutions for Ada, a modern programming language designed for large, long-lived applications where reliability, efficiency and safety are absolutely critical. AdaCore's flagship product is GNAT Pro, the commercial-grade open-source Ada development environment, which comes with expert online support and is available on more platforms than any other Ada technology. AdaCore has customers worldwide; see http://www.adacore.com/customers.php for more information.

Use of Ada and GNAT Pro continues to grow in high-integrity and safety-critical applications, including commercial and defence aircraft avionics, air traffic control, railroad systems, financial services and medical devices. AdaCore has North American headquarters in New York and European headquarters in Paris.

[Cf. "Public Release of GNAT Programming System IDE (GPS)" in AUJ 26-1 (Mar 2005), p.13 and "AdaCore - GPS 2.1.0" in AUJ 25-4 (Dec 2004), pp.193-194. -- su]

AdaCore - Ada Answers

URL: http://www.adacore.com/pressroom_18.php

AdaCore Shines Spotlight on Ada for Broad Range of Leading-Edge Applications

Salt Lake City, Systems & Software Technology Conference - April 18th, 2005

AdaCore today introduced Ada Answers, a web portal aimed at providing managers and software developers with a comprehensive knowledge base about Ada, the programming language most often used to implement high-integrity, safety-critical and real-time systems.

The Ada Answers web portal is dedicated to keeping developers and project managers informed about Ada and its forthcoming Ada 2005 revision, showcasing the strengths and benefits of this extremely powerful programming language. The site includes examples of companies and organizations that are successfully using Ada, highlighting some of these companies in a growing collection of video interviews. Also on video is a series of university lectures and conference presentations given by some of the foremost experts on the language, as well as an up-to-date list of Ada materials, resources and web links.

"As the need for robust and reliable software systems rapidly expands, Ada continues to prove itself an excellent answer for many of today's largest and most complex programming challenges," said AdaCore president Robert Dewar. "Ada Answers helps explain Ada's distinct qualities and technical features and how they can translate into bottom-line business benefits."

More than any other language, Ada was specifically designed to address issues of testing, quality assurance, functionality upgrades, platform portability, multi-language support, and similar "back end" activities. Its consistent software engineering principles make Ada, the world's first object-oriented programming language, intuitive and easy to learn. Its expressive features and strong checking make Ada a "think first, code later" discipline that translates into fewer bugs and higher productivity.

"Ada has always been an attractive choice where reliability has been the overriding requirement," says Dewar. "Historically this has been in the defence and aerospace industry, but Ada is also increasingly used commercially in fields like avionics, power plants, transportation systems, communications, medical instruments, and finance. Ada includes numerous built-in features specifically optimized for financial data."

AdaCore will showcase videos from Ada Answers at SSTC, Booth 431. The companies and applications to be highlighted include:

* JEOL - Nuclear magnetic resonance for analysis of molecular structures
* New Trade Research - Automated securities trading systems
* Philips Semiconductor (ITEC Division) - Equipment used in the assembly of discrete semiconductors
* Vienna University, Austria - Astrophysics applications involving massively parallel computing

[Cf. same topic in AUJ 25-4 (Dec 2004), p.193. -- su]

AdaCore - G++/GNAT Pro Joint Edition

URL: http://www.adacore.com/pressroom_17.php

AdaCore and CodeSourcery Join Forces to Create Open-Source Integrated Ada and C++ Development System

Salt Lake City, Systems & Software Technology Conference - April 18th, 2005

AdaCore (New York) and CodeSourcery (Granite Bay, CA) have partnered to create the G++/GNAT Pro Joint Edition, the first open-source development environment for native and embedded applications that use both Ada and C++ programming languages. The G++/GNAT Pro Joint Edition is based on a combination of several GNU-based, open-source technologies - AdaCore's GNAT Pro Ada toolsuite, and CodeSourcery's G++ Pro compiler. The result is a full-featured, modern IDE that includes source editors, configuration management facilities, source-level debuggers, source navigation and queries, and user extensibility. It is ideal for complex, mission-critical systems that must efficiently unite software developed in Ada and C++.

"Over the past decade, modern defense systems have created large and sophisticated bodies of software in a variety of high-level languages, including Ada and C++," said Robert Dewar, president of AdaCore. "The partnership between AdaCore and CodeSourcery provides defence contractors with an open-source, feature-rich development platform for a broad range of native and cross-development scenarios."

The open-source principles that underlie the G++/GNAT Pro Joint Edition allow third-party and proprietary tools to be used from within the IDE framework, while maintaining compliance with recognized industry standards. As a result, it is particularly well- suited for long-term defence programs. The U.S. Army's Future Combat System (FCS) and the U.S. Navy's DD(X) System have publicly recognized the advantages of, and are committed to, using open-source software.

"Today's announcement reaffirms our commitment to the continued development of GNU, Ada, and C++," said Mark Mitchell, founder and chief sourcerer of CodeSourcery. "By entering into strategic partnerships with dedicated language and toolset experts, such as AdaCore, we can provide our customers with the highest quality support, and can reduce compiler and development risks throughout a program's entire lifecycle."

About CodeSourcery
CodeSourcery, LLC was founded in 1997 to provide high-quality tools and consulting services that improve the productivity of software developers. The company's uncompromising standards of engineering excellence are the fundamental framework behind its software design, software engineering, and open-source project management products and services. CodeSourcery, whose customers include Fortune 500 companies, the United States government, and other leaders in the computer industry, is a privately held company registered in the state of California. www.codesourcery.com

Aonix - ObjectAda for PikeOS

URL: http://www.aonix.com/pr_03.07.05b.html

Aonix Delivers Real-Time Java and Ada Applications for PikeOS

Embedded Systems Conference, San Francisco, CA, March 7, 2005

SYSGO and AONIX today announced a cooperation to provide the PERC VM and ObjectAda environments for PikeOS. The implementation makes full use of PikeOS' multi-OS capabilities, thus allowing real-time Java, Ada and traditional Linux applications to run reliably side by side in different partitions.

Aonix, a provider of complete solutions for safety- and mission-critical applications, is well known for its Ada and PERC products. PERC, a clean-room Virtual Machine (VM), supports the execution of Java platform applications in embedded systems, without sacrificing the integrity, performance, or real-time benefits of legacy approaches.

PikeOS incorporates software partitioning, enabling developers to run multiple operating system APIs on top of a microkernel, forming so-called "OS-personalities." Aonix's PERC VM and Ada run-time environments will add new personalities to go along with Linux, POSIX and OSEK personalities that PikeOS already supports. The PERC solution will be available Q2 2005 with the Ada solution by the end of the year.

"We are proud to join forces with Aonix," states Detlev Schaadt, CTO of SYSGO. "Their reputation in our market is outstanding, and their Ada and PERC implementations fit perfectly into our technology. They have earned a solid reputation for developing innovative and proven solutions that matches our own product approach."

"Both companies have considerable expertise in developing safety-critical software," comments Jacques Brygier, Aonix' VP of Marketing. "SYSGO has a long record of delivering quality solutions and strong technical support to their customers. By combining our technologies, we offer some very solid product offerings to our customers."

About SYSGO AG
SYSGO provides software solutions for Industrial Systems and Embedded Devices. The company's product and service offerings focus on the most important building blocks in any successful project-the low-level system software, such as Firmware, Operating Systems, and Device Drivers. Founded in 1991, SYSGO became a leading company for safety-critical system software and Embedded Linux in Europe, growing to 70 employees in six offices in Europe. SYSGO's new product-PikeOS-represents the companies' experience with both software certification and embedded Linux. SYSGO boasts OEM customers like Siemens, DaimlerChrysler, Rockwell-Collins, EADS and Raytheon and hardware vendors such as Motorola, AMCC and Kontron.

About Aonix
Aonix offers mission- and safety-critical solutions primarily to the military and aerospace, telecommunications and transportation-related industries. Aonix delivers the leading high-reliability, real-time embedded Java solution deployed today and has the largest number of certified Ada applications at the highest level of criticality. Our unique modeling solution features UML 2.0 profiles and MDA tailored for the mission- and safety-critical space. Aonix products include PERC®, RAVEN, and Ameos. Headquartered in San Diego, CA and Paris, France, Aonix operates sales offices throughout North America and Europe in addition to offering a network of international distributors. For more information, visit www.aonix.com.

Green Hills Software - Ada, Ravenscar Ada and SPARK for INTEGRITY 178B RTOS

URL: http://www.ghs.com/news/20050502_embedded.html

Green Hills Software Announces Embedded C++ for the INTEGRITY-178B Safety Critical Operating System

SANTA BARBARA, CA, - May 3, 2005 -Green Hills Software, Inc., the technology leader in embedded software development tools and real-time operating systems (RTOS), today announced the availability of DO-178B Level A certifiable Embedded C++ (EC++) for its safety-critical INTEGRITY 178B RTOS. Green Hills Software is first to offer safety critical developers a choice of C, C++ and Ada, all developed and supported by a single vendor and integrated into a single, multiple language development environment-Green Hills Software's MULTI. Additionally, Green Hills Software supports specialized versions of these languages for safety critical development-MISRA C, Embedded C++, SPARK Ada and Ravenscar Ada.

"The addition of EC++ to the languages already supported for the INTEGRITY-178B operating system provides a powerful new capability for the development of safety-critical applications," said Greg Gicca, director of product marketing for safety critical products at Green Hills Software. "INTEGRITY-178B now supports the development of a single system with a mix of several languages. This allows developers to select the language best suited for their application development needs and designated safety level."

Multiple applications can now be developed in different languages and deployed in separate INTEGRITY-178B partitions running on the same computer. A high-integrity DO-178B Level A application might be developed using EC++. An application with lower criticality might be developed using full Ada and deployed in its own separate partition.

With EC++, object oriented programming capabilities benefit from improved efficiency over full ANSI C++. EC++ removes ANSI C++ features that are not typically used in embedded or safety critical systems development, thereby generating much smaller and faster programs. EC++ improvements range from 30-50 percent in code size and run-time efficiency over full ANSI C++.

About INTEGRITY-178B
INTEGRITY-178B is a powerful, safety-critical, DO-178B Level A certified RTOS. It offers full time and memory partitioning as well as an ARINC-653-1 APEX interface. Three programming languages are now available to safety-critical developers supporting their development needs, and INTEGRITY-178B provides the protection between applications with its full partitioning support.

Availability
DO-178B Level A certifiable EC++ support for the INTEGRITY-178B RTOS is available now.

About Green Hills Software
Founded in 1982, Green Hills Software, Inc. is the technology leader for real-time operating systems and software development tools for 32- and 64-bit embedded systems. Our royalty-free INTEGRITY® RTOS, velOSity microkernel, compilers, MULTI® and AdaMULTI Integrated Development Environments and TimeMachine debugger offer a complete development solution that addresses both deeply embedded and high-reliability applications. Green Hills Software is headquartered in Santa Barbara, CA, with European headquarters in the United Kingdom. Visit Green Hills Software on the web at www.ghs.com.

RainCode - RainCode Engine free of charge

URL: http://www.raincode.com/newslfeb05.html

February 2005 - Major Change in License Agreement

As from February 2005, the RainCode Engine for Ada, C, and COBOL under Win9X, NT, and Unixes are available FREE of charge.

These tools are used to perform automatic source code analysis or transformation tasks, such as:

* implementation of coding guidelines;
* development of specific code transformations and restructuring;
* source code migrations;
* Etc.

These freely available tools are FULLY FUNCTIONAL and offered freely WITHOUT ANY LIMITATIONS. All is needed to start using the RainCode Engine is to go through a no-nonsense license agreement.

Various forms of maintenance contracts are available to support you in your RainCode projects, but the financial aspect of the license itself has turned to a non-issue.

The ambition of RainCode's new strategy to offer free licenses of its main product is to distribute the RainCode Engine more widely, and to give a much wider base of users the opportunity to benefit from RainCode's very fine code-analysis capability and great ability to manipulate and transform source code in an intelligent way.

To get your free RaiCode Engine License Log on RainCode Online: http://www.raincode.com/online

On the home page, choose: "RainCode product line". Then click on "Downloads" to find the RainCode Engine of your choice.

URL: http://www.raincode.com/adaengine.html

RainCode parses Ada 83 and Ada 95 sources. Typical uses of RainCode include:

* Quality assessment for outsourced work. When a project is submitted to an external company, precise coding guidelines can be defined formally as part of the assignment. Such guidelines can then be checked continuously or at delivery time on the entire source code, rather than relying on random sampling. Besides, an exhaustive analysis provides a number of metrics that quantify the degree of compliance, which gives you a precise estimation of the amount of work required to achieve full compliance, rather than a binary answer.

* In a migration project, performing large amounts of updates automatically, identifying the places that must be updated manually, predicting the total effort required for this manual update, and monitoring the progress of the entire migration. Examples of such migrations cover the replacement of a compiler by another, the change of platforms, TP monitor or database.

* Integration in a compilation chain to ensure that every source code is compliant before it can even be tested. The standard language within an organisation or a project moves from plain Ada to Ada restricted by the coding guidelines.

* Implementation of a complete metrics and quality strategy. This includes monitoring the quality and maintainability of all components continuously, predicting maintenance costs, detecting abnormal entropy within systems before it becomes a maintenance issue. Metrics, such as cyclomatic complexity of functions, can be performed on a daily basis.

SofCheck - European distribution of SofCheck's technology

URL: http://www.sofcheck.com/news/praxispressrelease.html

SofCheck Expands International Reach Through Distribution Agreement with Praxis High Integrity Systems

Praxis HIS to bring SofCheck's technology to safety critical markets in the U.K. and Europe.

BURLINGTON, Mass. (March 28, 2005) - SofCheck, Inc., an independent provider of software analysis and verification technology, has signed a software distribution license agreement with Praxis High Integrity Systems (Praxis HIS), a provider of high integrity systems engineering solutions. Under the agreement, Praxis HIS will offer SofCheck's technology to clients in the U.K. and Europe, across a range of safety critical markets, including, aerospace and defense, automotive, rail, nuclear, telecommunications and finance.

"We were looking for a distribution partner that could not only expand our reach, but that also shared our market focus and dedication to quality," said Stephen F. Clairmont, senior vice president, Sales & Strategic Alliances, SofCheck. "Praxis HIS has earned a reputation as a provider of the highest quality IT consulting and engineering solutions to the safety critical markets that can benefit from SofCheck technology. We are very pleased to be working with them, and anticipate a rewarding relationship."

"Our goal is always to deliver the right solution at the right quality," said Rod Chapman, Products Manager at Praxis High Integrity Systems. "We are delighted to be adding SofCheck's technology to the range of tools we can apply to our client solutions."

About Praxis HIS
Praxis High Integrity Systems is part of the Altran Group, a recognized global leader in innovative engineering. Praxis HIS leads the High Technology Engineering service line for Altran's U.S. and European business. The company delivers expertise into client projects, as well as Intellectual Property in the form of software products, training courses and tool templates. Praxis HIS has industry-leading expertise and capability in Software Engineering, Systems Engineering (including Requirements Engineering), Project/Operational Risk Management, Programme Delivery Management, Safety Engineering, Security Engineering and Human Factors. Visit www.praxis-his.com for more information about the company and its range of services.

About SofCheck
Founded in 2002, SofCheck develops technology that enables software developers and IT organizations to detect and eliminate bugs that can cause crashes or numeric overflows earlier in the development cycle, improving overall software quality and reducing time-to-market. SofCheck's flagship product, SofCheck Inspector, is a complement to traditional runtime testing tools, employing advanced static error detection technology and pushbutton convenience to find lurking defects in software. SofCheck is a privately held company whose clients include: Raytheon, Northrop Grumman and United Technologies. To learn more, visit www.sofcheck.com, or contact SofCheck by phone +1 (781) 750-8068, Fax +1 (781) 750-8064 or E-mail info@sofcheck.com.

McKae Technologies - DTraq

From: Marc A. Criley <mc@mckae.com>
Subject: Announce: DTraq 0.986a now Available
Date: Mon, 18 Apr 2005 15:22:35 -0500
Newsgroups: comp.lang.ada

McKae Technologies announces the release of version 0.986a of DTraq, an Ada 95 data logging and review tool.

DTraq is a data logging and playback debugging tool providing near realtime data logging and analysis to aid debugging and validation. Captured, or 'tapped' data from a program can be viewed live while the program is running or, since it is being logged to a file, played back or printed out later for off-line review and analysis.

DTraq differs from other logging and playback tools in that no data layout maps or byte interpretations or "data dumpers" need to be manually created. Nor is the application responsible for converting the raw binary data to text form before logging it. DTraq handles all conversion automatically by scanning the application's source code, identifying tapped data items, and extracting the information it needs to properly convert and display the logged items-simple scalar items as well as arrays and records. When the layout of data items change, rescanning automatically picks up the changes.

DTraq requires GNAT 3.15p due to its reliance on the Ada Semantic Interface Specification (ASIS) and has been validated on Red Hat 9 Linux.

Source and executables are available on the DTraq home page: http://www.mckae.com/dtraq.html, along with the comprehensive and up-to-date user manual -- http://www.mckae.com/dtq_common/DTraq.pdf.

DTraq usage is described, and screenshots provided, starting at http://www.mckae.com/dtq_usage/tapping.html.

Updates to DTraq 0.986a (versus 0.986):

- Added a Data Item Reviewing capability to live data monitoring to permit quick reviewing of recently logged and displayed data items. This eliminates the need to stow the current logfile, suspend logging, and go into playback mode to just look back at a recently received item.

- Reverted the DTraq.Tap tap ID type back to the standard Positive type.

- Cleaned up version handling.

[Cf. same topic in AUJ 25-4 (Dec 2004), p.196. -- su]

McKae Technologies - XPath In Ada (XIA)

From: Marc A. Criley <mc@mckae.com>
Date: Sun, 06 Mar 2005 13:38:28 GMT
Subject: Announce: XIA 1.00 Now Available
Newsgroups: comp.lang.ada

Version 1.00 of XIA (XPath In Ada) is now available on the McKae Technologies website at www.mckae.com/xia.html.

This version of XIA completes the initial Ada implementation and release of the XPath 1.0 specification.

Bug reports and suggestions for improvement are welcome, with optimizations and improvements to be incorporated into subsequent releases. Please see the XIA page at mckae.com for contact information.

An example driver, test_xpath, and a test script that submits over 160 queries accompanies the distribution in the 'test' subdirectory.

From: Marc A. Criley <mc@mckae.com>
Date: Thu, 10 Mar 2005 08:24:42 -0600
Subject: Re: Announce: XIA 1.00 Now Available
Newsgroups: comp.lang.ada

Okay, so you now have a native Ada implementation for XPath querying, so what?

Well, Ada programmers in general like to work with software written in Ada so they can look at it and more easily see what's going on, hence writing XIA in Ada.

What does XPath buy me?

If you're working with XML documents, there are two standard approaches for interacting with such documents, SAX and DOM.

SAX is oriented towards stream-oriented processing, meaning that you process the contents of a document as it streams through your application, there's no innate retention of the content once it's been processed. This is good for doing things like transformations and especially when working with documents of very large size.

DOM is tree-oriented access to the document, where the entire XML document is loaded into an in-memory tree, and can now be walked through, randomly accessed, manipulated, and even easily written back out to a document.

Thanks for the info, but what does XPath buy me?

XPath is a standard approach for selecting nodes out of an XML document. Instead of you having to write your own code to go and search for nodes (elements and/or attributes) of interest, you write an XPath query and bang it up against the document--and back comes a list of nodes that meet the query's criteria.

XPath can work with both DOM and SAX approaches to XML document processing, but SAX, being a 1-way stream oriented mechanism, means that you either have to restrict yourself to an XPath subset (eliminating queries that involve elements that would have already gone past), or do some gnarly query preprocessing, along with maintenance and pruning of lists of potential node matches, etc.

Due to the latter complications of dealing with SAX interaction, XIA works strictly with the DOM model of XML document access.

[Cf. "XIA - XPath In Ada" in AUJ 26-1 (Mar 2005), p.11. -- su]

McKae Technologies - XML EZ Out

From: Marc A. Criley <mc@mckae.com>
Date: Fri, 08 Apr 2005 13:57:01 -0500
Subject: XML EZ Out 1.00
Newsgroups: comp.lang.ada

XML EZ_Out is a small set of packages intended to aid the creation of XML-formatted output from within Ada programs. It basically wraps the tags and data provided to it with XML syntax and writes them to a user-supplied medium.

This medium can be any sort of writable entity, such as a file, a memory buffer, or even a communications link, such as a socket. The only functionality required of the medium is that it supply a meaningful "Put" (for writing a string) and "New_Line" procedure.

XML EZ Out is available at http://www.mckae.com/xml_ezout.html.

The key facilitator of making XML EZ_Out usage readable when generating XML documentation is the overloading of a number of variations of the "=" function. By doing this, a simple XML element having no content, such as:

<player lastName="Cuddyer" firstName="Michael" team="Twins"/>

can be generated as:

```
Output_Tag
  (F, "player",
     ("lastName"  = "Cuddyer",
      "firstName" = "Michael",
      "team"      = "Twins"));
```

To simplify the specification of the attributes, variations of "=" are provided. Given these declarations:

```
Batting_Average : Float;
At_Bats         : Natural;
```

One can directly reference the variables:

```
Output_Tag
(F, "stats",
 ("battingAvg" = Batting_Average,
  "atBats"     = At_Bats));
```

From: Marc A. Criley <mc@mckae.com>
Newsgroups: comp.lang.ada
Subject: XML EZ_Out 1.01 Update
Date: Wed, 13 Apr 2005 14:04:59 -0500
Organization: UseNetServer.com

XML EZ_Out is a small set of packages intended to aid the creation of XML-formatted output from within Ada programs. It basically wraps the tags and data provided to it with XML syntax and writes them to a user-supplied medium.

This medium can be any sort of writable entity, such as a file, a memory buffer, or even a communications link, such as a socket. The only functionality required of the medium is that it supply a meaningful "Put" (for writing a string) and "New_Line" procedure.

XML EZ Out is available at http://www.mckae.com/xml_ezout.html.

Revision History
Changes since 1.00:
- Fixed problem with attributes being given negative numeric values. The minus sign was being dropped.
- If an attribute value is an empty string ("") or Null_Unbounded_String, then generation of that attribute specification is skipped.

From: Marc A. Criley <mc@mckae.com>
Date: Mon, 02 May 2005 08:49:37 -0500
Subject: XML EZ_Out 1.02 Available
Newsgroups: comp.lang.ada

XML EZ_Out is a small set of packages intended to aid the creation of XML-formatted output from within Ada programs. It basically wraps the tags and data provided to it with XML syntax and writes them to a user-supplied medium.

XML EZ Out is available at http://www.mckae.com/xml_ezout.html.

McKae Technologies "eats its own dog food", meaning that the utilities and tools that are distributed are actually used internally. Hence XML EZ Out 1.02 :-)

This version simply adds attribute assignment ("=") functions for directly handling single character attribute values.

It was thought to also add similar functions for Long_Float and Long_Integer, but this introduces ambiguities when assigning attribute values that are numeric literals, e.g., is "0.0" a Float or a Long_Float?

While this would be dealt with using Qualification -- Long_Float'(0.0) -- the initial addition of such functions would break existing code, so they were omitted.

If the need for such values as attribute values is needed, simply take the 'Image of the value (or invoke "Put" from a suitably instantiated IO package) and use the result as the attribute value.

Ada and GNU/Linux

Ada Usage in Debian

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Date: Thu, 24 Feb 2005 22:15:21
Subject: Re: Source code of large programs wanted
Newsgroups: comp.lang.ada

Here is a site that I just stumbled upon the other day: http://libresoft.dat.escet.urjc.es/debian-counting/

This guy must be nuts, he set out to measure the SLOC count of the last four stable releases of *Debian*, perhaps the largest single collection of programs in the world. The total for Sarge is roughly 218 million lines!

And I am happy to report that Ada is doing rather well in Sarge, as the count went from 0.5 million to 2.5 million lines of source text (I hate to call it "code"), and that Ada is now the 9th most used language in Debian (*up* two places since Woody).

If anyone feels like packaging even more Ada software for Debian, *please* do! There must be a way to beat Fortran and become #8, we need a mere 256 kSLOC. Anyone for PolyORB? That's 115 kSLOC by itself. Or Dtraq? 30 kSLOC. Or Adagio? 32 kSLOC. These fine programs, and others, deserve to be spread to the world.

I also noticed that on the Linux counter (http://counter.li.org), Debian now exceeds 21% of the installed base, now surpassing Red Hat and Fedora Core combined and becoming the #1 distribution!

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Date: Thu, 24 Feb 2005 22:30:24
Subject: Re: Source code of large programs wanted
Newsgroups: comp.lang.ada

Florian Weimer wrote:

> Debian unstable/experimental contains four additional copies of GNAT (with 400 kSLOC each), which accounts for 1.6 million lines of code. 8-/

The experimental packages won't go into Sarge, by definition. And, GCC also contains lots of C, C++, Java and Fortran. So this is unlikely to change the ranking.

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Date: Thu, 24 Feb 2005 22:33:35
Subject: Re: Source code of large programs wanted
Newsgroups: comp.lang.ada

Not to mention that I consider this cheating ☺ it's easy to package N versions of the same thing to swell the SLOC count, but there is little pride to be gained from that. For example, I could have kept libgtkada1 and gvd but decided not to, now there's libgtkada2 and gnat-gps instead.

[Cf. "Debian Policy for Ada" in AUJ 25-3 (Sep 2004), p.126. -- su]

References to Publications

DDC-I Online News

[Extracts from the table of contents. See elsewhere in this news section for selected items. -- su]

From: jc <jcus@ddci.com>
To: 22 February 2005 Online News US <jcus@ddci.com>
Date: Tue, 1 Feb 2005 17:35:30
Organization: DDC-I
Subject: Real-Time Industry Updates

DDC-I Online News, Real-Time Industry Updates - February 2005, Volume 6, Nr 2 [http://www.ddci.com/news_vol6num2.shtml] A monthly news update dedicated to DDC-I customers & registered subscribers.

This Month:

* 2.2 Billion Miles & Counting -- Riding High With Cassini-Huygens Software Coded With DDC-I Tools is Successfully Orbiting Saturn
* Precision Customer Service for Swiss Developers Legacy System DACS-8086 Ported from Solaris to Windows
* Italian Distributor -- ARTiSAN Software Tools Srl -- Relocates New Location Meets the Demands of Continued Growth
* Tech Talk: Saving and Restoring State A Nice Feature in the SCORE(R) Debugger
* Unusual Solutions - Part 1
Learn From the Past. It's the Best Way to Grow

From: jc <jcus@ddci.com>
To: 23 March 2005 Online News US <jcus@ddci.com>
Date: Tue, 1 Mar 2005 16:46:07
Organization: DDC-I
Subject: Real-Time Industry Updates

DDC-I Online News - Real-Time Industry Updates - March 2005, Volume 6, Number 3 - [http://www.ddci.com/news_vol6num3.shtml] A monthly news update dedicated to DDC-I customers & registered subscribers.

This Month:

* Secure C ... Not A Moment Too Soon
Perhaps the world of C is finally ready to listen!
* Upgrading DACS' Microsoft Visual Studio Support
Addressing issues that customers experience when using a combination of tools hosted on Windows
* DDC-I In Action
S/H-92 Helicopter Leading the Charge in Safety and Standardization
* Thoughts From Thorkil
Floating Point Concepts and the 80x86 Implementation (2)
* Unusual Solutions - Part 2
Never doubt that a small, committed group of people will change the world!

From: jc <jcus@ddci.com>
To: 26 April 2005 Online News US <jcus@ddci.com>
Date: Mon, 4 Apr 2005 14:36:45
Organization: DDC-I
Subject: Real-Time Industry Updates

DDC-I Online News - Real-Time Industry Updates - April 2005, Volume 6, Nr 4 [http://www.ddci.com/news_vol6num4.shtml] A monthly news update dedicated to DDC-I customers & registered subscribers.

This Month:

* Partner Update - Wind River
Unite with DDC-I & Your Peers at The Wind River 2005 Worldwide User Conference
* In The News
Will 64 Bit Embedded Systems Soon Become Common Place?
* C>Prompt
Routine Maintenance YOU Can Do
* Tech Talk
Migration of DACS Source to SCORE(R) With the Ada 83 Switch
* Something To Think About
Great Idea! And... It's The Best Way To Learn

SPARK team newsletter

SPARK news - May 2005

Please find below the latest instalment of SPARK-related news and information from the SPARK team here at Praxis.

Ada community award
SPARK Team was awarded the 2004 ACM SIGAda award for outstanding contribution to the Ada community. At the SPARK User Group meeting, the award was dedicated to Professor Bernard Carré - the founder of Program Validation Limited and principal designer of SPARK.

Ada Usage Survey 2005
The Ada Resource Association is attempting to quantify the global market for Ada. We'd like to encourage people to fill in their Ada usage survey. This is important to show the outside world that people really are still using Ada, and that SPARK forms a significant share of that market! We hope you can fill the survey in for your projects. The results will be presented at Ada Europe 2005 in York.

GPS Pro 3.0.0 and SPARK
The recent release of GPS Pro 3.0.0, AdaCore's free multi-language IDE, includes a SPARK customization file. This creates a SPARK menu, and allows users to run their SPARK tools from inside GPS - a help to anyone who prefers their life a little more GUI-fied.

UML and SPARK
A joint development effort between I-Logix and Praxis now enables the generation of SPARK Ada code directly from UML models. This involves a new capability to I-Logix' UML Model-Driven Development (MDD) product, Rhapsody, that facilitates the development of fully SPARK compliant Ada applications.

This development complements the existing support of SPARK in the ARTiSAN Real-Time Studio product line.

A new one-day "UML to SPARK" course has been designed for customers wishing to take advantage of these developments. The first public "UML to SPARK" course will take place in September in Bath.

SPARK black belt training
The new SPARK Black Belt course - designed for advanced users and focusing on proof - has been very successful. Feedback has been overwhelmingly positive.

The next public SPARK and black belt courses are scheduled for September - book now to guarantee your place.

Spreading the word
In April, Praxis hosted a highly successful one-day seminar for senior managers and senior engineers to describe the Correctness by Construction approach. This took place in the National Cryptological Museum in Maryland, USA, and featured guest speaker Randy Johnson from the NSA.

Forthcoming conferences include AdaEurope, where SPARK team will be giving two tutorials, presenting a paper and exhibiting. We will also be giving a tutorial at Formal Methods '05 in the UK in July.

Academic developments
In October, Praxis announced a joint academic initiative with AdaCore, as part of our ongoing attempt to promote SPARK within universities.

The primary objective of AdaCore's Ada Academic Initiative is to provide a collaborative platform where educational materials, knowledge, resources and fresh ideas can be developed and shared. This is perfectly complemented by Praxis' offer of a fully supported professional SPARK toolset offered free-of-charge to university faculty members for teaching and/or research.

Release 7.2
Release 7.2 went out to customers in January this year. Hopefully everyone is now using this upgrade and enjoying the

enhancements, particularly in the Simplifier. Upgrades for the "book" demo toolset to release 7.2 are also available

Team news

The SPARK team has been joined for 6 months by research student Bill Ellis. Bill is exploiting research from a previous project (called NuSPADE) to extend the proof capabilities of the SPARK toolset.

Emphasis is being placed on addressing the practicalities of an industrial system, rather than discovering new proof strategies.

Carys Ottner has joined the team from within Praxis. Carys is mainly working on support, marketing, training and porting the Proof Checker to a new PROLOG compiler.

New Publications

February 2005 - New Paper now available

Peter Amey's invited keynote address "Dear Sir, Yours Faithfully: an Everyday Story of Formality" from the 2004 Safety Critical Systems Symposium is now available on the Publications page.

[http://www.praxis-his.com/sparkada/pdfs/dear_sir.pdf -- su]

New York - May 23, 2005 - Dynamic Plug-in Loading with Ada - Paper Available

Maintenance of high-availability systems (e.g., servers) requires the ability to modify, enhance, or correct parts of the application without needing to shut down and re-link the entire system. This is relatively straightforward in an interpreted or virtual-machine based language such as Java, in which new code is loaded upon demand. In a language with static executable images this capability can be realized though dynamically loaded / linked libraries ("DLLs"). However, in practice this causes problems, because the protocol for invoking subprograms in a DLL is very low-level and sacrifices type safety.

Object-oriented programming makes this approach practical by using dynamic dispatching to invoke dynamically loaded functions with a more robust, high-level protocol. In an OO paradigm, a “plug-in” contains new classes that enrich the class set of the original application. Calls to subprograms in the shared library (plug-in) are done implicitly through dynamic dispatching which is much simpler, more transparent to the programmer, more type-safe, and thus much safer. A paper by Cyrille Comar and Pat Rogers shows how Ada, a statically-typed, statically-built, object-oriented language, can fully implement dynamic plug-ins as in Java, but without needing to rely on a comparatively inefficient virtual machine. This paper, which will be available on the AdaCore website, shows how to use GNAT Pro to build an extensible application and illustrates adding new functionality at run time through plug-ins, without needing to shut down the program.

Download the paper at [http://www.adacore.com/multimedia/pdfs/dynamic_plugin_loading_with_ada.pdf -- su]

Ada Wikibook in the Top 3

From: Martin Krischik <martin@krischik.com>
Date: Wed, 16 Feb 2005 14:07:40
Subject: Wikibooks:Top_active/Listing_January_2005
Newsgroups: comp.lang.ada

After all gloomy statistics here something good.

Provided that you only count books which start with "Programming:" then Ada has made it to place 3 on the Wikibooks:Top_active list: http://en.wikibooks.org/wiki/Wikibooks:Top_active/Listing_January_2005

Since the statistic is based on the count of different users contributing I consider it quite good. However don't trust a statistic you have not falsified yourself - do read my discussion entry as well.

[Cf. "Ada at Wikipedia & Wikibooks" in AUJ 26-1 (Mar 2005), p.8. -- su]

Ada Inside

Eurofighter selects Ada

URL: http://www.ghs.com/news/20050307_eurofighter.html

Eurofighter Selects Green Hills Software's INTEGRITY Real-Time Operating System

Key to mission-critical systems

SAN FRANCISCO, CA - March 7, 2005

Green Hills Software, Inc., the technology leader in embedded software development tools and real-time operating systems, today announced that its INTEGRITY Real-Time Operating System (RTOS) and AdaMULTI Integrated Development Environment (IDE) are being used in the development and implementation of mission-critical systems deployed in the latest Eurofighter Typhoon aircraft. The INTEGRITY RTOS is being incorporated as a critical component in a number of the aircraft's "line-replaceable" items.

David Smith, Software Manager, of Eurofighter GmbH, the consortium managing Eurofighter's development and production, said, "The INTEGRITY RTOS is an essential element in delivering the levels of system availability and reliability that are required by a number of computers being developed for Eurofighter Typhoon."

"We are confident that the decision to use Green Hills Software's RTOS and tools will contribute to the success of the second stage of the Eurofighter program," commented Jon Williams, European Director of Safety Critical Business for Green Hills Software. "This evolution, which includes a move to PowerPC devices, greatly benefits from the use of the INTEGRITY RTOS, which maximizes security and reliability and leverages the hardware memory protection facilities of the PowerPC processor. In addition, the AdaMULTI IDE provides an intuitive tool for testing during both software design and production."

About INTEGRITY

The royalty-free INTEGRITY RTOS is a scalable, ROMable, and memory-protected RTOS. Leveraging the hardware memory protection facilities of the PowerPC processor's Memory Management Unit (MMU), the INTEGRITY RTOS maximizes security and reliability by building a firewall between the kernel and user tasks. This prevents errant or malicious tasks from corrupting user data, the kernel, interprocess communications, device drivers, and other user tasks. In addition, the INTEGRITY RTOS guarantees the availability of system resources like the CPU and memory to application processes, making it far more secure and deterministic than conventional embedded operating systems.

About AdaMULTI

The AdaMULTI IDE is a complete, integrated set of tools for the development of embedded applications using Ada 95, C, C++, Embedded C++, and FORTRAN. The AdaMULTI IDE runs on Windows, Linux, and UNIX hosts and supports cross-debugging to a variety of target environments. The AdaMULTI IDE contains all of the tools needed to debug and deploy major programming project including: source level debugger, project builder, event analyzer, performance profiler, run-time error checker, and non-intrusive field debugging.

About Eurofighter Typhoon

Eurofighter Typhoon is the world's most capable and dynamic swing-role combat aircraft. Developed by Germany, Italy, Spain and the UK, the Eurofighter Typhoon will fulfil European Air Force requirements well into the mid-21st Century. The aircraft is in full production and has been in service with all partner Air Forces since 2004. 638 aircraft are under contract for the four Nations and Austria, the first export customer.

Aonix Tools Selected for European Satellite Launcher

Vega project chooses Aonix Ada development tools and real-time executive

Nuremberg, Germany, February 22, 2005

Aonix, a provider of complete safety- and mission-critical solutions, announces the selection of its tools by the European Space Agency's (ESA) Vega Programme, a satellite launcher for satellites weighing one ton or less that are used for scientific Earth observation, telecommunications, and technology applications in low-Earth orbits. Aonix tools have been selected for Ada application development along with its highly reliable real-time executive.

The Vega Programme came into being in the early 1990s when several European countries began to investigate the possibility of complementing the Ariane launchers' family with a capability for smaller payloads. These preparatory activities concluded in 1998, and in 2000, the member countries approved the launcher's full development phase. Aonix COTS products were selected as the mission-critical software phase of the project opened up. The AdaWorld Solaris to ERC-32 product implements the board segment software that controls and monitors all phases of the launcher's take off and flight until the satellite is in orbit.

"Thanks to a long history of certifiable applications, Aonix's tools have been selected for application development and real-time control of the Vega launcher," said Jacques Brygier, VP of Marketing at Aonix. "We are honoured to develop a launcher specifically designed for scientific and low-orbit missions and to work with prestigious companies like ELV, EADS Space Transportation, and Saab Ericsson Space who also value high standards of real-time safety-critical development."

"Aonix has built a strong reputation within the Space industry," noted Maurizio Porfiri, System Software Architect at ELV, main contractor of the Vega project. "We were impressed by the quality and robustness of their environment and appreciated the high level of expertise Aonix developed with space technology. Aonix expertise and experience are key factors in the future success of the Vega project."

"Aonix has played an instrumental role in major space projects such as the Ariane 5 launcher and the Automated Transfer Vehicle (ATV)," said Christophe Goarin, On Board Software Integration Manager at EADS Space Transportation. "Aonix products have always satisfied our demanding requirements and met the high-quality standards we expect from our suppliers."

To reduce costs, the Vega Programme uses a flexible modular approach that employs advanced low-cost technologies and takes advantage of existing production facilities used for Ariane launchers. Aonix provides COTS products for onboard software development, based on technology already proven in previous Ariane launchers. The first qualification launch for Vega is planned in 2006 from the French Guiana Space Center. Following this, there will be an average of three to four launches each year.

Utilizing Ada in the Airbus A380

URL: http://www.vectors.com/pdf/ nord_micro_vector_testimonial_final.pdf

NORD-MICRO USES VECTOR SOFTWARE'S VECTORCAST FOR TESTING OF A380 CABIN PRESSURE CONTROL SYSTEM

North Kingstown, RI June 1, 2005 Vector Software, a leading provider of software test tools for embedded systems, today announced that Nord-Micro has used its VectorCAST test solution on the flight software for the A380-Cabin Pressure Control System project currently undergoing certification. VectorCAST was used for module and SW-integration testing as required by DO-178B for levels B and C. The software tested with VectorCAST was written in Ada utilizing the AdaCore GNAT Pro HIE (High-Integrity Edition) compiler, the JTAG interface, and debugger from Abatron.

According to Bill McCaffrey, Director of Sales for Vector Software, Inc., We are extremely proud of our role as a technology supplier to Nord-Micro. Vector is pleased that VectorCAST has been chosen by Nord-Micro as a critical component for their software development and testing. About Nord-Micro's Cabin Pressure Control System (CPCS) Hamilton Sundstrand's Nord-Micro business unit in Frankfurt, Germany is a leading designer and manufacturer of cabin pressure control systems (CPCS) and ventilation system components for commercial aircraft. Nord-Micro supplies CPCS systems for the majority of the Airbus fleet, including all models of the A320 and A330/340 family. Nord-Micro has also been the supplier for the Boeing 737 CPCS for many years and is now working with Boeing to define and develop the CPCS for the Boeing 787 aircraft. About Vector Software Vector Software, Inc. is a leading independent provider of automated test tools for software developers. Established in 1989 as a consulting and service organization, Vector's product focus is to empower software professionals to deliver the highest quality software in the least amount of time. Vector's "VectorCAST" line of products, reduce the burden placed on individual developers by automating and standardizing application component level testing. This innovative technology developed by Vector represents the "next generation" of intelligent embedded software test tools. The tools support Ada 83/95, C/C++ and Embedded C++ (EC++). The market focus of Vector is on companies performing embedded systems development for aerospace, military, medical, telecom, and process control related projects. Vector Software s Product Family VectorCAST/Ada VectorCAST/C VectorCAST/RSP VectorCAST/Cover MC/DC add-on capabilities DO-178B Qualification Packages

Indirect Information on Ada Usage

[Extracts from and translations of job-ads and other postings illustrating Ada usage around the world. -- su]

Date: Tue, 08 Mar 2005 18:00:00

Job Summary: Position requires performance of all duties related to the RSA products transitioning to the Range and of RSA product sustainment activities after transition.

* Review, preparation and presentation of various technical documents and formal briefings for delivery to the Government and SLRSC management.

* Familiarization and compliance with the established SLRSC Policies and Procedures, always exercising initiative, good judgment and discretion.

Qualifications: BS degree in a technical field plus 5 years of applicable experience. Must have:

* Five years of experience with UNIX/AIX/Linux/VxWorks systems throughout the project life cycle.

* Experience with Ada, C, C++, as well as an understanding of Object Oriented design and development.

* Knowledge of and experience with Range Systems and the Space and Missile Systems Center Organizations.

* An understanding of system specifications and requirements allocations.

* Strong analytical skills and an understanding of component interaction and sub-system hardware and software functionalities.

Date: Wed, 16 Mar 2005, 14:00:00

[...] candidate must have a solid background in simulation software development. Experience with ground force gunnery simulation would be a plus. Skill Set: Ada, Ada 95, Simulation software experience, B.S. Computer Science.

For immediate consideration.

Date: Tue, 12 Apr 2005, 12:00:00

Praxis High Integrity Systems have vacancies in the UK in Bath, Loughborough and London.

The prime requirements are:
* a passionate desire to put engineering into software engineering;
* a willingness to learn; and
* flexibility in the kind of work done which may range from software development through to process consultancy.

We are interested in people with a broad understanding of engineering discipline and familiarity with entire development lifecycles. We are not simply looking for "Ada programmers"; however, the likely jobs have a software bias and those who appreciate the benefits of Ada are likely to be the kind of people we want.

Date: Tue, April 21 2005, 14:00:00

[...] Senior Software Engineer position developing in Ada. The position will involve the design, development and test of software utilized by the U.S. military and other customers for tactical communications. These are long term projects that have been ongoing for several years. A solid background in programming in the Ada language is required. Development experience using Rational APEX environment is preferred. The position also requires experience in the full software lifecycle. Responsibilities will also include measuring and evaluating product defects, applying rigor in design and coding practices, evaluating risks, evaluation of project metrics, and interfacing with Systems Engineering and Program Management in the discussion/resolution of issues and the management of interdependencies. Effective communications skills are necessary. Secret clearance will be required. Though the position will be primarily Ada development, there will be some C++ development as well. So the following will also be considered in the evaluation of candidates: C++, XML, Rational Rose, MS Visual Studio, MFC, Windows NT/XP, OOA/D. Usually requires 8-12 years experience in the following: Ada, C++, Object Oriented Analysis, Object Oriented Design, Secret Clearance, SEI/CMM, Software Engineer, Visual C++, VxWorks and XML.

Ada in Context

Language Portability

From: Lionel Draghi <Lionel.Draghi@Ada-France.org>
Date: Fri, 29 Apr 2005 23:23:30
Subject: Re: Ada.Text_IO and protected objects (Was: [newbie question] tasks and protected types)
Newsgroups: comp.lang.ada

> I think it actually does work in Janus/Ada and in GNAT, but (in our case at least), we aren't making any promises that that will remain the case. And its always bad practice to write code that will work only on a few compilers, especially something as hard to track down as this would be.

I fully agree with you, […] but it's quite difficult to ensure that no compiler dependant behaviour is used in the code, because no compiler I am aware of is kind enough to put a warning on all compiler/platform dependencies.

When porting, some problems will be caught at compilation time (for example representation clause that do not compile with another compiler). Some will be caught early at run time (for example elaboration order issue). But some, like this one, may not be noticed before delivery, and will be really difficult to track down, as you said.

And here is the limit of Ada portability Myth. OK, Ada is probably the most portable industrial compiled language. OK, porting an Ada code is much faster than any other compiled language. But the situation is not perfect. Ada is just 99.5% portable :-)

Those "inaccuracies" in the Ada semantics are possibly inevitable, precisely because of portability across platform, and maybe also to preserve different compiler implementation options.

So, the only way I see to ensure that an general Ada code is fully portable is an ASIS tools that will warn about all risky code. A kind of portability lint.

From: Lionel Draghi <Lionel.Draghi@Ada-France.org>
Date: Sat, 30 Apr 2005 09:21:08
Subject: Re: Ada.Text_IO and protected objects (Was: […])
Newsgroups: comp.lang.ada

> Isn't there in GCC/GNAT any options like "-pedantic" and "-ansi" in order to reject all programs that do not strictly follow the standard?

But non-portable programs follow the standard, that's my point. (Fortunately, in the Ada world, not following the standard is not an option).

That's why you may write code that perfectly run with GNAT, unaware that it won't run that fine with another one.

Multicore CPUs vs. Hyperthreading

From: Bini <fracttcarf@yahoo.co.kr>
Date: 31 Mar 2005 17:54:47 -0800
Subject: ada and multicore
Newsgroups: comp.lang.ada

What is a special benefit than C or C++ on multicore CPUs? Will Ada be a little more popular language? I used Ada 3 years, and I like Ada more than any other programming language...

From: Tom Moran <tmoran@acm.org>
Date: Fri, 01 Apr 2005 02:56:36 -0600
Subject: Re: ada and multicore
Newsgroups: comp.lang.ada

Ada has robust multitasking built in, which makes it simpler and more natural to write programs that take advantage of multiple processors. Ada programmers are thus also more likely to be comfortable with writing for multitasking.

In another thread here, some Ada versions of a simple word counting benchmark were compared to C versions. But a multitasking Ada version, a simple modification of the straightforward single tasking version, was shown to run about 50% faster on a dual CPU system.

From: Marin David Condic
Subject: Re: ada and multicore
Date: Fri, 01 Apr 2005 13:19:34
Newsgroups: comp.lang.ada

Of course, the caveat has to be that the compiler and possibly the underlying OS (if there is one - the RTK if not) has to have adequate support for multi-threaded applications or you're not going to realize a performance advantage. Naturally, this is also true for any other language that has some kind of multi-threading capability.

If one is truly interested in getting the advantages Ada can offer in terms of performance via multitasking, be sure to do adequate research up front and make sure you REALLY understand tasking & its possible implications. I've seen programs that have numerous tasks in them designed by people who probably didn't really understand what they were doing and they don't get a performance gain - or they take a performance hit.

Check the compiler & OS first. Start with *simple* uses of multitasking. Learn and understand what it does before trying to use it in critical applications.

From: Tom Moran <tmoran@acm.org>
Date: Fri, 25 Mar 2005 18:28:23 -0600
Subject: Gnat 3.15p & Windows & Hyperthreading Q
Newsgroups: comp.lang.ada

I'm told that a multitask program compiled with Gnat 3.15p and run under Windows XP Pro on a hyperthreaded machine, runs in the same total time as the same program using a single task. OTOH, when compiled with GNAT 5.02a1 and run on a dual-processor 400 MHz Celeron running Mandrake Linux 8.2, it runs about 50% faster with two rather than one tasks. Is the problem "hyperthreading", Windows, or Gnat 3.15p?

From: Steve <steved94@comcast.net>
Date: Fri, 25 Mar 2005 20:45:41 -0800
Subject: Re: Gnat 3.15p & Windows & Hyperthreading Q
Newsgroups: comp.lang.ada

Another data point, but not one you asked for...

I have seen Gnat 3.15p take advantage of dual Xeons on a W2k machine. So I don't think there is anything wrong in general with Gnat 3.15p on dual processors.

From: Wiljan Derks <Wiljan.Derks@zonnet.nl>
Date: Tue, 29 Mar 2005 20:33:05
Subject: Re: Gnat 3.15p & Windows & Hyperthreading Q
Newsgroups: comp.lang.ada

I did some performance tests on XP professional with hyperthreading on a 2.8Ghz P4. For that I used some code that checks the CPU performance. I did this by making some compute procedure and check how often it is calculated.

In my program I did make multiple tasks to check the functioning of hyper threaded.

It turns out that hyper threading does not help at all with the total performance that is available.

My conclusions where as follows:
* When hyperthreading is turned of, the system can do X computations.
* When hyperthreading is turned on and both threads are loaded, each of them can typically do less then X/2 computations. Thus turning on hyper threading gives basically two CPU which are both half speed when being used. When one of the CPU's is free, the other is faster. So it looks like one CPU is multiplexed in time (but very fast).

The only advantage that hyper threading has, is that the system might be more responsive.

Thus when accidentally locking one CPU at high priority, one can still break into the system (using the other CPU).

From: Adrien Plisson <aplisson-news@stochastique.net>
Newsgroups: comp.lang.ada
Subject: Re: Gnat 3.15p & Windows & Hyperthreading Q
Date: Wed, 30 Mar 2005 10:48:50

OK, let's release some misconceptions about HyperThreading: A HyperThreading enabled processor DOES NOT have 2 cores.

It's not clear at all when you look at Intel's overview of HyperThreading. They really like to tell it runs as if it were 2 processors. This may look like true for simple office tasks, which are not computationally intensive. Unfortunately, for software developers used to multitasking and trying to get the most of their system, it is evident that it is NOT 2 processors.

Actually, an HT processor has one core, which makes it no faster than a single processor. What's different is that it has 2 sets of "states", allowing for more efficient context switches. As noted by Wiljan, this makes the system more responsive.

Recompilation of Large Projects

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Date: Thu, 17 Mar 2005 20:47:20
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

> I have a 1mio LOC project and it takes multiple hours to build and rebuild. The C/C++ world has nice tools (ccache is well known here) which cache the output of the compiler. This especially speeds up the time for a "make clean;make".
> Is there such a thing for Ada in general and gcc's gnat specially? Any other way to speed up the compilation?

ccache does not work with GNAT (I tried it). However, "make clean" is seldom necessary where I work, and "gnatmake -m" does minimal recompilation. This really speeds things up.

From: Simon Wright <simon@pushface.org>
Date: 18 Mar 2005 19:22:00
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

We absolutely rely on ["gnatmake -m"]; the handwritten code is in separate proper bodies, 99% of specs and package bodies are generated, and the only way to change them is to change the model and re-generate.

One thing to watch out for is that when deciding whether a unit needs to be recompiled GNAT first checks the timestamp of each dependency; it the timestamp is different it checks the contents of the dependency and only recompiles if there's a semantic difference.

So recreating lots of identical source will mean that the dependencies have to be at least parsed. This can take quite a time (I think mainly the overhead of opening/closing the files).

We had an interesting bug with 3.16a1 on Windows where compilations ran slower if the compiler had been installed in the winter months (outside daylight saving time) -- a feature involving the Windows installer, I think; the timestamps in the library were all one hour out.

From: Robert A Duff <bobduff@shell01.TheWorld.com>
Date: 17 Mar 2005 14:40:48 -0500
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

Well, I had never heard of ccache before, so I just went and read about it. I am rather mystified by the *point* of it. I mean, if I don't want to recompile stuff that hasn't changed, I just don't type "make clean". If for some reason I want to rebuild everything from scratch, I type "make clean" (or rm -rf build_area or whatever) -- but then ccache *defeats* that, and does *not* rebuild everything from scratch. I don't see the point of that. It seems like a contradiction -- I want to rebuild everything from scratch while avoiding rebuilding everything!

The only thing I can think of is that if your makefile is buggy (missing dependences) it won't rebuild some things that depend on some .h files, and you'll get mysterious bugs. So rebuilding from scratch is more reliable. And ccache actually decides whether to run the compiler *reliably* (whereas 'make' does not). Is *that* why people use ccache?

Don't people automatically generate the make-file dependences for C code these days? It seems crazy to try to keep the make file in synch with the #includes by hand!

Anyway, the buggy make file problem does not happen with Ada. All Ada compilers come with a build tool that reliably decides what needs to be recompiled, and does it. In AdaMagic, it's called adabuild. In GNAT it's called gnatmake. Also, if you try to link an Ada program where the parts are inconsistently compiled you will get an error at link time. You never need to invoke the compiler directly -- just use the build tool.

If you're writing a make file for an Ada program, you should *not* put in dependencies for the Ada code! Instead, use a .PHONY rule to unconditionally invoke the build tool. Don't type "make clean" (except perhaps in your nightly test script, where you don't have to wait for it), but trust the builder to decide what needs recompiling and to keep things consistent.

You don't even need to use 'make' with Ada at all, if all you want to do is rebuild. But it's convenient to have make-file targets like "rebuild if necessary and run the regression tests".

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Date: Thu, 17 Mar 2005 21:04:56
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

Yes, Ada defines separate compilation cleanly. C does not, and Makefiles were created to make up for it. Makefiles can become very complex in the presence of preprocessors or rules that build a program that generates code which is compiled later on (this happens e.g. in the GCC bootstrap process).

It takes a lot of care to get these Makefiles right, and even then they tend to be very brittle. ccache was created to make up for *that*.

The Ada model is indeed clean, simple and reliable. Sometimes, it is necessary to supplement it to provide pre-processing facilities, or to select the units to be compiled from a CM system. Simple scripts or minimalist makefiles can do that.

From: i-google-iasuhdkajsh@rf.risimo.net
Date: 17 Mar 2005 22:58:18 -0800
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

There are two main gains scenarios where a cache helps.

1) make clean;make
You think that there are no reason for this. However I disagree. There are cases you need to rerun configure and afterwards the make clean;make idiom. Reasons for this could be:
- you have changed the version/flags of tools you use
- you have changed the configuration somehow (adding sound support for example)
- poor/bad make files for non-Ada code

In my project are multiple compiler generators and C/C++ and Java source code. So there is a configure and make clean;make is a good way to make sure you have a good build.

2) Sharing between trees
I also want to give here two examples. The first is when a group of local developers share such a cache. If now an external CVS change comes in only one developer has to wait. In the best case you could do a background cache filling based on new CVS code with cron and co.

The second case for sharing is multiple trees. Let me give you an example:

```
$ cd tree.clean
$ make
$ cd ..; cp -a tree.clean tree.patched; cd tree.patched
$ patch -p1 <....
$ make
# nothing gained so far. Two days later however there
# is an external CVS change.
$ cd ../tree.clean
$ cvs up; make
# testing ... ok. Let's test the patched version
$ cd ../tree.patched
$ cvs up
$ make
# The last make should really be sped up.
```

So you see there are more or less valid reasons for a compile cache.

Thanks for the -m switch hint. I will try it.

From: Martin Krischik <martin@krischik.com>
Date: Fri, 18 Mar 2005 09:07:17
Subject: Re: How to cache output of the compiler aka ccache
Newsgroups: comp.lang.ada

Tools like ccache or precompiled header files usually optimise "#include" which is indeed the main performance killer in C/C++ compilations - however Ada does not have "#include" - the with command works differently.

That much for a general comment - for more specific help we would need to know which compiler you are using.

From: Ludovic Brenta <ludovic.brenta@insalien.org>
Subject: Re: How to cache output of the compiler aka ccache
Date: Fri, 18 Mar 2005 02:24:28
Newsgroups: comp.lang.ada

>> The Ada model is indeed clean, simple and reliable. Sometimes, it is necessary to supplement it to provide preprocessing facilities, or to select the units to be compiled from a CM system. Simple scripts or minimalistic makefiles can do that.

> Yeah. And nowadays makefiles is mostly used as an output language from Configure. Write-only makefiles. Completely unreadable and unmantainable. And in my experience, more often than not these makefiles don't work. For example, building Graphviz, or ZLib on a Mac OS X. I end up studying the macrostructure of the code (the C units, which ones are libraries, which ones are programs) and then compiling by hand, or writing a small, readable makefile by hand, and eventually using Libtool (and sometimes Ranlib, another idiotic tool). For me Configure was a waste of time. Its 72-page manual is in the bin.

Hehe... C had separate compilation, but no dependency management. To overcome this limitation, Makefiles were created, but they were not portable enough. To overcome this limitation, configure scripts were created. At first simple, they had to take into account the idiosyncrasies of more and more host and target platforms and became unwieldy and fragile. To overcome this limitation, autoconf was created, but it still cannot possibly know everything about every host and target platform. To overcome this limitation, automake was created.

Now, autoconf and automake work hand in hand, each using a mix of several cryptic languages. They have become so fiendishly difficult to maintain that few people if any at all understand how they work. To overcome this limitation, ...

This isn't the Tao of Programming ☺

Who Will Champion Ada?

From: Michael Card <thehouseofcards@mac.com>
Date: Thu, 10 Mar 2005 02:33:15 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

It seems that everywhere I look, I see articles about the DoD world being anxious to purge Ada from all their systems in favor of C++ and Java. For example, see: http://www.cotsjournalonline.com/home/article.php?id=100149

This article references the Navy Open Architecture Computing Environment (NOACE -love that acronym) which specifically calls for a move away from Ada and requires all new software to be done in C++ or Java ("the C++ mandate"?)

My question is this: why are so many in the DoD itself and the contractors world opposed to Ada? The projects I have been on that used Ada got good results out of it in terms of system performance and development schedule. In both of these regards, the results were generally better than comparable C/C++ projects. So, was my experience unique? Were there great Ada failures (huge cost over-runs, bad performance, etc) that left such a bad taste in people's mouths that even nice products like modern Ada 95 compilers are unwelcome?

Also, as Ada is being abandoned in the aerospace industry, is there evidence that it is being picked up elsewhere?

From: Michael Card <thehouseofcards@mac.com>
Date: Thu, 10 Mar 2005 13:42:34 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

Whenever I have searched for the justification for these decisions, what you hear is stuff like "we want to use what everyone in the commercial world is using because it's easier to find programmers." It's rather a fad-chasing mentality, which seems strange given what we're talking about building. I find it hard to believe the use of C/C++ actually saves money vs. Ada no matter how many C++ programmers are on monster.com.

Does anyone know of any actual cost data that supports the "fad-du-jour language X is cheaper than Ada due to availability of programmers" argument?

From: <svaa@ciberpiula.net>
Date: 12 Mar 2005 11:08:11 -0800
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

Yes, Ada is still running because of inertia, and because of inertia it still will run for a long time. But it's a fact that there are very little people interested in Ada (companies, professional programmers, students or just curious).

The DoD is moving to other languages, and there are a few big projects out of DoD, but they eventually will move into other popular languages.

You shouldn't need to read this article to realize that Ada is almost irrelevant, and that's the trend.

Why? How Ada has reach that point of irrelevance? What can be done to change the trend?

Those are common threads in this forum. Just search.

From: Stephen Leake <stephen_leake@acm.org>
Date: Sat, 12 Mar 2005 20:59:10 -0500
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

I was just at a talk by Rod Chapman of Praxis Critical Systems; they sell SPARK and services using SPARK. They are growing, and several other tool vendors are starting to support SPARK; Ilogix, for one.

SPARK is a statically analyzable subset of Ada; the safety and security critical fields are beginning to realize that can save them money. AdaCore is also growing.

Hmm. Perhaps _you_ need to read some _other_ articles ☺

Pay attention to what's really going on.

From: <svaa@ciberpiula.net>
Subject: NOACE- End of the road for Ada?
Date: 13 Mar 2005 04:44:28 -0800
Newsgroups: comp.lang.ada

Denying reality is not a way to solve problems.

Borland is [also growing], so Sun does, so C++ does, so Java does, so others do.

The fact that AdaCore is growing may only mean that AdaCore is collecting all potential Ada customers that don't have any other company. Perhaps AdaCore is growing not because a new Ada golden age, but at expenses of companies that don't work with Ada anymore. The market of Ada is so small the there is only room for a few companies. When a company stops developing with Ada, the rest of companies, that still use Ada, grow a little.

You live in bubble. You should read another articles too. Not only those that tell that Ada is lingering, but those about Java, about C++, about C, about PHP about Perl, about Ruby, about Python...

This look like Esperanto. I played a little with Esperanto. Thanks to the Internet Esperanto is growing. So what?. If you live inside esperanto movement, the Esperanto has a lot of associations, literature etc. You see esperanto everywhere, and you conclude that esperanto is quite alive. If you look esperanto from outside, esperanto is irrelevant.

If you program most of time with Ada, work on a company/organization that works with Ada, you read articles that support Ada, you go to conferences about Ada, accept good news about Ada, but filter bad news about Ada. You will conclude that Ada is quite alive.

If you look Ada from outside, you see that Ada is lingering, that it's difficult to find a job for Ada, and if you find it, 99% will be to support legacy systems, and probably until they move to another language. You can find a thousand tools and libraries for any language and choose. For Ada you must go to half a dozen sites/companies and take what you find there.

NOACE movement is a good show of what's going on related to Ada. For each new project in Ada with a big hype in Ada related conferences, congresses, and websites, you can find 100 projects that are giving up Ada silently. In demography, more deaths than births is called negative growth.

From: Stephen Leake <stephen_leake@acm.org>
Date: Sun, 13 Mar 2005 09:22:26 -0500
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

I did not deny anything, I merely pointed out some sources of information that you appeared to be unaware of.

Rod Chapman made an interesting point. Ada has less than 0.1% of the programmer market. BMW has less than 0.1% of the car market. Is BMW a failure, or a successful niche player? Ada is a successful niche player.

The market for high-end cars is also small. Hmm. Does "irrelevant" mean "dead"? I don't think so.

If you don't want to join us in our Ada "bubble", fine. But have the grace to leave us alone ☺

"A job for Ada"? Do you mean "a job that requires Ada knowledge"?

Any problem that requires programming is potentially "a job for Ada"; those are certainly not hard to find.

Any job that requires a particular programming language is not one I'm interested in; I'm interested in using the best tool for the job.

If that means I'm in a "bubble", fine.

If that statistic were true for the last several years, no Ada company would be in business now, since no company can lose 99% of its business several years in a row and survive. That is demonstrably false; just look at the AdaIC list of Ada compiler companies; it has been stable for the last several years.

So I conclude your statistic is not true.

From: Dr. Adrian Wrigley <amtw@linuxchip.demon.co.uk.uk.uk>
Date: Sun, 13 Mar 2005 23:20:32 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

If there had been 1,000 projects, losing 100 projects silently, and gaining one project (with fan boy fanfare) gives 901 projects remaining. This could last all of... ten years before annihilation. It's even plausible, if you count projects (e.g.) >5M SLOC. But it is not a 99% loss each year.

Curiously, in the world of hardware/chip design, the same debate about VHDL (with Ada's discipline/syntax) vs. Verilog takes place. But VHDL has a large (not majority) base. Nobody seems to worry much about VHDL being a DoD language. And its fanaticism for precision and reliability isn't seen as useless, redundant or lacking "power". But the "new" upstart in hardware design (amazingly) is "C" (subsetted, tweaked). (The hope is you can get programmers to design hardware!)

I've always thought that Ada would benefit by being much more closely associated (even merged) with VHDL. But (AFAICT) few VHDL users have ever used Ada, and vice-versa. Given that they are nearly identical, why are no synergies found?

The [debate of the thread "Teaching new tricks to an old dog (C++ -->Ada)"] shows how amazingly ill-informed people are about the Ada language features. (people say "do not think it supports generic programming", '"manually added checks" in C++ would be identically eliminated to the automatic checks in Ada', 'what's the use of rep. specs, except to restrict portability(?)' etc.)

Clearly the beliefs and reputation are a major part in language choice.

The three ways you can make a popular language are:
1) Extend a popular language (C++, F77)
2) Start from scratch with big budget (Java, C#)
3) Fill a big market vacuum with something that works (Fortran, C, Cobol, PostScript, Perl <at various times>)

I'm not aware of any popular languages that came about in any other way. Ada tried to be 2, 3. But the market vacuum was in the eyes of the DoD, not the users/contractors. Ada has failed to become a popular language (in terms of users), and now none of these three possibilities can be used to rectify the situation.

Any language designer/advocate who wants to promote the Ada ideals would be best trying again (don't start from here!). For example:

- find a popular language and transplant Ada features (C99 with tasks, arrays, generics etc? (a real bastard)(too late?) takes us back to the infamous "Ada syntax turns people off!)

- get a big backer to force a new Ada-inspired language into the market (too late for C# or Java, but they could easily have taken much more from Ada, if enough of the right persons had been there!)

- think up something radical and new in programming, and infuse it with Ada principles. (we had 4GL and Fifth Generation, what next? 6GL? Wikipedia doesn't yet have "Sixth-Generation languages" entry!) (my personal view is that a decent "visual programming language" could find a market vacuum sometime in the next thirty years, and is ready to be invented. Nothing so far has been terribly useful or general, so the field has been written off.) (any more ideas on this topic?)

back to the original topic... NOACE does seem to be a real step backwards. It looks a lot like a "Java Mandate", but acknowledges that there will be many exceptions, which C++ would probably meet. I think it's very risky, since newer languages tend to have a shorter lifespan and change faster than mature languages. It clearly is motivated by much more than the technical merits of the language. But if colleges switch to teaching "C2#" or "Guam" a decade from now, they might be stuck with a poor technical solution, serviced by a declining programmer base. And if they have to have specialized variants of Java for their high reliability, sub-microsecond real-time applications, they risk having a total "language isolate" on their hands.

Interesting that Boeing doesn't like Ada or C++. It'd be interesting to understand why each of these fails to meet their needs. Particularly since both languages' advocates usually say they are much more suitable than Java for almost any application!

From: Michael Card <thehouseofcards@mac.com>
Date: Mon, 14 Mar 2005 00:25:49 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

Good post, my experience has been consistent with the experiences cited by Richard Riehle in his post on this thread.

The only rationale I can come up with to explain the DoD's anti-Ada bias is that some high-up folks in the Pentagon didn't like it, maybe because it was from DISA (I get the feeling DISA is one of the lesser-loved branches of the DoD). This distaste for Ada by the services themselves (USN, USA, USMC, USAF) would certainly be quickly mirrored by the contractors, since the services are the ones paying them, not DISA. So, a dislike for DISA mandates on behalf of the armed services ends up being reflected in their contractors, who want to "suck up" (not in the bad sense) to their customers as much as possible to win contracts.

This is the only explanation I can come up with, because I have never seen a study that says "we did this multi-million line job in C++ and boy are we glad we did; we saved so much $ vs. past similar jobs we have done in Ada". In fact, the studies out there that I am aware of say just the opposite.

I am wide open to receiving a C++ cost savings study, however; I just haven't been able to find it. If someone can send me link I'd be happy to read it and be educated about it. My experience with C/C++ on large DoD style projects suggests the usual culprits [memory corruption and concurrency problems] make it a more expensive choice than Ada because it takes longer to get the bugs out.

From: Ed Falis <falis@verizon.net>
Date: Mon, 14 Mar 2005 02:11:33 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

From the COTS Journal article:
"The Navy, for example, has crafted its Navy Open Architecture Computing Environment (NOACE) to be the standard for all future software systems on Navy warships. That includes shipboard weapon systems, such as anti-aircraft cannon controls as well as avionics systems aboard naval aircraft. The standard calls for all new software to develop in either C++ or Java, and makes specific mention of moving away from Ada. They plan to continue to use Ada only as required to support legacy systems that have already been developed."

I've heard that the NOACE document has not yet been released - that what COTS Journal has been making such a big deal about was a leaked draft and not the final statement of policy.

Perhaps it's better to wait for the final document to be published?

From: Dr. Adrian Wrigley <amtw@linuxchip.demon.co.uk.uk.uk>
Subject: NOACE- End of the road for Ada?
Date: Mon, 14 Mar 2005 02:29:53 GMT

On the contrary! One of the reasons for leaks is to allow outsiders to comment on plans before they are committed to. If a document is officially released, changing it may be politically sensitive and/or slow. Much more convenient to listen to the backlash from a leaked draft, deny the plans, fix them and release the "final" document with the bugs fixed. This makes commenting on leaked drafts a good idea!

Here in Britain, deliberate leaks seem to be a major part of government policymaking. With NOACE however, I think we'd be whistling in the wind.

From: Peter C. Chapin <pchapin@sover.net>
Date: Sun, 13 Mar 2005 19:58:43 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

I noted with interest that the recent release of KDevelop offers Ada support (no doubt due to gnat being a part of gcc). Is Ada gaining favor in the open source community? I should think that the availability of a good quality, free compiler would be attractive to open source developers.

Some people I talk to say things like, "Ada is that DoD language, and I'm not into military stuff." It seems like Ada's association with the DoD has not helped it... at least not lately. However, if Ada attracts the attention of students and hackers (I'm using the positive meaning of "hacker" here), that might be very good for its long-term future.

Peter (who is a student thinking about using Ada in a class project)

From: Pascal Obry <pascal@obry.org>
Date: 13 Mar 2005 21:14:50
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

I hope so too. It seems to be right. Despite what can be read here and there I found that Ada is used more and more these days. The traffic here in comp.lang.ada is bigger than what it used to be (5 or 10 years ago). And we have some great Open Source projects around... All this led me to think that Ada is on the right tracks...

From: Richard Riehle <adaworks@sbcglobal.net>
Date: Sun, 13 Mar 2005 18:42:35 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

The move toward Java has nothing to do with whether Java is superior to Ada. It's not. Is it easier to learn than Ada? No. Is it more efficient than Ada? Certainly not. Is it easier to code than Ada? Not at all. Does it produce better executables? Not at all.

So why is it taking over the programming landscape like kudzu or crabgrass on an Alabama lawn?

I attended a seminar presented by a U.S. Navy Admiral a couple of years ago on the subject of software in the Navy. He droned on for a while about his view on this subject and finally came to Ada. His opening remarks to this topic, "And then there was the Ada fiasco!" In his comments he noted that Ada was hard to learn, even after hiring the best teachers the Navy could find, there were no good tools available for development and maintenance, all the programmers hated it, no one wanted to support it, everything they did related to Ada created more trouble than it was worth.

This perception of Ada throughout much of the Navy, and throughout much of the DoD persists. I work daily with DoD people who believe Ada was one of the most idiotic initiatives the DoD ever pursued. At the school where I teach, Ada was once required. Now it is hardly mentioned (except in some of my classes). Sometimes, when I visit the office of one of my colleagues, I see old copies of Ada books (most Ada 83) on the bookshelves. The only two languages most people want to acknowledge are Java and C++, and of those, Java gets the larger share of attention.

Java, for all its faults, is the current darling of decision-makers and academics. Many of my students find Ada easier to learn after they have learned Java. Most of them hate C++, but the have to learn it to successfully complete their required class in computer graphics. There are almost no circumstances where they must use Ada, let alone know anything about it.

I continue to believe that Ada is as good, often better, as a programming language than either Java or C++. But that is not a widespread belief throughout the DoD. Rather, the more dominant view is that Ada is now an old-fashioned language, more in the category of PL/I, COBOL, old versions of Fortran, etc. It is seen as old, in part because it is regarded as a language of the early 1980's. Java is the language of now. Ada is the language of then. For many, C++ is also the language of then.

There is no large company currently pushing Ada. There are no substantial financial resources behind it. Even the companies that publish Ada compilers, with the exception of AdaCore, RR Software, and Irvine Compiler, are focusing their attention and their advertising dollars on other products.

One Navy official said to me a couple of years ago, "In five years you won't be able to find anyone supporting Ada." That was nearly five years ago, and he was wrong. But how wrong was he? Does IBM take its (Rational) Ada compiler seriously anymore?

Ada certainly does not deserve the reputation it has among DoD officials. But, as long as the majority of promotional dollars are devoted to touting the (dubious) benefits of technologies, even as those technologies are inferior to Ada for military software, Ada will suffer.

Who will champion Ada? Currently, no one with influence or power will come forward to encourage the use of Ada. Sometimes I speak with developers who prefer Ada and still choose C++, not because they prefer it but because it is the easiest choice to make. Courage is not a common characteristic of DoD developers. To preach too openly the benefits of Ada in the halls of a contractor's office or the corridors of a DoD facility is to risk being branded "some kind of nut." I have been called an "Ada bigot," more times than I can count -- this, in spite of my continual assertion that we should pick the right tools for the right job -- and the right tool is often, but not always, Ada.

From: Jared
Date: Mon, 14 Mar 2005 05:13:22 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

This is wrong. I'm sorry. I like Ada. I want to believe it is right, but it isn't. I could be wrong about all of this. I often am.

Sun is behind Java. They've been hyping it. They've been pushing it. They've been doing everything they can to keep it in the press. That's great, and it gives Java a chance to succeed, but that's all it gives it. Java really is better, in a marketing sense, and sort of better in a conceptual sense.

Rant abstract: Java is more popular because its syntax better represents OO. If you're bored of that argument, there is another, much shorter one, at the first separator. If you're bored of that one, there's a third one, but it isn't very constructive.

Java, as has been pointed out elsewhere on this group, is basically C++ with all the bad parts removed and with garbage collection added. Java is comfortable to all the C people, because of its syntax, because of its culture, and because it pretends to share the C++ idioms. Nobody is going to switch to a language that feels uncomfortable unless it has some really neat gimmick. (For example, Oz and Piccola have really neat gimmicks.) In a market sense, even without the hype, this makes Java better.

It's worse than that, though. Java got namespaces mostly right. It got packaging mostly right. A package isn't a namespace. It's really tempting to identify them, but the real namespaces are the variables.

Why do you think Object.Method syntax is so popular? What was the big deal with the 'use type' clause? And what are all those Smalltalk people yammering about when they talk about sending messages to objects? The variable is the namespace. The method exists within it. It isn't in the package. The package isn't really a real thing; it's a variable of anonymous type. The reason that Object.Method and Package.Method are indistinguishable it that they are indistinct.

So, for example, suppose I have:

```
package Thing is
  A : Integer := 0;
  procedure Do_Something;
end Thing;
....
type Thing_Type is tagged
   record
      A : Integer := 0;
   end record;
procedure Do_Something
   (T : in out Thing);
Thing : Thing_Type;
```

There's not much difference when I declare it, except that the latter is much more verbose and has to be embedded in a package. There's no difference at all when I use it. There is a difference when I try to extend it, but was anyone paying attention when Dmitry suggested that 'use' be transitive?

What I mean by getting packaging mostly right should be clear by now. The type (class, or whatever one wants to call it) is the natural unit of packaging. Consider the following common idiom:

```
package Shoes is
  type Shoe is tagged record
   with private;
  ...
end Shoes;
```

With no other types declared in the package. Wouldn't it just be easier to let the type be a package? That's about all I have to say about that.

So, in conclusion, Java represents a cleaner presentation of OO than Ada.

But that's all it is. Java's model is not really any better from Ada's; it just looks better, or rather, it's presented in a way that people prefer. It's just presentation, but presentation is critical. Many industries exist almost entirely on presentation, with substance coming in a distant second.

Yes, I know Ada was meant to be legible. But it has a low signal to noise ratio.

Want Ada to be more popular? Write an alternate syntax. Play up objects. Make them the focus. Read up on the pi calculus, and compare with protected types and streams. Above all, if something doesn't need to be said more than once, it shouldn't be. [...]

Actually, I do have one argument besides the old "the syntax sucks" dead horse. Think about Perl.

Perl was created to solve a problem. It thrives because it filled a niche, and did it well. Ada didn't have that kind of focus. The ARG needs to find new niches and fill them well. It needs to defend hard real-time and do so quickly, because that's being lost to Java. A garbage collected language! For real-time systems!

Government contracting isn't a niche. It's a hog trough; a place to become bloated and lazy.

And that's about all I have to say, unless somebody needs a clarification. That quite likely; I doubt the previous made much more sense than it did the first time I tried to formulate it.

Now, you don't have to buy into any of that. So here's an alternate theory, based on the grass analogy. From an article in a university newspaper:

"One of the more common questions I'm asked is how to control Bermuda grass in the lawn. My standard reply is 'asphalt'," says Jerry Goodspeed, Utah State University Extension horticulturist. Unfortunately, Bermuda grass can grow through asphalt and really thrive.

There you go. Java is the weed that won't die and Sun (pun not intended) is making sure it stays that way.

Myopically focusing on Java and C++ is a good way to ensure that Ada is never more than marginally better than either. There's a lot of research going on out there.

Much of this research is geared at making functional and denotational languages more efficient. I have full confidence that the Mozart/Oz people will screw up their chance to become the Next Big Thing. But somebody won't, and that could happen tomorrow.

Where are the big bucks behind, say, Ruby? What is Ruby's growth rate, compared to Ada's? What can be learned here?

Who will champion Ada? AWS. APQ, maybe. The other projects. You. Maybe me. We don't need big companies. We don't need money. We need, if you'll forgive me for putting it this way, the coolness factor.

Step 2: ???
Step 3: Profit!

I'm still working on step 2.

From: Marin David Condic
Date: Sun, 13 Mar 2005 17:23:52 GMT
Subject: NOACE- End of the road for Ada?
Newsgroups: comp.lang.ada

I personally think your criticism is well thought out and makes some valid points. Irrational exuberance and rose coloured glasses will not save Ada or make it more relevant. I work in a DoD related field and I can see the customers I have packing their bags and moving on to other languages. I can try to influence that decision towards Ada, but they are not in a position to spit into the wind and utilize a language without much following in the general computing world unless there is some compelling reason. It is difficult to find compelling reasons to offer them when all the economics tend to get stacked against Ada.

That said, let me offer this: It doesn't help to be negative about it, nor does it help to spend hours worrying about whether or not someone likes you. If one gets stuck in a rut of saying "Its all hopeless!!!" then ipso facto, it becomes hopeless. If one sits around all day thinking "Why doesn't anyone like me? What can I do to get people to like me?" it is similarly self defeating. You'll never get everyone to like you and trying will only expend your efforts in a bunch of futile dodges. While we're at it, being a Pollyanna about it ("Everything with Ada is WONDERFUL in my little pastel coloured, unicorn infested, rainbow, gumdrop world!") doesn't help either. One denies the obvious problems and refuses to take action to make it better.

Some suggestions that might actually help:

1) Do things in Ada that you want to do and ignore those who keep saying its going to hell in a handcart. Make as much Ada code as possible. Make it as useful as possible. Make it as available as possible. The more Ada there is out there, the more likely Ada has a sound future.

2) Quit thinking about making more software technology or remaking things that already exist in other languages. Dream up things to make out of Ada that aren't already done and that address some bigger need. We keep thinking in terms of "Here's this cool app someone wrote in C. Let me rewrite it in Ada..." Hint: NOBODY CARES THAT IT IS WRITTEN IN ADA OR ANYTHING ELSE. They care that it does some job. Reinventing network tools or software development tools or any other batch of stuff that programmer-geeks like to build doesn't really help if there are thousands of them out there already and you have nothing new & innovative to offer. Its also a small market compared to the wider world of general computer users. Think about it this way: Build a better mousetrap. What about a better office suite? What about a better accounting package? What about a better statistics tool? What about a better structural analysis tool? What about a better "Simulink"? (I'd like to see one - and one that generates Ada instead of C) Make some better mousetrap that has usefulness beyond the interest of a few programmer-geeks.

3) Think about starting a business that makes some useful product with Ada as part of its technology. If Ada has so many advantages, it ought to be a competitive edge. If you build some sort of commercial software or embedded system or other useful product with Ada as a component, then you create a market for Ada tool vendors and a job market for Ada programmers. The people who program in C or C++ generally are not so concerned about the language, per se. They're busy building some cable TV network or computational fluid dynamics analysis tool or automotive control & diagnostic computer. They sell that stuff and hence have money to spend on stuff like compilers and programmers.

4) Don't worry if the DoD guys want to abandon Ada. Their motivation is one of economics (primarily). Make Ada economical and they'll come back. It was and is a mistake to rely on them to create the market for Ada. Ada has to have a utilization in the greater world and not just rely on the DoD. If the DoD contractors find that some commercial sector that is doing something similar to what they want to do are using Ada as part of their toolset, they'll follow.

Think about this for a minute: Say I'm a DoD contractor and I have an application that involves graphics in some regard. They look at what guys in the private sector are using - the GUI building tools, the graphics libraries, etc., and they go do the same. Why? Because they can readily get the tools and readily get the people who know how to use them and since it is technology out there in the field, it is low risk to their project. If Ada had the same tools and libraries & skilled people out there in real-world projects, they'd go for that. But their objective is not to use Ada, but to get a graphics job done. If some Ada fan(s) were building the world's coolest video games in Ada and making money doing so & employing people to do it and generating/licensing the technology, wouldn't DoD contractors go follow suit?

In the world I live in, I see a bunch of tools that are variations on Simulink for designing plant models & control systems. Pretty much across the board, these tools are designed to work in a style akin to 1960's era Fortran programs. They pretty much suck stylistically in that they don't support most of the software engineering kinds of features we've developed since the 1960's. But they basically do a job: Someone can model a plant and model a control and test it out on a workstation. Then the pressure becomes to use the C code (few if any still output Ada) they generate to be the actual control code. That has problems, but hopefully you can understand that pressure: the model already exists and it already works and there is already a test suite, so why not dump it into the control & scab up some more C code around it to run the real time control?

I can imagine a *much* superior design & modeling tool that might utilize lots of Ada concepts like packages & tasking and sophisticated data types and all sorts of stuff. I can imagine a much superior simulation environment that would buy numerous improvements in flowing the design into the actual box & testing it with greater efficiency. If such a system got built in Ada and generated Ada and was based on Ada concepts and *if it helped do someone’s job better* than the existing technology, it might worm its way into the control software market. Perhaps finding users in the automotive and aerospace industries. It might secure a niche for Ada. This would be an example of something that was being built for reasons other than just to use Ada or make Ada popular. It would be getting built to make a better mousetrap and might have the beneficial side effect of promoting more Ada use. That kind of thinking might get Ada somewhere.

Conference Calendar

This is a list of European and large, worldwide events that may be of interest to the Ada community. Further information on items marked ♦ is available in the Forthcoming Events section of the Journal. Items in larger font denote events with specific Ada focus. Items marked with ☺ denote events with close relation to Ada.
The information in this section is extracted from the on-line *Conference announcements for the international Ada community* at: http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/events/list.html on the Ada-Belgium Web site. These pages contain full announcements, calls for papers, calls for participation, programs, URLs, etc. and are updated regularly.

2005

☺ July 06-08 — 17th **Euromicro Conference on Real-Time Systems** (ECRTS'2005), Palma de Mallorca, Spain

July 06-12 — 17th **International Conference on Computer-Aided Verification** (CAV'2005), Edinburgh, Scotland, UK. Topics include: Algorithms and tools for verifying models and implementations, Program analysis and software verification, Applications and case studies, Verification in industrial practice, etc.

☺ July 11-14 — OMG Annual **Workshop on Distributed Object Computing for Real-time and Embedded Systems**, Washington, DC, USA. Topics include: Real-time systems; Embedded systems; Fault-tolerant systems; High-availability systems; Safety-critical systems; Real-time middleware, including real-time CORBA; Modelling notations (including Unified Modelling Language, UML); High-level real-time programming models; etc.

July 11-15 — 32nd **International Colloquium on Automata, Languages and Programming** (ICALP'2005), Lisbon, Portugal. Topics include: Parallel and Distributed Computing; Principles of Programming Languages; Formal Methods; Program Analysis and Transformation; Specifications, Verifications and Secure Programming; etc. Affiliated Workshops on July 9-10 and 16-17, 2005.

July 11-15 — 1st **International Conference on Open Source Systems** (OSS'2005), Genova, Italy. Topics include: Introduction of OSS in companies and Public Administrations, Empirical analysis of OSS, Case studies and experiments, etc.

> July 11 — **Workshop on Evolution of Open-Source Code Bases** (EVOSC'2005). Topics include: evolving open-source code bases, without losing the benefit of a community working on the same software, etc.

July 17-20 — 24th Annual ACM **SIGACT-SIGOPS Symposium on Principles of Distributed Computing** (PODC'2005), Las Vegas, Nevada, USA. Topics include: all areas of distributed systems; including: Distributed applications; Specification, semantics, and verification; Distributed middleware platforms; etc.

July 18-22 — 13th **International Symposium of Formal Methods Europe** (FM'2005), Newcastle upon Tyne, UK. Topics include: introducing formal methods in industrial practice (technical, organizational, social, psychological aspects); reports on practical use and case studies (reporting positive or negative experiences); tool support and software engineering; environments for formal methods; etc.

> July 18-19 — 3rd International Workshop on Formal Aspects in Security & Trust (FAST'2005)
>
> ☺ July 18 — Grand Challenge 6 **Workshop on Dependable Systems Evolution** (GC6). Topics include: the current state of the art in strong software engineering tool-sets, and their application to systems that have been deployed in practice.
>
> July 19 — **Workshop on Rigorous Engineering of Fault-Tolerant Systems** (REFT'2005). Topics include: Development and application of tools supporting rigorous design of dependable systems; Case studies demonstrating rigorous development of fault tolerant systems; etc.

July 20-29 — **Summer School on Reliable Computing**, Eugene, Oregon, USA. Topics include: current research in reliability of software systems ranging from foundational materials on type systems, program analyses, and model checking to advanced applications of the techniques in practice.

July 25-28 — 29th Annual **International Computer Software and Applications Conference** (COMPSAC'2005), Edinburgh, Scotland, UK. Theme: "High Assurance Software Systems". Topics include: Dependable service provision, Trustworthy software, Software safety, Software fault tolerance, High performance software, Component-based software, Design patterns, Software certification, Software standards, Software engineering education, Embedded systems, Middleware systems, Automotive telematics, etc.

☺ July 25-29 — 19th **European Conference on Object-Oriented Programming** (ECOOP'2005), Glasgow, Scotland, UK. Topics include: Concurrent, real-time and parallel systems; Design patterns; Distributed systems; Frameworks and software architectures; Language design and implementation; Programming environments; Adaptability; Formal methods; Software evolution; etc.

- ☺ July 25 — **Workshop on Exception Handling in Object Oriented Systems** (EHOOS'2005). Topics include: Programming constructs for exception handling, Experience reports, etc.
- ☺ July 25 — 9th **Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts**. Topics include: successfully used exercises, examples, and metaphors; approaches and tools for teaching (basic) object-oriented concepts; approaches and tools for teaching analysis and design; ordering of topics, in particular when to teach analysis and design; teaching outside the CS curriculum; experiences with innovative CS1 curricula and didactic concepts; etc.
- ☺ July 25 — 4th **Workshop on Parallel/High-Performance Object-Oriented Scientific Computing** (POOSC'2005). Topics include: frameworks and tools for scientific object-oriented computing; tried or proposed programming language alternatives to C++; performance issues and their realized or proposed resolution; issues specific to handling or abstracting parallelism; existing, developing, or proposed software; etc.
- ☺ July 26 — **Workshop on Practical Problems of Programming in the Large** (PPPL'2005). Topics include: The role of the software-architect in the phases requirements engineering, software design and development; Negative results: what went wrong although it should have worked according to software engineering folklore; Keeping systems with large amounts of classes / objects / modules / components organised; Refactoring, software evolution and migration; etc.
- ☺ July 26 — 2nd **Workshop on Programming Languages and Operating Systems** (PLOS'2005). Topics include: type-safe languages for OS; domain-specific languages for OS development; language support for OS verification, testing, and debugging; etc.

August 23-26 — 16th **International Conference on Concurrency Theory** (CONCUR'2005), San Francisco, CA, USA.

Aug. 29-Sept. 02 — 13th IEEE **International Requirements Engineering Conference** (RE'2005), Paris, France.

- August 30 — 4th International Workshop on Requirements for High Assurance Systems (RHAS'05).

☺ Aug. 30-Sept. 02 — 11th **International Conference on Parallel and Distributed Computing** (Euro-Par'2005), Lisboa, Portugal. Topics include: Support Tools and Environments; Scheduling and Load Balancing; Compilers for High Performance; Distributed Systems and Algorithms; Parallel Programming: Models, Methods, and Languages; etc.

Aug. 30-Sept. 03 — 31st **EUROMICRO Conference on Software Engineering and Advanced Applications** (EUROMICRO'2005), Porto, Portugal. Topics include: Component-Based Software Engineering, Software Process and Product Improvement, Component Models for Dependable Systems, Value-based Software Engineering, etc.

☺ September 05-06 — 10th **International Workshop on Formal Methods for Industrial Critical Systems** (FMICS'2005), Lisbon, Portugal. Topics include: Verification and validation of complex, distributed, real-time systems and embedded systems; Verification and validation methods that aim at circumventing shortcomings of existing methods in respect to their industrial applicability; Case studies and project reports on formal methods related projects with industrial participation (e.g. safety critical systems, mobile systems, object-based distributed systems); etc.

September 05-09	5th joint meeting of the **European Software Engineering Conference** and the **Foundations of Software Engineering Conference** (ESEC/FSE'2005), Lisbon, Portugal. Topics include: Software Specification and Verification, Component-based Software Engineering, Software Engineering Tools and Environments, Software Frameworks and Middleware, Software Engineering and Security, Software Safety and Reliability Engineering, Reengineering and Software Maintenance, Generative Programming and Techniques, Software Evolution and Change Management, Software Economics, etc. Deadline for early registration: July 29, 2005.
	Sept. 05-06 **International Workshop on Principles of Software Evolution** (IWPSE'2005). Topics include: evolution of requirements and environments; architecture for evolution, evolution of architecture; methodology for evolutional design and development; validation and verification of evolution; experience reports and lessons learned from evolutional software systems; etc.
September 07-09	3rd IEEE **International Conference on Software Engineering and Formal Methods** (SEFM'2005), Koblenz, Germany. Topics include: software architectures and their description languages; software specification, validation and verification; integration of formal and informal methods; integration of different formal methods; formal aspects of security and mobility; program analysis; fault-tolerant, real-time and hybrid systems; analysis of safety-critical systems; light-weight formal methods; CASE tools and tool integration; application to industrial cases; socio-economic implications of the use of formal methods; etc.
September 07-09	12th **International Static Analysis Symposium** (SAS'2005), London, UK.
September 08-09	2nd **International Workshop on Rapid Integration of Software Engineering techniques** (RISE'2005), Heraklion, Crete, Greece. Topics include: Software reuse, Lightweight or practice-oriented formal methods, Software processes and software metrics, Design patterns, Defensive programming, Software entropy and software re-factoring, Programming languages, Software dependability and trustworthiness, High-availability or mission-critical systems, Embedded systems and applications, Development environments, Enterprise computing and applications, etc. Deadline for submissions: July 5, 2005.
☺ September 11-14	**Workshop on Language-Based Parallel Programming Models** (WLPP'2005), Poznan, Poland. Topics include: Language and library implementations; Proposals for, and evaluation of, language extensions; Applications development experiences; Comparisons between programming models; Compiler Implementation and Optimization; etc.
September 12-13	5th **International Workshop on Automated Verification of Critical Systems** (AVoCS'2005), Warwick, UK. Deadline for paper submissions: July 25, 2005 (short presentation abstracts).
☺ September 12-14	18th **International Conference on Parallel and Distributed Computing Systems** (PDCS'2005), Las Vegas, Nevada, USA. Topics include: Parallel and Distributed Systems Software; Languages, Compilers and Operating Systems; Libraries and Programming Environments; Message Passing and Distributed Shared Memory Paradigms; Software Development, Services, Support, and Tools; Middleware for Parallel and Distributed Computing; Embedded Systems; Parallel and Distributed Applications; etc.
September 12-16	**International Conference on Practical Software Quality and Testing** (PSQT'2005 North), Minneapolis, Minnesota, USA. Theme: "Software Testing: From Art to Engineering"
☺ September 13-16	**International Conference on Parallel Computing 2005** (ParCo2005), Malaga, Spain. Topics include: applications; software engineering methodologies, methods and tools for developing and maintaining parallel software, incl. parallel programming models and paradigms, development environments, languages, compiling and run-time tools; etc. Deadline for submissions: July 31, 2005 (draft full papers).
September 14-16	10th **European Symposium on Research in Computer Security** (ESORICS'2005), Milan, Italy. Topics include: dependability, formal methods in security, language-based security, etc.
September 18-21	6th **conference on Communicating Process Architectures** (CPA'2005), Eindhoven, The Netherlands. Topics include: Concurrent design patterns and tools; Modelling concurrent software architectures; Safety and security issues (race-hazards, deadlock, livelock, process starvation, ...); Language issues; Applications; etc.

September 19-21	5th **International Conference on Quality Software** (QSIC'2005), Melbourne, Australia. Topics include: Software quality (reliability, safety and security, ...); Methods and tools (design tools, quality tools, ...); Evaluation of software products and components (static and dynamic analysis, validation and verification); Formal methods (program analysis, ...); Applications (component-based systems, distributed systems, embedded systems, enterprise applications, safety critical systems, ...); etc.
September 19-22	11th **International Software Metrics Symposium** (Metrics'2005), Como, Italy. Topics include: Effort and cost estimation; Defect rate and reliability prediction; Quality Assurance; Empirical studies of global software development projects, open source software projects, agile development projects; etc.
September 19-23	9th IEEE **International Enterprise Distributed Object Computing Conference** (EDOC'2005), Enschede, The Netherlands.
September 19-22	2nd **International Workshop on Software Quality** (SOQUA'2005), Erfurt, Germany. Topics include: Communication of current trends related to software quality, Identification of future trends and problems, Metrics for software quality, Formal methods, etc.
September 20-22	8th **International Conference for Young Computer Scientists** (ICYCS'2005), Beijing, China. Topics include: Distributed and parallel processing, Fault-tolerance techniques, Software methodology and engineering techniques, Software reuse, Object-oriented programming, Middleware techniques, Robotics, etc.
September 25-30	21st IEEE **International Conference on Software Maintenance** (ICSM'2005), Budapest, Hungary. Topics include: issues related to maintaining, modifying, enhancing, and testing operational systems, and designing, building, testing, and evolving maintainable systems.
	Sept. 30-Oct. 01 5th IEEE **International Workshop on Source Code Analysis and Manipulation** (SCAM'2005). Topics include: program transformation, abstract interpretation, program slicing, source level software metrics, decompilation, source level testing and verification, source level optimization and program comprehension.
September 26-30	3rd **World Conference for Software Quality** (3WCSQ), Munich, Germany. Topics include: Software Construction, Integration and Testing, Verification and Validation, Risk Management and Problem resolution, Training and Education, Maintenance and Customer Support, Reliability Engineering, Embedded Systems, Medical Devices, Automotive and Automation, Avionics and Transportation Systems, etc.
Sept. 29-Oct. 01	4th **International Conference on Generative Programming and Component Engineering** (GPCE'2005), Tallinn, Estonia. Topics include: Generative techniques for Product lines and architectures, Embedded systems, etc.; Component-based software engineering (Reuse, distributed platforms, distributed systems, evolution, analysis and design patterns, development methods, formal methods); Integration of generative and component-based approaches; Industrial applications; etc. Deadline for early registration: July 29, 2005
October 02-05	25th IFIP WG 6.1 **International Conference on Formal Techniques for Networked and Distributed Systems** (FORTE'2005), Taiwan. Topics include: formal description techniques, embedded systems, tool supports, case studies on industrial projects, etc.
October 02-07	8th **International Conference on Model Driven Engineering Languages and Systems** (MoDELS'2005), Montego Bay, Jamaica. Formerly the UML series of conferences. Topics include: Model-driven development methodologies, approaches, and languages; Empirical studies of modeling and model-driven development; Tool support for any aspect of model-driven development or model use; Semantics of modeling languages; etc.
October 03-07	19th **Brazilian Symposium on Software Engineering** (SBES'2005), Uberlândia, Brazil. Topics include: Distributed Software Engineering; Generative Software Development; Multi-paradigm and Multi-language Modelling and Programming; Object-oriented Techniques; Software Engineering for Embedded and Real-time Software; Software Engineering Tools and Environments; Software Maintenance; Software Quality; Software Reuse; Software Safety and Reliability; Software Security; Software Verification, Validation and Inspection; etc.

October 13-14	**Workshop "Zuverlässigkeit in eingebetteten Systemen"**, Aachen, Germany. Organized by Gesellshaft für Informatik e.V. Fachgruppe "Ada", and Gesellshaft Mess- und Automatisierungstechnik Fachausschus 5.11 "Embedded Software". Topics include (in German): Programmiersprache Ada und Profile (Raven, SPARK), angeladene Hauptvorträge zu Real Time Scheduling und zu Ada0Y, etc. Deadline for submissions: July 31, 2005 (position papers).
☺ October 13-14	3rd **Workshop on Object-oriented Modeling of Embedded Real-Time Systems** (OMER-3), Paderborn, Germany. Topics include: Architectures/frameworks for platform independent, reusable software components; Code-generation; Component interoperability; Formal verification at the model and code level; Software components as products; Software quality; Standards and guidelines; Respective trends in automotive software development; etc.
☺ October 16-20	20th Annual ACM SIGPLAN **Conference on Object-Oriented Programming, Systems, Languages, and Applications** (OOPSLA'2005), San Diego, California, USA. Sponsored by ACM SIGPLAN in cooperation with SIGSOFT.
	☺ October 16 **Workshop on Synchronization and Concurrency in Object-Oriented Languages** (SCOOL'2005). Topics include: Compiler transformations, Concurrent data structure implementations, Expression of concurrency-related design intent, Languages and semantics, Memory models for concurrent object-oriented languages, Synchronization abstractions, etc. Deadline for submissions: July 29, 2005.
October 19-21	17th **Nordic Workshop on Programming Theory** (NWPT'2005), Copenhagen, Denmark. Topics include: Program verification, Formal specification of programs, Real-Time and hybrid systems, Modeling of concurrency, Programming methods, Tools for program construction and verification, etc. Deadline for submissions: September 19, 2005.
October 25-26	**International Conference on Software Testing** (ICSTEST-E'2005), Bilbao, Spain. Topics include: Transportation and Safety-Critical Systems, Industry real experiences, Verification and Validation, Techniques for real time systems, Static and Dynamic analysis, Norms and standards, etc.
October 26-28	20th **International Symposium on Computer and Information Sciences** (ISCIS'2005), Istanbul, Turkey. Topics include: Parallel and Distributed Computing, Programming Languages and Algorithms, Software Engineering, etc.
October 27-28	6th **International Workshop on Advanced Parallel Processing Technologies** (APPT'2005), Hong Kong, China. Topics include: Middleware, Software Tools and Environments, Parallelizing Compilers, Software Engineering issues, Task Scheduling and Load Balancing, Fault tolerance and dependability, etc.
Oct, 30-Nov. 03	24th **Digital Avionics Systems Conference** (DASC'2005), Washington D.C., USA. Theme: "Avionics in a Changing Market Place - Safe and Secure?". Topics include: Software Engineering: Development of large-scale, flight-critical software systems, including processes and formal methods for design, testing and certification; Lean Avionics: Application of continuous improvement principles/practices (lean, six sigma, TQM, CMM, CMMI) to the design, development and sustainment of mission critical avionics systems; Flight Critical Systems: Methods, techniques, and tools for the design, verification, integration, validation, and certification of complex and highly integrated flight critical systems; etc.
☺ Oct. 31-Nov. 04	7th **International Symposium on Distributed Objects and Applications** (DOA'2005), Agia Napa, Cyprus. Topics include: Application case studies of distribution technologies; Design patterns for distributed systems; Distribution technologies for embedded systems; Interoperability between object systems and complementary technologies; Real-time solutions for distributed objects; Scalability for distributed objects and object middleware; Security for distributed object systems; Specification and enforcement of Quality of Service; Technologies for reliability and fault-tolerance; etc.
November 01-04	4th **International Symposium on Formal Methods for Components and Objects** (FMCO'2005), Amsterdam, the Netherlands. Deadline for submissions: September 5, 2005 (title and short abstract), February 28, 2006 (tutorial papers).
November 01-04	7th **International Conference on Formal Engineering Methods** (ICFEM'2005), Manchester, UK. Topics include: all aspects of formal engineering methods, from theoretical work that promises various benefits, to application to real production systems.

☺ November 02-05	3rd **International Symposium on Parallel and Distributed Processing and Applications** (ISPA'2005), Nanjing, China. Topics include: Parallel/distributed system architectures; Tools and environments for software development; Parallel/distributed algorithms; Parallel compilers; Parallel programming languages; Distributed systems; Reliability, fault-tolerance, and security; Parallel/distributed applications; etc.
November 08-11	16th IEEE **International Symposium on Software Reliability Engineering** (ISSRE'2005), Chicago, Illinois, USA. Theme: "Developing High Reliability for Ubiquitous Mobile Applications". Topics include: Software safety analysis, Formal reliability assurance methods, Software testing and verification, Empirical reliability studies, Reliability measurement, Tools and automation, Fault-tolerant and robust software, Security testing, Software certification, End-to-end dependability, etc. Deadline for submissions: August 1, 2005 (industry practice presentations, student papers, fast abstracts).
November 08-11	12th **Working Conference on Reverse Engineering** (WCRE'2005), Pittsburgh, PA, USA. Theme: "Recovering and Reclaiming Architecture" Topics include: Software architecture recovery; Program transformation and refactoring; Object and aspect identification; Preprocessing, parsing and fact extraction; Reverse engineering tool support; Program slicing; Redocumenting legacy systems; Program analysis; Reengineering patterns; etc. Deadline for submissions: September 2, 2005 (workshop papers, tool descriptions).
November 09-11	**European Software Process Improvement and Innovation Conference** (EuroSPI'2005), Budapest, Hungary. Deadline for early registration: October 1, 2005.
♦ November 13-17	2005 ACM Annual **SIGAda International Conference** (SIGAda'2005), Atlanta, Georgia, USA. Sponsored by ACM SIGAda; in cooperation with SIGAPP, SIGCAS, SIGCSE, SIGPLAN, SIGSOFT, and Ada-Europe (ACM approval pending; Cooperation approvals pending).
November 17-18	**XP Day Benelux 2005**, Rotterdam, The Netherlands. Deadline for submissions: July 11, 2005 (sessions).
Nov. 28-Dec. 02	ACM/IFIP/USENIX **International Middleware Conference** (Middleware'2005), Grenoble, France. Deadline for submissions: August 26, 2005 (doctoral symposium).
Nov. 29-Dec. 01	18th **International Conference on Software & Systems Engineering and their Applications** (ICSSEA'2005), Paris, France.
Nov. 29-Dec. 02	5th **International Conference on Integrated Formal Methods** (IFM'2005), Eindhoven, The Netherlands.
☺ December 05-08	6th **International Conference on Parallel and Distributed Computing, Applications, and Techniques** (PDCAT'2005), Dalian, China. Topics include: Formal methods and programming Languages, Parallelizing compilers, Component-based and OO Technology, Tools and environments for software development, etc.
December 10	Birthday of Lady Ada Lovelace, born in 1815. Happy Programmers' Day!
December 12-14	11th **International Symposium Pacific Rim Dependable Computing** (PRDC'2005), Changsha, Hunan, China. Topics include: Software and hardware reliability, testing, verification and validation; Dependability measurement, modeling and evaluation; Safety-critical systems and software; Tools for design and evaluation of dependable systems; Dependability issues in distributed and parallel systems; Dependability issues in real-time systems; etc.
December 15-17	12th **Asia-Pacific Software Engineering Conference** (APSEC'2005), Taipei, Taiwan. Topics include: Software Formal Methods, Software Process Improvement, Cost Estimation, Risk Management, Quality Management, Object-Oriented Technology, etc. Deadline for submissions: August 15, 2005 (papers).
December 18-21	12th IEEE **International Conference on High Performance Computing** (HiPC'2005), Goa, India. Topics include: Scientific/Engineering Applications, System Design for High Reliability, Parallel and Distributed Computing, Heterogeneous Computing, Embedded Applications and Systems, Parallel Languages and Programming Environments, Load Balancing and Scheduling, etc.

2006

January 04-07	**Software Technology Track** of the 39th **Hawaii International Conference on System Sciences** (HICSS-39), Kauai, Haway, USA. Includes mini-tracks on: Strategic Software Engineering; Adaptive and Evolvable Software Systems; etc.
January 11-13	33rd Annual ACM SIGPLAN-SIGACT **Symposium on Principles of Programming Languages** (POPL'2006), Charleston, South Carolina, USA. Topics include: fundamental principles and important innovations in the design, definition, analysis, transformation, implementation and verification of programming languages, programming systems, and programming abstractions. Deadline for submissions: July 18, 2005.
January 14	**2006 International Workshop on Foundations and Developments of Object-Oriented Languages** (FOOL/WOOD'2006), Charleston, South Carolina, USA. Following POPL'2006. Topics include: language semantics, type systems, program analysis and verification, concurrent and distributed languages, language-based security, etc. Deadline for submissions: October 3, 2005.
February 13-17	5th **International Conference on COTS-Based Software Systems** (ICCBSS'2006), Orlando, Florida, USA. Theme: "Pushing the COTS Envelope". Deadline for submissions: July 15, 2005 (technical papers, experience reports, panels, tutorials, workshops).
☺ April 18	**Workshop on Secure Software Engineering Education & Training** (WSSEET'2006), Oahu, Hawaii, USA. Topics include: experience, current situation, and future of education and training in software engineering of (more) secure software. Deadline for submissions: October 13, 2005 (position papers, papers, panels).
April 23-27	21st ACM **Symposium on Applied Computing** (SAC'2006), Dijon, France. Includes tracks on: Software Engineering, etc. Deadline for submissions: September 3, 2005 (software engineering track papers).
♦ June 05-09	11th **International Conference on Reliable Software Technologies** - Ada-Europe'2006, Porto, Portugal. Sponsored by Ada-Europe, in cooperation with ACM SIGAda (approval pending). Deadline for submissions: October 30, 2005 (papers, tutorials, workshops).
December 10	Birthday of Lady Ada Lovelace, born in 1815. Happy Programmers' Day!

2007

June 09-16	3rd **History of Programming Languages Conference** (HOPL-III), San Diego, CA, USA. Co-located with FCRC'2007. Deadline for submissions: July 8, 2005 (1 page abstract), August 15, 2005 (full papers), August 2006 (reworked full papers).

Call for Papers

11th International Conference on Reliable Software Technologies – Ada-Europe 2006

5-9 June 2006, Porto, Portugal

`http://www.ada-europe.org/conference2006.html`

General Information

The 11th International Conference on Reliable Software Technologies (Ada-Europe 2006) will take place in Porto, Portugal. Following the usual style, the conference will span a full week, including a three-day technical program and vendor exhibitions from Tuesday to Thursday, along with parallel workshops and tutorials on Monday and Friday.

Schedule

30 October 2005	Submission of papers, workshop/tutorial proposals
20 January 2006	Notification to authors
20 February 2006	Camera-ready papers required
5-9 June 2006	Conference

Topics

In the last decade the conference has established itself as an international forum for providers and practitioners of, and researchers into, 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 variety of application domains. The program will allow ample time for keynotes, Q&A sessions, panel discussions and social events. Participants will include practitioners and researchers from industry, academia and government organizations interested in furthering the development of reliable software technologies. To mark the completion of the technical work for the Ada language standard revision process, contributions that present and discuss the potential of the revised language are particularly sought after.

For papers, tutorials, and workshop proposals, the topics of interest include, but are not limited to:

- **Methods and Techniques for Software Development and Maintenance**: Requirements Engineering, Object-Oriented Technologies, Formal Methods, Re-engineering and Reverse Engineering, Reuse, Software Management Issues
- **Software Architectures:** Patterns for Software Design and Composition, Frameworks, Architecture-Centered Development, Component and Class Libraries, Component-Based Design
- **Enabling Technology**: CASE Tools, Software Development Environments and Project Browsers, Compilers, Debuggers, Run-time Systems
- **Software Quality:** Quality Management and Assurance, Risk Analysis, Program Analysis, Verification, Validation, Testing of Software Systems
- **Critical Systems**: Real-Time, Distribution, Fault Tolerance, Information Technology, Safety, Security
- **Mainstream and Emerging Applications**: Multimedia and Communications, Manufacturing, Robotics, Avionics, Space, Health Care, Transportation
- **Ada Language and Technology**: Programming Techniques, Object-Oriented Programming, Concurrent Programming, Distributed Programming, Bindings and Libraries, Evaluation & Comparative Assessments, Critical Review of Language Enhancements, Novel Support Technology, HW/SW platforms
- **Experience Reports**: Experience Reports, Case Studies and Comparative Assessments, Management Approaches, Qualitative and Quantitative Metrics, Experience Reports on **Education and Training** Activities with bearing on any of the conference topics

Conference Chair

Luís Miguel Pinho
Polytechnic Institute of Porto, Portugal
lpinho@dei.isep.ipp.pt

Program Co-Chairs

Luís Miguel Pinho
Polytechnic Institute of Porto, Portugal
lpinho@dei.isep.ipp.pt

Michael González Harbour
Universidad de Cantabria, Spain
mgh@unican.es

Tutorial Chair

Jorge Real
U. P. Valencia, Spain
jorge@disca.upv.es

Exhibition Chair

José Ruiz
AdaCore, France
ruiz@adacore.com

Publicity Chair

Dirk Craeynest
Aubay Belgium & K.U.Leuven, Belgium
Dirk.Craeynest@cs.kuleuven.be

Local Chair

Sandra Almeida
Polytechnic Institute of Porto, Portugal
salmeida@dei.isep.ipp.pt

Ada-Europe Conference Liaison

Laurent Pautet
Telecom Paris, France
pautet@enst.fr

In cooperation with

SIGAda
(approval pending)

Submissions

Authors are invited to submit original contributions. Paper submissions shall be in English, should be complete and should not exceed 20 double-spaced pages in length. Authors should submit their work via the Web submission system accessible from the conference Home page. The preferred format for submission is PDF. Postscript can also be accepted, as long as it was generated selecting the "optimize for portability" option in the used printer driver. Submissions by other means and formats will not be accepted. If you do not have easy access to the Internet, or you do not have an appropriate Web browser, please contact the Program Co-Chair Luís Miguel Pinho, whose address details are on the flip side of this call as well as on the conference Home page.

Proceedings

The authors of accepted papers shall prepare their camera-ready submissions in full conformance with the LNCS style, not exceeding 12 pages and strictly by February 20, 2006. For format and style guidelines authors should refer to: http://www.springer.de/comp/lncs/authors.html. Failure to comply will prevent the paper from appearing in the conference proceedings. The conference proceedings including all accepted papers will be published in the Lecture Notes in Computer Science (LNCS) series by Springer Verlag, which will be available at the start of the conference.

Awards

Ada-Europe will offer honorary awards for the best paper and the best presentation, which will be presented during the banquet and at the close of the conference respectively.

Call for Tutorials

Tutorials should address subjects that fall within the thrust of the conference and may be proposed as either half- or a 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 Jorge Real. The providers of 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 accordingly be halved. The Ada User Journal will offer space for the publication of summaries of the accepted tutorial in issues preceding and/or following the conference.

Call for Workshops

Workshops on themes within the conference scope may be arranged to discuss matters of immediate technical interest as well as to foster action on longer-term technical objectives. Proposals may be submitted for half- or full-day workshops, to be scheduled on either ends of the main conference. Workshop proposals should be submitted by e-mail to the Conference Chair Luís Miguel Pinho The workshop organiser shall also commit to preparing proceedings for timely publication in the Ada User Journal.

Exhibition

Commercial exhibitions will span the three days of the main conference. Vendors and providers of software products and services should contact the Exhibition Chair José Ruiz as soon as possible for further information and for allowing suitable planning of the exhibition space and time.

Reduced Fees for Students

A small number of grants are available for students who will (co-)author and present papers at the conference. A reduction of 25% will be made to the conference fee. Contact the Conference Chair Luís Miguel Pinho for details.

Rationale for Ada 2005: 2 Access Types

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements concerning access types for Ada 2005.

Ada 2005 permits all access types to be access to constant types and to indicate that null is not an allowed value in all contexts. Anonymous access types are permitted in more contexts than just as access parameters and discriminants; they can also be used for variables and all components of composite types. This further use of access types is of considerable value in object oriented programming by reducing the need for (unnecessary) explicit type conversions.

A further major improvement concerns access to subprogram types which are now allowed to be anonymous in line with access to object types. This permits so-called "downward closures" and allows the flexible use of procedures as parameters of subprograms and thereby avoids excessive use of generic units.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] does not specifically mention access types as an area needing attention. Access types are, of course, more of a tactical detail than a strategic issue and so this is not surprising.

However, the guidance document strongly emphasizes improvements to object oriented programming and the use of access types figures highly in that area. Indeed one of the motivations for changes was to reduce the number of explicit access type conversions required for OOP.

The guidance document also asks for "improvements that will remedy shortcomings in Ada". The introduction of anonymous access-to-subprogram types comes into that category in the minds of many users.

The following Ada issues cover the relevant changes and are described in detail in this paper:

230 Generalized use of anonymous access types

231 Access to constant parameters, null-excluding types

254 Anonymous access to subprogram types

318 Limited and anonymous access return types

363 Eliminating access subtype problems

382 Current instance rule and anonymous access types

384 Discriminated type conversion rules

385 Stand-alone objects of anonymous access types

392 Prohibit unsafe array conversions

402 Access discriminants of nonlimited types

404 Not null and all in access parameters and types

406 Aliased permitted with anonymous access types

409 Conformance with access to subprogram types

416 Access results, accessibility and return statements

420 Resolution of universal operations in Standard

423 Renaming, null exclusion and formal objects

These changes can be grouped as follows.

First, there is a general orthogonalization of the rules regarding whether the designated type is constant and whether the access subtype includes null (231, part of 404, part of 423).

A major change is the ability to use anonymous access types more widely (230, part of 318, 385, 392, part of 404, 406, part of 416, part of 420). This was found to require some redefinition of the rules regarding the use of a type name within its own definition (382). Access discriminants are now also permitted with nonlimited types (402).

The introduction of anonymous access-to-subprogram types enables local subprograms to be passed as parameters to other subprograms (254, 409). This has been a feature of many other programming languages for over 40 years and its omission from Ada has always been both surprising and irritating and forced the excessive use of generics.

Finally there are some corrections to the rules regarding changing discriminants which prevent attempting to access components of variants that do not exist (363). There is also a change to the rules concerning type conversions and discriminants to make them symmetric (384).

2 Null exclusion and constant

In Ada 95, anonymous access types and named access types have unnecessarily different properties. Furthermore anonymous access types only occur as access parameters and access discriminants.

Anonymous access types in Ada 95 never have null as a value whereas named access types always have null as a value. Suppose we have the following declarations

```
type T is
  record
    Component: Integer;
  end record;

type Ref_T is access T;
T_Ptr: Ref_T;
```

Note that T_Ptr by default will have the value **null**. Now suppose we have a procedure with an access parameter thus

```
procedure P(A: access T) is
  X: Integer;
begin
  X := A.Component;     -- read a component of A
                        -- no check for null in 95
  ...
end P;
```

In Ada 95 an access parameter such as A can never have the value null and so there is no need to check for null when doing a dereference such as reading the component A.Component. This is assured by always performing a check when P is called. So calling P with an actual parameter whose value is null such as P(T_Ptr) causes Constraint_Error to be raised at the point of call. The idea was that within P we would have more efficient code for dereferencing and dispatching at the cost of just one check when the procedure is called. Such an access parameter we now refer to as null-excluding.

Ada 2005 extends this idea of null-excluding access types to named access types as well. Thus we can write

```
type Ref_NNT is not null access T;
```

In this case an object of the type Ref_NNT cannot have the value null. An immediate consequence is that all such objects should be explicitly initialized – they will otherwise be initialized to null by default and this will raise Constraint_Error.

Since the null excluding property can now be given explicitly for named types, it was decided that for uniformity, anonymous access types should follow the same rule whenever possible. So, if we want an access parameter such as A to be null excluding in Ada 2005 then we have to indicate this in the same way

```
procedure PNN(A: not null access T) is
  X: Integer;
begin
  X := A.Component;     -- read a component of A
                        -- no check for null in 2005
  ...
end PNN;
```

This means of course that the original procedure

```
procedure P(A: access T) is
  X: Integer;
begin
  X := A.Component;     -- read a component of A
                        -- check for null in 2005
  ...
end P;
```

behaves slightly differently in Ada 2005 since A is no longer of a null-excluding type. There now has to be a check when accessing the component of the record because null is now an allowed value of A. So in Ada 2005, calling P with a null parameter results in Constraint_Error being raised within P only when we attempt to do the dereference, whereas in Ada 95 it is always raised at the point of call.

This is of course technically an incompatibility of an unfortunate kind. Here we have a program that is legal in both Ada 95 and Ada 2005 but it behaves differently at execution time in that Constraint_Error is raised at a different place. But of course, in practice if such a program does raise Constraint_Error in this way then it clearly has a bug and so the difference does not really matter.

Various alternative approaches were considered in order to eliminate this incompatibility but they all seemed to be ugly and it was felt that it was best to do the proper thing rather than have a permanent wart.

However the situation regarding controlling access parameters is somewhat different. Remember that a controlling parameter is a parameter of a tagged type where the operation is primitive – that is declared alongside the tagged type in a package specification (or inherited of course). Thus consider

```
package PTT is
  type TT is tagged
    record
      Component: Integer;
    end record;

  procedure Op(X: access TT); -- primitive operation
  ...
end PTT;
```

The type TT is tagged and the procedure Op is a primitive operation and so the access parameter X is a controlling parameter.

In this case the anonymous access (sub)type is still null excluding as in Ada 95 and null is not permitted as a parameter. The reason is that controlling parameters provide the tag for dispatching and null has no tag value. Remember that all controlling parameters have to have the same tag. We can add **not null** to the parameter specification if we wish but to require it explicitly for all controlling parameters was considered to be too much of an incompatibility. But in newly written programs, we should be encouraged to write **not null** explicitly in order to avoid confusion during maintenance.

Another rule regarding null exclusion is that a type derived from a null-excluding type is also null excluding. Thus given

```
type Ref_NNT is not null access T;
type Another_Ref_NNT is new Ref_NNT;
```

then Another_Ref_NNT is also null excluding. On the other hand if we start with an access type that is not null excluding then a derived type can be null excluding or not thus

```
type Ref_T is access T;
type Another_Ref_T is new Ref_T;
type ANN_Ref_T is new not null Ref_T;
```

then Another_Ref_T is not null excluding but ANN_Ref_T is null excluding.

A technical point is that all access types including anonymous access types in Ada 2005 have null as a value whereas in Ada 95 the anonymous access types did not. It is only subtypes in Ada 2005 that do not always have null as a value. Remember that Ref_NNT is actually a first-named subtype.

An important advantage of all access types having null as a value is that it makes interfacing to C much easier. If a parameter in C has type *t then the corresponding parameter in Ada can have type **access** T and if the C routine needs null passed sometimes then all is well – this was a real pain in Ada 95.

An explicit null exclusion can also be used in object declarations much like a constraint. Thus we can have

```
type Ref_Int is access Integer;
X: not null Ref_Int := Some_Integer'Access;
```

Note that we must initialize X otherwise the default initialization with **null** will raise Constraint_Error.

In some ways null exclusions have much in common with constraints. We should compare the above with

```
Y: Integer range 1 .. 10;
...
Y := 0;
```

Again Constraint_Error is raised because the value is not permitted for the subtype of Y. A difference however is that in the case of X the check is Access_Check whereas in the case of Y it is Range_Check.

The fact that a null exclusion is not actually classified as a constraint is seen by the syntax for subtype_indication which in Ada 2005 is

```
subtype_indication ::=
                [null_exclusion] subtype_mark [constraint]
```

An explicit null exclusion can also be used in subprogram declarations thus

```
function F(X: not null Ref_Int) return not null Ref_Int;
procedure P(X: in not null Ref_Int);
procedure Q(X: in out not null Ref_Int);
```

But a difference between null exclusions and constraints is that although we can use a null exclusion in a parameter specification we cannot use a constraint in a parameter specification. Thus

```
procedure P(X: in not null Ref_Int);       -- legal
procedure Q(X: in Integer range 1 .. N);   -- illegal
```

But null exclusions are like constraints in that they are both used in defining subtype conformance and static matching.

We can also use a null exclusion with access-to-subprogram types including protected subprograms.

```
type F is access function (X: Float) return Float;
Fn: not null F := Sqrt'Access;
```

and so on.

A null exclusion can also be used in object and subprogram renamings. We will consider subprogram renamings here and object renamings in the next section when we discuss anonymous access types. This is an area where there is a significant difference between null exclusions and constraints.

Remember that if an entity is renamed then any constraints are unchanged. We might have

```
procedure P(X: Positive);
...
procedure Q(Y: Natural) renames P;
...
Q(0);                  -- raises Constraint_Error
```

The call of Q raises Constraint_Error because zero is not an allowed value of Positive. The constraint Natural on the renaming is completely ignored (Ada has been like that since time immemorial).

We would have preferred that this sort of peculiar behaviour did not extend to null exclusions. However, we already have the problem that a controlling parameter is always null excluding even if it does not say so. So the rule adopted generally with null exclusions is that "null exclusions never lie". In other words, if we give a null exclusion then the entity must be null excluding; however, if no null exclusion is given then the entity might nevertheless be null excluding for other reasons (as in the case of a controlling parameter).

So consider

```
procedure P(X: not null access T);
...
procedure Q(Y: access T) renames P;          -- OK
...
Q(null);               -- raises Constraint_Error
```

The call of Q raises Constraint_Error because the parameter is null excluding even though there is no explicit null exclusion in the renaming. On the other hand (we assume that X is not a controlling parameter)

```
procedure P(X: access T);
...
procedure Q(Y: not null access T) renames P;  -- NO
```

is illegal because the null exclusion in the renaming is a lie.

However, if P had been a primitive operation of T so that X was a controlling parameter then the renaming with the null exclusion would be permitted.

Care needs to be taken when a renaming itself is used as a primitive operation. Consider

```
package P is
  type T is tagged ...
  procedure One(X: access T);              -- is null excl

  package Inner is
    procedure Deux(X: access T);           -- not null excl
    procedure Trois(X: not null access T);    -- null excl
  end Inner;

  use Inner;

  procedure Two(X: access T) renames Deux;     -- NO
  procedure Three(X: access T) renames Trois;  -- OK
  ...
```

The procedure One is a primitive operation of T and its parameter X is therefore a controlling parameter and so is null excluding even though this is not explicitly stated. However, the declaration of Two is illegal. It is trying to be a dispatching operation of T and therefore its controlling parameter X has to be null excluding. But Two is a renaming of Deux whose corresponding parameter is not null excluding and so the renaming is illegal. On the other hand the declaration of Three is permitted because the parameter of Trois is null excluding.

The other area that needed unification concerned **constant**. In Ada 95 a named access type can be an access to constant type rather than an access to variable type thus

```
type Ref_CT is access constant T;
```

Remember that this means that we cannot change the value of an object of type T via the access type.

Remember also that Ada 95 introduced more general access types whereas in Ada 83 all access types were pool specific and could only access values created by an allocator. An access type in Ada 95 can also refer to any object marked **aliased** provided that the access type is declared with **all** thus

```
type Ref_VT is access all T;
X: aliased T;
R: Ref_VT := X'Access;
```

So in summary, Ada 95 has three kinds of named access types

```
access T;            -- pool specific only, read & write
access all T         -- general, read & write
access constant T    -- general, read only
```

But in Ada 95, the distinction between variable and constant access parameters is not permitted. Ada 2005 rectifies this by permitting **constant** with access parameters. So we can write

```
procedure P(X: access constant T);     -- legal 2005
procedure P(X: access T);
```

Observe however, that **all** is not permitted with access parameters. Ordinary objects can be constant or variable thus

```
C: constant Integer := 99;
V: Integer;
```

and access parameters follow this pattern. It is named access types that are anomalous because of the need to distinguish pool specific types for compatibility with Ada 83 and the subsequent need to introduce **all**.

In summary, Ada 2005 access parameters can take the following four forms

```
procedure P1(X: access T);
procedure P2(X: access constant T);
procedure P3(X: not null access T);
procedure P4(X: not null access constant T);
```

Moreover, as mentioned above, controlling parameters are always null excluding even if this is not stated and so in that case P1 and P3 are equivalent. Controlling parameters can also be constant in which case P2 and P4 are equivalent.

Similar rules apply to access discriminants; thus they can be null excluding and/or access to constant.

3 Anonymous access types

As just mentioned, Ada 95 permits anonymous access types only as access parameters and access discriminants. And in the latter case only for limited types. Ada 2005 sweeps away these restrictions and permits anonymous access types quite freely.

The main motivation for this change concerns type conversion. It often happens that we have a type T somewhere in a program and later discover that we need an access type referring to T in some other part of the program. So we introduce

```
type Ref_T is access all T;
```

And then we find that we also need a similar access type somewhere else and so declare another access type

```
type T_Ptr is access all T;
```

If the uses of these two access types overlap then we will find that we have explicit type conversions all over the place despite the fact that they are really the same type. Of course one might argue that planning ahead would help a lot but, as we know, programs often evolve in an unplanned way.

A more important example of the curse of explicit type conversion concerns object oriented programming. Access types feature quite widely in many styles of OO programming. We might have a hierarchy of geometrical object types starting with a root abstract type Object thus

```
type Object is abstract;
type Circle is new Object with ...

type Polygon is new Object with ...
type Pentagon is new Polygon with ...

type Triangle is new Polygon with ...
type Equilateral_Triangle is new Triangle with ...
```

then we might well find ourselves declaring named access types such as

```
type Ref_Object is access all Object'Class;
type Ref_Circle is access all Circle;
type Ref_Triangle is access all Triangle'Class;
type Ref_Equ_Triangle is access all Equilateral_Triangle;
```

Conversion between these clearly ought to be permitted in many cases. In some cases it can never go wrong and in others a run time check is required. Thus a conversion between a Ref_Circle and a Ref_Object is always possible because every value of Ref_Circle is also a value of Ref_Object but the reverse is not the case. So we might have

```
RC: Ref_Circle := A_Circle'Access;
RO: Ref_Object;
...
RO := Ref_Object(RC); --- explicit conversion, no check
...
RC := Ref_Circle(RO);  -- needs a check
```

However, it is a rule of Ada 95 that type conversions between these named access types have to be explicit and give the type name. This is considered to be a nuisance by many programmers because such conversions are allowed without naming the type in other OO languages. It would not be quite so bad if the explicit conversion were only required in those cases where a run time check was necessary.

Moreover, these are trivial (view) conversions since they are all just pointers and no actual change of value takes place anyway; all that has to be done is to check that the value is a legal reference for the target type and in many cases this is clear at compilation. So requiring the type name is very annoying.

In fact the only conversions between named tagged types (and named access types) that are allowed implicitly in Ada are conversions to a class wide type when it is initialized or when it is a parameter (which is really the same thing).

It would have been nice to have been able to relax the rules in Ada 2005 perhaps by saying that a named conversion is only required when a run time check is required. However, such a change would have caused lots of existing programs to become ambiguous.

So, rather than meddle with the conversion rules, it was instead decided to permit the use of anonymous access types in more contexts in Ada 2005. Anonymous access types have the interesting property that they are anonymous and so necessarily do not have a name that could be used in a conversion. Thus we can have

```
RC: access Circle := A_Circle'Access;
RO: access Object'Class;          -- default null
...
RO := RC;       -- implicit conversion, no check
```

On the other hand we cannot write

```
RC := RO;       -- implicit conversion, needs a check
```

because the general rule is that if a check is required then the conversion must be explicit. So typically we will still need to introduce named access types for some conversions.

We can of course also use null exclusion with anonymous access types thus

```
RC: not null access Circle := A_Circle'Access;
RO: not null access Object'Class;           -- careful
```

The declaration of RO is unfortunate because no initial value is given and the default of null is not permitted and so it will raise Constraint_Error; a worthy compiler will detect this during compilation and give us a friendly warning.

Note carefully that we never write **all** with anonymous access types.

We can of course also use **constant** with anonymous access types. Note carefully the difference between the following

```
ACT: access constant T := T1'Access;
CAT: constant access T := T1'Access;
```

In the first case ACT is a variable and can be used to access different objects T1 and T2 of type T. But it cannot be used to change the value of those objects. In the second case CAT is a constant and can only refer to the object given in its initialization. But we can change the value of the object that CAT refers to. So we have

```
ACT := T2'Access;         -- legal, can assign
ACT.all := T2;            -- illegal, constant view
CAT := T2'Access;         -- illegal, cannot assign
CAT.all := T2;            -- legal, variable view
```

At first sight this may seem confusing and consideration was given to disallowing the use of constants such as CAT (but permitting ACT which is probably more useful since it protects the accessed value). But the lack of orthogonality was considered very undesirable. Moreover Ada is a left to right language and we are familiar with equivalent constructions such as

```
type CT is access constant T;
ACT: CT;
```

and

```
type AT is access T;
CAT: constant AT;
```

(although the alert reader will note that the latter is illegal because I have foolishly used the reserved word **at** as an identifier).

We can of course also write

```
CACT: constant access constant T := T1'Access;
```

The object CACT is then a constant and provides read-only access to the object T1 it refers to. It cannot be changed to refer to another object such as T2 nor can the value of T1 be changed via CACT.

An object of an anonymous access type, like other objects, can also be declared as aliased thus

```
X: aliased access T;
```

although such constructions are likely to be used rarely.

Anonymous access types can also be used as the components of arrays and records. In the Introduction we saw that rather than having to write

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Integer;
  end record;
```

we can simply write

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

and this not only avoids have to declare the named access type Cell_Ptr but it also avoids the need for the incomplete type declaration of Cell.

Permitting this required some changes to a rule regarding the use of a type name within its own declaration – the so-called current instance rule.

The original current instance rule was that within a type declaration the type name did not refer to the type itself but to the current object of the type. The following task type declaration illustrates both a legal and illegal use of the task type name within its own declaration. It is essentially an extract from a program in Section 18.10 of [2] which finds prime numbers by a multitasking implementation of the Sieve of Eratosthenes. Each task of the type is associated with a prime number and is responsible for removing multiples of that number and for creating the next task when a new prime number is discovered. It is thus quite natural that the task should need to make a clone of itself.

```
task type TT (P: Integer) is
  ...
end;

type ATT is access TT;

task body TT is
  function Make_Clone(N: Integer) return ATT is
  begin
    return new TT(N);  -- illegal
  end Make_Clone;

  Ref_Clone: ATT;
  ...
begin
  ...
  Ref_Clone := Make_Clone(N);
  ...
  abort TT;                -- legal
  ...
end TT;
```

The attempt to make a slave clone of the task in the function Make_Clone is illegal because within the task type its name refers to the current instance and not to the type. However, the abort statement is permitted and will abort the current instance of the task. In this example the solution is simply to move the function Make_Clone outside the task body.

However, this rule would have prevented the use of the type name Cell to declare the component Next within the type Cell and this would have been infuriating since the linked list paradigm is very common.

In order to permit this the current instance rule has been changed in Ada 2005 to allow the type name to denote the type itself within an anonymous access type declaration (but not a named access type declaration). So the type Cell is permitted.

Note however that in Ada 2005, the task TT still cannot contain the declaration of the function Make_Clone. Although we no longer need to declare the named type ATT since we can now declare Ref_Clone as

```
Ref_Clone: access TT;
```

and we can declare the function as

```
  function Make_Clone(N: Integer) return access TT is
  begin
    return new TT(N);
  end Make_Clone;
```

where we have an anonymous result type, nevertheless the allocator **new** TT inside Make_Clone remains illegal if Make_Clone is declared within the task body TT. But such a use is unusual and declaring a distinct external function is hardly a burden.

To be honest we can simply declare a subtype of a different name outside the task

```
subtype XTT is TT;
```

and then we can write **new** XTT(N); in the function and keep the function hidden inside the task. Indeed we don't need the function anyway because we can just write

```
Ref_Clone := new XTT(N);
```

in the task body.

The introduction of the wider use of anonymous access types requires some revision to the rules concerning type comparisons and conversions. This is achieved by the introduction of a type *universal_access* by analogy with the types *universal_integer* and *universal_real*. Two new equality operators are defined in the package Standard thus

```
function "=" (Left, Right: universal_access)
                                        return Boolean;
function "/=" (Left, Right: universal_access)
                                        return Boolean;
```

The literal **null** is now deemed to be of type universal_access and appropriate conversions are defined as well. These new operations are only applied when at least one of the arguments is of an anonymous access types (not counting **null**).

Interesting problems arise if we define our own equality operation. For example, suppose we wish to do a deep comparison on two lists defined by the type Cell. We might decide to write a recursive function with specification

```
function "=" (L, R: access Cell) return Boolean;
```

Note that it is easier to use access parameters rather than parameters of type Cell itself because it then caters naturally for cases where null is used to represent an empty list. We might attempt to write the body as

```
function "=" (L, R: access Cell) return Boolean is
begin
  if L = null or R = null then      -- wrong =
    return L = R;                   -- wrong =
  elsif L.Value = R.Value then
    return L.Next = R.Next;         -- recurses OK
  else
    return False;
  end if;
end "=" ;
```

But this doesn't work because the calls of "=" in the first two lines recursively call the function being declared whereas we want to call the predefined "=" in these cases.

The difficulty is overcome by writing Standard."=" thus

```
  if Standard."=" (L, null) or Standard."=" (R, null) then
    return Standard."=" (L, R);
```

The full rules regarding the use of the predefined equality are that it cannot be used if there is a user-defined primitive equality operation for either operand type unless we use the prefix Standard. A similar rule applies to fixed point types as we shall see in a later paper.

Another example of the use of the type Cell occurred in the previous paper when we were discussing type extension at nested levels. That example also illustrated that access types have to be named in some circumstances such as when they provide the full type for a private type. We had

```
package Lists is
  type List is limited private;     -- private type
  ...
private
  type Cell is
    record
      Next: access Cell;            -- anonymous type
      C: Colour;
    end record;

  type List is access Cell;         -- full type
end;

package body Lists is
  procedure Iterate(IC: in Iterator'Class; L: in List) is
    This: access Cell := L;         -- anonymous type
  begin
    while This /= null loop
      IC.Action(This.C);            -- dispatches
      This := This.Next;
    end loop;
  end Iterate;
end Lists;
```

In this case we have to name the type List because it is a private type. Nevertheless it is convenient to use an anonymous access type to avoid an incomplete declaration of Cell.

In the procedure Iterate the local variable This is also of an anonymous type. It is interesting to observe that if This had been declared to be of the named type List then we would have needed an explicit conversion in

```
      This := List(This.Next);      -- explicit conversion
```

Remember that we *always* need an explicit conversion when converting to a named access type. There is clearly an art in using anonymous types to best advantage.

The Introduction showed a number of other uses of anonymous access types in arrays and records and as function results when discussing Noah's Ark and other animal situations. We will now turn to more weighty matters.

An important matter in the case of access types is accessibility. The accessibility rules are designed to prevent dangling references. The basic rule is that we cannot create an access value if the object referred to has a lesser lifetime than the access type.

However there are circumstances where the rule is unnecessarily severe and that was one reason for the introduction of access parameters. Perhaps some recapitulation of the problems would be helpful. Consider

```
type T is ...
Global: T;
type Ref_T is access all T;
Dodgy: Ref_T;

procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);              -- dynamic check
end P;

procedure Q(Ptr: Ref_T) is
begin
  ...
  Dodgy := Ptr;                     -- legal
end Q;
...
declare
  X: aliased T;
begin
  P(X'Access);                      -- legal
  Q(X'Access);                      -- illegal
end;
```

Here we have an object X with a short lifetime and we must not squirrel away an access referring to X in an object with a longer lifetime such as Dodgy. Nevertheless we want to manipulate X indirectly using a procedure such as P.

If the parameter were of a named type such as Ref_T as in the case of the procedure Q then the call would be illegal since within Q we could then assign to a variable such as Dodgy which would then retain the "address" of X after X had ceased to exist.

However, the procedure P which uses an access parameter permits the call. The reason is that access parameters carry dynamic accessibility information regarding the actual parameter. This extra information enables checks to be performed only if we attempt to do something foolish within the procedure such as make an assignment to Dodgy. The conversion to the type Ref_T in this assignment fails dynamically and disaster is avoided.

But note that if we had called P with

```
P(Global'Access);
```

where Global is declared at the same level as Ref_T then the assignment to Dodgy would be permitted.

The accessibility rules for the new uses of anonymous access types are very simple. The accessibility level is simply the level of the enclosing declaration and no dynamic information is involved. (The possibility of preserving dynamic information was considered but this would have led to inefficiencies at the points of use.)

In the case of a stand-alone variable such as

```
V: access Integer;
```

then this is essentially equivalent to

```
type anon is access all Integer;
V: anon;
```

A similar situation applies in the case of a component of a record or array type. Thus if we have

```
type R is
  record
    C: access Integer;
    ...
  end record;
```

then this is essentially equivalent to

```
type anon is access all Integer;
type R is
  record
    C: anon;
    ...
  end record;
```

Further if we now declare a derived type then there is no new physical access definition, and the accessibility level is that of the original declaration. Thus consider

```
procedure Proc is
  Local: aliased Integer;
  type D is new R;
  X: D := D'(C => Local'Access, ... );        -- illegal
begin
  ...
end Proc;
```

In this example the accessibility level of the component C of the derived type is the same as that of the parent type R and so the aggregate is illegal. This somewhat surprising rule is necessary to prevent some very strange problems which we will not explore in this paper.

One consequence of which users should be aware is that if we assign the value in an access parameter to a local variable of an anonymous access type then the dynamic accessibility of the actual parameter will not be held in the local variable. Thus consider again the example of the procedure P containing the assignment to Dodgy

```
procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);              -- dynamic check
end P;
```

and this variation in which we have introduced a local variable of an anonymous access type

```
procedure P1(Ptr: access T) is
  Local_Ptr: access T;
begin
  ...
  Local_Ptr := Ptr;                 -- implicit conversion
  Dodgy := Ref_T(Local_Ptr);        -- static check, illegal
end P1;
```

Here we have copied the value in the parameter to a local variable before attempting the assignment to Dodgy. (Actually it won't compile but let us analyze it in detail anyway.)

The conversion in P using the access parameter Ptr is dynamic and will only fail if the actual parameter has an accessibility level greater than that of the type Ref_T. So it will fail if the actual parameter is X and so raise Program_Error but will pass if it has the same level as the type Ref_T such as the variable Global.

In the case of P1, the assignment from Ptr to Local_Ptr involves an implicit conversion and static check which always passes. (Remember that implicit conversions are never allowed if they involve a dynamic check.) However, the conversion in the assignment to Dodgy in P1 is also static and will always fail no matter whether X or Global is passed as actual parameter.

So the effective behaviours of P and P1 are the same if the actual parameter is X (they both fail, although one dynamically and the other statically) but will be different if the actual parameter has the same level as the type Ref_T such as the variable Global. The assignment to Dodgy in P will work in the case of Global but the assignment to Dodgy in P1 never works.

This is perhaps surprising, an apparently innocuous intermediate assignment has a significant effect because of the implicit conversion and the consequent loss of the accessibility information. In practice this is very unlikely to be a problem. In any event programmers are aware that access parameters are special and carry dynamic information.

In this particular example the loss of the accessibility information through the use of the intermediate stand-alone variable is detected at compile time. More elaborate examples can be constructed whereby the problem only shows up at execution time. Thus suppose we introduce a third procedure Agent and modify P and P1 so that we have

```
procedure Agent(A: access T) is
begin
   Dodgy := Ref_T(A);              -- dynamic check
end Agent;

procedure P(Ptr: access T) is
begin
   Agent(Ptr);                     -- may be OK
end P;

procedure P1(Ptr: access T) is
   Local_Ptr: access T;
begin
   Local_Ptr := Ptr;               -- implicit conversion
   Agent(Local_Ptr);               -- never OK
end P1;
```

Now we find that P works much as before. The accessibility level passed into P is passed to Agent which then carries out the assignment to Dodgy. If the parameter passed to P is the local X then Program_Error is raised in Agent and propagated to P. If the parameter passed is Global then all is well.

The procedure P1 now compiles whereas it did not before. However, because the accessibility of the original parameter is lost by the assignment to Local_Ptr, it is the accessibility level of Local_Ptr that is passed to Agent and this means that the assignment to Dodgy always fails and raises Program_Error irrespective of whether P1 was called with X or Global.

If we just want to use another name for some reason then we can avoid the loss of the accessibility level by using renaming. Thus we could have

```
procedure P2(Ptr: access T) is
   Local_Ptr: access T renames Ptr;
begin
   ...
   Dodgy := Ref_T(Local_Ptr);      -- dynamic check
end P2;
```

and this will behave exactly as the original procedure P.

As usual a renaming just provides another view of the same entity and thus preserves the accessibility information.

A renaming can also include **not null** thus

```
Local_Ptr: not null access T renames Ptr;
```

Remember that not null must never lie so this is only legal if Ptr is indeed of a null excluding type (which it will be if Ptr is a controlling access parameter of the procedure P2).

A renaming might be useful when the accessed type T has components that we wish to refer to many times in the procedure. For example the accessed type might be the type Cell declared earlier in which case we might usefully have

```
Next: access Cell renames Ptr.Next;
```

and this will preserve the accessibility information.

Anonymous access types can also be used as the result of a function. In the Introduction we had

```
function Mate_Of(A: access Animal'Class)
                                 return access Animal'Class;
```

The accessibility level of the result in this case is the same as that of the declaration of the function itself.

We can also dispatch on the result of a function if the result is an access to a tagged type. Consider

```
function Unit return access T;
```

We can suppose that T is a tagged type representing some category of objects such as our geometrical objects and that Unit is a function returning a unit object such as a circle of unit radius or a triangle with unit side.

We might also have a function

```
function Is_Bigger(X, Y: access T) return Boolean;
```

and then

```
Thing: access T'Class := ... ;
...
Test: Boolean := Is_Bigger(Thing, Unit);
```

This will dispatch to the function Unit according to the tag of Thing and then of course dispatch to the appropriate function Is_Bigger.

The function Unit could also be used as a default value for a parameter thus

```
function Is_Bigger(X: access T;
                   Y: access T := Unit) return Boolean;
```

Remember that a default used in such a construction has to be tag indeterminate.

Permitting anonymous access types as result types eliminates the need to define the concept of a "return by reference" type. This was a strange concept in Ada 95 and primarily concerned limited types (including task and protected types) which of course could not be copied. Enabling us to write **access** explicitly and thereby tell the truth removes much confusion. Limited types will be discussed in detail in a later paper.

Access return types can be a convenient way of getting a constant view of an object such as a table. We might have an array in a package body (or private part) and a function in the specification thus

```
package P is
  type Vector is array (Integer range <>) of Float;

  function Read_Vec return access constant Vector;
  ...
private

end;
```

```
package body P is

  The_Vector: aliased Vector :=   ;

  function Read_Vec return access constant Vector is
  begin
    return The_Vector'Access;
  end;
  ...
end P;
```

We can now write

```
X := Read_Vec(7);               -- read element of array
```

This is strictly short for

```
X := Read_Vec.all(7);
```

Note that we cannot write

```
Read_Vec(7) := Y;               -- illegal
```

although we could do so if we removed **constant** from the return type (in which case we should use a different name for the function).

The last new use of anonymous access types concerns discriminants. Remember that a discriminant can be of a named access type or an anonymous access type. Discriminants of an anonymous access type are known as access discriminants. In Ada 95, access discriminants are only allowed with limited types. Discriminants of a named access type are just additional components with no special properties. But access discriminants of limited types are special. Since the type is limited, the object cannot be changed by a whole record assignment and so the discriminant cannot be changed even if it has defaults. Thus

```
type Minor is ...

type Major(M: access Minor) is limited
  record
    ...
  end record;

Small: aliased Minor;
Large: Major(Small'Access);
```

The objects Small and Large are now bound permanently together.

In Ada 2005, access discriminants are also allowed for nonlimited types. However, defaults are not permitted so that the discriminant cannot be changed so again the objects are bound permanently together. An interesting case arises when the discriminant is provided by an allocator thus

```
Larger: Major(new Minor( ... ));
```

In this case we say that the allocated object is a coextension of Larger. Coextensions have the same lifetime as the major object and so are finalized when it is finalized. There are various accessibility and other rules concerning objects which have coextensions which prevent difficulty when returning such objects from functions.

4 Downward closures

This section is really about access to subprogram types in general but the title downward closures has come to epitomize the topic.

The requirements for Ada 83, (Strawman .. Steelman) were strangely silent about whether parameters of subprograms could themselves be subprograms as was the case in Algol 60 and Pascal. Remember that Pascal was one of the languages on which the designs for the DoD language were to be based.

The predictability aspects of the requirements were interpreted as implying that all subprogram calls should be identified at compilation time on the grounds that if you didn't know what was being called than you couldn't know what the program was going to do. This was a particularly stupid attitude to take. The question of predictability (presumably in some safety or security context) really concerns the behaviour of particular programs rather than the universe of all programs that can be constructed in a language.

In any event the totality of subprograms that might be called in a program is finite and closed. It simply consists of the subprograms in the program. Languages such as Ada are not able to construct totally new subprograms out of lesser components in the way that they can create say floating point values.

So the world had to use generics for many applications that were natural for subprograms as parameters of other subprograms. Thankfully many implementers avoided the explosion that might occur with generics by clever code sharing which in a sense hid the parameterization behind the scenes.

The types of applications for which subprograms are natural as parameters are any where one subroutine is parameterized by another. They include many mathematical applications such as integration and maximization and more logical applications such as sorting and searching and iterating.

As outlined in the Introduction, the matter was partly improved in Ada 95 by the introduction of named access-to-subprogram types. This was essentially done to allow program call back to be implemented.

Program call back is when one program passes the "address" of a subprogram within it to another program so that this other program can later respond by calling back to the first program using the subprogram address supplied. This is often used for communication between an Ada application program and some other software such as an operating system which might even be written in another language such as C.

Named access to subprogram types certainly work for call back (especially with languages such as C that do not have nested subprograms) but the accessibility rules which followed those for general access to object types were restrictive. For example, suppose we have a general library level function for integration using a named access to subprogram type to pass the function to be integrated thus

```
type Integrand is access function(X: Float) return Float;

function Integrate(Fn: Integrand; Lo, Hi: Float)
                                        return Float;
```

then we cannot even do the simplest integration of our own function in a natural way. For example, suppose we wish to integrate a function such as Exp(X**2). We can try

```
with Integrate;
procedure Main is
  function F(X: Float) return Float is
  begin
    return Exp(X**2);
  end F;

  Result, L, H: Float;
begin
  ...              -- set bounds in L and H say
  Result := Integrate(F'Access, L, H);     -- illegal in 95
  ...
end Main
```

But this is illegal because of the accessibility check necessary to prevent us from writing something like

```
Evil: Integrand;
X: Float;
...
declare
  Y: Float;
  function F(X: Float) return Float is
    ...
    Y := X;                 --assign to variable in local block
    ...
  end F;
begin
  Evil := F'Access:         -- illegal
end;
  X := Evil(X);             -- call function out of context
```

Here we have attempted to assign an access to the local function F in the global variable Evil. If this assignment had been permitted then the call of Evil would indirectly have called the function F when the context in which F was declared no longer existed; F would then have attempted to assign to the variable Y which no longer existed and whose storage space might now be used for something else. We can summarise this perhaps by saying that we are attempting to call F when it no longer exists.

Ada 2005 overcomes the problem by introducing anonymous access to subprogram types. This was actually considered during the design of Ada 95 but it was not done at the time for two main reasons. Firstly, the implementation problems for those who were using displays rather than static links were considered a hurdle. And secondly, a crafty technique was available using the newly introduced tagged types. And of course one could continue to use generics. But further thought showed that the implementation burden was not so great after all and nobody understood the tagged type technique which was really incredibly contorted. Moreover, the continued use of generics when other languages forty years ago had included a more natural mechanism was tiresome. So at long last Ada 2005 includes anonymous access to subprogram types.

We rewrite the integration function much as follows

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float is
  Total: Float;
  N: constant Integer := ... ;          -- no of subdivisions
  Step: Float := (Hi - Lo) / Float(N);
  X: Float := Lo;                       -- current point
begin
  Total := 0.5 * Fn(Lo);                -- value at low bound
  for I in 1 .. N-1 loop
    X := X + Step;                      -- add values at
    Total := Total + Fn(X);             -- intermediate points
  end loop;
  Total := Total + 0.5 * Fn(Hi);        -- add final value
  return Total * Step;                  -- normalize
end Integrate;
```

The important thing to notice is the profile of Integrate in which the parameter Fn is of an anonymous access to subprogram type. We have also shown a simple body which uses the trapezium/trapezoid method and so calls the actual function corresponding to Fn at the two end points of the range and at a number of equally spaced intermediate points.

(NB It is time for a linguistic interlude. Roughly speaking English English trapezium equals US English trapezoid. They both originate from the Greek τραπεζα meaning a table (literally with four feet). Both originally meant a quadrilateral with no pairs of sides parallel. In the late 17th century, trapezium came to mean having one pair of sides parallel. In the 18th century trapezoid came to mean the same as trapezium but promptly faded out of use in England whereas in the US it continues in use. Meanwhile in the US, trapezium reverted to its original meaning of totally irregular. Trapezoid is rarely used in the UK but if used has reverted to its original meaning of totally irregular. A standard language would be useful. Anyway, the integration is using quadrilateral strips with one pair of sides parallel.)

With this new declaration of Integrate, the accessibility problems are overcome and we are allowed to write Integrate(F'Access, ...) just as we could write P(X'Access) in the example in the previous section where we discussed anonymous access to object types.

We still have to consider how a type conversion which would permit an assignment to a global variable is prevented. The following text illustrates both access to object and access to subprogram parameters.

```
type AOT is access all Integer;
type APT is access procedure (X: in out Float);

Evil_Obj: AOT;
Evil_Proc: APT;

procedure P(Objptr: access Integer;
            Procptr: access procedure (X: in out Float)) is
begin
  Evil_Obj := AOT(Objptr);        -- fails at run time
  Evil_Proc := APT(Procptr);      -- fails at compile time
end P;

declare
  An_Obj: aliased Integer;
  procedure A_Proc(X: in out Float) is
  begin ... end A_Proc;
begin
  P(An_Obj'Access, A_Proc'Access);       -- legal
end;

Evil_Obj.all := 0;          -- assign to nowhere
Evil_Proc.all( ... );       -- call nowhere
```

This repeats some of the structure of the previous section. The procedure P has an access to object parameter Objptr and an access to subprogram parameter Procptr; they are both of anonymous type. The call of P in the local block passes the addresses of a local object An_Obj and a local procedure A_Proc to P. This is permitted. We now attempt to assign the parameter values from within P to global objects Evil_Obj and Evil_Proc with the intent of assigning indirectly via Evil_Obj and calling indirectly via Evil_Proc after the object and procedure referred to no longer exist.

Both of these wicked deeds are prevented by the accessibility rules.

In the case of the object parameter Objptr it knows the accessibility level of the actual An_Obj and this is seen to be greater than that of the type AOT and so the conversion is prevented at run time and in fact Program_Error is raised. But if An_Obj had been declared at the same level as AOT and not within an inner block then the conversion would have been permitted.

However, somewhat different rules apply to anonymous access to subprogram parameters. They do not carry an indication of the accessibility level of the actual parameter but simply treat it as if it were infinite (strictly – deeper than anything else). This of course prevents the conversion to the type APT and all is well; this is detected at compile time. But note that if the procedure A_Proc had been declared at the same level as APT then the conversion would still have failed because the accessibility level is treated as infinite.

There are a number of reasons for the different treatment of anonymous access to subprogram types. A big problem is that named access to subprogram types are implemented in the same way as C *func in almost all compilers. Permitting the conversion from anonymous access to subprogram types to named ones would thus have caused problems because that model does not work especially for display based implementations. Carrying the accessibility level around would not have prevented these conversions. The key goal was simply to provide a facility corresponding to that in Pascal and not to encourage too much fooling about with access to subprogram types. Recall that the attribute Unchecked_Access is permitted for access to object types but was considered far too dangerous for access to subprogram types for similar reasons.

The reader may be feeling both tired and that there are other ways around the problems of accessibility anyway. Thus the double integration presented in the Introduction can easily be circumvented in many cases. We computed

$$\int_0^1 \int_0^1 xy \, dy \, dx$$

using the following program

```
with Integrate;
procedure Main is
  function G(X: Float) return Float is
    function F(Y: Float) return Float is
    begin
      return X*Y;
    end F;
  begin
    return Integrate(F'Access, 0.0, 1.0);
  end G;

  Result: Float;
begin
  Result:= Integrate(G'Access, 0.0, 1.0);
  ...
end Main;
```

The essence of the problem was that F had to be declared inside G because it needed access to the parameter X of G. But the astute reader will note that this example is not very convincing because the integrals can be separated and the functions both declared at library level thus

```
function F(Y: Float) return Float is
begin
  return Y;
end F;

function G(X: Float) return Float is
begin
  return X;
end G;

Result:= Integrate(F'Access, 0.0, 1.0) *
                          Integrate(G'Access, 0.0, 1.0);
```

and so it all works using the Ada 95 version of Integrate anyway.

However, if the two integrals had been more convoluted or perhaps the region had not been square but triangular so that the bound of the inner integral depended on the outer variable as in

$$\int_0^1 \int_0^x xy\,dy\,dx$$

then nested functions would be vital.

We will now consider a more elegant example which illustrates how we might integrate an arbitrary function of two variables $F(x, y)$ over a rectangular region.

Assume that we have the function Integrate for one dimension as before

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

Now consider

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float
                   LoY, HiY: Float) return Float is
  function FnX(X: Float) return Float is
    function FnY(Y: Float) return Float is
    begin
      return Fn(X, Y);
    end FnY;
  begin
    return Integrate(FnY'Access, LoY, HiY);
  end FnX;
begin
  Integrate(FnX'Access, LoX, HiX);
end integrate;
```

The new function Integrate for two dimensions overloads and uses the function Integrate for one dimension (a good example of overloading). With this generality it is again impossible to arrange the structure in a manner which is legal in Ada 95.

We might use the two-dimensional integration routine to solve the original trivial problem as follows

```
function F(X, Y: Float) return Float is
begin
  return X*Y;
end F;
...
Result := Integrate(F'Access, 0.0, 1.0, 0.0, 1.0);
```

As an exercise the reader might like to rewrite the two dimensional function to work on a non-rectangular domain. The trick is to pass the bounds of the inner integral also as functions. The profile then becomes

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float
              LoY, HiY: access function(X: Float) return Float)
                                                  return Float;
```

In case the reader should think that this topic is all too mathematical it should be pointed out that anonymous access to subprogram parameters are widely used in the new container library thereby saving the unnecessary use of generics.

For example the package Ada.Containers.Vectors declares procedures such as

```
procedure Update_Element
  (Container: in Vector; Index: in Index_Type;
   Process: not null access
      procedure (Element: in out Element_Type));
```

This updates the element of the vector Container whose index is Index by calling the procedure Process with that element as parameter. Thus if we have a vector of integers V and we need to double the value of those with index in the range 5 to 10, then we would first declare a procedure such as

```
procedure Double(E: in out Integer) is
begin
  E := 2 * E;
end Double ;
```

and then write

```
for I in 5 .. 10 loop
  Update_Element(V, I, Double'Access);
end loop;
```

Further details of the use of access to subprogram types with containers will be found in a later paper.

Finally it should be noted that anonymous access to subprogram types can also be used in all those places where anonymous access to object types are allowed. That is as stand-alone objects, as components of arrays and records, as function results, in renamings, and in access discriminants.

The reader who likes long sequences of reserved words should realise by now that there is no limit in Ada 2005. This is because a function without parameters can return an access to function as its result and this in turn could be of a similar kind. So we would have

```
type FF is access function return access function
                return access function ...
```

Attempts to compile such an access to function type will inevitably lead to madness.

5 Access types and discriminants

This final topic concerns two matters. The first is about accessing components of discriminated types that might vanish or change mysteriously and the second is about type conversions.

Recall that we can have a mutable variant record such as

```
type Gender is (Male, Female, Neuter);

type Mutant(Sex: Gender := Neuter) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
      when Neuter =>
        null;
    end case;
  end record;
```

This represents a world in which there are three sexes, males which can have beards, females which can bear children, and neuters which are fairly useless. Note the default value for the discriminant. This means that if we declare an unconstrained object thus

```
The_Thing: Mutant;
```

then The_Thing is neuter by default but could have its sex changed by a whole record assignment thus

```
The_Thing := (Male, The_Thing.Birth, True);
```

It now is Male and has a beard but the date of birth retains its previous value.

The problem with this sort of object is that components can disappear. If it were changed to be Female then the beard would vanish and be replaced by children. Because of this ghostly behaviour certain operations on mutable objects are forbidden.

One obvious rule is that it is not permissible to rename components which might vanish. So

```
Hairy: Boolean renames The_Thing.Bearded;    -- illegal
```

is not permitted. This was an Ada 83 rule. It was probably the case that the rules were watertight in Ada 83. However, Ada 95 introduced many more possibilities. Objects and components could be marked as **aliased** and the Access attribute could be applied. Additional rules were then added to prevent creating references to things that could vanish.

However, it was then discovered that the rules in Ada 95 regarding access types were not watertight. Accordingly various attempts were made to fix them in a somewhat piecemeal fashion. The problems are subtle and do not seem worth describing in their entirety in this general presentation. We will content ourselves with just a couple of examples.

In Ada 95 we can declare types such as

```
type Mutant_Name is access all Mutant;
type Things_Name is access all Mutant(Neuter);
```

Naturally enough an object of type Things_Name can only be permitted to reference a Mutant whose Sex is Neuter.

```
Some_Thing: aliased Mutant;
Thing_Ptr: Things_Name := Some_Thing'Access;
```

Things would now go wrong if we allowed Some_Thing to have a sex change. Accordingly there is a rule in Ada 95 that says that an aliased object such as Some_Thing is considered to be constrained. So that is quite safe.

However, matters get more difficult when a type such as Mutant is used for a component of another type such as

```
type Monster is
  record
    Head: Mutant(Female);
    Tail: aliased Mutant;
  end record;
```

Here we are attempting to declare a nightmare monster whose head is a female but whose tail is deceivingly mutable. Those with a decent education might find that this reminds them of the Sirens who tempted Odysseus by their beautiful voices on his trip past the monster Scylla and the whirlpool Charybdis. Those with an indecent education can compare it to a pantomime theatre horse (or mare, maybe indeed a nightmare). We could then write

```
M: Monster;
Thing_Ptr := Monster.Tail'Access;
```

However, there is an Ada 95 rule that says that the Tail has to be constrained since it is aliased so the type Monster is not allowed. So far so good.

But now consider the following very nasty example

```
generic
  type T is private;
  Before, After: T;
  type Name is access all T;
  A_Name: in out Name;
procedure Sex_Change;

procedure Sex_Change is
  type Single is array (1..1) of aliased T;
  X: Single := (1 => Before);
begin
  A_Name := X(1)'Access;
  X := (1 => After);
end Sex_Change;
```

and then

```
A_Neuter: Mutant_Name(Neuter);         -- fixed neuter

procedure Surgery is new Sex_Change(
        T => Mutant,
        Before => (Sex => Neuter),
        After => (Sex => Male, Bearded, True),
        Name => Mutant_Name,
        A_Name => A_Neuter);

Surgery;         -- call of Surgery makes A_Neuter hairy
```

The problem here is that there are loopholes in the checks in the procedure Sex_Change. The object A_Name is assigned an access to the single component of the array X whose value is Before. When this is done there is a check that the component of the array has the correct subtype. However the subsequent assignment to the whole array changes the value of the component to After and this can change the subtype of X(1) surreptitiously and there is no check concerning A_Name. The key point is that the generic doesn't know that the type T is mutable; this information is not part of the generic contract.

So when we call Surgery, the object A_Neuter suddenly finds that it has grown a beard!

A similar difficulty occurs when private types are involved because the partial view and full view might disagree about whether the type is constrained or not. Consider

```
package Beings is
  type Mutant is private;
  type Mutant_Name is access Mutant;
  F, M: constant Mutant;
private
  type Mutant(Sex: Gender := Neuter) is
    record
      ...                  -- as above
    end record;

  F: constant Mutant := (Female, ... );
  M: constant Mutant := (Male, ... );
end Beings;
```

Now suppose some innocent user (who has not peeked at the private part) writes

```
Chris: Mutant_Name := new Mutant'(F);    --OK
...
Chris.all := M;            -- raises Constraint_Error
```

This is very surprising. The user cannot see that the type Mutant is mutable and in particular cannot see that M and F are different in some way. From the outside they just look like constants of the same type. The big trouble is that there is a rule in Ada 95 that says that an object created by an allocator is constrained. So the new object referred to by Chris is permanently Female and therefore the attempt to assign the value of M with its Bearded component to her is doomed.

Attempting to fix these and related problems with a number of minimal rules seemed fated not to succeed. In the end the approach has been taken of getting to the root of the matter in Ada 2005 and disallowing access subtypes for general access types that have defaults for their discriminants. So both the explicit Things_Name and also Mutant_Name(Neuter) are forbidden in Ada 2005.

Moreover we cannot even have an access type such as Mutant_Name when the access type completes a private view that has no discriminants.

By removing these nasty access subtypes it is now possible to say that heap objects are no longer considered constrained in this situation.

The other change in this area concerns type conversions. A variation on the gender theme is illustrated by the following

```
type Gender is (Male, Female);

type Person(Sex: Gender) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
    end case;
  end record;
```

Note that this type is not mutable so all persons are stuck with their sex from birth.

We might now declare some access types

```
type Person_Name is access all Person;
type Mans_Name is access all Person(Male);
type Womans_Name is access all Person(Female);
```

so that we can manipulate various names of people. We would naturally use Person_Name if we did not know the sex of the person and otherwise use Mans_Name or Womans_Name as appropriate. We might have

```
It: Person_Name := Chris'Access;
Him: Mans_Name := Jack'Access;
Her: Womans_Name := Jill'Access;
```

If we later discover that Chris is actually Christine then we might like to assign the value in It to a more appropriate variable such as Her. So we would like to write

```
Her := Womans_Name(It);
```

But curiously enough this is not permitted in Ada 95 although the reverse conversion

```
It := Person_Name(Her);
```

is permitted. The Ada 95 rule is that any constraints have to statically match or the conversion has to be to an unconstrained type. Presumably the reason was to avoid checks at run time. But this lack of symmetry is unpleasant and the rule has been changed in Ada 2005 to allow conversion in both directions with a run time check as necessary.

The above example is actually Exercise 19.8(1) in the textbook [2]. The poor student was invited to solve an impossible problem. But they will be successful in Ada 2005.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment.*

[2] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 3 Structure and visibility

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in the areas of structure and visibility for Ada 2005.

The most important improvement is perhaps the introduction of limited with clauses which permit types in two packages to refer to each other. A related addition to context clauses is the private with clause which just provides access from a private part.

There are also important improvements to limited types which make them much more useful; these include initialization with aggregates and composition using a new form of return statement.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies the solution of the problem of mutually dependent types as one of the two specific issues that need to be addressed in devising Ada 2005.

Moreover the guidance document also emphasizes

> Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, including adding a Java-like interface feature and improved interfacing to other OO languages.

OO is largely about structure and visibility and so further improvements and in particular those that remedy shortcomings are desirable.

The following Ada issues cover the relevant changes and are described in detail in this paper:

217 Mutually recursive types – limited with

262 Access to private units in the private part

287 Limited aggregates allowed

318 Limited and anonymous access return types

326 Tagged incomplete types

412 Subtypes and renamings of incomplete entities

These changes can be grouped as follows.

First there is the important solution to the problem of mutually dependent types across packages provided by the introduction of limited with clauses (217). Related changes are the introduction of tagged incomplete types (326) and the ability to have subtypes and renamings of incomplete views (412).

Another improvement to the visibility rules is the introduction of private with clauses (262).

There are some changes to aggregates. These were triggered by problems with limited types but apply to aggregates in general (part of 287).

An important area is that of limited types which are somewhat confused in Ada 95. There are two changes which permit limited values to be built *in situ*. One is the use of aggregates for initialization and the other is a more elaborate return statement which enables the construction of limited values when returning from a function (287, 318).

2 Mutually dependent types

For many programmers the solution of the problem of mutually dependent types will be the single most important improvement introduced in Ada 2005.

This topic was discussed in the Introduction using an example of two mutually dependent types, Point and Line. Each type needed to refer to the other in its declaration and of course the solution to this problem is to use incomplete types. In Ada 95 there are three stages. We first declare the incomplete types

```
type Point;                -- incomplete types
type Line;
```

Suppose for simplicity that we wish to study patterns of points and lines such that each point has exactly three lines through it and that each line has exactly three points on it. (This is not so stupid. The two most fundamental theorems of projective geometry, those of Pappus and Desargues, concern such structures and so does the simplest of finite geometries, the Fano plane.)

Using the incomplete types we can then declare

```
type Point_Ptr is access Point;  -- use incomplete types
type Line_Ptr is access Line;
```

and finally we can complete the type declarations thus

```
type Point is                    -- complete the types
  record
    L, M, N: Line_Ptr;
  end record;

type Line is
  record
    P, Q, R: Point_Ptr;
  end record;
```

Of course, in Ada 2005, as discussed in the previous paper, we can use anonymous access types more freely so that the second stage can be omitted in this example. As a consequence the complete declarations are simply

```
type Point is                        -- complete the types
  record
    L, M, N: access Line;
  end record;

type Line is
  record
    P, Q, R: access Point;
  end record;
```

This has the important advantage that we do not have to invent irritating identifiers such as Point_Ptr.

But we will stick to Ada 95 for the moment. In Ada 95 there are two rules

- the incomplete type can only be used in the definition of access types;
- the complete type declaration must be in the same declarative region as the incomplete type.

The first rule does actually permit

```
type T;
type A is access procedure (X: in out T);
```

Note that we are here using the incomplete type T for a parameter. This is not normally allowed, but in this case the procedure itself is being used in an access type. The additional level of indirection means that the fact that the parameter mechanism for T is not known yet does not matter.

Apart from this, it is not possible to use an incomplete type for a parameter in a subprogram in Ada 95 except in the case of an access parameter. Thus we cannot have

```
function Is_Point_On_Line(P: Point; L: Line)
                                     return Boolean;
```

before the complete type declarations.

It is also worth pointing out that the problem of mutually dependent types (within a single unit) can often be solved by using private types thus

```
  type Point is private;
  type Point_Ptr is access Point;
  type Line is private;
  type Line_Ptr is access Line;
private

  type Point is
    record
      L, M, N: Line_Ptr;
    end record;

  type Line is
    record
      P, Q, R: Point_Ptr;
    end record;
```

But we need to use incomplete types if we want the user to see the full view of a type so the situation is somewhat different.

As an aside, remember that if an incomplete type is declared in a private part then the complete type can be deferred to the body (this is the so-called Taft Amendment in Ada 83). In this case neither the user nor indeed the compiler can see the complete type and this is the main reason why we cannot have parameters of incomplete types whereas we can for private types.

We will now introduce what has become a canonical example for discussing this topic. This concerns employees and the departments of the organization in which they work. The information about employees needs to refer to the departments and the departments need to refer to the employees. We assume that the material regarding employees and departments is quite large so that we naturally wish to declare the two types in distinct packages Employees and Departments. So we would like to say

```
with Departments;  use Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee;
                            D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee)
                                   return Dept_Ptr;
  ...
end Employees;

with Employees;  use Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department;
                           M: in out Employee);
  ...
end Departments;
```

We cannot write this because each package has a with clause for the other and they cannot both be declared (or entered into the library) first.

We assume of course that the type Employee includes information about the Department for whom the Employee works and the type Department contains information regarding the manager of the department and presumably a list of the other employees as well – note that the manager is naturally also an Employee.

So in Ada 95 we are forced to put everything into one package thus

```
package Workplace is
  type Employee is private;
  type Department is private;
  procedure Assign_Employee(E: in out Employee;
                            D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee)
                                   return Dept_Ptr;
  procedure Choose_Manager(D: in out Department;
                                        M: in out Employee);
private
  ...
end Workplace;
```

Not only does this give rise to huge cumbersome packages but it also prevents us from using the proper abstractions. Thus the types Employee and Department have to be declared in the same private part and so are not protected from each others operations.

Ada 2005 solves this by introducing a variation of the with clause – the limited with clause. A limited with clause enables a library unit to have an incomplete view of all the visible types in another package. We can now write

```
limited with Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee;
                    D: access Departments.Department);
  type Dept_Ptr is access all Departments.Department;
  function Current_Department(E: Employee)
                                        return Dept_Ptr;
  ...
end Employees;

limited with Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department;
                    M: access Employees.Employee);
  ...
end Departments;
```

It is important to understand that a limited with clause does not impose a dependence. Thus if a package A has a limited with clause for B, then A does not depend on B as it would with a normal with clause, and so B does not have to be compiled before A or placed into the library before A.

If we have a cycle of packages we only have to put **limited with** on one package since that is sufficient to break the cycle of dependences. However, for symmetry, in this example we have made them both have a limited view of each other.

Note the terminology: we say that we have a limited view of a package if the view is provided through a limited with clause. So a limited view of a package provides an incomplete view of its visible types. And by an incomplete view we mean as if they were incomplete types.

In the example, because an incomplete view of a type cannot generally be used as a parameter, we have had to change one parameter of each of Assign_Employee and Choose_Manager to be an access parameter.

There are a number of rules necessary to avoid problems. A natural one is that we cannot have both a limited with clause and a normal with clause for the same package in the same context clause (a normal with clause is now officially referred to as a nonlimited with clause). An important and perhaps unexpected rule is that we cannot have a use package clause with a limited view because severe surprises might happen.

To understand how this could be possible it is important to realise that a limited with clause provides a very restricted view of a package. It just makes visible

- the name of the package and packages nested within,
- an incomplete view of the types declared in the visible parts of the packages.

Nothing else is visible at all. Now consider

```
package A is
  X: Integer := 99;
end A;

package B is
  X: Integer := 111;
end B;

limited with A, B;
package P is
  ...                   -- neither X visible here
end P;
```

Within package P we cannot access A.X or B.X because they are not types but objects. But we could declare a child package with its own with clause thus

```
with A;
package P.C is
  Y: Integer := A.X;
end P.C;
```

The nonlimited with clause on the child "overrides" the limited with clause on the parent so that A.X is visible.

Now suppose we were allowed to add a use package clause to the parent package; since a use clause on a parent applies to a child this means that we could refer to A.X as just X within the child so we would have

```
limited with A, B;
use A, B;               -- illegal
package P is
  ...                   -- neither X visible here
end P;

with A;
package P.C is
  Y: Integer := X;      -- A.X now visible as just X
end P.C;
```

If we were now to change the with clause on the child to refer to B instead of A, then X would refer to B.X rather than A.X. This would not be at all obvious because the use clause that permits this is on the parent and we are not changing the context clause of the parent at all. This would clearly be unacceptable and so use package clauses are forbidden if we only have a limited view of the package.

Here is a reasonably complete list of the rules designed to prevent misadventure when using limited with clauses

- a use package clause cannot refer to a package with a limited view as illustrated above,

```
limited with P; use P;          -- illegal
package Q is ...
```

the rule also prevents

```
limited with P;
package Q is
  use P;                        -- illegal
```

- a limited with clause can only appear on a specification – it cannot appear on a body or a subunit,

```
limited with P;                 -- illegal
package body Q is ...
```

- a limited with clause and a nonlimited with clause for the same package may not appear in the same context clause,

```
limited with P; with P;         -- illegal
```

- a limited with clause and a use clause for the same package or one of its children may not appear in the same context clause,

```
limited with P; use P.C;        -- illegal
```

- a limited with clause may not appear in the context clause applying to itself,

```
limited with P;                 -- illegal
package P is ...
```

- a limited with clause may not appear on a child unit if a nonlimited with clause for the same package applies to its parent or grandparent etc,

```
with Q;
package P is ...

limited with Q;                 -- illegal
package P.C is ...
```

but note that the reverse is allowed as mentioned above

```
limited with Q;
package P is ...

with Q;                         -- OK
package P.C is ...
```

- a limited with clause may not appear in the scope of a use clause which names the unit or one of its children,

```
with A;
package P is
  package R renames A;
end P;

with P;
package Q is
  use P.R;                      -- applies to A
end Q;

limited with A;                 -- illegal
package Q.C is ...
```

without this specific rule, the use clause in Q which actually refers to A would clash with the limited with clause for A.

Finally note that a limited with clause can only refer to a package declaration and not to a subprogram, generic declaration or instantiation, or to a package renaming.

We will now return to the rules for incomplete types. As mentioned above the rules for incomplete types are quite strict in Ada 95 and apart from the curious case of an access to subprogram type it is not possible to use an incomplete type for a parameter other than in an access parameter.

Ada 2005 enables some relaxation of these rules by introducing tagged incomplete types. We can write

```
type T is tagged;
```

and then the complete type must be a tagged type. Of course the reverse does not hold. If we have just

```
type T;
```

then the complete type T might be tagged or not.

A curious feature of Ada 95 was mentioned in the Introduction. In Ada 95 we can write

```
type T;
...
type T_Ptr is access all T'Class;
```

By using the attribute Class, this promises in a rather sly way that the complete type T will be tagged. This is strictly obsolescent in Ada 2005 and moved to Annex J. In Ada 2005 we should write

```
type T is tagged;
...
type T_Ptr is access all T'Class;
```

The big advantage of introducing tagged incomplete types is that we know that tagged types are always passed by reference and so we are allowed to use tagged incomplete types for parameters.

This advantage extends to the incomplete view obtained from a limited with clause. If a type in a package is visibly tagged then the incomplete view obtained is tagged incomplete and so the type can then be used for parameters.

Returning to the packages Employees and Departments it probably makes sense to make both types tagged since it is likely that the types Employee and Department form a hierarchy. So we can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee;
              D: in out Departments.Department'Class);
  type Dept_Ptr is
              access all Departments.Department'Class;
  function Current_Department(E: Employee)
                                         return Dept_Ptr;
  ...
end Employees;
```

```
limited with Employees;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager(D: in out Department;
                  M: in out Employees.Employee'Class);
   ...
end Departments;
```

The text is a bit cumbersome now with Class sprinkled liberally around but we can introduce some subtypes in order to shorten the names. We can also avoid the introduction of the type Dept_Ptr since we can use an anonymous access type for the function result as mentioned in the previous paper. So we get

```
limited with Departments;
package Employees is
   type Employee is tagged private;
   subtype Dept is Departments.Department;
   procedure Assign_Employee(E: in out Employee;
                                 D: in out Dept'Class);
   function Current_Department(E: Employee)
                            return access Dept'Class;
   ...
end Employees;

limited with Employees;
package Departments is
   type Department is tagged private;
   subtype Empl is Employees.Employee;
   procedure Choose_Manager(D: in out Department;
                                 M: in out Empl'Class);
   ...
end Departments;
```

Observe that in Ada 2005 we can use a simple subtype as an abbreviation for an incomplete type thus

```
subtype Dept is Departments. Department;
```

but such a subtype cannot have a constraint or a null-exclusion. In essence it is just a renaming. Remember that we cannot have a use clause with a limited view. Moreover, many projects forbid use clauses anyway but permit renamings and subtypes for local abbreviations. It would be a pain if such abbreviations were not also available when using a limited with clause.

It's a pity we cannot also write

```
subtype A_Dept is Departments.Department'Class;
```

but then you cannot have everything in life.

A similar situation arises with the names of nested packages. They can be renamed in order to provide an abbreviation.

The mechanism for breaking cycles of dependences by introducing limited with clauses does not mean that the implementation does not check everything thoroughly in a rigorous Ada way. It is just that some checks might have to be deferred. The details depend upon the implementation.

For the human reader it is very helpful that use clauses are not allowed in conjunction with limited with clauses since it eliminates any doubt about the location of types involved. It probably helps the poor compilers as well.

Readers might be interested to know that this topic was one of the most difficult to solve satisfactorily in the design of Ada 2005. Altogether seven different versions of AI-217 were developed. This chosen solution is on reflection by far the best and was in fact number 6.

A number of loopholes in Ada 95 regarding incomplete types are also closed in Ada 2005.

One such loophole is illustrated by the following (this is Ada 95)

```
package P is
   ...
private
   type T;                          -- an incomplete type
   type ATC is access all T'Class;  -- it must be tagged
   X: ATC;
   procedure Op(X: access T);       -- primitive operation
   ...
end P;
```

The incomplete type T is declared in the private part of the package P. The access type ACT is then declared and since it is class wide this implies that the type T must be tagged (the reader will recall from the discussion above that this odd feature is banished to Annex J in Ada 2005). The full type T is then declared in the body. We also declare a primitive operation Op of the type T in the private part.

However, before the body of P is declared, nothing in Ada 95 prevents us from writing a private child thus

```
private package P.C is
   procedure Naughty;
end P.C;

package body P.C is
   procedure Naughty is
   begin
      Op(X);              -- a dispatching call
   end Naughty;
end P.C;
```

and the procedure Naughty can call the dispatching operation Op. The problem is that we are required to compile this call before the type T is completed and thus before the location of its tag is known.

This problem is prevented in Ada 2005 by a rule that if an incomplete type declared in a private part has primitive operations then the completion cannot be deferred to the body.

Similar problems arise with access to subprogram types. Thus, as mentioned above, Ada 95 permits

```
type T;
type A is access procedure (X: in out T);
```

In Ada 2005, the completion of T cannot be deferred to a body. Nor can we declare such an access to subprogram type if we only have an incomplete view of T arising from a limited with clause.

Another change in Ada 2005 can be illustrated by the Departments and Employees example. We can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee;
              D: in out Departments.Department'Class);
  type Dept_Ptr is
              access all Departments.Department'Class;
  ...
end Employees;

with Employees; use Employees;
procedure Recruit(D: Dept_Ptr; E: in out Employee) is
begin
  Assign_Employee(E, D.all);
end Recruit;
```

Ada 95 has a rule that says "thou shalt not dereference an incomplete type". This would prevent the call of Assign_Employee which is clearly harmless. It would be odd to require Recruit to have a nonlimited with clause for Departments to allow the call of Assign_Employee. Accordingly the rule is changed in Ada 2005 so that dereferencing an incomplete view is only forbidden when used as a prefix as, for example, in D'Size.

3 Visibility from private parts

Ada 95 introduced public and private child packages in order to enable subsystems to be decomposed in a structured manner. The general idea is that

- public children enable the decomposition of the view of a subsystem to the user of the subsystem,
- private children enable the decomposition of the implementation of a subsystem.

In turn both public and private children can themselves have children of both kinds. This has proved to work well in most cases but a difficulty has arisen regarding private parts.

Recall that the private part of a package really concerns the implementation of the package rather than specifying the facilities to the external user. Although it does not concern algorithmic aspects of the implementation it does concern the implementation of data abstraction. During the original design of Ada some thought was given to the idea that a package should truly be written and compiled as three distinct parts. Perhaps like this

```
with ...
package P is
  ...           -- visible specification
end;

with ...
package private P is          -- just dreaming
  ...           -- private part
end;

with ...
package body P is
  ...           -- body
end;
```

Each part could even have had its own context clause as shown.

However, it was clear that this would be an administrative nightmare in many situations and so the two-part specification and body emerged with the private part lurking at the end of the visible part of the specification (and sharing its context clause).

This was undoubtedly the right decision in general. The division into just two parts supports separate compilation well and although the private part is not part of the logical interface to the user it does provide information about the physical interface and that is needed by the compiler.

The problem that has emerged is that the private part of a public package cannot access the information in private child packages. Private children are of course not visible to the user but there is no reason why they should not be visible to the private part of a public package provided that somehow the information does not leak out. Thus consider a hierarchy

```
package App is
  ...
private
  ...
end App;

package App.Pub is
  ...
private
  ...
end App.Pub;

private package App.Priv is
  ...
private
  ...
end App.Priv;
```

There is no reason why the private parts of App and App.Pub and the visible part of the specification of App.Priv should not share visibility (the private part of App.Priv logically belongs to the next layer of secrecy downwards). But this sharing is not possible in Ada 95.

The public package App.Pub is not permitted to have a with clause for the child package App.Priv since this would mean that the visible part of App.Pub would also have visibility of this information and by mechanisms such as renaming could pass it on to the external user.

The specification of the parent package App is also not permitted to have a with clause for App.Priv since this would break the dependence rules anyway. Any child has a dependence on its parent and so the parent specification has to be compiled or entered into the program library first.

Note that the private part of the public child App.Pub does automatically have visibility of the private part of the parent App. But the reverse cannot be true again because of the dependence rules.

Finally note that the private child App.Priv can have a with clause for its public sibling App.Pub (it creates a dependence of course) but that only gives the private child visibility of the visible part of the public child.

So the only visibility sharing among the three regions in Ada 95 is that the private part of the public child and the visible part of the private child can see the private part of the parent.

The practical consequence of this is that in large systems, information which should really be lower down the hierarchy has to be placed in the private part of the ultimate parent. This tends to mean that the parent package becomes very large thereby making maintenance more difficult and forcing frequent recompilations of the parent and thus the whole hierarchy of packages.

The situation is much alleviated in Ada 2005 by the introduction of private with clauses.

If a package P has a private with clause for a package Q thus

```
private with Q;
package P is ...
```

then the private part of P has visibility of the visible part of the package Q, whereas the visible part of P does not have visibility of Q and so visibility cannot be transmitted to a user of P. It is rather as if the with clause were attached to just the private part of P thus

```
package P is
   ...
with Q;                    -- we cannot write this
private
   ...
end P;
```

This echoes the three-part decomposition of a package discussed above.

A private with clause can be placed wherever a normal with clause for the units mentioned can be placed and in addition a private with clause which mentions a private unit can be placed on any of its parent's descendants.

So we can put a private with clause for App.Priv on App.Pub thereby permitting visibility of the private child from the private part of its public sibling. Thus

```
private with App.Priv;
package App.Pub is
   ...                     -- App.Priv not visible here
private
   ...                     -- App.Priv visible here
end App.Pub;
```

This works provided we don't run afoul of the dependence rules. The private with clause means that the public child has a dependence on the private child and therefore the private child must be compiled or entered into the program library first.

We might get a situation where there exists a mutual dependence between the public and private sibling in that each has a type that the other wants to access. In such a case we can use a limited private with clause thus

```
limited private with App.Priv;
package App.Pub is
   ...                     -- App.Priv not visible here
private
   ...                     -- limited view of App.Priv here
end App.Pub;
```

The child packages are both dependent on the parent package and so the parent cannot have with clauses for them. But a parent can have a limited with clause for a public child and a limited private with clause for a private child thus

```
limited with App.Pub; limited private with App.Priv;
package App is
   ...                     -- limited view of App.Pub here
private
   ...                     -- limited view of App.Priv here
end App;
```

A simple example of the use of private with clauses was given in the Introduction. Here it is somewhat extended

```
limited with App.User_View;
limited private with App.Secret_Details;
package App is
   ...  -- limited view of type Outer visible here
private
   ...  -- limited view of type Inner visible here
end App;

private package App.Secret_Details is
   type Inner is ...
   ...  -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

   type Outer is private;
   ...  -- various operations on Outer visible to the user

        -- type Inner is not visible here
private
        -- type Inner is visible here

    type Outer is
      record
         X: Secret_Details.Inner;
         ...
      end record;
   ...
end App.User_View;
```

In the previous section we observed that there were problems with interactions between use clauses, nonlimited with clauses, and limited with clauses. Those rules also apply to private with clauses where a private with clause is treated as a nonlimited with clause and a limited private with clause is treated as a limited with clause. In other words private is ignored for the purpose of those rules.

Moreover, we cannot place a package use clause in the same context clause as a private with clause (limited or not). This is because we would then expect it to apply to the visible part as well which would be wrong. However, we can always put a use clause in the private part thus

```
private with Q;
package P is
  ...                      -- Q not visible here
private
  use Q;
  ...                      -- use visibility of Q here
end P;
```

At the risk of confusing the reader it might be worth pointing out that strictly speaking the rules regarding private with are treated as legality rules rather than visibility rules. Here is an example which illustrates this subtlety and the dangers it avoids

```
package P is
  function F return Integer;
end P;

function F return Integer;

with P;
private with F;
package Q is
  use P;
  X: Integer := F;         -- illegal
  Y: Integer := P.F;       -- legal
private
  Z: Integer := F;         -- legal, calls the library F
end Q;
```

If we treated the rules regarding private with as pure visibility rules then the call of F in the declaration of X in the visible part would be a call of P.F. So moving the declaration of X to the private part would silently change the F being called – this would be nasty. We can always write the call of F as P.F as shown in the declaration of Y.

So the rules regarding private with are written to make entities visible but unmentionable in the visible part. In practice programmers can just treat them as visibility rules so that the entities are not visible at all which is how we have described them above.

A useful consequence of the unmentionable rather than invisible approach is that we can use the name of a package mentioned in a private with clause in a pragma in the context clause thus

```
private with P; pragma Elaborate(P);
package Q is ...
```

Private with clauses are in fact allowed on bodies as well, in which case they just behave as a normal with clause. Another minor point is that Ada has always permitted several with clauses for the same unit in one context clause thus

```
with P; with P; with P, P;
package Q is ...
```

To avoid complexity we similarly allow

```
with P; private with P;
package Q is
```

and then the private with is ignored.

We have introduced private with clauses in this section as the solution to the problem of access to private children from the private part of the parent or public sibling. But they have other important uses. If we have

```
private with P;
package Q is ...
```

then we are assured that the package Q cannot inadvertently access P in the visible part and, in particular, pass on access to entities in P by renamings and so on. Thus writing **private with** provides additional documentation information which can be useful to both human reviewers and program analysis tools. So if we have a situation where a private with clause is all that is needed then we should use it rather than a normal with clause.

In summary, whereas in Ada 95 there is just one form of with clause, Ada 2005 provides four forms

```
with P;                  -- full view

limited with P;          -- limited view

private with P;          -- full view from private part

limited private with P;  -- limited view from private part
```

Finally, note that if a private with clause is given on a specification then it applies to the body as well as to the private part.

4 Aggregates

There are important changes to aggregates in Ada 2005 which are very useful in a number of contexts. These were triggered by the changes to the rules for limited types which are described in the next section, but it is convenient to first consider aggregates separately.

The main change is that the box notation <> is now permitted as the value in a named aggregate. The meaning is that the component of the aggregate takes the default value if there is one.

So if we have a record type such as

```
type RT is
  record
    A: Integer := 7;
    B: access Integer;
    C: Float;
  end record;
```

then if we write

```
X: RT := (A => <>, B => <>, C => <>);
```

then X.A has the value 7, X.B has the value **null** and X.C is undefined. So the default value is that given in the record type declaration or, in the absence of such an explicit default value, it is the default value for the type. If there is no explicit default value and the type does not have one either then the value is simply undefined as usual.

The above example could be abbreviated to

```
X: RT := (others => <>);
```

The obvious combinations are allowed

```
(A => <>, B => An_Integer'Access, C => 2.5)
(A => 3, others => <>)
(A => 3, B | C => <>)
```

The last two are the same. There is a rule in Ada 95 that if several record components in an aggregate are given the same expression using a | then they have to be of the same type. This does not apply in the case of <> because no typed expression is involved.

The <> notation is not permitted with positional notation. So we cannot write

```
(3, <>, 2.5)                -- illegal
```

But we can mix named and positional in a record aggregate as usual provided the named components follow the positional ones, so the following are permitted

```
(3, B => <>, C => 2.5)
(3, others => <>)
```

A minor but important rule is that we cannot use <> for a component of an aggregate that is a discriminant if it does not have a default. Otherwise we could end up with an undefined discriminant.

The <> notation is also allowed with array aggregates. But in this case the situation is much simpler because it is not possible to give a default value for array components. Thus we might have

```
P: array (1.. 1000) of Integer := (1 => 2, others => <>);
```

The array P has its first component set to 2 and the rest undefined. (Maybe P is going to be used to hold the first 1000 prime numbers and we have a simple algorithm to generate them which requires the first prime to be provided.) The aggregate could also be written as

```
(2, others => <>)
```

Remember that **others** is permitted with a positional array aggregate provided it is at the end. But otherwise <> is not allowed with a positional array aggregate.

We can add **others** => <> even when there are no components left. This applies to both arrays and records.

The box notation is also useful with tasks and protected objects used as components. Consider

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
    Finished: Boolean := False;
  end record;
```

As explained in the next section, we can now use an aggregate to initialize an object of a limited type. Although we cannot give an explicit initial value for a Semaphore we would still like to use an aggregate to get a coverage check. So we can write

```
X: PT := (Guard => <>, Count => 0, Finished => <>);
```

Note that although we can use <> to stand for the value of a component of a protected type in a record we cannot use it for a protected object standing alone.

```
Sema: Semaphore := <>;          -- illegal
```

The reason is that there is no need since we have no coverage check to concern us and there could be no other reason for doing it anyway.

Similarly we can use <> with a component of a private type as in

```
type Secret is private;

type Visible is
  record
    A: Integer;
    S: Secret;
  end record;

X: Visible := (A => 77; S => <>);
```

but not when standing alone

```
S: Secret := <>;                -- illegal
```

It would not have any purpose because such a variable will take any default value anyway.

We conclude by mentioning a small point for the language lawyer. Consider

```
function F return Integer;

type T is
  record
    A: Integer := F;
    B: Integer := 3;
  end record;
```

Writing

```
X: T := (A => 5, others => <>);   -- does not call F
```

is not quite the same as

```
X: T;                             -- calls F
...
X.A := 5;  X.B := 3;
```

In the first case the function F is not called whereas in the second case it is called when X is declared in order to default initialize X.A. If it had a nasty side effect then this could matter. But then programmers should not use nasty side effects anyway.

5 Limited types

The general idea of a limited type is to restrict the operations that a user can do on the type to just those provided by the author of the type and in particular to prevent the user from doing assignment and thus making copies of objects of the type.

However, limited types have always been a problem. In Ada 83 the concept of limitedness was confused with that of private types. Thus in Ada 83 we only had limited private types (although task types were inherently limited).

Ada 95 brought significant improvement by two changes. It allowed limitedness to be separated from privateness. It also allowed the redefinition of equality for all types whereas Ada 83 forbade this for limited types. In Ada 95, the key property of a limited type is that assignment is not predefined and cannot be defined (equality is not predefined either but it can be defined). The general idea of course is that there are some types for which it would be wrong for the user to be able to make copies of objects. This particularly applies to types involved in resource control and types implemented using access types.

However, although Ada 95 greatly improved the situation regarding limited types, nevertheless two major difficulties have remained. One concerns the initialization of objects and the other concerns the results of functions.

The first problem is that Ada 95 treats initialization as a process of assigning the initial value to the object concerned (hence the use of := unlike some Algol based languages which use = for initialization and := for assignment). And since initialization is treated as assignment it is forbidden for limited types. This means that we cannot initialize objects of a limited type nor can we declare constants of a limited type. We cannot declare constants because they have to be initialized and yet initialization is forbidden. This is more annoying in Ada 95 since we can make a type limited but not private.

The following example was discussed in the Introduction

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

Note that this type is explicitly limited (but not private) but its components are not limited. If we declare an object of type T in Ada 95 then we have to initialize the components (by assigning to them) individually thus

```
  X: T;
begin
  X.A := 10;  X.B := True;  X.C := 45.7;
```

Not only is this annoying but it is prone to errors as well. If we add a further component D to the type T then we might forget to initialize it. One of the advantages of aggregates is that we have to supply all the components which automatically provides full coverage analysis.

This problem did not arise in Ada 83 because we could not make a type limited without making it also private and so the individual components were not visible anyway.

Ada 2005 overcomes the difficulty by stating that initialization by an aggregate is not actually assignment even though depicted by the same symbol. This permits

```
X: T := (A => 10,  B => True,  C => 45.7);
```

We should think of the individual components as being initialized individually *in situ* – an actual aggregated value is not created and then assigned.

The reader might recall that the same thing happens when an aggregate is used to initialize a controlled type; this was not as Ada 95 was originally defined but it was corrected in AI-83 and consolidated in the 2001 Corrigendum [2].

We can now declare a constant of a limited type as expected

```
X: constant T := (A => 10,  B => True,  C => 45.7);
```

Limited aggregates can be used in a number of other contexts as well

- as the default expression in a component declaration,

  so if we nest the type T inside some other type (which itself then is always limited – it could be explicitly limited but there is a general rule that a type is implicitly limited if it has a limited component) we might have

  ```
  type Twrapper is
    record
      Tcomp: T := (0, False, 0.0);
    end record;
  ```

- as an expression in a record aggregate,

  so again using the type Twrapper as in

  ```
  XT: Twrapper := (Tcomp => (1, True, 1.0));
  ```

- as an expression in an array aggregate similarly,

  so we might have

  ```
  type Tarr is array (1 .. 5) of T;

  Xarr: Tarr := (1 .. 5 => (2, True, 2.0));
  ```

- as the expression for the ancestor part of an extension aggregate,

  so if TT were tagged as in

  ```
  type TT is tagged limited
    record
      A: Integer;
      B: Boolean;
      C: Float;
    end record;

  type TTplus is new TT with
    record
      D: Integer;
    end record;
  ```

```
...
XTT: TTplus := ((1, True, 1.0) with 2);
```

- as the expression in an initialized allocator,

  so we might have

```
type T_Ptr is access T;
XT_Ptr: T_Ptr;
...
XT_Ptr := new T'(3, False, 3.0);
```

- as the actual parameter for a subprogram parameter of a limited type of mode in

```
procedure P(X: in T);
...
P((4, True, 4.0));
```

- as the result in a return statement

```
function F( ... ) return T is
begin
  ...
  return (5, False, 5.0);
end F;
```

  this really concerns the other major change to limited types which we shall return to in a moment.

- as the actual parameter for a generic formal limited object parameter of mode in,

```
generic
  FT: in T;
package P is ...
...
package Q is new P(FT => (7, True, 7.0));
```

The last example is interesting. Limited generic parameters were not allowed in Ada 95 at all because there was no way of passing an actual parameter because the generic parameter mechanism for an in parameter is considered to be assignment. But now the actual parameter can be passed as an aggregate. An aggregate can also be used as a default value for the parameter thus

```
generic
  FT: in T := (0, False, 0.0);
package P is ...
```

Remember that there is a difference between subprogram and generic parameters. Subprogram parameters were always allowed to be of limited types since they are mostly implemented by reference and no copying happens anyway. The only exception to this is with limited private types where the full type is an elementary type.

The change in Ada 2005 is that an aggregate can be used as the actual parameter in the case of a subprogram parameter of mode **in** whereas that was not possible in Ada 95.

Sometimes a limited type has components where an initial value cannot be given as in

```
protected type Semaphore is ... ;

  type PT is
    record
      Guard: Semaphore;
      Count: Integer;
      Finished: Boolean := False;
    end record;
```

Since a protected type is inherently limited the type PT is also limited because a type with a limited component is itself limited. Although we cannot give an explicit initial value for a Semaphore, we would still like to use an aggregate to get the coverage check. In such cases we can use the box symbol <> as described in the previous section to mean use the default value for the type (if any). So we can write

```
  X: PT := (Guard => <>, Count => 0, Finished => <>);
```

The major rule that must always be obeyed is that values of limited types can never be copied. Consider nested limited types

```
  type Inner is limited
    record
      L: Integer;
      M: Float;
    end record;

  type Outer is limited
    record
      X: Inner;
      Y: Integer;
  end record;
```

If we declare an object of type Inner

```
  An_Inner: Inner := (L => 2, M => 2.0);
```

then we could not use An_Inner in an aggregate of type Outer

```
  An_Outer: Outer := (X => An_Inner, Y => 3);  -- illegal
```

This is illegal because we would be copying the value. But we can use a nested aggregate as mentioned earlier

```
  An_Outer: Outer := (X => (2, 2.0), Y => 3);
```

The other major change to limited types concerns returning values from functions.

We have seen that the ability to initialize an object of a limited type with an aggregate solves the problem of giving an initial value to a limited type provided that the type is not private.

Ada 2005 introduces a new approach to returning the results from functions which can be used to solve this and other problems.

We will first consider the case of a type that is limited such as

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

We can declare a function that returns a value of type T provided that the return does not involve any copying. For example we could have

```
function Init(X: Integer; Y: Boolean; Z: Float) return T is
begin
  return (X, Y, Z);
end Init;
```

This function builds the aggregate in place in the return expression and delivers it to the location specified where the function is called. Such a function can be called from precisely those places listed above where an aggregate can be used to build a limited value in place. For example

```
V: T := Init(2, True, 3.0);
```

So the function itself builds the value in the variable V when constructing the returned value. Hence the address of V is passed to the function as a sort of hidden parameter.

Of course if T is not private then this achieves no more than simply writing

```
V: T := (2, True, 3.0);
```

But the function Init can be used even if the type is private. It is in effect a constructor function for the type. Moreover, the function Init could be used to do some general calculation with the parameters before delivering the final value and this brings considerable flexibility.

We noted that such a function can be called in all the places where an aggregate can be used and this includes in a return expression of a similar function or even itself

```
function Init_True(X: Integer; Z: Float) return T is
begin
  return Init(X, True, Z);
end Init_True;
```

It could also be used within an aggregate. Suppose we have a function to return a value of the limited type Inner thus

```
function Make_Inner(X: Integer; Y: Float) return Inner is
begin
  return (X, Y);
end Make_Inner;
```

then not only could we use it to initialize an object of type Inner but we could use it in a declaration of an object of type Outer thus

```
An_Inner: Inner := Make_Inner(2, 2.0);
An_Outer: Outer := (X => Make_Inner(2, 2.0), Y => 3);
```

In the latter case the address of the component of An_Outer is passed as the hidden parameter to the function Make_Inner.

Being able to use a function in this way provides much flexibility but sometimes even more flexibility is required. New syntax permits the final returned object to be declared and then manipulated in a general way before finally returning from the function.

The basic structure is

```
function Make( ... ) return T is
begin
  ...
  return R: T do          -- declare R to be returned
    -- here we can manipulate R in the usual way
    -- in a sequence of statements
  end return;
end Make;
```

The general idea is that the object R is declared and can then be manipulated in an arbitrary way before being finally returned. Note the use of the reserved word **do** to introduce the statements in much the same way as in an accept statement. The sequence ends with **end return** and at this point the function passes control back to where it was called. Note that if the function had been called in a construction such as the initialization of an object X of a limited type T thus

```
X: T := Make( ... );
```

then the variable R inside the function is actually the variable X being initialized. In other words the address of X is passed as a hidden parameter to the function Make in order to create the space for R. No copying is therefore ever performed.

The sequence of statements could have an exception handler

```
  return R: T do
    ...                   -- statements
  exception
    ...                   -- handlers
  end return;
```

If we need local variables within an extended return statement then we can declare an inner block in the usual way

```
  return R: T do
    declare
      ...                 -- local declarations
    begin
      ...                 -- statements
    end;
    ...
  end return;
```

The declaration of R could have an initial value

```
  return R: T := Init( ... ) do
    ...
  end return;
```

Observe that these extended return statements cannot be nested but could have simple return statements inside

```
return R: T := Init( ... ) do
   if ... then
      ...
      return;                     -- result is R
   end if;
   ...
end return;
```

Note that return statements inside an extended return statement do not have an expression since the result returned is the object R declared in the extended return statement itself.

Although extended return statements cannot be nested there could nevertheless be several in a function, perhaps in branches of an if statement or case statement. This would be quite likely in the case of a type with discriminants

```
type Person(Sex: Gender) is ... ;

function F( ... ) return Person is
begin
   if ... then
      return R: Person(Sex => Male) do
         ...
      end return;
   else
      return R: Person(Sex => Female) do
         ...
      end return;
   end if;
end F;
```

This also illustrates the important point that although we introduced these extended return statements in the context of greater flexibility for limited types they can be used with any types at all such as the nonlimited type Person. The mechanism of passing a hidden parameter which is the address for the returned object of course only applies to limited types. In the case of nonlimited types, the result is simply delivered in the usual way.

We can also rename the result of a function call – even if it is limited.

The result type of a function can be constrained or unconstrained as in the case of the type Person but of course the actual object delivered must be of a definite subtype. For example suppose we have

```
type UA is array (Integer range <>) of Float;
subtype CA is UA(1 .. 10);
```

Then the type UA is unconstrained but the subtype CA is constrained. We can use both with extended return statements.

In the constrained case the subtype in the extended return statement has to statically match (typically it will be the same textually but need not) thus

```
function Make( ... ) return CA is
begin
   ...
   return R: UA(1 .. 10) do       -- statically matches
      ...
   end return;
end Make;
```

In the unconstrained case the result R has to be constrained either by its subtype or by its initial value. Thus

```
function Make( ... ) return UA is
begin
   ...
   return R: UA(1 .. N) do
      ...
   end return;
end Make;
```

or

```
function Make( ... ) return UA is
begin
   ...
   return R: UA := (1 .. N => 0.0) do
      ...
   end return;
end Make;
```

The other important change to the result of functions which was discussed in the previous paper is that the result type can be of an anonymous access type. So we can write a function such as

```
function Mate_Of(A: access Animal'Class)
                              return access Animal'Class;
```

The introduction of explicit access types for the result means that Ada 2005 is able to dispense with the notion of returning by reference.

This does, however, introduce a noticeable incompatibility between Ada 95 and Ada 2005. We might for example have a pool of slave tasks acting as servers. Individual slave tasks might be busy or idle. We might have a manager task which allocates slave tasks to different jobs. The manager might declare the tasks as an array

```
Slaves: array (1 .. 10) of TT;      -- TT is some task type
```

and then have another array of properties of the tasks such as

```
type Task_Data is
   record
      Active: Boolean := False;
      Job_Code: ... ;
   end record;

Slave_Data: array (1 .. 10) of Task_Data;
```

We now need a function to find an available slave. In Ada 95 we write

```
function Get_Slave return TT is
begin
   ...                     -- find index K of first idle slave
   return Slaves(K);       -- in Ada 95, not in Ada 2005
end Get_Slave;
```

This is not permitted in Ada 2005. If the result type is limited (as in this case) then the expression in the return statement has to be an aggregate or function call and not an object such as Slaves(K). In Ada 2005 the function has to be rewritten to honestly return an access value thus

```
function Get_Slave return access TT is
begin
  ...                          -- find index K of first idle slave
  return Slaves(K)'Access;          -- in Ada 2005
end Get_Slave;
```

and all the calls of Get_Slave have to be changed to correspond as well.

This is perhaps the most serious incompatibility between Ada 95 and Ada 2005. But then, at the end of the day, honesty is the best policy.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment.

[2] ISO/IEC 8652:1995/COR 1:2001, *Ada Reference Manual – Technical Corrigendum 1.*

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Ada Reuse Guidelines

Muthu Ramachandran

School of Computing, Leeds Metropolitan University,The Headingley Campus, Beckett Park, Leeds LS6 3QS, UK; Email: m.ramachandran@leedsmet.ac.uk

Abstract

In this paper, we discuss the general area of software development for reuse and reuse guidelines. We identify, in detail, language-oriented and domain-oriented guidelines whose effective use affects component reusability. We also discuss the application domain of abstract data structures and propose an alternative view of reusability to that of Booch [3], whose work is well known in this area. Our guidelines are used as an effective technique for domain analysis. We have developed a prototype software system which takes Ada components and provides objective detailed advice on how to construct reusable components.

Keywords *Software reuse guidelines, domain analysis, Ada reuse guidelines, reuse improvement*

1. Introduction

Software component reuse is the key to significant gains in productivity. However, to achieve its full potential, we need to focus our attention on *development for reuse*, which is a process of producing potentially reusable components. We know clearly the difficulties that are faced when trying to reuse a component that is not designed for reuse. Therefore, the emphasis of the research described here is on development for reuse rather than *development with reuse*, which is a process of normal systems development (i.e., existing form of reuse). The process of developing potentially reusable components depends solely on defining their characteristics such as language features and domain abstractions. Reuse guidelines can represent such characteristics clearly. Therefore, we need to formulate objective and automatable reuse guidelines.

There have been previous studies on reuse guidelines [3,10,5, 8, 12, 13, &34], but these authors emphasise on general advice including design, documentation and management issues.

In this paper, we will explore the general area of development for reuse and discuss how we can formulate realisable and objective reuse guidelines. We will also review some of these existing guidelines and present our guidelines. Why do we need such objective and realisable reuse guidelines? They are important for:

- Assessing the reusability of software components against objective reuse guidelines.
- Providing reuse advice and analysis.
- Improving components for reuse which is the process of modifying and adding reusability attributes.

In our work, reuse guidelines fall into two classes:

Ada reuse guidelines: Most existing programming languages including object-oriented languages provide features that support reuse. However, simply writing code in those languages doesn't promote reusability. Components must be designed for reusability using those features. Such features must be listed as a set of design techniques for reusability before design takes place.

Domain-oriented reuse guidelines: Guidelines which are relevant to a specific application domain. We discuss more on this in a later section of this paper.

The language we have chosen for study is Ada, and the application domain chosen is components of abstract data structures (ADS). The main reason for choosing Ada is because of its explicit technical support for reuse, features such as the packaging mechanism, generics, support for abstraction, exceptions, parameterisation, building blocks, and information hiding. The reason for choosing ADS as the application domain is partly because, as computer scientists, we might be considered domain experts ourselves in this area and partly because it has been extensively studied and documented. These components are the fundamental building blocks for many applications.

2. Development for reuse

We argue that reuse impacts the software development process in two distinct ways (Sommerville and Ramachandran [31]):

1. *Development with reuse*.

Software design takes place in an environment where a significant number of potentially reusable components are available. It is an existing form of software reuse practice. An example of this kind is UNIX environment. The objective is to produce a software product. During the past years of active research on reuse, most emphasis has been given to development with reuse. As a result, there is no large body of components, except in specific domains such as mathematical libraries, which have been generalised for reuse.

2. *Development for reuse*.

An objective of the design process is to produce components which are potentially reusable. These components form building blocks for future development (over the long term) and are applicable for various situations and perhaps across application domains.

In development with reuse, reuse is desirable but there need be no resources expended in creating new reusable components. Development for reuse implies expending resources specifically to increase the reusability of components. In many cases, this process might follow development with reuse where components generated during normal system development are made more reusable by generalisation and improvement.

Our notion of the development for reuse process is shown in Figure (1), in which there are a number of stages to be followed which start from identifying an application domain, identify & classify reusable abstractions, domain-oriented reuse, language-oriented reuse, design components, assessment for reuse, improvement for reuse, and deliver potentially reusable components. The idea is to identify a number of frequently reusable domain-specific abstractions (using classification or by interview with domain experts) and then to apply domain-specific and language-specific criteria that are defined by the reuse guidelines.

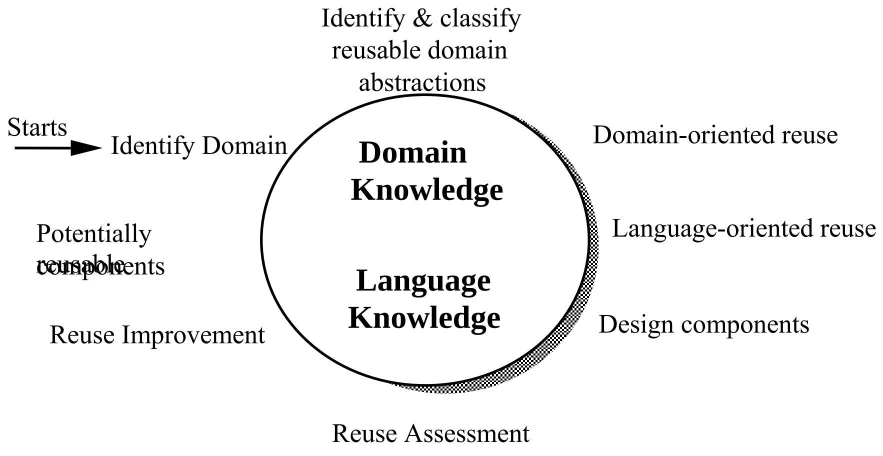

Figure 1 The process of development for reuse

The development for reuse process proceeds in a number of stages.

1. Identify domain. Domain analysis has been identified as essential for effective reuse. The first step is to identify a specific application domain and define its boundary.

2. Identify and classify (frequently) reusable abstractions. To identify potentially reusable components, the reuse assessor must know what the important domain abstractions are and how frequently these abstractions are used in systems developed for that domain. There is not much point in devoting a lot of effort in producing a reusable domain abstraction if that abstraction is rarely used. Domain classification helps to identify effective reusable abstractions. This stage involves interviewing domain experts, surveying domain literature and studying existing systems.

3. Identify design/programming language constructs that support reuse. Selecting an appropriate language is an important part of development for reuse. We should be able to express our reuse guidelines effectively using language mechanisms.

4. Study and formulate language reuse guidelines (rules concerning language support for reuse). This emphasises the effective use of language features for reuse. This process includes studies of existing techniques and appropriate modifications to them.

5. Study and formulate domain reuse guidelines (rules concerning the domain characteristics for reuse). This emphasises the reusable domain abstractions that are identified in the application domain. Guidelines should not just be general advice but should be specific and verifiable for creating potentially reusable components. Design/redesign components based on these guidelines.

6. The next step, known as reuse assessment is a process of assessing components based on the number of guidelines satisfied against the total number of guidelines that are applicable, and then produce an assessment report. This is where we need to automate this process. The outcome of this process is to make sure that the components designed for reuse satisfy some of the key characteristics.

7. The final step, known as reuse improvement is a process of modifying and improving these components for reuse by adding attributes of an abstraction for reuse. This process is based on the assessment report produced during the previous step. The reuse improver must know what attributes of an abstraction must be generalised to make it reusable. Again, an automatic reuse improvement is essential. Finally, produce potentially reusable components.

8. Automate, where possible, these two processes of assessing and improving components for reuse.

It is unrealistic to expect reusable components to be produced as a side-effect of normal systems development. The reasons for this are partly technical and partly managerial. Technically, the notion of what constitutes a reusable component is not well-understood and engineers working on a project cannot be expected to wrestle with this problem while developing to a given set of requirements. Furthermore, the requirements for a particular project may be such that components have to be very specific in order to satisfy them.

Furthermore, a project manager's principal responsibility is to deliver the required software system on time and within budget. Creating reusable components requires additional effort to be expended which is of no immediate benefit to that project. The project manager cannot reasonably be expected to give reusable component production a high priority.

Thus, we believe that the normal mode of production of reusable components should be to take existing components and to add reusability to them. This extra cost for reuse must be an organisational rather than a project responsibility. Reusability is an attribute which can be added at any level from the specification through to the implementation. In our work, we are principally concerned with the reusability of compilable components. However, we believe that the approach discussed is equally applicable to formal specifications and software designs.

Design for reusable component is dependent on the effective use of the programming language used to implement the component and application domain knowledge. Application domain knowledge allows the abstractions in a domain to be identified and encoded as a set of reusable components. The objective of our work is to use language and domain knowledge to assess, with automatic assistance, the reusability of a component and to suggest to the software engineer how that component may be made more reusable.

3. Reuse Guidelines

Development for reuse requires that the language features must be used effectively. The objective of language-oriented reusability is to exploit the use of language support for reuse and to capture the domain knowledge efficiently. There have been experiments conducted to show that experienced programmers can reuse better than novices (Soloway and Ehrlich 1984). The idea is to formulate a set of verifiable reuse guidelines (derived from experts and existing systems and literature, for example available Ada reuse guidelines).

The major technical problems of development for reuse are:

- How to identify the characteristics of a reusable component?
- How to assess and improve reusability attributes of a component automatically?
- How to encode and analyse application domain knowledge?

The work described here addresses these problems and hence considers factors affecting reusability such as language factors and domain factors. We believe objective and realisable guidelines will help to solve these problems. Existing studies on creating reusable components [12, 34, 10, 3, 8, &5] fall into the following classes:

1. *Highly Conceptual* studies which try to be language independent but very abstract. For example, all such studies say reusable components should be:

- Highly cohesive, meaning that they should represent a single abstraction.
- Loosely coupled, meaning that they should be largely independent of any other abstraction.

There are other three such criteria proposed by Gargaro and Pappas [9] specifically for Ada programs. A reusable program should be:

- Transportable
- An orthogonal (context-independent) composition, and
- Independent of the runtime system.

More recently, Hollingsworth (1992) proposed a set of discipline for constructing high-quality components:

- Correctness
- Composability
- Reusability
- Understandability

Again, this is interesting but these are general programming principles rather than a discipline for reuse. Similarly, Tracz [32] and Weide at al. [34] have proposed a framework based on a highly abstract idea, known as the 3C model:

- *Concept*: a statement of what a piece of software does, factoring out how it does it; abstract specification of functional behaviour.
- *Content*: a statement of what a piece of software achieves the behaviour defined in its concept; the code to implement a functional specification.
- *Context*: aspects of the software environment relevant to the definition of concept or content that are explicitly part of the concept or content.

2. *Language oriented* studies which produce guidelines for a specific programming language and suggest how features of that language affect reusability. Gautier and Wallis [10] have done extensive studies on Ada reuse guidelines. However, some of their guidelines are interesting advice rather than practical and realisable reuse guidelines. For example, they say:

- Avoid taking a design decision that the reuser can take later. Where possible allow the reuser to defer decisions until runtime.
- Avoid implementing a package in such a way that it maintains state in private variables.

Their guidelines on Ada generics say:

- Components should be generic, even if no parameterisation is required.
- When generic components have many formal parameters consider specifying the components as a nested generic.
- Do not unnecessarily restrict generic formal types. Limited private types provide for maximum reuse potential.

Let us also review some of the existing guidelines on the *design of abstract data types*. For example, studies by

Gautier and Wallis [10] and Braun and Goodenough [5] say:

- Components should model a single abstraction.
- Develop models by generalising from real problems [5].

These guidelines seem to contradict another of their own guidelines, which says,

- Where possible make local abstractions separate components.
- For a package that models an abstract data type, instantiation of the package should be used in preference to deriving the type representing the abstract values.

Similarly, Hollingsworth [12] has proposed some forty guidelines that are similar to the above guidelines. For example:

Principle 1 - Make generic package the unit of modularity.

Principle 2 - Export a type so that abstract state is maintained in variables of that type, not in package instances.

Principle 13 - Parameterise the component by each ADT that it manipulates but does not export.

Principle 16 - Formally specify the behaviour of each component using a mathematical specification language.

As we can see these are clearly general programming principles rather than a discipline for reuse. Like others, his guidelines are also difficult to automate.

In general, existing guidelines do not address our real problem of defining and identifying reusable attributes. How would we generalise an abstraction? How do we assess and add reusable attributes automatically? To address all these problems, we need to formulate practical and objective reuse guidelines.

Our work has taken these existing studies as a starting point and has attempted to produce more detailed and practical guidelines on the way in which language and domain features affect reusability. Compared to these existing reuse guidelines [12, 34, 10, 3,8, 5], our guidelines are, practical and objective, domain-specific, comprehensive, classified, support design for reuse, and have been implemented for automated improvement for reuse.

In our work, we classify these into language-oriented and domain-oriented reuse guidelines. Language-oriented guidelines are further classified into language-independent reuse guidelines that are realisable in all programming languages and language-dependent reuse guidelines that are specific to Ada. In the domain of abstract data structures (ADS), reuse guidelines are classified into guidelines on sequential and concurrent structures. Guidelines on sequential structures are further classified into guidelines on linear and non-linear structures which are further classified into guidelines on static ADS and dynamic ADS.

4. Ada Reuse Guidelines as Knowledge Representation

To support the process of development for reuse, we need to represent reuse knowledge effectively. Objective reuse guidelines represent the reuse characteristics of language knowledge and domain knowledge. In our work, guidelines are represented as rules because these are collected as verifiable rules. In this section, we discuss some of these knowledge guidelines for reuse.

Language-oriented reusability is a process in which the language support for reuse is analysed. For example, recent studies by Sommerville [30] show that C++ is being used in a way which is very similar to C programming. As a result, components are designed without using specific C++ features for reuse. In Ada, the effective use of generics and the packaging mechanism support reusability.

1. *Language-independent and Language-dependent reuse guidelines (such as Ada Reuse Guidelines).* Language-independent guidelines are concerned with the effective language features which are common across languages. For example, one of our guidelines says, "Always provide a means to discover the array size". The purpose of this guideline is to ensure any previous values are not retained. In C, this guideline can be implemented by passing a parameter which is the array size.

In Ada, the same can be said:

• Always use the FIRST, LAST and RANGE attributes to discover the lower bounds, the upper bounds and the size of an array.

Ada's predefined attributes must be used to predict the array size directly. The array structure is supported across the programming languages and it is used across applications.

2. *Ada reuse guidelines*. Language-specific reuse guidelines are concerned with the effective use of Ada's support for reuse. Ada reuse guidelines emphasise rules concerning its support for reuse. For this work we have used Ada 83 (mainly) and Ada 95 (some) mainly because of our experience in using them. Here, we present only a sample of our guidelines. In our work, Ada reuse guidelines fall into a number of classes based on various Ada constructs:

- Design of Ada packages
- Design of Ada types
- Design of Ada generics
- Design of Subprogram interfaces
- Design of Ada tasks

1. *Ada packages*. The packaging mechanism of Ada supports the representation of abstract data types and reuse of building blocks. Hence, it is important to formulate guidelines and design components based on the knowledge about abstract data types. One of our guidelines on the use of Ada packages says:

• Always hide information by using access types for detailed structural representation.

This guideline supports reuse of specification, abstract data types, the principle of information hiding, and reuse of building blocks. This allows reusers to modify and defer detailed design decisions in the package body without affecting its specification and other parts of the system. This guideline must be satisfied if a component has to be certified as reusable. It emphasises that access types should be used to hide structural implementation. This can be done by defining structure type as private in the visible part of the Ada package and the detailed design information should be defined in the package body.

2. *Private and limited private types*. Ada supports the use of private types and limited private types to build abstract data types. It is sometimes difficult to choose between these and there are merits and demerits of both. A limited private type can't be assigned but can provide automatic initialisation and control of allocation of all objects whereas a private type allows assignment of objects but can't support automatic initialisation and allocation of all objects. This selection depends on a design rationale which needs to be addressed here.

Let us look at what some of the existing guidelines by Gautier and Wallis 1990 [10] say on this design rationale on selecting appropriate private types.

• A private type should be implemented either as an access type or as a record type with a default component value which enables an uninitialised object to be detected.

• Export abstract data types as limited private types.

These guidelines do not say clearly when to choose private and limited private types. The first one provides advice rather than a reuse guideline because Ada provides automatic initialisation for all composite types. The last one says to choose always 'limited private' which is not practical for all structures. In Ada, choosing a particular private type involves a heuristic design decision to be made, so we need a precise design rationale.

For example, one of our Ada reuse guidelines says,

- Always select a private type for all objects which are static structures and limited private type for all objects which are dynamic structures.

This guideline provides a checkable design rationale for choosing an appropriate private type. It says to choose a private type for which the representation is static and to choose a limited private type for which the representation is dynamic. Ada's private types export an abstraction of a component, reveal the properties of a component to be reused, allow you to build a structure of a structure, and adopt the information hiding principle. This guideline emphasises that you should choose private type for static structures because they allow you to assign objects individually whereas for dynamic structures we would not normally make such assignments. Hence you choose limited private types for dynamic structures.

When a type is private then all the predefined operations are available on the type outside the package to the reuser. For this reason, a private type is preferred for static structures which might require individual assignment and manipulation using those operations. On the other hand, if we choose a private type for dynamic structures then the objects can't be controlled and it is unsafe because all the predefined operations are available on the type outside the package to the reuser. For this reason, a limited private type is preferred for dynamic structures.

4.1 Guidelines on Abstract Data Structures

Domain classification is an important and difficult part of modern domain engineering. It helps to identify effective reusable abstractions and model the problem domain. Booch [3] has proposed a classification scheme, known as Booch's components. In his scheme, components are classified into *structures, tools, and subsystems*. He has characterised a structure as an ADT (abstract data type) or ASM (abstract state machine). Most of the ADS are considered as monolithic or polylithic components. Monolithic components are stacks, strings, queues, dequeues, rings, maps, sets, and bags. Polylithic components are lists, trees, and graphs. Tools are utilities, filters, pipes, sorting, searching, and pattern matching. Again these are further classified into various *forms of a component*, which represent variations on the theme of components for differences on time and space requirements. The forms are sequential, guarded, concurrent, and multiple.

Booch's work [3] has been used as a starting point for constructing reusable components. However, his notion of forms represents only minor variations in implementation and is cumbersome for the reuser to choose a particular implementation because there are too many variants. For example there are more than twenty-six variant forms of stack components.

Our objective is to formulate realisable domain reuse guidelines to represent the design of reusable components of abstract data structures (ADS). These reuse guidelines are kept as general as possible, and not specific to any particular language, but specific to this domain of ADS. The main purposes of these guidelines are firstly, to support development for reuse in the application domain of ADS. Secondly, to estimate the reuse potential of a program automatically, and followed by this to improve components for reuse by representing these guidelines within this domain. Domain reuse guidelines are based on a proposed classification scheme.

In our work, we have proposed a classification scheme for the domain of abstract data structures (ADS) as shown in Figure 2. In this scheme, ADS have been classified into sequential and concurrent structures. The sequential structure is further classified into linear, and non-linear structures. An important further sub-classification is static and dynamic abstractions which can be kept together as a single abstraction. This classification is important for the following reasons.

Guidelines that have been formulated refer to specific parts of the classification structure, mainly sequential structures.

Sub-classification is limited to static and dynamic structures which are single, generalised, and easy to reuse.

A single and generalised abstraction is more reusable than an abstraction with several versions, which are called forms in Booch's components [3].

The domain boundary is clearly defined which is important to do domain analysis effectively, whereas one of the Booch's sub-classification known as Subsystem is left undefined.

Booch's sub-taxonomy needs further refinement and his classification scheme is far too general (structures, tools, and subsystems) which makes the domain boundary and scope undefined and divergent. There are too many forms in Booch's scheme whereas our scheme proposes only two distinct forms namely static and dynamic, and it is based on a specific application domain (the domain boundary is clearly defined and limited) rather than general.

There are good reasons for keeping abstractions together rather than having several versions (or forms) for each minor variation. It may be difficult for the reuser to understand each of these minor variations before reusing a component. For example, Booch's has provided with a notion of bounded and unbounded for monolithic components and controlled and uncontrolled for concurrent structures. These can lead to unmanageable components with variations. We believe these should always be designed as manageable. Similarly, variations for iterator and noniterator, there is no need for noniterator. Our conclusion is that most of these forms would never be reused. In our work on domain analysis, support is provided in identifying frequently reusable abstractions.

4.2 Knowledge representation

Probably there is no best and easy method of domain representation. Research is underway on how to do domain analysis, and on domain representation (Prieto-Diaz 1990). In our work, the approach taken is rule-based representation. Reuse guidelines are represented as rules. An example of the rule is:

```
IF    abstract structure is complex AND
        all operations are independent of the type of the
        structure element THEN
        Component should be implemented as a generic
        package with the element type
        as a generic parameter;
END IF;
```

However, automating some of these guidelines breaches this rule. For example, one of our guidelines on defining the list of operations on object creation, termination, object inquiry, and state change, involves more than one interaction and transformations. Hence it breaches our single if-then rule and depends on applying domain knowledge for further transformations. This information is modelled using a component template and the reusability is assessed and improved by comparing the component with that template.

Some of our guidelines are illustrated here:

1. *Design of abstract data types*. The notion of an abstract data type allows you to express real world
2. *Entities of an application domain*. It allows you to separate a specification from an internal representation of a structure (principle of information hiding). It means we are able to specify an abstraction of a component in terms of its actual interface descriptions together which is useful to generalise that abstraction for reuse. It allows the designer to view a system at a more abstract level and to change the representation of ADS without affecting their use in other parts of the system.

One of our guidelines on ADS says,
• For all complex structures, provide two representations such as static and dynamic structures for each domain abstraction.

This guideline says, for each structure, provide two abstractions such as static which is represented using an array structure and dynamic which is represented using dynamic structure (access/pointer). This provides a choice and maximum flexibility for the reuser with improved reuse potential. For example, in Ada, we can design two packages for each structure implemented statically and dynamically. If an abstraction is to be represented in Ada then we can apply various Ada reuse guidelines. For example, one on the rationale for choosing private types say, choose limited private for complex and dynamic structures, and choose private type for static structures. However, the Ada library mechanism is inadequate in that it rises naming conflict when there are two library units with similar names which mean that the implementation of similar components must have different names. This has been solved in Ada 95 with the use of child packages.

Another important guideline (Braun and Goodenough 1985) on the design of abstract data structures emphasises the need for providing methods for a list of operations such as object creation, object termination, state change, state inquiry, and input and output. They have not considered operations on exceptions that deal with error conditions. We believe that the operations on exceptions and handling are significant for reusable and reliable components. In our work we have extended this guideline to include operations on exceptions handling.

Our extended guideline on ADS says,

• The components should be provided with the following operations on ADS.

• Creation

• Termination

• Conversion

• State inquiry

• State change

• Input/ output representation, and

• Exceptions

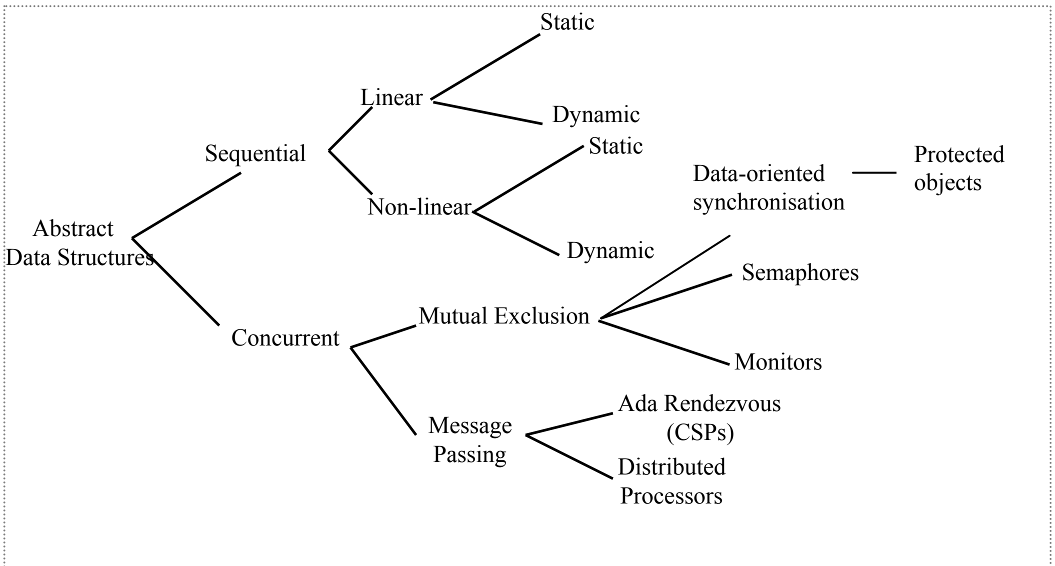

Figure 2 Classification of Abstract Data Structures

Creation involves both creating and initialising an object, termination is a means of making the object inaccessible for the remainder of its scope, conversion allows for the change of representation from one type to another, state inquiry functions allow the user to determine the state of the object and boundary conditions, state change functions allow modifying or changing the contents of the object, input/ output representations are primarily useful for debugging purposes, and exceptions deal with error conditions and exception handling procedures. Each operation emphasises one or more functionality so that the services offered by the component are increased thus leading to improved reusability. Sometimes components which do not provide all these operations may well be reused. In such cases, the component has to be measured based on the degree of reusability.

2. *Other guidelines*. Our guidelines on the design of reusable static and dynamic structures, and on space management are essential, objective and realisable. Complete set of guidelines can be found in Ramachandran (1992). Some of our important domain guidelines are,

- Always define a constrained array structure to represent a component of static structure.

Always select dynamic object representation for all complex structures and hide detailed structural information.

If the abstract structure is complex and all operations are independent of the type of the structure element then that component should be implemented as a generic package with the element type as a generic parameter.

- Always provide a procedure to record the maximum size of the free list with a counter so that the user may increase or decrease the size of the free list. when decreasing the free list size, space in excess of the new size is returned to the system.

Always provide a procedure to release the free list, so that all space in the free list is returned to the system completely.

- For each exception, provide an exception handler.

In the following section we will see how these guidelines can be implemented as a tool for automated improvement and advisory system which can take Ada code and provides an assessment and improvement for reuse.

5. Automation

The guidelines discussed in this paper have been partially or completely automated in our system for which a prototype has been developed as shown in Figure 3. Some of them involve straightforward transformation and others might need user interaction and domain knowledge. This system takes an Ada component, checks through various reuse guidelines that are applicable, provides reuse advice and analysis to the reuser, and generates that component which is improved for reuse. Ada components are modelled using component templates and reuse guidelines are checked objectively against that template. Some of these domain reuse guidelines have been represented and analysed using component templates. For most of these guidelines, automation depends on some user interactions and domain knowledge.

One of the major objective of this system is to demonstrate, how well-defined reuse guidelines can be used to automate the process of reuse assessment by providing support for language analysis and domain analysis. For example, this system takes an Ada component specification, assesses it through two analysis phases, estimates its reusability according to how well it satisfies a set of reuse guidelines and generates a component which is improved for reuse The system interacts with the engineer to discover information that can't be determined automatically. The conclusion of this first pass is an estimate of how many guidelines are applicable to the component and how many of these have been breached. The report generator produces a report with all the information that has been extracted about that component and

changes that have been made for reuse.

The second pass involves applying domain knowledge to the system. The component templates have been modelled representing static and dynamic structures. Their reusability is assessed by comparing the component with that template. The support provided by the system ensures that the reuse

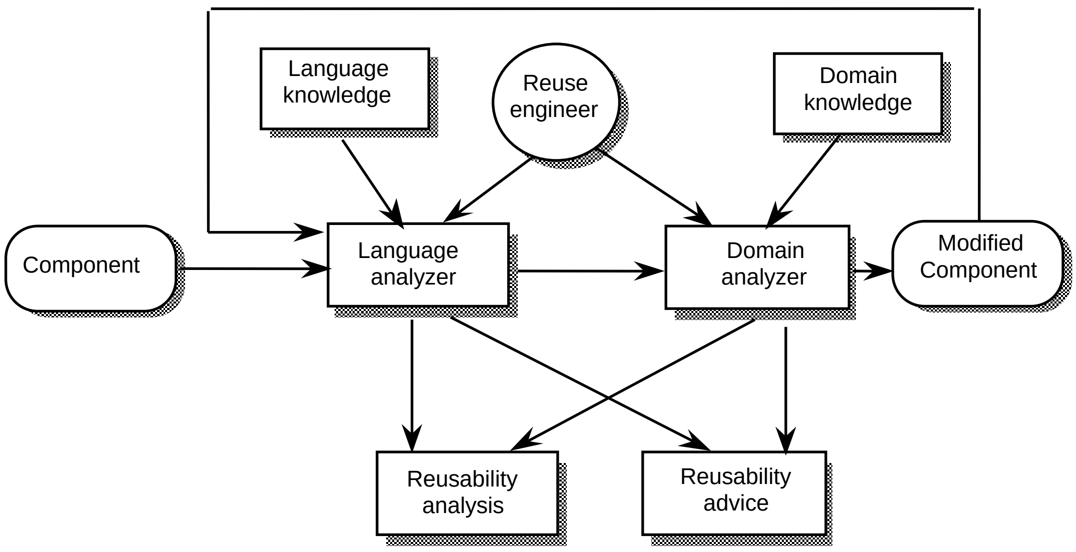

Figure 3 The reuse assessor and improver system

engineer carries out a systematic analysis of the component according to the suggested guidelines. He or she need not be a domain expert. Again, an analysis is produced which allows the engineer to assess how much work is required to improve system reusability.

For example, a scheme for automating one of our domain guideline is shown algorithmically in Figure 4. This scheme involves identification of procedures and domain related information against a component template, and adds operations automatically to those components with perhaps some human assistance.

Guidelines for automation are represented in two distinct ways:

- Wherever possible, a rule-based representation is used so that it is clear when a guideline should be applied. We have found that rule-based representations are mostly applicable for language-oriented guidelines.
- For domain-oriented guidelines, we are mostly concerned with checking that a component fits a model of a reusable domain abstraction. In this case, we have developed templates of these abstractions which represent the reuse guidelines.

However, it remains to see how many numbers of guidelines are significant for reuse, and further

investigation is underway to improve its limitations. We find our approach interesting and the system has demonstrated that it is possible to formulate and automate practical and objective reuse guidelines supporting the development of potentially reusable software components.

6. Conclusion

Reusable components can be produced and re-engineered effectively in a large scale if we can formulate objective and realisable guidelines and apply them systematically. Existing guidelines are general advice and often not checkable. Domain analysis can play a major role in supporting development for reuse in the near future. Our approach to the production of reusable components has proved to be practical and effective to that of Booch's approach [3].

Our classification scheme is domain specific, and has well-defined scope and boundary. We have taken an alternative view to that of existing studies on reuse guidelines [12, 34, 10, 3, 5, 8], in which we have applied reuse guidelines to model and analyse domain-oriented reusability and language-oriented reusability. In our work, guidelines are also adopted for knowledge representation. Our conclusion is that it is possible to produce a set of objective and practical reuse guidelines which can be applied systematically to improve reusability. We also believe that our approach is applicable to other languages, methods, tools, and application systems.

```
For each ADS do
  Analyse abstractions and advice;
  Identify attributes for static and dynamic
          structures with user interactions;
  Identify procedures against Braun and
                  Goodenough's list;
  Add operations that are missing;
  Follow further schemes for Static and/ or Dynamic
                        Structures;
  Produce reports and potentially reusable
                   components;
End loop;
```

Figure 4 Scheme for automating domain guidelines

Acknowledgement

The author would like to thank Prof I Sommerville, Department of Computing, University of Lancaster for his valuable comments provided on this paper.

7. References

[3] Biggerstaff, T.J. and Perlis, A.J. (1984), "Foreword to the special issue on software reusability", IEEE trans. on software engineering, September.

[4] Biggerstaff, T.J. and Perlis, A.J. (Editors) (1989), "Software Reusability: Concepts and Models", Vol.I & II, ACM Press, Addison-Wesley.

[5] Booch, G. (1987), "Software Components with Ada", Benjamin/Cummings.

[6] Bott, M.F. and Wallis, P.J.L. (1988), "Ada and software reuse", Software Engineering Journal, September.

[7] Braun, C.L. and Goodenough, J.B. (1985), "Ada Reusability Guidelines", Report 3285-2-208/2, USAF.

[8] CAMP (1987), "Common Ada Missile Packages", Final Technical Report, Vols. 1, 2 and 3. AD-B-102 654, 655, 656, Airforce Armament Laboratory, FL.

[9] Carter, J.R. (1990), "The Form of reusable Ada Components for Concurrent Use", Ada Letters, vol.X, No.1, Jan/Feb.

[10] Dennis, R.J.St. (1987), "Reusable Ada (R) software guidelines", proc. of the 12th annual Hawaii International conference on system sciences, pp.513-520.

[11] Gargaro, A. and Pappas, T.L. (1987), "Reusability issues and Ada", IEEE software, pp.43-51, July.

[12] Gautier, R.J. and Wallis, P.J.L. (Editors) (1990), "Software Reuse with Ada", Peter Peregrinus Ltd for IEE/BCS.

[13] Genillard, C., Ebel, N., and Strohmeier, A. (1989), "Rational for the design of reusable abstract data types implemented in Ada", Ada letters, vol.IX, No.2, March/April.

[14] Hollingsworth, J (1992). Software components design for reuse: a language independent discipline applied to Ada, PhD thesis, Dept. of computing and Information, Ohio State Univ., Columbus, December.

[15] Hooper, J. W. and Chester, R. O. (1991). Software Reuse: Guidelines and Methods, Plenum Press.

[16] Keenan, P. (1987), "Reuse of Designs as a First Step Towards the Introduction of Ada Component Reuse", IEE Colloquium on Reusable Software Components, May.

[17] Krueger, C (1992) Software Reuse, ACM Surveys, Vol. 24, No. 2, June 1992.

[18] Latour, L. (1991), " A methodology for the design of reuse engineered Ada components", Ada Letters, spring.

[19] Lubars, M. (1991), Domain analysis and domain engineering in IDeA, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

[20] Maiden, N A M and Sutcliffe, A G (1992) Exploiting reusable specifications through analogy, Communications of the ACM 34(5), May, 1992.

[21] McCain, R. (1985), "Reusable Software Component Construction: A Product Oriented Paradigm", In Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference, Long Beach, CA, 125-135, October 21-23.

[22] Moore, J M and Bailin, S C 1991. Domain Analysis: Framework for reuse, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

[23] Neighbors, J.M. (1984), "The Draco Approach to constructing Software from reusable components", IEEE Trans. on Software Engineering, vol.SE-10, No.5, pp.564-574, September.

[24] Prieto-Diaz, R and Frakes, W. B (1993) Advances in software reuse, Proc. of the second international workshop on software reusability (IWSR-II Lucca, Italy, March 1993) IEEE Computer Society Press, March 1993.

[25] Prieto-Diaz, R. (1990), "Domain Analysis: An Introduction", ACM SIGSOFT, Software Engineering Notes, vol 15, no.2, Page 47, April.

[26] Prieto-Diaz, R. and Arango, G (1991) Software Modelling and Domain Analysis, IEEE Computer Society Press Tutorial.

[27] Ramachandran, M. (1992) An Investigation into Tool Support for the Development of Reusable Software, PhD thesis, May, Lancaster University.

[28] Ramachandran, M. (1994) Knowledge-based support for reuse, Proceedings of Intl. conf. on software engineering and knowledge engineering (SEKE94), Latvia, July.

[29] Schafer, W., Prieto-Diaz, R., and Matsumoto, M. (1994). Software Reusability, Ellis Horwood.

[30] Simos, M. (1991), The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

[31] Soloway, E and Ehrlich, K. (1984), "Emprical studies of programming knowledge", IEEE Transactions on Software Engineering, Vol. SE-10, No.5, September.

[32] Sommerville, I. and Morrison, R. (1987), "Software Development with Ada", Addison-Wesley.

[33] Sommerville, I. and Ramachandran, M. (1991), "Reuse Assessment", First International Workshop on Software Reuse, Dortmund, Germany, July.

[34] Tracz, W. (1990), "The 3 Cons of Software Reuse," in the proceedings of the *Third Annual Workshop on Software Reuse*, July, Syracuse, NY.

[35] Wartik S and Prieto-Diaz, R. (1992), Criteria for comparing reuse-oriented domain analysis approaches, Intl. J. of Soft. Eng. and knowledge Eng., Vol 2, No. 3.

Weide, B.W et al. (1991) Reusable software components, Advances in Computers, Yovits, M. C (ed.), Vol. 33, Academic Press.

Ada-Europe 2005 Sponsors

ACT Europe *Contact: Zépur Blot*	8 Rue de Milan, F-75009 Paris, France Tel: +33-1-49-70-67-16 Email: sales@act-eurpoe.fr	 Fax: +33-1-49-70-05-52 URL: www.act-europe.fr
Aonix *Contact: Jacques Brygier*	66/68, Avenue Pierre Brossolette, 92247 Malakoff, France Tel: +33-1-41-48-10-10 Email : info@aonix.fr	 Fax: +33-1-41-48-10-20 URL : www.aonix.com
Artisan Software Tools Ltd *Contact: Emma Allen*	Suite 701, Eagle Tower, Montpellier Drive, Cheltenham, GL50 1TA, UK Tel: +44-1242-229300 Email : info.uk@artisansw.com	 Fax: +44-1242-229301 URL : www.artisansw.com
Esterel Technologies *Contact: Ian Hodgson*	PO Box 7995, Crowthorne, RG45 9AA, UK Tel: +44-1344-780898 Email : sales@esterel-technologies.com	 Fax: +44 1344 780898 URL : www.esterel-technologies.com
Green Hills Software Ltd *Contact: Christopher Smith*	Dolphin House, St Peter Street, Winchester, Hampshire, SO23 8BW, UK Tel: +44-1962-829820 Email :	 Fax: +44-1962-890300 URL : www.ghs.com
I-Logix *Contact: Martin Stacey*	1 Cornbrash Park, Bumpers Way, Chippenham, Wiltshire, SN14 6RA, UK Tel: +44-1249-467-600 Email : info_euro@ilogix.com	 Fax: +44-1249-467-610 URL : www.ilogix.com
LDRA Ltd *Contact: Brenda Pedryc*	24 Newtown Road, Newbury, Berkshire, RG14 7BN, UK Tel: +44-1635-528-828 Email: info@ldra.com	 Fax: +44-1635-528-657 URL: www.ldra.com
Praxis High Integrity Systems Ltd *Contact: Rod Chapman*	20 Manvers Street, Bath, BA1 1PX, UK Tel: +44-1225-466-991 Email : sparkinfo@praxis-his.com	 Fax: +44-1225-469-006 URL : www.sparkada.com
Silver Software *Contact: Steve Billet*	Riverside Buisness Park, Malmsebury, SN16 9RS, UK Tel: +44-1666-580-000 Email: enquiries@silver-software.com	 Fax: +44-1666-580-001 URL: www.silver-software.com
TNI Europe Limited *Contact: Pam Flood*	Triad House, Mountbatten Court, Worrall Street, Congleton, CW12 1DT, UK Tel: +44-1260-29-14-49 Email: info@tni-europe.com	 Fax: +44-1260-29-14-49 URL: www.tni-europe.com