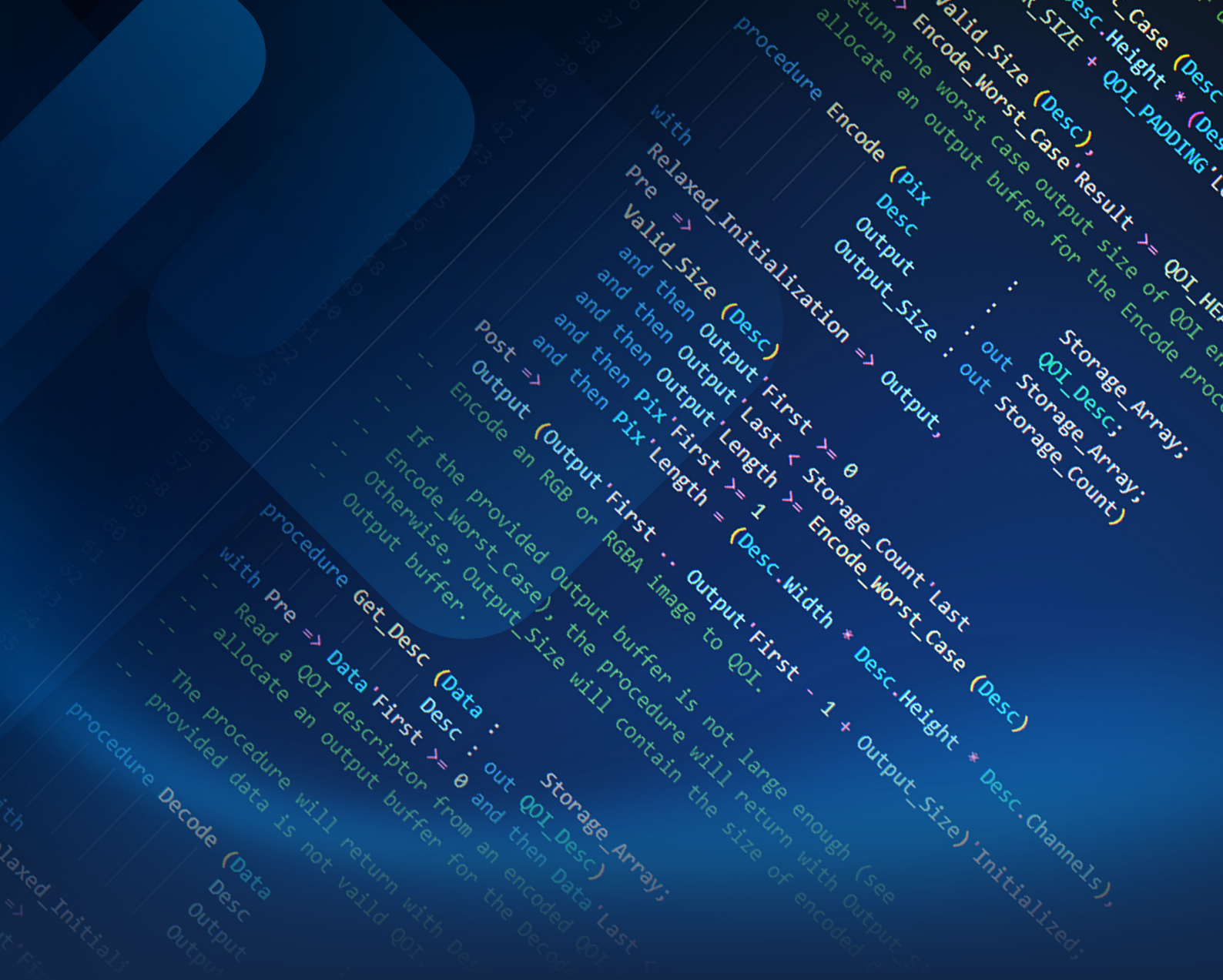


AdaCore TECH PAPER

Should I choose Ada, SPARK, or Rust over C/C++?

by Quentin Ochem

Should I choose Ada, SPARK, or Rust over C/C++?

Author: Quentin Ochem, Chief Product and Revenue Officer at AdaCore

Abstract

At AdaCore, we're in the business of supporting people who develop high-integrity software, in particular for embedded systems. In terms of programming languages, this means supporting the most commonly found candidates, which in 2024 include C/C++, Ada/SPARK, and Rust. If you've already made your decision, we will support you. However, in a number of situations, people ask us: "What should we do? What's the best out there?". While it's difficult to give a one-size-fits-all answer, there are some strategic elements to consider.

C/C++ - a risky default solution

In the embedded domain, you're more likely to look at C/C++ than anything else. This is the option "by default". A large portion of your software is likely to already be in C/C++. Your staff is trained in this language, tools, and processes are in place, and development costs are known and deterministic. Why change?

There is a growing body of evidence, both qualitative and quantitative, that shows that C/C++ is making the production of safe and secure software more difficult than it should be. Decades of research and investment have still not yielded a "safe C/C++" that is cost-effective, flexible, and reliable.

The good news is that today, depending on what you may want to do, you have better options.

Rust and Ada - improving traditional development processes

Teams that are looking at alternative programming languages have two options today: Ada and Rust. Both languages raise the bar in terms of safety and security compared to C/C++; each has unique strengths.

Consider ecosystems and communities. Rust has a vibrant community that has developed a huge amount of resources over a short period of time. However, its commercial ecosystem is still in the process of organizing itself. AdaCore has a role to play in this, but filling some of the gaps is going to take some time. In contrast, Ada has a smaller community - it has been growing over the years but much more slowly. However Ada has a complete and mature ecosystem both in terms of toolchain availability and certification documentation.

Or consider language capabilities. Rust pushes memory safety very far and provides a more flexible memory model than most programming languages today. Ada has an unmatched specification language that allows one to express and check software and hardware constraints at various levels.

These are just two examples. Below, we present a table that compares other aspects of Ada and Rust to help you select the language best suited to your needs.

SPARK - industrial-strength formal methods

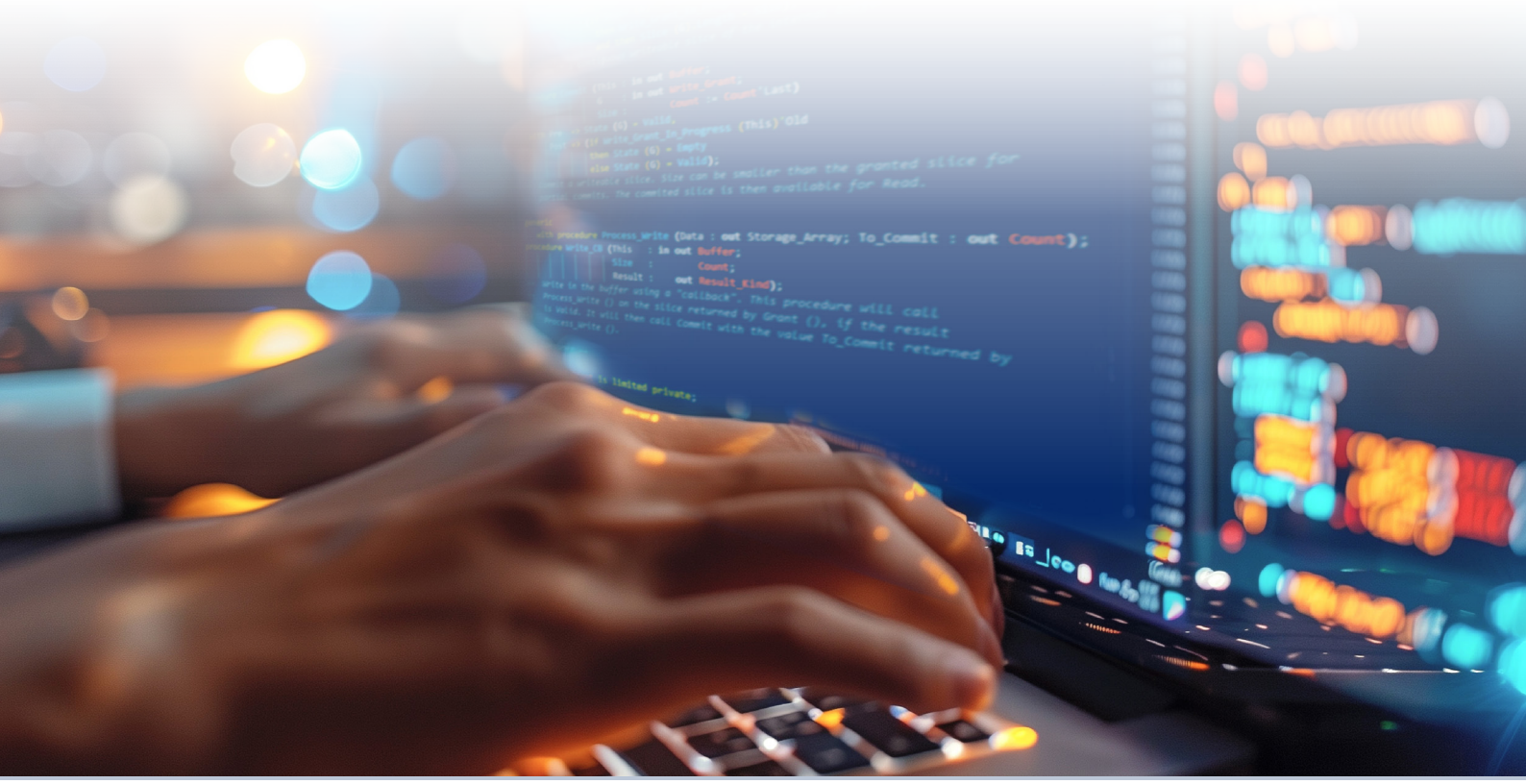
If you're prepared to look at alternative programming languages to avoid the costs and risks of C/C++, SPARK offers an opportunity to go much further than Ada or Rust. SPARK, which is based on Ada, offers industrial-strength formal methods: an opportunity for you to prove mathematically that your software is safe and secure. This paradigm shift in software development methodology offers significant cost savings for high-integrity software.

Using SPARK, you identify properties that can be formalized and proved true throughout an entire program - statically, i.e., at compile time. Ada and Rust offer some basic properties that are checked statically, such as the specification of hardware constraints in Ada or memory safety via borrow checking in Rust. SPARK takes these approaches to the limit, allowing the full range of Ada's specification language to be used to formalize properties that are proved, automatically. The result is comprehensive proven properties across a whole application.

The first set of properties SPARK demonstrates pertains to vulnerabilities inherent to the programming language itself. For example, there's no guarantee that the index used to access an array element is within range. While many programming languages guarantee that an out of range access will yield an exception at run-time, SPARK will prove that there's no possible out of range index statically, at compile-time.

SPARK also allows expression of custom properties and verifies that the code complies with them in all possible cases. These properties can range from simple cases (are the callees to mutexes balanced? Is the array after this sort call really sorted?) to more complex relationships between function input and output.

Ultimately, using SPARK allows you to eliminate various checkers (think MISRA-C checkers). By verifying properties with 100% certainty via mathematical proof, you can eliminate many unit-level tests. This yields direct cost savings and ensures an overall higher level of integrity.



So what's the best choice?

Choosing between Ada, Rust, and SPARK is a complex discussion. The key is, what is the team looking at achieving, and what is the potential appetite for change? The chart below provides some elements that can serve as the basis of a discussion. Different companies may allocate different weights to different elements. This is the way we view things at AdaCore and for our customers:

	Ada		SPARK
Community	Small	Large	Small
Toolchain Embedded Ecosystem	Mature	In development	Mature
Certification	Off-the-shelf	In development	Off-the-shelf
Libraries available	Limited	Large	Limited
Programming paradigm	Imperative system-level	Imperative system-level	Imperative system-level
Mitigation of programming errors	Yes	Yes	Yes
Strong Typing	Yes	Limited	Yes
Data constraints, hardware / software data consistency	Yes	No	Yes
Guaranteed absence of run-time errors	Run-time checks	Run-time checks	Static, via Proof
Contract language (pre- post- conditions, invariants, predicates...)	Yes, checked at run-time	No	Yes, checked statically, via Proof
Memory safety	Pointer avoidance Accessibility checks Dynamic checks	Borrow checker Lifetimes	Borrow checker Pointer avoidance Accessibility checks
Cost of adoption	Language change	Language change	Methodology change
Expected benefits	Mitigation of programming errors Constraint checks	Mitigation of programming errors Memory safety	Mitigation of programming errors Memory Safety Guarantee of absence of run-time errors Guarantee of formal properties Guarantee of constraint checks Testing reduction

Community

One of the big strengths of the Rust programming language is its large and vibrant community. It's easy to find resources on the language and people who have a true passion for it. Tools and libraries constantly emerge from the hobbyist scene, which can be easily leveraged when starting projects.

The Ada and SPARK communities are comparatively smaller. While excellent resources exist for learning Ada and SPARK, there are fewer community-provided tools and libraries. However, they've endured the test of time for over four decades and are composed of dedicated individuals. Today, they are also infused with new members and on the rise.

Overall, both communities benefit (albeit at different intensities) from the same underlying force: an increase in the demands for technology that provides safety and security.

Embedded Toolchain Ecosystem

Ada and SPARK have a very mature ecosystem, in particular in the embedded space. Besides the familiar Linux and Windows environment, compilers exist for many real-time operating systems and hardware architecture, and tools cover the whole range of needs from static analysis to dynamic analysis. All of this comes with industrial support. The tools also come with long term support, which means that you can select one version and stay supported for years or decades.

Rust today is becoming established in the native / server environments. Embedded RTOSes and architecture support are in the process of being developed. It's a lengthy process, due to the number of environments to support and their accessibility and specificities. Commercial support is also in development - some environments are already available off-the-shelf while others are being put together (including by AdaCore). The question of long-term support is also important, as some providers tend to update their toolchain very frequently, which may make long-lasting projects difficult to support. AdaCore addresses this specific issue with our products.

Undoubtedly, both languages will converge over time as far as toolchain goes - the choice criterion is more whether it's important to have all of the answers today or if the adopting team can wait.

Certification

The situation on certification mimics the situation in toolchain support. Ada and SPARK, having been around for quite some time, have qualification and certification evidence for a wide variety of standards, notably the most common ones in the embedded world such as avionics (DO-178), automotive (ISO 26262), railway (EN-50128), space (ECSS-E-ST-40C and ECSS-Q-ST-80C) and others.

Rust is a much younger technology, so it hasn't had the same amount of time for these to emerge. We're starting to see some automotive ISO-26262 evidence, which is currently limited to some environments and for some subsets of the Rust toolchain.

Libraries

One of the very strong attributes of the Rust programming language is the large number of libraries available through its cargo package manager. Pretty much anything that you can think of is covered one way or another. However, a number of these libraries are developed by hobbyists and many of the most popular libraries have yet to reach version 1.0.

Ada and SPARK have fewer off-the-shelf libraries available. The Alire package manager started in 2021, counts about 400 packages at the date of writing. This lack of native libraries is usually offset by binding from Ada and SPARK to pre-existing C or C++ libraries.

However, for both languages, constraints may come from regulatory or certification requirements, as publicly available libraries are usually not suitable for safety- or security-certified embedded development.

Programming paradigm

There's no fundamental underlying difference in terms of the programming paradigms between Ada, SPARK, and Rust. They all are imperative modular languages, providing variations around the concepts of object orientation and other similar capabilities (as opposed to e.g. functional languages). They're all statically compiled directly into machine code (as opposed to, e.g., interpreted languages).

Mitigation of programming errors

All three languages provide various mechanisms for avoiding or mitigating programming errors. For example, all three languages provide arrays as first-class citizens, containing boundaries and allowing for dynamic index checking. They also provide various ways to avoid uninitialized variables, data races, null pointer dereferencing, etc.

Strong Typing

Strong typing ensures that you can determine at compile time the specific type of an object and that you can check the integrity of its values throughout its usage. C is notably weakly typed: while variables are typed, implicit conversions allow you to mix numbers with different representations without the developer's oversight (for example, when adding integers and floats). This may result in various issues such as overflow, underflow or rounding errors. Treating arrays like pointers is another example of an issue that arises from weak typing.

Rust's typing is stronger in this regard. Different types can't be mixed together without explicit conversion, and arrays are first-class citizens. This allows programmers to avoid a number of common programming mistakes.

Ada and SPARK go further. Types become fundamental elements of the software design. They are named, associated with a number of properties, and checked for consistency statically and dynamically. For example, you can declare a float to be a distance in miles and another float to be a percentage and then make sure that, without explicit conversion, there's no risk of mixing them up by mistake. Similarly, a latitude and longitude floating-point type could be defined and the type system would prevent mixing them up in subprogram calls or in arithmetic. Strong typing allows us to detect not only coding errors but also design inconsistencies.

Data constraints, hardware/software data consistency

The Ada and SPARK type system allows programmers to associate a number of properties, such as data ranges, representation constraints, or validity predicates with types. For example, a percentage type can be constrained to be between 0 and 100; a latitude could be constrained to be between -90 and 90 degrees; and a longitude could be constrained to be between -180 and 180 degrees. These constraints can be checked statically and dynamically, depending on the verification strategy. Data structures can be specified at the bit level in memory with specific endianness, avoiding common mistakes related to bitwise operations. Consistency of specification representation is checked statically (e.g., that there is no overlap between fields of structures, the number of bits specified for a type match is enough for the values that it can take, and the precision for a floating point value can be implemented in hardware).

Rust doesn't natively provide these capabilities. When needed, these capabilities could be implemented through more traditional design patterns, such as structures with appropriate methods.

Guaranteed absence of run-time errors

Run-time errors refer to errors that can be detected by checks and assertions while a program is running. For example, checking that an index used to access an array element is valid. Ada and Rust both provide run-time checks that verify the validity of data and would either raise an exception or issue a panic in case of a failure. While they incur a small footprint in code of code size and performance, they protect the code against much more adversarial vulnerabilities such as buffer overflows.

SPARK formally proves absence of run-time errors, at compile time. For example, SPARK statically checks that there are no code paths that can bring values out of bounds. This has the advantage of not only avoiding performance penalties, but also ensuring proper execution of the code against potential exceptions/panic - at the cost of more work on the programmer side to express additional constraints, assertions, and contracts (see later sections).

Contract language (pre- post- conditions, invariants, predicates...)

Ada and SPARK are unique in that they allow the description of contracts around software entities, notably types and functions. This allows constraints and dynamic behavior expectations to be encoded as part of the specification and checked for validity across the entire application.

Ada translates these contracts into dynamic checks that are verified at runtime. While this has a code size and performance footprint, it helps during testing, debugging, and integration phases and can be stripped out (fully or partially) at compilation time before deployment. Failure in contracts will typically be translated into exceptions.

SPARK allows formal, mathematical proof that contracts are always satisfied by the application, ensuring that, regardless of what value is manipulated, specified constraints and functional requirements are met. In this case, there's no need to compile these contracts into dynamic checks, obviating the space and performance penalties.

SPARK thus offers a paradigm shift for the programmer, who becomes much more verification-oriented, which is extremely valuable in high-integrity environments.

There's no specification language in Rust that can be leveraged to implement these capabilities today. They can be emulated to some extent through defensive code and assertion for the purpose of dynamic checking. However, there's no technology today that allows formal proof of these kinds of properties like in SPARK.

Memory safety

One of Rust's most powerful capabilities is its ability to avoid memory errors through its ownership model of memory. This eliminates the most significant source of security vulnerabilities in software, simply by adopting rust, following the ownership model, and satisfying the borrow checker.

Ada, in its latest definition (2022), offers a pointer avoidance strategy that mitigates the risk of memory corruption. However, some programming patterns require the use of pointers, and in the absence of, an explicit ownership model borrow checker, memory issues are possible.

SPARK, on the other hand, adds a strong ownership model and borrow checker, providing the same level of guarantees as Rust.

Cost of adoption

The cost of adopting Ada and Rust is similar. In both cases, programmers must learn a new language and teams must deploy a new toolchain. While far from insignificant, programmers keep their overall programming processes more or less the same.

The cost of adopting SPARK is probably higher. At the outset, the toolchain and language considerations are pretty close to those of Ada. However, really adopting SPARK means adopting a different way of programming. To be effective, formal verification should be integrated into the development process and change the way software is designed, as well as bringing a number of verification steps earlier in the process. This is not an all-or-nothing decision; depending on the expected trade-off, more or less emphasis can be put on the properties to prove. The benefits can however be significant.

Expected benefits

Benefits depend on context - here we're looking at languages in the context of high-integrity development.

At the coarse-grained level, the benefit of adopting Ada or Rust should be pretty similar. Both languages greatly reduce the odds of programming errors. Both languages address memory safety, albeit in different ways. When applicable, the Rust memory model will go further than the current Ada pointer-avoidance strategy, but Ada's strong specification and typing allow consistency checking in places where Rust can't yet. Literature on Ada highlights up to 40% development-cost savings compared to C. This does not account for the reduction of residual bugs that are less likely to make it into production.

Because SPARK delivers industrial-strength formal methods, it has the potential to exceed the benefits of Ada or Rust significantly. While a number of verification activities will be front-loaded during development, some testing and checking activities will be eliminated - those that would check constraints and properties expressed in SPARK. Deviations against specified behavior are not mitigated, they are eliminated from production. In the context of applications where defect cost is high and whose lifetime is counted in years or decades, this can yield significant gains, beyond the gains of a simple language change.

About Quentin Ochem

Quentin Ochem is the Chief Product and Revenue Officer at AdaCore, overseeing marketing, sales, and product management. His involvement with AdaCore began in 2002 during his school years, officially joining in 2005 to work on IDE and cross-language bindings. Quentin has a background in software engineering, particularly in high-integrity domains like avionics and defense. His roles expanded to include training and technical sales, leading him to build the technical sales department and global product management in the US. In 2021, he stepped into his current role, steering the company's strategic initiatives.

Quentin holds a master's degree in Computer Engineering from Polytech Marseille, awarded in 2005.

AdaCore

adacore.com