Thoughts on Ada and Language Technology in Our Time

Franco Gasperoni, AdaCore, gasperoni@adacore.com

It is fascinating to open the second edition of John Barnes' book "Programming in Ada". If you are lucky to own a copy I encourage you to read the "Foreword" by Jean Ichbiah, the two prefaces, section 1.1 "History", and the last page in the "Finale". John's witty style makes the reading very enjoyable. Upon completing the reading, almost 20 years after my initial one, I emerged with a number of thoughts described in the following pages that could very succinctly be summarized as follows:

- Ada is meant for industrial systems and that is just fine. Ada does not have to be used everywhere to be useful and successful.
- Modeling and certifiable/verifiable components are and will be two important realities in the industrial domain and Ada can and must play a role there.
- The future of any programming language lies in the young generations of programmers (be they engineering or science graduates) and we, as a community, must do everything that is possible to make Ada appealing to them.

"Programming in Ada" by John Barnes, 2nd Edition, October 1983

Here are some interesting excerpts from John's book:

From the Foreword by Jean Ichbiah: "*Here is a major contradiction in any design work. On the one hand, one can only reach an harmonious integration of several features by immersing oneself into the logic of the existing parts; it is only in this way that one can achieve a perfect combination. On the other hand, this perception of perfection, and the implied acceptance of certain unconscious assumptions, will prevent further progress*."

From the preface to the first edition: "*This book is about Ada, the new and powerful programming language originally developed on behalf of the US Department of Defense, for use in embedded systems. Typical of such systems are those of process control, missile guidance or even the sequencing of a dishwasher. […] Although originally intended for embedded systems, it is a general purpose language and could, in time, supersede FORTRAN and even COBOL."*

From Section 1.1 "History" in the "Introduction": "*The story of Ada goes back to about 1974 when the United States Department of Defense realized that it was spending far too much on software. It carried out a detailed analysis of how its costs were distributed over the various application areas and discovered that over half of them were directly attributed to embedded systems.*

Further analysis was directed towards the programming languages in use in the various areas. It was discovered that COBOL was the universal standard for data processing and FORTRAN was a similar standard for scientific and engineering computation. Although these languages were not modern, the fact that they were uniformly applied in their respective areas meant that unnecessary and expensive duplication was avoided.

The situation with regard to embedded systems was however quite different. The number of languages in use was enormous. Not only did each of the three Armed Services have their own favorite high level languages, but they also used many assembly languages as well. Moreover, the high level languages had spawned variants. It seemed that successive contracts had encouraged the development of special versions aimed at different applications. The net result was that a lot of money was being spent on an unnecessary number of compilers. There were also all the additional costs of training and maintenance associated with a lack of standardization."

Ada 1.0

Ada 1.0 (also known as Ada 83) was a language that was way too powerful for its time. Compiling it for an 8 bit microcontroller was a daunting task and never really happened until recently when Ada 3.0 (Ada 2005) was made available on Atmel's AVR.

In the mid-80s Ada was ahead of what compiler and computer technology could do. In addition, the sociology of a fundamental element in collaborative software development was broken. Unlike its 1970s C predecessor, Ada 1.0 had a collaboration bottleneck: the order-of-compilation model and its centralized "program library file" assumption. This is one of those "*unconscious assumptions"* Jean Ichbiah talks about in the Foreword of John's 1983 book. From the Ada 1.0 reference manual:

> *10.4. The Program Library*
>
> *Compilers are required to enforce the language rules in the same manner for a program consisting of several compilation units (and subunits) as for a program submitted as a single compilation. Consequently, a library file containing information on the compilation units of the program library must be maintained by the compiler or compiling environment. This information may include symbol tables and other information pertaining to the order of previous compilations.*
>
> *A normal submission to the compiler consists of the compilation unit(s) and the library file. The latter is used for checks and is updated for each compilation unit successfully compiled.*

Today, programming languages foster the same sociological "gestalt" for collaborative software development: the sources, nothing-but the sources. Some readers may not understand what I am talking about as this is so obvious today. How could we have lived through a take-your-turn-to-compile-and-if-your-colleague-recompiles-you-may-have-to-recompile-too? This started from a noble intention and a critical Ada insight: type safety (type-checking) must operate on the overall program, across "compilation units" in Ada's parlance. This was completely novel at the time. What was unfortunate is the approach "à la Colbert[1]" that was taken in Ada 1.0. In fact, instead of sticking to a purely regulatory philosophy as it was subsequently done in Ada 2.0, Ada 1.0 strongly implied an

[1] Jean-Baptiste Colbert served as the Minister of Finances of France under King Louis XIV from 1665 to 1683. Colbert is referenced here because of his doctrine in which the State exerts a strong directive influence on the economy as opposed to a merely regulatory role. His doctrine is also known as "Colbertism" or "dirigism". If you are still wondering why I mentioned Colbert in the context of the Ada 1.0 centralized program library, consider the following. Jean Ichbiah, French born, was a pure product of "Colbertism". Ichbiah had graduated from an elite college created by Napoleon (who had very much been influenced by Colbert). The Ada 1.0 centralized program library assumption was the approach "à la Colbert" towards ensuring that an Ada program was type safe as a whole.

implementation approach to program-wide type safety. This implied approach was sociologically broken. This unfortunate oversight was fixed in the early 90s thanks to the intervention of Richard Stallman and Robert Dewar. Richard Stallman was adamant that Ada's type-safety-across-the-program rule did not create order of compilation dependencies (a sociological bottleneck). Robert Dewar, leveraging on the evolution of computers, found a way to achieve Ada's overall type safety rule with a pure source-based model by adjoining Ada type information to the object code generated by the compiler and introduced a tool to check the overall type safety at link time. Today we can have the cake and eat it too: C's freedom and Ada's type safety.

Apart from this, Ada 1.0 was visionary on the fundamental properties that a programming language for industrial systems had to possess. Decade after decade these properties pay back their dividends to users of Ada. My favorite property is the ability to communicate to other humans the key elements of what is being computed and have the compiler check their consistent use. This property has evolved and strengthened from Ada 1.0 (1983), to Ada 2.0 (1995), to Ada 3.0 (2005), to today's Ada 4.0 (2012).

General-Purpose Does Not Mean Universal

When Ada 1.0 came about, the US Department of Defense (DoD) was the biggest software contractor. Everything had to be programmed. No spreadsheets. To compute a simple linear regression we had to do it with pencil and paper and a hand calculator. The luckier ones had access to statistical packages on minicomputers or mainframes. Programming languages were the heart of the matter: they were the only way to get anything done with a large, bulky, slow, expensive, unconnected computer. Programming was the realm of mathematicians, physicists, chemists, engineers. A programming language, a dumb editor, and a compiler were all that was available then. No IDEs, no components, no frameworks.

When "Green" was designed, the attitude towards developing an embedded application was: let's do it from scratch using the best possible language, a language that would decrease the chance of writing "wrong" code, a language that would make abstractions clear, a language that would facilitate the reading, use, and re-use of software.

From the mid-50s onwards, computer scientists nurtured the dream that a sound, general-purpose, programming language would be the key to computing salvation. In the 60s general-purpose did not include embedded systems, which were just starting to emerge. IBM's PL/I effort was focused around IBM's concerns of the time and did not have embedded systems in mind. A general-purpose language targeting embedded systems was needed. In the wildest of dreams that language could be used from embedded real-time systems to accounting applications in the US DoD.

Ada substantiated the dream that a general-purpose programming language could be used universally to program all computing devices and applications. This idea strengthened throughout the 80s, 90s, and a portion of this century. After placing that hope on Ada 1.0, the community placed its bets on C++ and then Java, hoping to find the programming language that does it all. This never happened.

The message of this section is that we cannot expect a general-purpose programming language, be it Ada, C++, or Java, to be used universally. The spread and usage of a language is correlated with the

economics and evolution of the application domain which gave birth to that language. In this respect some languages such as Ada and C++ compete because their application domains overlap, while neither is a serious contender in web-centric applications.

Because there is no "universal language", systems are being written using several idioms and approaches. For this to be viable, languages should be able to talk to each other. Ada realized the importance of this in its 2.0 release and today Ada interfaces very well with subsystems written in other languages. In the end what matters is being a good play-mate: if you are, everyone wants to play with you.

Raising the Level of Abstraction: Model It

The myth of a universal programming language is slowly fading (I wrote universal not general-purpose). Universality comes at the cost of expressivity. Imagine doing math, physics, or engineering without mathematical notation. Imagine having to spell everything out in plain English. Sure we can do that. English is a general-purpose (and as close as we can get universal) language. But how expressive is it to talk math, physics, and engineering? Likewise, how can a team of scientists and engineers model a "phenomenon"? What language can the team use to devise that model? The key is in the meaning of the word ontology. From Wikipedia:

"*In computer science and information science, an ontology formally represents knowledge as a set of concepts within a domain, and the relationships between those concepts. It can be used to reason about the entities within that domain and may be used to describe the domain. In theory, an ontology is a "formal, explicit specification of a shared conceptualization*". *An ontology renders shared vocabulary and taxonomy which* ***models*** *a domain with the definition of objects and/or concepts and their properties and relations. Ontologies are the structural frameworks for organizing information and are used in artificial intelligence, the Semantic Web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture as a form of knowledge representation about the world or some part of it."*

Conventional programming languages such as FORTRAN, COBOL, C … have been used to create, express, maintain and evolve the ontology of the "phenomenon" that we want to model. To identify and communicate among humans the patterns of interactions between the elements of the ontology there has been a race to design the "best" high-level general-purpose programming language: Ada, C++, Java …. These languages have grown out of the Church–Turing computability thesis. The Church-Turing thesis tells us that to be processed mechanically a "phenomenon" must be modeled as computable mathematical functions. Although accurate, this view can limit the horizon of our possibilities. In fact, to create a computable model of the "phenomenon" we may want to use human-understandable languages that are not computable.

To raise the level of abstraction beyond general-purpose programming languages we could start from the ontology of the application domain and model the "phenomenon" using the language and symbols that are part of the application domain, i.e. its "natural" ontology. We could use that ontology to design, communicate, and convince others and ourselves that our model of the "phenomenon" is faithful. The last step would be to translate the model into a language that machines can understand. This last step could be done by humans, machines themselves, or a mixture of both. Welcome back domain-specific languages (DSL) also known as modeling languages

(and 4GL before that): UML, AADL, Simulink, Modelica … As a side note, when human intervention is required to go from the model to the machine the use of a high-level programming language such as Ada keeps translation and maintenance costs down.

When the ontology of our "phenomenon" is clearly defined and well established in the application domain, DSL are an attractive complement and even a substitute for general-purpose programming languages. In application domains without an obvious domain-specific language to express the ontology, high-level general-purpose programming languages are the best we have. In many cases we need a mixture of domain-specific and general-purpose programming languages. Note that a DSL that is mechanically translatable to machine language is nothing more than a high-level programming language that does away with generality in favor of expressivity for the application domain.

As for programming languages, modeling languages come in many shapes and forms and UML is no more a Universal Modeling Language than Esperanto is a universal natural language. Modeling business interactions and machine flight are fundamentally different activities and are so at the modeling level.

Given the multi-language nature of large systems today, a language that plays well with others and recognizes the existence of other languages (DSL and otherwise) has a definite advantage. Unsurprisingly, the message of this section is that in addition to being a good play-mate with other programming languages, Ada needs to meld well with DSL for the domains Ada has been designed for: industrial systems. See http://www.open-do.org/projects/p/ for a possible approach in this area.

Raising the Level of Abstraction: Brick by Brick

There is a constant race to raise the level of abstraction. In the previous section we have looked at DSL as a possible way to raise the level of abstraction. Another way is brick by brick. If there are libraries, components, frameworks with the desired functional and extra-functional (safety, security …) behavior and properties that can be acquired cost-effectively we may as well use them. This will reduce our time to delivery and it will increase the quality of our apps while allowing us to keep the costs under control: the programming language here becomes the cement between the bricks.

Apart from things like standard libraries and containers, components are domain-specific. We are unlikely to find high-quality components covering a large spectrum of application domains that can all be used effortlessly in a single programming language. There is an interesting circular dependency (a bootstrap problem if you prefer) between the application-domain of a component and the language it is written in. We are back at the generality vs. universality dilemma.

In today's systems of systems with many connected devices, security issues are a growing concern. In addition to Ada's orientation towards safety, Ada could play the role of a glue language for certifiable/provable components with the desired security properties. Ada 4.0 has certainly made this possible. In this respect it is fascinating to go back to the "Finale" of John's 1983 book:

"*Indeed, in the imagined future market for software components it is likely that packages of all sorts of generalities and performance will be available. We conclude by imagining a future conversation in our local software shop.*

Customer: *Could I have a look at the reader writer package you have in the window?*

Server: *Certainly sir. Would you be interested in this robust version – proof against abort? Or we have this slick version for trusty callers. Just arrived this week.*

Customer: *Well – it's for a cooperating system so the new one sounds good. How much is it?*

Server: *It's 250 Eurodollars but as it's new there is a special offer with it – a free copy of this random number generator and 10% off your next certification.*

Customer: *Great. It is validated?*

Server: *All our products conform to the highest standards sir. The parameter mechanism conforms to ES98263 and it has the usual multitasking certificate.*

Customer: *OK, I'll take it.*

Server: *Will you take it as is or shall I instantiate it for you?*

Customer: *As it is please. I prefer to do my own instantiation.*

John and the Ada community had sensed the growing role that components were to play in the coming decades. Because large industrial systems is the domain where Ada made its debut and showed its strengths, significant sets of component libraries have not emerged from Ada to date. This state-of-affairs is part of the socio-economics of the domains Ada has targeted. This is very different from the status of the Java context where a large set of business-oriented components and frameworks have appeared: in the last two decades business apps have dominated the software and service industry.

Still what are we to do with the "certification" or "provable properties" aspects of the components John talks about in his fictional dialog? That is an interesting alley where Ada 4.0 (with its assertions, pre/post conditions, and type invariants) should be leveraged on in the realm of provable/certifiable components in safety-critical and security-critical domains (for safety-critical domains DO-178C has created new opportunities for Ada 4.0 to lower the costs of certification). Efforts are ongoing in these areas at Kansas State University and other places such as in the Hi-Lite project undertaken by AdaCore, Altran-Praxis, Astrium Space Transportation, CEA-LIST, INRIA, and Thales Communications (see http://www.open-do.org/projects/hi-lite/). These efforts have their foundations in the SPARK language and its vision.

The objective of Hi-Lite is to combine testing and formal methods to lower the cost of verification. The enabler is an "executable annotation language", which allows writing contracts on types and subprograms for unit testing (because it is executable) and unit proof (because it has a logic interpretation). Ada 4.0 comes with such an executable annotation language in the form of type invariants, pre and post-conditions for subprograms, and a rich expression language (if-expressions, case-expressions, quantified-expressions, expression-functions). Annotations can be written by the user, inferred by static analysis, or generated with the code from a model. Being able to apply formal verification to parts of a program and testing to the rest of the program will be key to lowering the costs of verification.

The message of this section is that Ada 4.0 (and beyond) could be used to create certifiable components (general-purpose and domain-specific: containers, TCP/IP stack …) for the domains Ada has been designed for: industrial systems.

Ada as a Pivot Language in Requirements-Based Development

An interesting role that Ada 4.0 can play in the context of safety-critical software is to facilitate collaboration and communication within a team and lower the cost for the production of certification artifacts. For more on this read the Ada Europe 2012 paper: "Source Code as Key Artifact in Requirements-Based Development: The Case of Ada 2012" by Comar, Ruiz, and Moy.

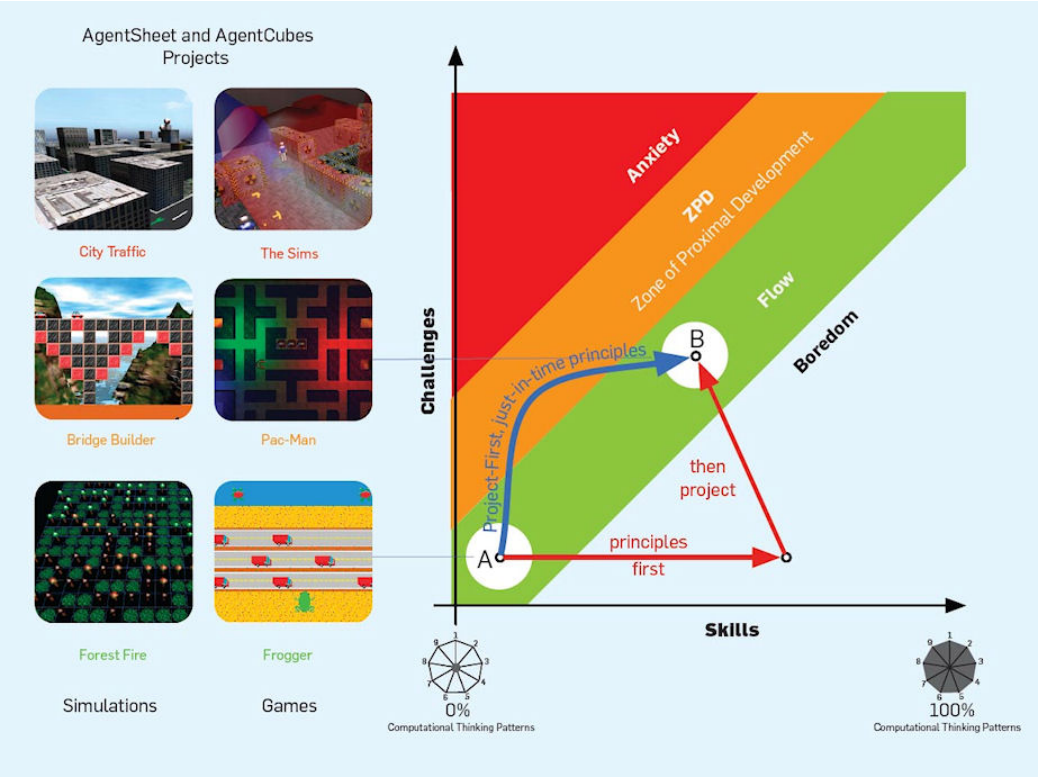
Ada for Digital Natives

Programming language experts focus on language purity, elegance, and completeness. What do our young programmers care about? And who are these programmers anyway? Is computer science a specialist-only discipline or is it a skill that most scientists and engineers need to master much like the ability to speak English? Today there is a broadening and blurring of engineering roles. Engineers are required to have a hand in multiple areas. As a result a programming language for industrial systems should be attractive to engineers as well as computer scientists.

How will these generations of new scientists and engineers learn programming? This decade presents a fantastic opportunity: web and tablet technologies allow to easily reach current and future programmers of industrial systems. To be vibrant the Ada experience must be readily available to younger generations. These digital natives are tech-savvy, plugged-in, and require quick and convenient feedback.

The design complexity of modern programming languages, be they Ada, C++, or Java, is significant. John's 1983 Edition of "Programming in Ada" was 367 pages; John's "Programing in Ada 2005" is 828 pages. For C++ it is terrifying: the book on "The C++ Standard Library: A Tutorial and Reference (2nd Edition)" is 1128 pages, and that is just the standard library. Most programmers don't want to be gurus. We have to develop short, interactive, design-elegant, self-contained, on-line tutorials that present subsets of Ada in which useful programs can be written. Not just one tutorial. A family of tutorials depending on the concepts each tutorial wants to convey.

Regarding the approach to tutorials and teaching (on-line and off-line), I recommend the reading of "Programming goes back to School" in the May 2012 edition of the Communications of the ACM. The article promotes a "project-first" approach instead of the more traditional "principles-first" methodology. This pedagogical style allows students to learn principles just-in-time which proves to be very beneficial from the viewpoint of captivating the audience (and I believe speed of learning for many). The following diagram from the article is particularly telling. The explanation of the diagram quoted from Webb, Repenning, and Koh is fascinating. The quote is an excerpt of their article: "Toward an emergent theory of broadening participation in computer science education" published in the ACM Special Interest Group on Computer Science Education Conference in 2012 (SIGCSE 2012).



The fundamental idea of the Project-first approach can be illustrated through what we call the Zones of Proximal Flow (ZPF) […]. Flow is an ideal condition for learning […]. The ZPD can be understood as an orchestration of participation in a rich set of carefully designed practices where forms of assistance and tool use are strategically employed. In the Zones of Proximal Flow diagram in Figure 1, the horizontal axis represents students' computational thinking (CT) skills and the vertical axis represents the level of the design challenge that would be intrinsic to a certain game or STEM simulation [STEM = science, technology, engineering, and mathematics]. *[…] As student acquisition of skills advances in response to the challenges, an ideal path in the flow region would progress from the origin to the upper right. Within this diagram, pedagogical approaches can now be described as instructional trajectories connecting a skill/challenge starting point (A) with destination point (B) in the Zones of Proximal Flow diagram. In many traditional CS education models, a principles-first approach would introduce students to a number of concepts such as AI search algorithms that may, or may not, be relevant for future projects. At some later stage, students receive the challenge of making a project such as a Pacman-like game.*

The acquisition of skills without the context of concrete challenges is not a bad pedagogical model, especially at the undergraduate CS level, but it runs the risk of seeming irrelevant, hence boring, for a broader audience of younger students if it does not go hand-in-hand with project based approaches. This assertion is consistent with the Flow model and with our own observations in classrooms. Instead of decoupling the acquisition of principles and the applications of these principles to a project, the project-first approach combines just-in-time CT skill acquisition with application to produce a tangible artifact.

In a nutshell: learn by doing, by example, by trial, by cut-paste-modify. Engage students in an exciting and feasible project. “I hear and I forget. I see and I remember. I do and I understand”, Confucius.

In the end scientists and engineers want to build things. Can we craft tutorials where students experience the exhilarating feeling of building a system in Ada? Simulations on tablets? Lego Mindstorms, train, UAV, or robotics projects?

If we look at this issue from a different angle, it is important that we help universities increase their focus on software development for industrial systems. In these contexts the use of Ada, as one of the tools in the Swiss army knife of the engineer, is appealing and is to be encouraged by providing to educators ready-made chunks of self-contained and student-engaging Ada training material.

The message of this section is that we should teach Ada providing a level of immediate engineering feedback and gratification. The entry point is the initial experience. If that experience is gratifying, if the student has learnt and built something by doing, Ada’s usage in industrial systems will broaden as new generations of professionals enter the workforce.

Concluding Thoughts

Ada has been created for industrial systems embedded in an airplane, train, satellite, helicopter, UAV, subway, automobile, radar, medical device, … for industrial systems controlling air traffic, power plants, railways, … as well as simulators for all of the above. Ada has been very successful in these areas. Ada’s strengths have shown that its use in other domains can bring significant advantages and rewards.

Today the software community is looking at the “Cloud”. The fact that these new celestial systems have taken over the financial and socio-dynamics of the computing industry and use languages and technologies that are intertwined with the history, evolution, and rise of the Web is not contradictory with the strengths and use of Ada for the domains it has been designed for: earthly industrial systems where software matters.

To keep playing an important role in future industrial systems, Ada’s level of abstraction in describing these systems should continue to rise, while attracting new generations of users. For this to happen, Ada language designers and tool providers should continue their cross-fertilization journey towards model-based and formal methods approaches. The Ada community must develop exciting Ada tutorials and should help teachers develop engaging courses on software development for industrial systems.

Thank You

Many thanks to Ed Schonberg, Ben Brosgol, Yannick Moy, Nicolas Setton, Ed Falis, Greg Gicca for their feedback on initial versions of this paper. Special thanks to John Barnes who decade after decade has helped the Ada community with his prolific and thought-provoking writings on Ada and SPARK.