AdaCore | Build Software that Matters

Case Study

Zenseact Chooses SPARK for Automotive Safety

adacore.com

Executive Summary

Zenseact is advancing road safety through the development of highly reliable Autonomous Driving (AD) and Advanced Driver Assistance Systems (ADAS). The company is committed to delivering software that enables safe, unsupervised automated driving.

To achieve this, Zenseact has adopted a rigorous approach to software development grounded in the use of high-integrity languages and formal verification. SPARK was selected after a detailed assessment of multiple formal-methods technologies. SPARK's ability to eliminate undefined behaviour and provide mathematical guarantees of correctness offers a practical and certifiable foundation for software aligned with ISO 26262 - particularly for ASIL C and D components, where formal verification is strongly recommended.

Zenseact's development strategy applies strict criticality classification across its systems. Components with the highest ASIL requirements are implemented in SPARK, enabling the organisation to verify functional correctness and absence of run-time errors across an entire subsystem. Lower-criticality components may be implemented in C++, but only where the risk profile permits. This structured approach ensures that verification efforts are directed where they have the greatest impact on safety.

The adoption of SPARK has been supported by targeted onboarding, internal knowledge sharing, and ongoing collaboration with AdaCore. Developers without prior experience in Ada and SPARK have adapted quickly to these languages. At the same time, the discipline of writing contracts and proofs has become a valued engineering approach that attracts and retains skilled practitioners. As the SPARK team grows, Zenseact is becoming increasingly capable of delivering product features that meet stringent safety requirements with confidence.

Building the safest cars in the world

zenseact

Zenseact is a safety-focused software company dedicated to building the safest autonomous driving software in the world.

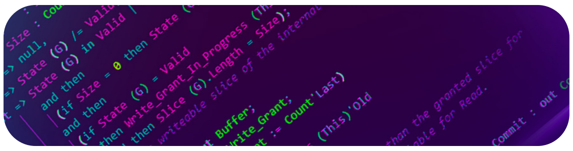
Zenseact is a safety-focused software company dedicated to building the safest autonomous driving software in the world.

When looking at the most common causes of car crashes, a pattern can be seen: the driver represents a single component at fault in all of these fatalities. Human reaction times are several orders of magnitude slower than those of a suitably designed computer system. While people's ability to assess risks in nominal settings can be debated, this judgment is often inhibited by intoxication, drowsiness, and distractions such as cell phones and entertainment systems. Computer systems do not suffer from these faults, which is why Zenseact's strategy is to push towards unsupervised automated driving – in other words, full self-driving. Zenseact believes that the more unsupervised self-driving that can be offered, the safer the car and the happier the customer.[1]

[1] Zenseact, Towards Zero. Faster – A Strategy Outline, https://zenseact.com/ (accessed 4 December 2025).

Developing safe software

Ada is a modern programming language designed for large, long-lived applications – and embedded systems in particular – where safety and security are essential.



Ada was developed in the late 1970s and early 1980s, when the programming language landscape was rapidly shifting. Methodological advances - structured programming, strong typing, encapsulation, object orientation, safe concurrency, genericity - allowed software developers to write code that was readable and reliable, expressed at a high level. The Ada design successfully integrated these advances into a coherent syntactic and semantic whole. At the same time, Ada allowed programmers to write low-level code when the requirements called for it, and ensured that compilers could generate efficient object code on target environments ranging from bare metal to mainframe processors and operating systems.

That era (late 1970s) was also witnessing ongoing research into formal proofs of program correctness. Although Ada is a full-featured general-purpose language, it includes features that support formal verification (such as an annex on safety and security, and a compiler directive that allows excluding features that might complicate analysis). The attention paid to high assurance in Ada's design made it an ideal candidate as the basis for a safe and secure subset; in fact, the SPARK language used Ada as its foundation.

SPARK: a tried and tested solution

SPARK is a programming language derived from Ada, designed to eliminate undefined behavior and enable formal verification. By restricting certain language features, SPARK guarantees determinism and analyzability while enriching Ada with precise semantics and annotations that allow developers to prove the absence of defects and verify functional correctness. Writing in SPARK means working in a full-featured imperative language that compiles directly to the hardware and delivers formally proven, high-performance code that can be trusted.

Investigating Formal Methods

In 2018, Zenseact investigated several formal verification methods. Following this exploration, SPARK was chosen as the best fit for the certification requirements.

Tamatea McGlinn, one of the developers involved with the adoption of SPARK at Zenseact for high-criticality components, explains the decision:

"I realised that high-criticality software in the automotive field faces the same challenges as that in other fields, such as military, aviation, spaceflight, and train-control, which already use Ada or SPARK. So why not use it in our cars, too?"

"I realised that high-criticality software in the automotive field faces the same challenges as that in other fields, such as military, aviation, spaceflight, and train-control, which already use Ada or SPARK. So why not use it in our cars, too?"

Since ISO 26262 recommends that ASIL C and D code be formally verified, SPARK was a natural choice. Moreover, unlike the other formal methods investigated, the SPARK tools are already qualified to the highest level required by ISO 26262.

"We needed a way of working that satisfies ISO 26262's recommendations for components with ASIL C and D requirements. Previously, we had thought this could be done from C++, but the complexity of the language makes this impractical."

Determining criticality

Not all code written at Zenseact is written in SPARK. For components deemed less critical (ASIL A and B) according to safety analysis, Zenseact continues to use C++. This approach is practical, since the majority of Zenseact's programmers are familiar with C++.

Zenseact uses SPARK for components that are deemed most critical (ASIL D). Although Ada has strong capabilities for linking with existing C++ code, Zenseact instead rewrites the component in SPARK with the aim of verifying that the entire component meets its safety requirements. Mixing SPARK and C++ would mean that the C++ elements would still require verification.

"We break up the whole system into subsystems and assign an Automotive Safety Integrity Level (ASIL) to the requirements of each, based on the potential severity and probability of exposure to risk. This criticality classification dictates our verification strategy: we prioritize the 'highly recommended' methods listed in the ISO-26262 standard for the specific ASIL, ensuring that any deviation is backed by a strong, documented rationale. Each subsystem has its own requirements and is further broken down into components, which are also paired 1-to-1 with requirements. Each requirement then corresponds to one contract, which is fed into the SPARK proof system to verify they hold and continue to hold, even as the code is changed to accommodate new requirements. Ultimately, the goal is not just to tick boxes in a table, but to rigorously fulfill the safety objectives defined for that specific phase of development."

Using Unit Tests

"For our lower criticality components, we do use unit testing at Zenseact in conjunction with other forms of testing. But for those components that we've determined are most critical to the system's safety, we have introduced formal verification."

In order to use software to assist or even take over the driving responsibility safely, Zenseact ensures the development process is sound all the way from requirements down to production code. Most software companies do this by constructing a large number of unit tests. This means setting up scenarios, each consisting of specific descriptions of exactly what will happen in what order, and expected outputs from the component.

For Example:

Component: Distance-evaluation function

Input: SensorReading = 0.10 m

Threshold: 0.50 m

Expected output: true
(indicating that the distance is unsafe)

This approach is time-tested, but it is also expensive, even when testing is automated. Reducing the amount of unit testing by using formal methods saves time and developer fatigue in the development process. Some unit tests will be replaced by formal proof, while some unit tests will be retained (for those requirements that cannot be or are too expensive to formalize).

Figure 1: Example of code

```
procedure Is_Accelerator_Pressed
 (Sample             : in Boolean;
  Accelerator_Pressed : out Boolean)
is
begin
   if Sample then
      if Current_Accelerator_Samples < Accelerator_Samples'Last then
         Current_Accelerator_Samples := Current_Accelerator_Samples + 1;
      end if;
   else
      Current_Accelerator_Samples := Accelerator_Samples'First;
   end if;

   Accelerator_Pressed :=
      Current_Accelerator_Samples = Accelerator_Samples'Last;
end Is_Accelerator_Pressed;
```

Using SPARK to Remove Undefined Behavior

SPARK proves that key properties hold throughout the code, providing strong guarantees that would be difficult to achieve through code review or unit testing alone. For example, the following procedure (from an earlier version of a component called the Ride Mode Manager) determines whether the accelerator pedal should be considered to be pressed based on consecutive input samples as in Figure 1 above.

This procedure counts consecutive "pressed" samples, prevents counter overflow, and sets Accelerator_Pressed to true only when a stable sequence of positive samples is observed.

SPARK ensures that:

- For all if-statements, there are possible input values that lead to the true condition, and also those that lead to the false condition
- No assignments are being done to input values (Sample:= 5; would be a mistake)
- All output values are being assigned to (Accelerator_Pressed)
- There are no implicit conversions
- There are no out-of-bounds array accesses, null pointers being dereferenced, or uninitialized values being used
- Side-effects are disallowed where they can cause undefined behaviour in C / C++

Without these checks, which are defined in the Ada language and proved statically by SPARK, Zenseact would have to rely solely on code review and/or testing to demonstrate the absence of undefined behavior, both of which are more error-prone, slower, and much more expensive. For more complex examples, dealing with calculations and especially floating point values, the list of subtleties the code reviewer must be aware of only grows. As a result, reviewers tend to accept code that has hidden vulnerabilities waiting to be triggered. Unit testing can only cover cases that the writer of the tests knows about. If the author has never heard of floating point underflow, chances are there will be no test for such conditions in their code, and again, the code reviewer may not correct this mistake. SPARK provides automated, static verification, reducing the risk of hidden vulnerabilities and ensuring that the code behaves as intended.

The SPARK adoption process

Companies can be hesitant about adopting a new language for their teams. Zenseact built a new team and shared its onboarding insights,

“From the beginning, the problem was always that it was hard to find developers with prior experience in Ada, but I have found that C++ developers only need to learn a few syntax changes, which seem large but actually are quite trivial. Competent programmers learn SPARK very rapidly and are immediately effective. The similarity to programming in other languages is a big advantage compared to model-based development methodologies, which require a very different way of working.”

```
procedure Grant (This : in out Framed_Buffer;
                 G    : in out Write_Grant;
                 Size :        Framed_Count)
is
   Hdr_Size : constant Count := Header_Size (Size);
begin
   if Size = 0 then
      BBqueue.Buffers.Grant (This.Buffer, G.Grant, Size);
      G.Header_Size := 0;
      return;
   end if;

   --  Save the worst case header size
   G.Header_Size := Hdr_Size;

   --  Request Size + worst case header size
   BBqueue.Buffers.Grant (This.Buffer, G.Grant, Size + H

   if State (G) = Valid then
      pragma Assert (G.Grant.Slice.Length = Size + Hd

      --  Change the slice to skip the header
      G.Grant.Slice.Length := G.Grant.Slice.Length
      G.Grant.Slice.Addr :=
        To_Address (To_Integer (G.Grant.Slice.Addr
   else
      --  Grant failed, no header
      er_Size := 0;
```

“Writing code contracts is difficult. It requires a level of reasoning that takes a lot of practice to master. On the other hand, we’ve found that within the existing teams, proofs present a unique puzzle that draws engineers all by itself. At Zenseact, we have a very fluid and flexible structure that allows engineers to find the problems they find most interesting, and an open environment that facilitates internal teaching and collaboration across disparate feature sets. I believe this is essential in fostering in-house SPARK expertise. The process of teaching and developing the necessary knowledge of working with proofs should not be underestimated; it is quite significant.”

“AdaCore has been very good in responding quickly and handling our issues, which have mostly just been in finding and understanding the correct documentation for the tools they provide.

The project is going well; we have a growing team of SPARK engineers, and the significant benefit is that we can now release product features with high criticality requirements.”

“AdaCore has been very good in responding quickly and handling our issues, which have mostly just been in finding and understanding the correct documentation for the tools they provide.

The project is going well; we have a growing team of SPARK engineers, and the significant benefit is that we can now release product features with high criticality requirements.”

Conclusion

Through the use of full program contracts and formal verification thereof using SPARK tools in the CI system, Zenseact is able to push the boundaries of software development. As the automotive industry continues its transition towards increasingly autonomous, software-defined vehicles, Zenseact’s use of SPARK shows how formal verification can be applied pragmatically to meet the highest safety requirements. By combining proven technology with disciplined engineering practice, Zenseact is building a foundation of trust that supports its ambition to deliver the safest cars in the world.

AdaCore | Build Software that Matters

adacore.com