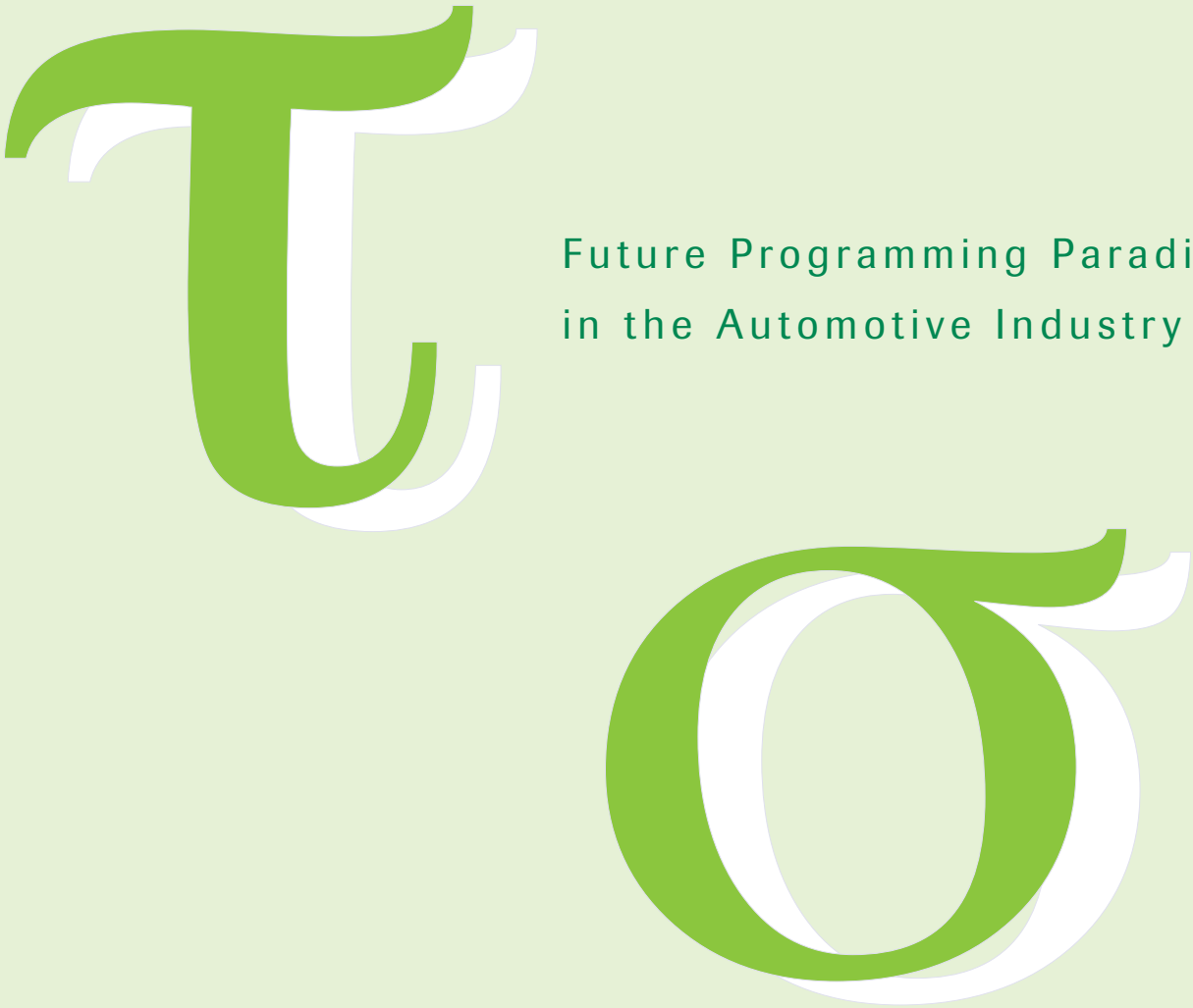
FORSCHUNGSVEREINIGUNG AUTOMOBILTECHNIK E.V.

FAT-SCHRIFTENREIHE

FAT 287



Future Programming Paradigms in the Automotive Industry

VDA Verband der Automobilindustrie

Future Programming Paradigms in the Automotive Industry

Forschungsstelle
fortiss GmbH

Autoren
Zaur Molotnikov
Konstantin Schorp
Vincent Aravantinos
Bernhard Schätz

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e. V. (FAT) gefördert.

Abstract

In this report, we deliver the results of a study on future programming language traits for the automotive industry. We briefly review the current status of programming languages used in automotive and come to the conclusion that an update is needed. This is due to the heavy and various requirements put on automotive software as well as its complexity. We analyze and compare a number of existing languages based on their features and propose some new traits not available in the current wide-spread programming languages, and tools for them in automotive. One conclusion of this study is that “there is no silver bullet”: There is not one single programming language able to directly and adequately cover all the needs of the automotive industry. Also, we do not advise to envisage a single language for the automotive industry to cover all aspects of software development in the future. For the future, we suggest research on layers of subdomain-specific programming languages and their interoperability.

Zusammenfassung

Software wird im Auto immer wichtiger. In Zukunft werden Autos immer mehr Konnektivität, Interaktion mit anderen Systemen und langfristig vollständig autonome Fahrfunktionen bieten. Das befördert die steigende Relevanz von Software in der Automobilindustrie und bringt neue Anforderungen. Kürzere Entwicklungszyklen, neue Sicherheitsanforderungen, Parallelität und Echtzeitfähigkeit sind Beispiele. Um den neuen Herausforderungen begegnen zu können, müssen die Softwareentwickler mit neuen Programmierparadigmen unterstützt werden. In dieser Studie analysieren und vergleichen wir etablierte und neue Programmiersprachen basierend auf ihren zugrundeliegenden Paradigmen und stellen neue für die Automotive-Domäne hilfreiche Merkmale vor, die in etablierten Programmiersprachen noch nicht eingesetzt werden. Ein Hauptergebnis dieser Studie ist, dass keine einzelne allgemein verwendbare Programmiersprache die Anforderungen aller Teildomänen der Automobilindustrie berücksichtigen kann. Um den Anforderungen moderner domänenübergreifender Funktionalität Rechnung tragen zu können, empfehlen wir weiterführende Forschungsaktivitäten zu an die Teildomänen angepassten Programmiersprachen und Mechanismen zu deren Interoperabilität.

Table of Contents

1 Introduction and Overview

1.1 Introduction

The importance of software in the modern automotive industry is paramount and growing. It is one of the primary innovation drivers today. Still, complex computerized automotive systems are built using mostly either non-state-of-the-art or narrowly specialized programming languages and tools which are not able to support all the combined needs of the automotive software development.

We take as an example a combination of C and Simulink. There are many code quality criteria and desired program features which are hard to achieve when using C and Simulink. We give a few examples:

Handling complex data structures in a way higher-order languages do is impossible, as C does not directly support polymorphic types, and Simulink is a control-flow language which can't manipulate such structures. Ensuring high program correctness is also more complicated in these formalisms. It is hard to write highly concurrent programs as both languages do not support concurrency natively. In C, expressing concurrency requires manual invocation of threads and their error-prone synchronization. Writing generic parameterized code is also impossible with C and Simulink as these languages lack mechanisms such as templates or interfaces. Implementing programs for safety-relevant domains is also hard in C and Simulink as there are no explicit constructs for safety, a weak type system and unrestricted low-level operations. Writing programs with real-time requirements is not trivial either. Time is not mentioned in C, and execution time analysis of C programs is complex. In combination, C and Simulink are even harder to analyze. Building well-structured object-oriented or functional-style programs is impossible too, as these paradigms are not supported by either C or Simulink.

Finally, handling modes and exceptions effectively appears to be extremely hard. Standardized error handling and mode-oriented programming are not supported in C and Simulink. Even encapsulation, the first step towards crystalizing modes out, is impossible with Simulink and C to the necessary degree.

In this report, we target the 5- to 10-years-remote future of programming languages for automotive applications. We investigate the progress in language design which has happened in the last years[1], but was not reflected in automotive software development.

[1] As C is the most popular language in automotive, we could as well convey the relevant features of languages starting from the C time – they are certainly not new in computer science, but are new for the automotive domain!

One conclusion of this study is that "there is no silver bullet":

There is not one single programming language able to directly and adequately cover all the needs of the automotive industry. Also, the authors do not advise to envisage a single language for the automotive industry to cover all aspects of software development in the future. Instead, languages, probably based on a common core, are to be tailored on per-subdomain basis and interaction is to be researched.

The reason for this is that automotive software development consists of a number of *sub-domains*:

- Real-time control
- Data-processing and infotainment
- Driver assistance
- Prototype building and research

These domains put their own strict and *conflicting* requirements on a programming language best suitable for a given subdomain.

This study is based on the notion of so-called programming paradigms. In this report, under programming paradigms we have agreed to understand programming language features only, we also call them "traits" sometimes. As they cannot exist separately we also discuss tools and processes, as well as feature interactions.

1.2 Document Structure

In Chapter 2 we build a system to compare programming languages. We start our study by listing language features in Chapter 2.1 and grouping them to form language *characteristics* – means to informally compare programming languages in Chapter 2.3.

The characteristics comparison charts in Chapter 2.3 support the intuition for our main "no silver bullet" thesis – although specialized languages exist that are topping some of the characteristics, they appear totally impractical when used in a foreign setting.

We briefly review the state of software development in automotive (Chapter 3) other industries (Chapter 4) in the context of programming languages. The "no silver bullet" thesis turns out not to be unique for the automotive setting.

Other industries give us inspiration for the automotive software development future as well. Some of the other industries have changed technologies rapidly, at a much faster pace than automotive. We suggest learning from their experience when planning for new languages in the automotive domain.

We held an on-line survey (Chapter 5) trying to identify the most important and problematic aspects of the automotive industry: *focus topics*. Professionals from the automotive industry

provide their feedback on the languages used currently and on the future language traits desired by them, depending on focus topics.

Following is the discussion on the new traits (Chapter 6), which are not there in the automotive programming landscape now, but could be of great use, to our opinion, for the modern and future automotive software development. The language features are presented in packages corresponding to focus topics. On side line of this discussion we highlight the need for modern tools supporting the language features and developers using them as well as for new processes. Next, in Chapter 6.2 language features are brought together to show their compatibility or incompatibilities. Incompatibilities depend on the implementation of the language often. This is the practical outcome of the “no silver bullet” statement. Once incompatibilities are sorted out we proceed mapping language characteristics to automotive subdomains in 6.3. Tracking back to the definitions of characteristics, one gets the language features necessary for a given automotive subdomain. Preparing a selected combination with the new language features offered in this report and capturing the output in a tool shall provide a basis for a next generation subdomain programming language. Migration to such a language is outlined in 6.4.

Chapter 7 wraps this report up highlighting the results and the open questions. In 7.3 we suggest a way to go ahead with the research on the automotive programming future. New research efforts should focus on interaction of a number of programming languages with different traits, each serving the needs of a single subdomain. A potential follow-up research activity is sketched: It discusses the architectural composition of multiple subdomains using a different programming language each.

Appendix I discusses the tool-prototype shipped with this report, demonstrating the possibility to build, with a relatively low effort, new languages as mixtures of existing traits as well as tooling for these languages.

Appendix II tells how to use the survey raw data which is also delivered.

Appendix III explains how to reproduce the language comparison and produce new images as we do in Chapter 2.

2 Programming Paradigms - State of the Art

In this chapter we discuss the modern state of the art in programming languages popular in practice or gaining popularity. Programming languages encapsulate programming paradigms as their features. Only a mixture of features makes up a programming language and defines its suitability for a particular purpose. We try to compare several languages and show their suitability in connection to the focus topics identified in the previous project phases.

On the languages choice. Before discussing the state of the art in the modern programming paradigms it is important to make an argument on the way the languages representing these paradigms are chosen. In the context of this project, we research the future programming paradigms *for automotive* meaning that many of the existing programming languages could still bear features which are *new in automotive*. We base our research on examining the existing mature programming languages *first*. Thus some of the technologies we cover are mature and practical enough to be subject to adoption in the automotive industry once project or subdomain demands match the traits of a given existing language. Overall here are the criteria for the languages we consider in this chapter:

- Established community or available commercial support shall exist to guarantee higher reliability and provide a mechanism to resolve questions or problems when using a programming language.
- Awareness on popular programming languages among industrial specialists should guarantee availability of experts on the market necessary to fulfill the need of a company, in case a language is chosen for development and longtime support.
- The language is representing a distinct class of languages, but is not a suitable solution with respect to the above criteria.

And *then* we make the restrictions above even more flexible. The languages which meet awareness only in their subdomain are also accepted (e.g., Haskell cannot be qualified as meeting the "awareness among industrial specialists" requirement in general, but it definitely qualifies if we restrict this requirement to the functional programming community).

Taking the course of high practicality and potential adoption we exclude several categories of languages, and thus do not pretend to cover the state of the art completely, but rather the cutting edge of the most practical programming practice. We intentionally exclude from consideration:

- Languages without a commercial body, or a numerous, thousands of people, community members behind them, the languages wide adoption of which is questionable.

- Highly-specialized non-universal programming languages, designed for a narrow application domain, e.g. shell scripting languages, application automation languages[2].
- Languages, the use of which requires high expertise, not generally available among the specialists on the market, e.g. the languages with heavy formal methods usage in the core.
- Esoteric languages[3], languages designed to be a proof of concept, other programming languages with low usability.

We take a number of programming languages using different programming paradigms as examples in order to cover the landscape of different language groups. We take C to compare the state of the practice language in many industries to more modern languages having similar objectives to C: Go, Rust, Swift. Several classical object-oriented or multi-paradigm programming languages are taken into comparison: C++, C# and Java are the representatives of this class. Higher-level programming languages, bringing new or different from above mentioned programming paradigms into the practice: Scala and Eiffel. A pure functional programming language, Haskell, is a representative of the functional group here. We consider Esterel and Lustre V6, synchronous programming languages, getting closer to the solution of real-time and concurrency problems. Simulink is considered as a representative of modeling (graphical) languages, using highly complex generators to produce executable code, it is used in automotive practice frequently.

Finally we use mbeddr[4]. It is, simply speaking, an improvement on C adding higher-level domain-specific languages (DSLs). Based on JetBrains MPS it is an example instantiation of modern language engineering technologies, allowing *creating and mixing freely* multiple domain-specific languages in one program.

Below we describe a method we developed to informally compare programming languages to each other. Although there is no known way to objectively compare programming languages, and such comparisons are always a subject of numerous discussions, we try to argue the way we try to systematically characterize the languages and make conclusions based on that. It is important to understand, that our comparison does not pretend to be decisive and even precise. However, we intend to give intuition on different programming languages, based on numbers, and the results we get from the systematic comparisons correspond sufficiently to our initial expectations so that we are able to make an argument based on that.

[2] We do so, because we have real and not indirect automotive programming, like programming tools for automotive, where many more languages might be of great use.

[3] Languages created as an exercise in compiling, or not having a purpose of being actually widely used in practice, e.g. FRACTRAN, Whitespace, Ook! and similar.

[4] See http://mbeddr.com/ for more information about mbeddr and MPS.

We see the following purposes of our results:

- One can compare C and Simulink to other languages.
- Starting from comparison diagrams one can get back to the definitions of characteristics and see the reason for the results, in other words which individual language features make a language of certain kind.
- Comparison shows that there is “no silver bullet”, we speculate that a combination of languages is an optimal solution for large and diverse software development domains, like automotive.

One benefit of our approach worth mentioning is its semi-automation[5], so that a new language can be added to the comparison with only a moderate effort.

2.1 Languages Features

It is a difficult and subjective task to compare programming languages. There is no precise, complete and objective approach to it. We wanted however to establish a quantitative method to assess the languages we compare. Although different projects might have specific requirements which predefine the language choice, we compare languages taking their features individually and weighting them equally. This equal weighting is where our comparison gets subjective and imprecise: various features *do have very different importance* depending on the usage goal. At first we will describe atomic language – the ingredients to build language characteristics.

2.1.1 Syntactical Complexity

We do not go into details comparing very carefully syntactical complexity (amount of keywords, their meaning in various contexts, clarity of syntax, punctuation, etc.) of different languages, but it's worth telling, that depending on the syntactic complexity languages themselves might *appear* more or less complex. It certainly should affect the ease of learning and readability of programs. We count the amount of keywords (or basic constructs) for the languages and compare them.

2.1.2 Type Systems

Type systems (TSs) are believed to be of great importance to the correctness of programs[6]. Here we define important characteristics of type systems we took into account when comparing the languages.

[5] The comparison is done in an automated table; the diagrams are produced automatically from it. Please, see the Appendix III for more details on how to use it.

[6] See for example: “Well-typed programs can't go wrong”, Milner, 1978

Static type system is one able to detect types and check the operations performed on them at compile time. This helps to avoid illegal operations before the program executions.

Dynamic type system performs type detection while executing the program. It allows for higher flexibility due to overloading or duck typing, but illegal type operations often result in runtime exceptional situations.

Nominal type systems require compatible types (and operations on them) to be explicitly declared. Such system makes sure that the programmer intentionally enables certain operations, and avoids unintended functionality.

A TS is called *„strong"* if it is highly likely to generate an error when a value of one type is used whenever a value of another type is expected (like Java or Go). Otherwise the TS is called weak (like in C or C++). There is no precise definition of strength or weakness of a TS.

Type safe type systems avoid unintended behavior of the program, when the types are used wrongly.

2.1.3 Memory Management

Pointer arithmetic is a language feature to manipulate directly the values of pointer variables with arithmetic operations. We differentiate between several *pointer arithmetic strength* classes: no pointer arithmetic (Go), limited pointer arithmetic (C#), fully powerful pointer arithmetic (C, C++).

Memory safe languages try to prevent or automate internally unsafe memory operations like: random pointer arithmetic, unsafe casting, manual allocation and deallocation.

Garbage collection is one of the mechanisms to step towards memory safety by automatically deallocating not necessary memory allocated before.

2.1.4 Supported Paradigms

We compare languages by different programming paradigms supported natively by them. We take into account *imperative, functional, object-oriented,* and *generic programming.*

Languages which support more than one of the paradigms are called *multi-paradigm.*

2.1.5 Overloading

Overloading usually stands for various language constructs changing their semantics depending on the context or the way they are used. We differentiate *operator overloading* and *method overloading*.

2.1.6 Concurrency

We differentiate languages by inclusion of a native support for concurrency in the language itself. Including locks in every structure, like Java or C# do would be a *native concurrency support*.

Separately highlighted are the languages which include *new programming paradigms for concurrency,* e.g. Go supports concurrence by go-routines, Scala by Actors and Haskell by Software Transactional Memory[7].

2.1.7 Language Mutability

Various languages include abilities to be modified by the language users. The simplest (and the most dangerous[8]) way of modification is usage of a *preprocessor*. The mentioned above overloading can also be considered a mean to mutate the language.

The ability to extend language with domain-specific construct we call *DSL Extensions Power.* If a language is internally extensible by secondary level constructs which compliant compilers may ignore, like annotations, we give assign to its extension power the value 1. The higher value of 2 goes to the languages, where new control constructs might be created. 3 goes to the DSL-centric systems, with the full power of language engineering[9] enabled.

One of the important features which we separately highlight in the language mutability is the ability to define *Safe Limitations*. The so-called unsafe features of the language are getting excluded from the limited language subset. This is typical for highly flexible, DSL-based languages, but is commonly required by safety standards for common languages as well.

2.1.8 Supported Modularity

Languages can support modularity by various methods. We distinguish here forbidding methods and structuring methods.

Forbidding methods include forbidding of *global variables* and data shared without any access control.

Structuring methods allow for better modularity via introduction of additional structure in the program. They can represent abilities to structure code in *functions*, *classes*, *modules*, *packages*, etc.

We rank languages on *global variables strength* like this: 0 means that the global variables are completely forbidden, and sharing has to be expressed with other means, 1 - allows only

[7] „Beautiful concurrency“, by Simon Peyto Jones, Microsoft Research, Cambridge, http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf

[8] The preprocessor directives usually disable IDE capabilities to perform any checks inside them. They can be implemented so, that the usage of them introduces semantic mistakes.

[9] M. Voelter et. al, “DSL Engineering – Designing, Implementing and Using Domain-Specific Languages.“, dsl-book.org, 2013.

explicit creation of global variables and controls their usage, and 2 – allows creation and unrestricted including unsafe use of global variables.

2.1.9 Internal Automations

Languages (and their runtime environments) can provide some useful automations internally. In simpler languages code has to be written manually to achieve the same functionality. We take (type-)reflection and serialization as examples here.

- *Reflection* is an ability to dynamically introspect the objects and their classes.
- Built-in *serialization* supports automatic marshalling and unmarshalling of any datatypes.

2.1.10 Virtualization

Running programs in a *virtual machine* contributes to higher flexibility and portability of the programs, but decreases performance in some cases, making resource consumption higher as a rule.

2.1.11 Applicability Features

Ability to handle *complex structures* (tuples, dictionaries, sets) natively in the language allows for higher applicability of the language itself. *Powerful IDEs, standards* and thus known to many users is also a plus when applying a language. We consider supporting *imperative* programming paradigm an important applicability improvement, as it is often the most natural way to express simple algorithms the nature of which lies close to the CPU organization principles.

2.1.12 Domain-Specificity

No matter that the languages discussed here are all Turing complete, some of them are targeting specific domains being for instance *synchronous* or oriented towards *signal processing,* designed for *embedded* development.

2.1.13 Other Criteria

Community around the language, large corporations supporting the language's development, or both make a language more stable and highly likely to be standardized. Community enables on-line support and easy resolution of frequent complications.

Programmer awareness should reflect the size of the available specialists pool aware of the language. It makes the adoption of the language easier, and guarantees availability of workforce needed to support larger projects.

We define another subjective criterion called *Learnability* and include in it the perceived complexity of the language (how hard it is perceived to learn), availability of documentation

and tutorials, and their average size. For example, Python is perceived to be easy to learn, it has a number of books describing the way it works, and they are usually observably big. In comparison a book on C++ is usually longer and can be subjectively harder to understand: unrestricted memory operations, complicated templates, diamond inheritance, etc.

2.1.14 Semantics Strength

We take a separate subsection for the following atomic features as we consider them very important.

Under the *defined semantics* feature we scale the languages from 1 to 3. With the highest score of 3 we mark the languages whose semantics is either formally defined, or is well described and fixed. The score of 2 get languages, the semantics of which might depend on the compiler implementation details, or on the language version. The score of 1 get the languages which are complex, have constructs without a clearly described semantics, whose semantics changes depending on the context, they are generally lower predictable languages.

Under the *generalized code generation* we describe languages which produce often executables with a lot of code to support generality of the language, and not directly related to the problems solved. Usually this is happening when the abstraction level of the language is high enough. Languages which allow for *low level optimizations* enable assembler insertions, and similar CPU and memory operations oriented towards low abstraction level CPU manipulation.

Languages (and IDEs for them) might include internally some analyses. Analysis makes it easier to (at least partially) enforce and check the semantics of a program in the language. We consider here only the *contract analysis*, e.g. the ability to formulate and check pre- and post-conditions for a program fragment. The score 0 is reserved for the languages which do not have any analysis features built-in. The score 1 is included for the languages with some kinds of analysis. The score 2 is for the languages which include various analyses and are intended to be used with analysis on a regular basis.

2.2 Language Characteristics

From the language atomic features we derived generalized language characteristics which should play a role in characterizing the language as a whole. Later in Chapter 6.3 we proceed to connecting the characteristics to automotive subdomains. The necessary characteristics are to be defined on a more narrow scope - per project.

Below follow the characteristics we derived with their definition, but first we discuss the methodology we used when summing up features into characteristics.

2.2.1 Methodology

To get the language characteristics like language-induced performance or correctness we want to aggregate individual atomic features. For this we list the features in a table and score languages as described above. When aggregating features into characteristics we simply sum up the scores which relate to the characteristic being built.

Naturally when summing up we expect the values to have the same measurement unit. We score all the features from 0 to 1 and some up these normalized scores. Like this we make sure that we do not prioritize one feature over another, although in particular circumstances this might be exactly what has to be done! For example, one atomic feature, synchrony, can be decisive when building software with hard real-time constraints. Here we lose precision and introduce subjectivity.

As a result, we get cumulative characteristics scores for different languages. It is important to understand what they mean and start from them when reasoning about the use of a given language in a project.

Once language A is scored higher than language B in, e.g. performance, it does not immediately imply that a program in a language A is going to be more performant then a similar program in the language B. And this is not only due to the fact of the programs being expressed differently, different skills of programmers performing the job, but it is due to individual atomic features might be more relevant than others in a given context.

The higher score in some characteristic just means that there are more reasons to be "better", and not that these reasons outweigh the lower-scoring language. Our method gives an intuition, an expectation, from which a detailed analysis might *start*, by having a look in the language comparison table we provide, and analyzing the features contributing to the result.

In this sense our diagrams which we show below, with comparisons of different languages are a *starting* point for a language choice question, and in no way the end result.

2.2.2 Language Internal Complexity

The following atomic features contribute to language internal complexity:

Positively	Negatively
+ Syntactical complexity + Dynamic Type System + Multiple Paradigms + Operator Overloading + Method Overloading + Pointer Arithmetic Strength + Global Variables Strength + Pre-processor + Reflection	- Strong Type System - Type Safety - Memory Safety - Learnability - Defined Semantics

The language internal complexity should informally describe how hard/easy it is to understand a program written in the language by a professional, as well as how hard it is to master the language.

Internal simplicity is the opposite of the internal complexity, a negation of the sum defined.

2.2.3 Correctness

We define correctness as the following sum:

Positively	Negatively
+ Static Type System + Nominal Type System + Strong Type System + Type Safety + Memory Safety + Native Support for Concurrency + Powerful IDE + DSL Extensions Power + Defined Semantics + Safe Limitations	- Pointer Arithmetic Strength - Global Variables Strength - Preprocessor - Imperative Paradigm - Low-Level Optimizations

Subjectively a more correct language should ensure more accurate code and forbid language elements where usual pitfalls happen.

2.2.4 Internal Performance

The following factors internal to the languages are able to influence the performance of programs in them.

Positively	Negatively
+ Low Level Optimizations + Pointer Arithmetic Strength	- Dynamic Type System - Garbage Collection - Reflection - Execution in a Virtual Machine - Generalized Code Generation

Performance depends a lot on the way the language is used and on the compiler and runtime implementations. This characteristic is subjective.

2.2.5 Code Portability

Portability is higher when a program in a language is easier to port to another platform. The following atomic features can influence portability.

Positively	Negatively
+ Strong Type System + Type Safety + Garbage Collection + Native Concurrency Constructs + Generic Programming + Execution in a Virtual Machine + Defined Semantics + Generalized Code Generation	- Pointer Arithmetic Strength - Low-Level Optimizations

Good portability is great for reuse.

2.2.6 Language Flexibility

The flexibility of a language allows easier use of the language in different contexts and goals. Flexible languages should support more naturally direct expressions of the programmers intent.

Positively	Negatively
+ Dynamic Type System + Garbage Collection + Native Concurrency Constructs + Multiple Paradigms Supported + Functional Programming + Generic Programming + Object-Oriented Programming + Operator Overloading + Method Overloading + Pointer Arithmetic Strength + Preprocessor + Reflection + Serialization + Native Support for Complex Structures + Imperative Programming + DSL Extensions Power	-

2.2.7 Concurrency Readiness

We include the following features into concurrency readiness.

Positively	Negatively
+ Native Support for Concurrency + New Paradigms for Concurrency + Functional Programming + Synchrony	-

It's worth mentioning that synchronous languages are of a special importance here.

2.2.8 Applied Practicality

We introduce a new characteristic here. We wanted to reflect, why some programming languages are considered to be „simple to use“ and „efficient” in solving many of the daily programming tasks. We aggregate here the features which make a language practical to use when solving problems without any special requirements (safety, real-time requirements, etc.). Such languages should be flexible and easy to use, they should incorporate paradigms with which a regular programmer is likely to be familiar.

Positively	Negatively
+ Multiple Paradigms Supported + Generic Programming + Object-Oriented Programming + Pointer Arithmetic Strength + Reflection + Serialization + Complex Data Structures + Powerful Applied Standard Library + Powerful IDE + Imperative Programming Paradigm + DSL Extension Power + Programmer Awareness + Standard Tool Chains + Learnability + Low-Level Optimizations	-

2.2.9 Resource Consumption

With this characteristic we try to describe the runtime environment needed to support programs in a language. Namely the runtime might need more or less computational resources (memory and processing power).

Positively	Negatively
+ Multiple Paradigms Supported + Generic Programming + Dynamic Type System + Garbage Collection + Functional Programming Paradigm + Reflection + Serialization + Running in a Virtual Machine + Support for Complex Data Structures + Generalized Code Generation	- Low-Level Optimizations - Pointer Arithmetic Strength

* On the charts we usually *invert* this characteristic, so that the higher ranking in resource consumption means lower actual resource consumption indeed – so the higher the invert is the “better” (the lower) the resource consumption is.

2.3 Language Characteristics Charts

Here we bring in charts from which a language analysis should start. After comparing a language with others on the charts it is possible to proceed to the comparison table to find out the exact details which lead to the results shown on the charts. The details give deeper insights in the language and its characteristics.

In the remarks section we will mention C and Simulink because they are used often in the automotive domain practice. We will also try to explain the interesting insights which we get ourselves from the charts. Doing this we show how the charts are to be used.

A curvy dashed line on the top right side of the charts should remind the reader of the Pareto-frontier concept. The languages placed close to this line should represent relatively optimal choices.

2.3.1 Simplicity vs. Correctness

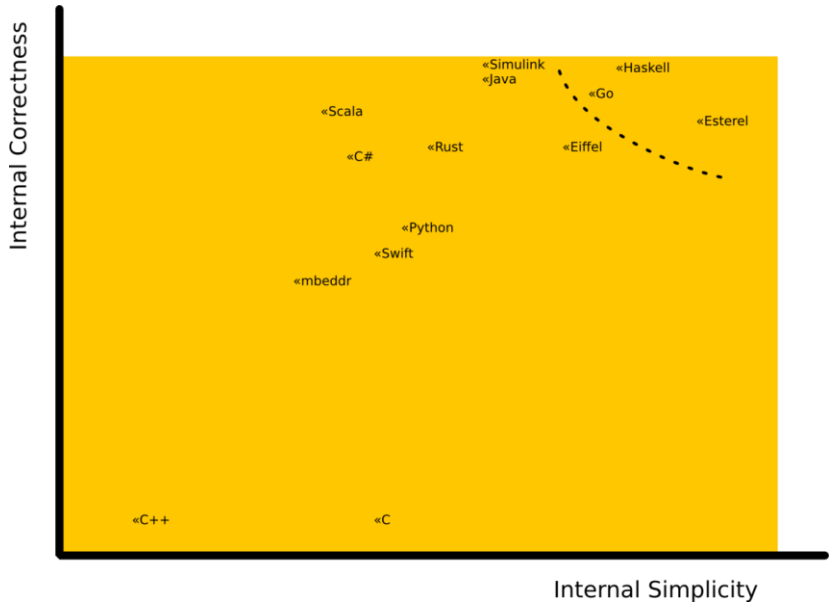


Figure 1- Internal Correctness vs. Internal Simplicity

Remarks. In this chart we see the higher distribution of languages close to the bisecting line. Thus we confirm our intuition: higher complexity of the language generally speaking negatively influences its correctness. The C programming language is noticeable for being less correct than many languages whose simplicity is comparable. Simulink stands good in this chart being correct and simple enough.

No silver bullet. We remember Haskell, Go, Simulink, Java and Eiffel to be “correct”.

2.3.2 Applied Practicality vs. Performance

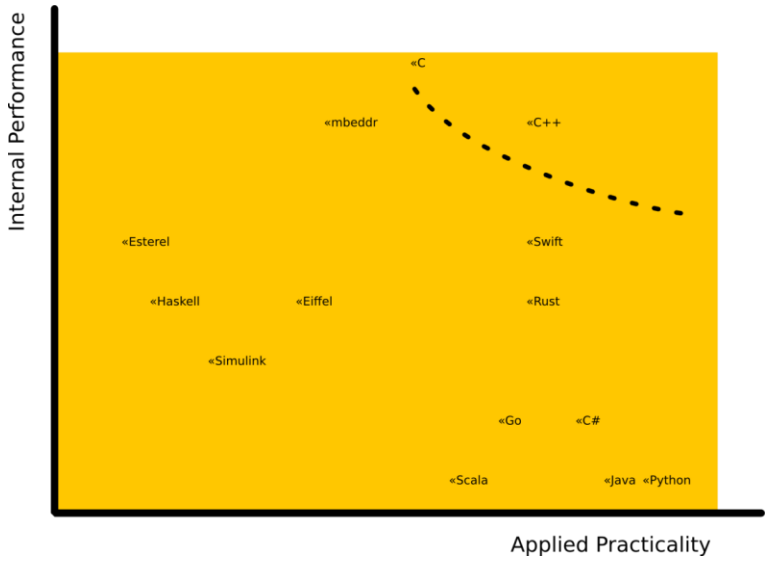


Figure 2.- Internal Performance vs. Applied Practicality

Remarks. C and C++ top the chart being performant and practical. Stepping down from the performance of complex C++ we retain the practicality level of it by using Swift and Rust.
Python appears to be the most practical of all the languages compared. Its applicability should not be confused with the language's low complexity, which is *not* the case. Python comes handy for solving many practical problems; the performance of it remains low though. Thus Python is not suitable for example for building real-time applications. Simulink has low practicality as it is not a purely general purpose language. The performance of it is low, as the generalized code generation is used.

No silver bullet. "Correct" languages" do not top the practicality and performance league. We get here C, C++, mbeddr, Swift and Rust as quick and useful.

2.3.3 Applied Practicality vs. Resource Consumption

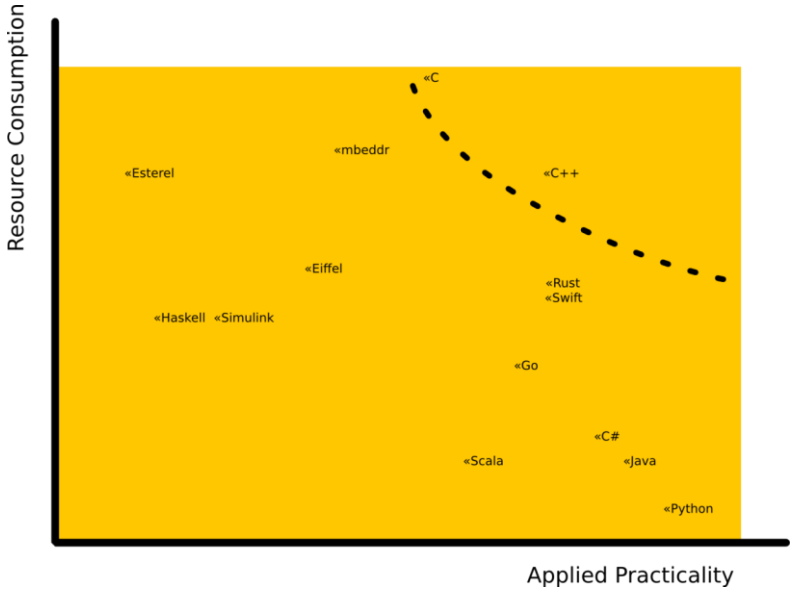


Figure 3.- Resource Consumption (inverted) vs. Applied Practicality.

Remarks. The classical (low) resource consumption champion C on this chart can be compared in practicality with heavier languages. Rust and Swift come handy, when complex C++ is not an option for the sake of higher correctness. Simulink can be dominated heavily in practicality being a one-paradigm language. The code generated from it (without modifications) could be compared in resource consumption as well, when it comes e.g. to mbeddr, which has higher practicality and still can incorporate domain-specific notions.

No silver bullet. "Correct" languages appear mediocre when it comes to the resource consumption.

Additionally we would like to mention that the costs of software development and maintenance grows with the use of a less practical and less correct language. Languages with lower resource consumption tend to be more difficult to maintain. And in the future of automotive industry this might have a bigger influence on the development costs than limiting the hardware computational resources of a car's embedded computer to run the software.

2.3.4 Correctness vs. Concurrency

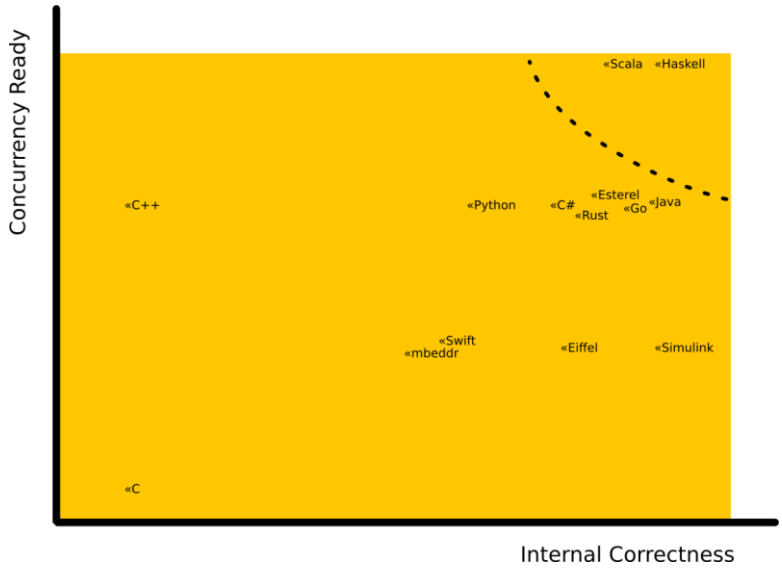


Figure 4 - Concurrency Readiness vs. Internal Correctness

Remarks. The C programming language does not have any support for concurrency and is not improving on the correctness of the programs. Once going concurrent, it makes sense to not to start with C. Simulink lies low in its concurrency support as well. Whereas Scala and Haskell have special support and extensions for concurrency. They are supporting higher correctness as well. When having stronger resource consumption limitations, it is then worth looking on Rust and Go for concurrent development.

No silver bullet. Light-weight languages are not concurrency ready. We remember Scala and Haskell as topping in correct concurrency.

2.3.5 Performance vs. Correctness

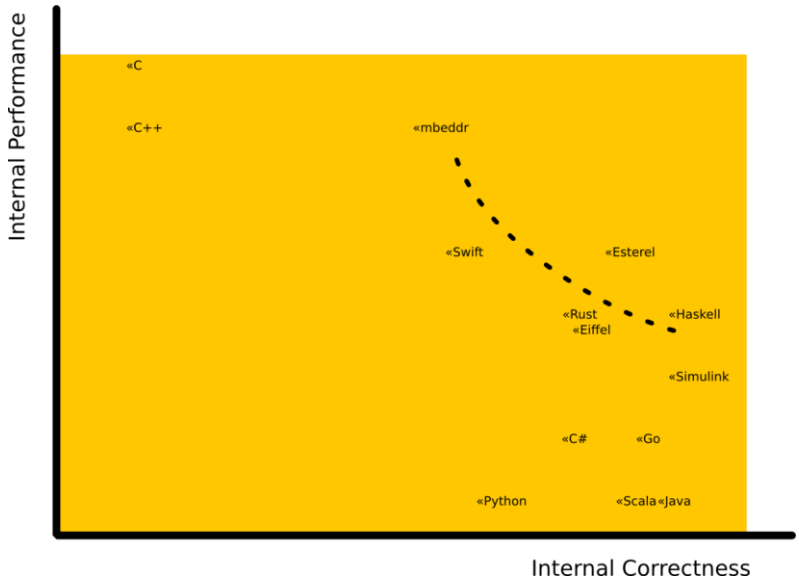


Figure 5 - Internal Performance vs. Internal Correctness.

Remarks. Both C and Simulink appear to be not optimal on this chart. When synchronous languages are possible for the application built, they could be a choice for correct and performant applications. Rust, Swift, mbeddr, Eiffel and Haskell represent in our chart correct and rather performant solutions.

No silver bullet. Correct and highly performant Esterel is not practical, concurrent Scala and Haskell are not in the top of performance still.

2.3.6 Portability vs. Correctness

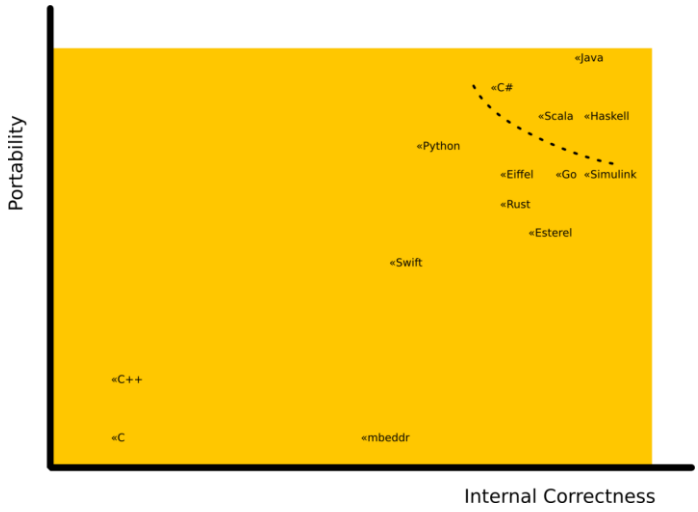


Figure 6.- Portability vs. Internal Correctness.

Remarks. C doesn't appear to be an optimal choice for producing correct and portable programs. Low-level programming language allows for creating machine-specific code with specific low abstraction level implementation details. C, thanks to its performance, could still be used further in drivers, as their code base is usually limited and is easier to verify, portability is not an objective.

Simulink in turn appears to be correct and portable. However other aspects of Simulink (performance, resource consumption) discussed here suggest for improvements. Thanks to high abstraction level the modern programming languages: Java, Scala, Haskell and C# represent a good choice for creating correct and portable code.

No silver bullet. Correct and portable means low performance and high resource consumption, low practicality otherwise.

2.3.7 Resource Consumption vs. Correctness

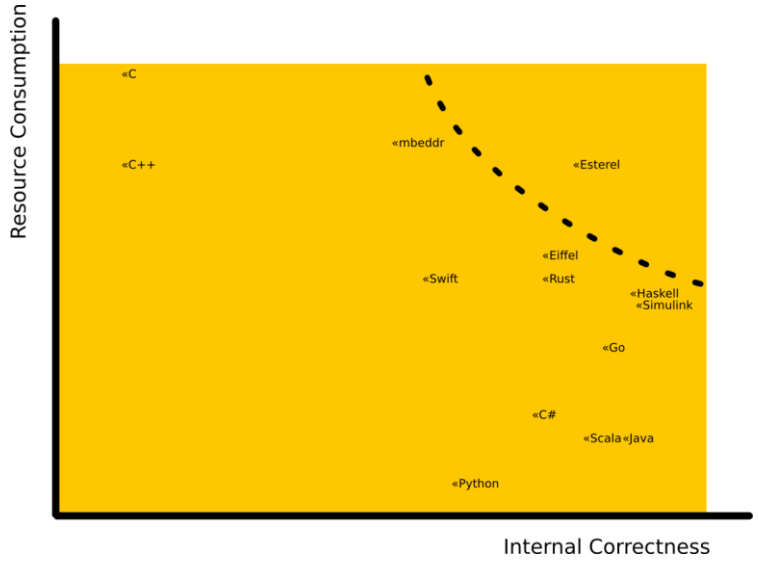


Figure 7 - Resource Consumption vs. Internal Correctness.

Remarks. When creating resource-constrained code which has to be correct domain-specific languages appear to be the optimal choice. Synchronous and simple Esterel, specialized Simulink and mbeddr stand close to the "Pareto frontier". Eiffel and Rust combine reasonable correctness and resource consumption. Haskell and Simulink stand on the correctness side taking more resources to finish the job. The best (meaning the lowest) resource consumption can be achieved with C (and C++) for the cost of potentially low correctness.

No silver bullet. Niche language Esterel shines here, but not anywhere else at the same degree.

2.3.8 Simplicity vs. Applied Practicality

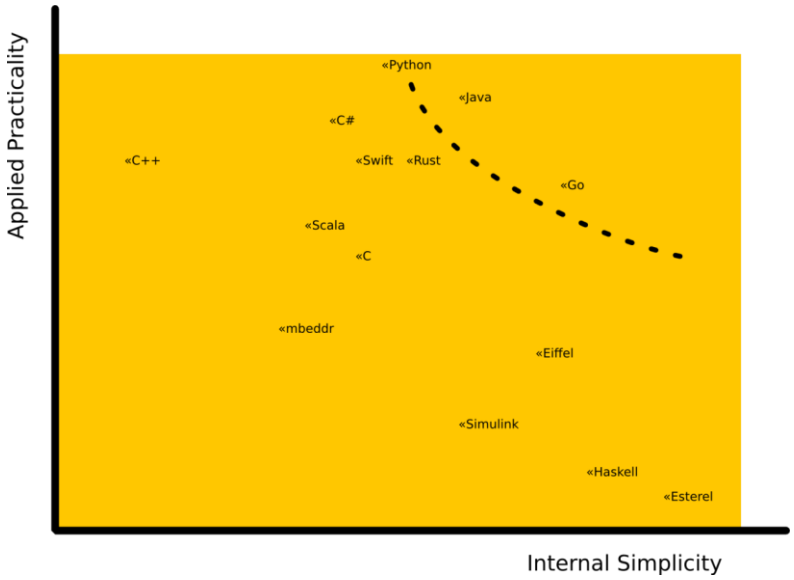


Figure 8 - Applied Practicality vs. Internal Simplicity.

Remarks. Python, Rust, Java and Go appear to be simple and practical languages. They can be recommended to solve data processing tasks with complex data structures. These languages are (rightfully) used often to create connected services. Looking on the concurrency readiness and correctness, the languages also appear beneficial there. C and Simulink are not an optimal choice here.

No silver bullet. Popular Python, Java and Go are popular for a reason they are simpler and practical. Performance is low though.

2.3.9 Simplicity vs. Flexibility

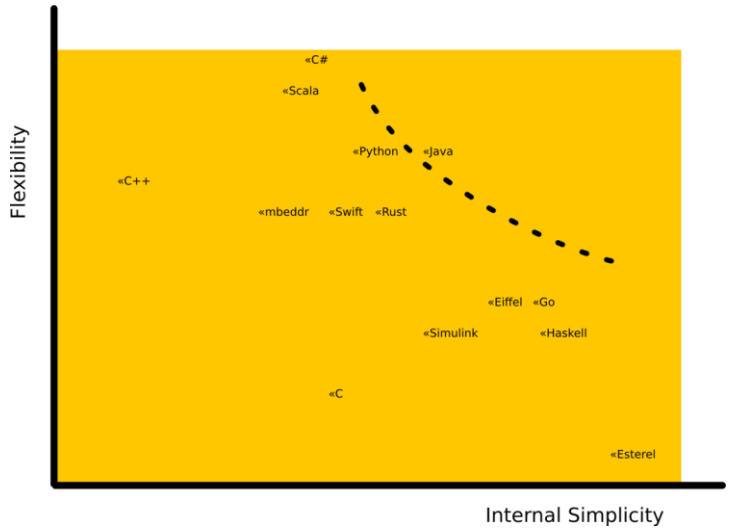


Figure 9 - Flexibility vs. Internal Simplicity.

Remarks. When looking for a language used in a larger team for creating a software product with a high number of different-by-nature features it is reasonable to look for a flexible language, simple enough to be used in a big team without a problem. Python and Java appear to be optimal in this context. Java with its packages and defined semantics, high portability and correctness being not a very modern language still appears to be an attractive choice generally. C# and Scala are flexible enough, but the simplicity is lower. Higher internal simplicity[10] for lower flexibility comes with Eiffel, Go and Haskell. C and Simulink both are not optimal in this chart.

No silver bullet. Flexible and simple Python, C# and Java are on the bottom in resource consumption.

[10] See the definition of internal simplicity above, to justify, why we call e.g. Haskell simple: its type system is strong and safe, the language is memory safe, it contains only one programming paradigm, it doesn't support (confusing) overloads, and has defined semantics, and these factors make the language moderately complex in our comparison.

2.3.10 Simplicity vs. Performance

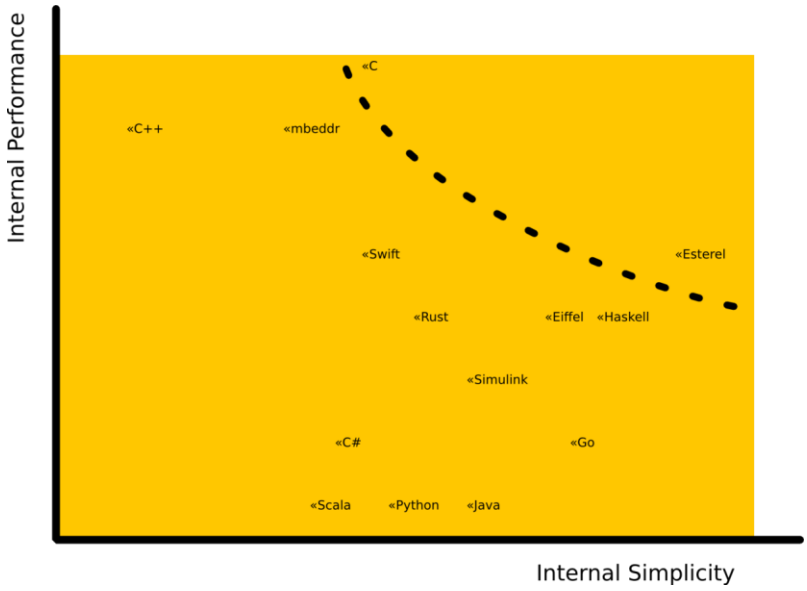


Figure 10 - Internal Performance vs. Internal Simplicity.

Remarks. When looking for a performant but simple language Rust and Swift come close to be optimal. Esterel is very simple and performant enough, its domain-specificity however doesn't allow the universal and convenient use of it. C outperforms other languages, but the complexity of it suggests another choice. Eiffel and Haskell allow reasonable performance while still being simple enough. Simulink is domain-specific and is not optimal on this chart.

No silver bullet. Simple and performant language sacrifices applicability, flexibility or correctness.

2.3.11 Simplicity vs. Portability

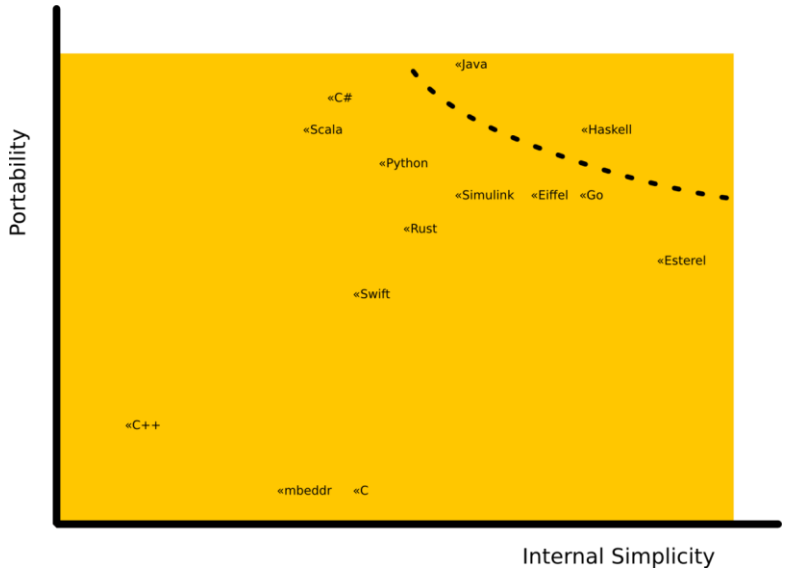


Figure 11 - Portability vs. Internal Simplicity.

Remarks. The ability to perform low-level operations and the type system of C make the language a bad candidate to express portable algorithms. Simulink is positioned much better in this chart. Still its domain-specificity makes it not naturally usable for any general algorithm. Java, Haskell, Eiffel and Go can be used to produce highly portable and simple code. Python, Scala and C# stand on a less simple, but still very portable side.

No silver bullet. Simple and performant languages sacrifice applicability, flexibility or correctness.

2.4 Summary

Programming language features in their combinations characterize a concrete programming language. An optimal language for *one selected* purpose starts to be less optimal for other purposes. Popular in automotive, C and Simulink represent an optimal choice very rarely when comparing multiple characteristics. For the automotive industry, depending on sub-domain, different languages could be used. It is possible to try to find the right mixtures of features, when the objective of a subdomain or a single project is defined. See the Chapter 6.3 for characteristics for subdomain recommendations.

As one single language cannot satisfy the demands of every domain, it is necessary to think of interoperability of different languages. It can be achieved using various methods in combination: inter-language interfaces, common platforms, environments supporting cross-language development and mixing of languages, orchestration languages – these are to be researched in the future, see Chapter 7.3.

3 Automotive Software Engineering Today

The development of vehicle functionality has largely been an evolutionary process in the past. Starting from a purely mechanical setup, electronics were added to the individual aggregates to increase the performance. These mechatronic systems were interconnected via communication busses throughout the vehicle in order to implement functions, which rely on a multitude of sensors and actuators from different domains of the vehicle. This trend continued and an increasing number of functionality was shifted from a mechanical realization towards software-based systems. Today, several million lines of code are running within each vehicle. Most of the innovation is contributed to additions and changes on the software level within tightly interconnected subsystems. The increasing number of new functions led to an increasing number of electronic control units to be integrated into the vehicle's architecture. The result is a highly complex electronic architecture with multiple dependencies.

A further increase of the vehicles' functionality by software and mechatronic systems will create new challenges for the design, specification, implementation, testing and integration of automotive software into a consistent architecture. The multifold dependencies between the systems are hard to handle and new concepts have to be developed to mitigate the increasing complexity on an electric, electronic and functional level. In this section, we give an overview about the specific characteristics of the automotive domain and the relevant features and trends that influence the state of the art and future challenges to automotive software engineering.

3.1 Domain Overview and domain-specific Requirements

The development of today's upper class vehicles is considered as one of the most challenging engineering tasks in product development. The automotive domain combines a wide variety of features from different areas[11]. Among others, prominent domain features are the heterogeneous nature of automotive software and hardware in the automotive subdomains, specific requirements with respect to reliability, safety and security, the division of labor between OEMs, suppliers and engineering consultants and the widely established unit-based cost model as well as the large number of variants per product line.

3.1.1 AUTOSAR

In an attempt to establish a common framework within the automotive sector to define automotive software and system architectures and to unify software development standards,

[11] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software engineering for automotive systems: A roadmap," *Futur. Softw. Eng. 2007. FOSE '07*, pp. 55–71, 2007.

the AUTOSAR consortium was founded by OEMs and suppliers to develop the necessary industry standards. AUTOSAR defined a layered ECU middleware architecture to provide generic application interfaces for function developers and a development process that supports implementation, testing and integration into vehicle architectures. One of the main objectives of AUTOSAR is to create a methodology that enables the reuse of generic application components independently from a specific hardware platform, although this is often inhibited by rigid inflexible application architectures and hardware-specific optimizations added to overcome hardware limitations. AUTOSAR heavily relies on C as language of choice for application development and an elaborate tool chain for modeling, configuration and generation of configuration as well as application code stubs. As C has no support for several paradigms that are included in AUTOSAR, such as modules or explicit description of runtime behavior, there is still a noticeable gap between the AUTOSAR objectives and the current state of practice.

3.1.2 Heterogeneous Nature of Automotive Software and Hardware

The current functional architecture of vehicles is the result of a historically grown architecture. Starting from purely mechanical systems, functionality was enhanced by the integration of electronics to individual components and later on by the interconnection of the electromechanical devices. Today, the E/E (electric/electronic) architecture of vehicles is traditionally divided into five different domains, which become more and more interconnected with each other[12]:

3.1.2.1 Chassis

Traditionally, the chassis domain was the base structure of a vehicle and typically made of a robust metal pipe frame. Today, the domains chassis and body merge as the vehicle usually contains a self-supporting body. However, the chassis domain is still a container for components that were formerly associated with that domain like suspensions and brakes. The functionality realized by these components is highly safety critical. Brakes as well as the steering system have to operate reliably at all times. These formerly purely mechanical components are now enhanced by electronic control systems to enhance comfort and safety.

Unlike traditional PC software, the control systems software employed in this domain usually realizes behavior that controls physical processes that are modeled as differential equations. Most of those *continuous real-time data processing tasks* require deterministic timing and task execution in order to ensure correct behavior as timing also influences the reaction to changes in the physical process.

Systems like ABS or ESP introduce high amounts of software that also has to fulfill strict safety standards when it comes to specification, implementation, testing and integration. However, as a last resort, there is still a purely mechanical fallback in all those functions. They are constructed in a way that at least degraded function of brakes and steering is always possi-

[12] Campetelli, A., Irlbeck, M., Bytschkow, D., Cengarle, M. V., & Schorp, K. Reference Framework for the Engineering of Cyber-Physical Systems: A First Approach.

ble to bring the vehicle into a safe state as it is demanded by regulations, even in case of total power loss or ECU malfunction.

Future trends show steps that lead away from the mechanical fallback towards X-by-wire systems where there is only a data connection ("wire") between the driver input from pedals or steering wheel. Here, the software system does not only have to detect errors, but needs recovery strategies that include fail-safe or even fail-operational behavior.

3.1.2.2 Body

Today, the body or comfort domain contains mainly secondary features of the vehicle, which are not directly associated with the driving capability like wipers, lights and window controls. A sharp differentiation between the chassis and body domain is not always possible. While in the past, many of those functions used to be controlled by simple switches, the state of those functions is nowadays controlled by software functions in the *Body Control Module* ECUs.

The embedded software that is required in this domain has to *control the transitions between states* on function level as well as the reaction on a small set of vehicle wide state changes, mostly consisting of starting, stopping or locking the car. There are *no hard real time constraints* and only basic requirements when it comes to functional safety, which are covered by standard QA procedures.

The body domain is a main location where software based features will be responsible for an increase of customer visible features and improvements in usability of the vehicle. While today, interaction between the functions in the body domain is limited to a certain extent, new connected functionality is expected that introduced high interconnectivity between functions in the body domain as well as to functions in other domains. Also safety-critical applications might become dependent on comfort functionality.

3.1.2.3 Powertrain

The powertrain domain includes all components related to the engine of a vehicle. Today, this usually includes a combustion engine with auxiliary aggregates and the manual or automatic transition. In case of a hybrid powertrain, the high voltage electric engine components are also part of this domain. To control the highly complex physical reactions in a combustion engine and to optimize power and efficiency, highly sophisticated control algorithms are implemented in the engine control system ECU. Similar to the chassis domain, the continuous control is *highly real time critical and highly safety critical*. Complex mathematical models of the engine are used to create the control algorithms, usually using model-based techniques provided by products such as Matlab/Simulink. The engine control is optimized for a specific engine model in an elaborate process of manual testing to find the perfect set of parameters for the engine in combination with the vehicle.

This domain is expected to become more complex in the future, as the variety of electric and hybrid vehicles will increase. The separation of the functions acceleration, braking and steer-

ing in different domains (chassis and powertrain) prevents the implementation of highly reactive safety systems that have control of the whole vehicle with little latency.

3.1.2.4 Passenger and Pedestrian Safety

The passenger and pedestrian safety domain includes components and functions for the active and passive safety such as airbags or emergency brake systems. Due to the immense development of safety features, this domain will gain a more important role in the control of the whole vehicle and in the coordination between other vehicles.

From a software development perspective, the integration of data from other vehicle and external domains and therefore the necessity to deal with other development styles for *highly safety relevant functionality* will have a huge impact on development efforts and costs.

3.1.2.5 Infotainment and Telematics

Systems of the domain infotainment and telematics are typically not safety-critical and include functions like navigation, traffic forecast, communication capabilities and passenger entertainment. Also, among all automotive subdomains, the infotainment domain is the least standardized one with a variety of brand-specific components and communication solutions. In this domain, also programming paradigms such as object-oriented programming are widely used. Software development in this domain is much more similar to software development for multi-purpose consumer devices, as there are *virtually no constraints with respect to safety or timing.*

However, the complexity of this domain will also increase in the future as the cooperation of vehicle among each other, to the infrastructure and centralized services will increase. Also, as this domain will have high interaction with other domains that require stricter software quality standards, compatible data models and programming paradigms that allow the interaction of e.g. the navigation system with the powertrain for energy optimization.

3.1.2.6 Cross-Domain Functionality

A visualization of the interconnection between the vehicle domains according to the AUTOSAR industry standard[13] is given in Figure 12. It is clearly visible that the domains are heavily interconnected today. This trend continues as newly integrated functions, especially advanced driver assistant systems (ADAS), require data from a multitude of domains while controlling a set of aggregates also from different domains of the vehicle. Already today, driver assistant systems are considered cross-domain functionality as they combine interaction with the driver, control of safety-critical vehicle functionality and rely on sensor hardware that is part of the chassis or body domain.

[13] AUTOSAR Group, AUTomotive Open System ARchitecture (AUTOSAR) Release 4.1 (2013).

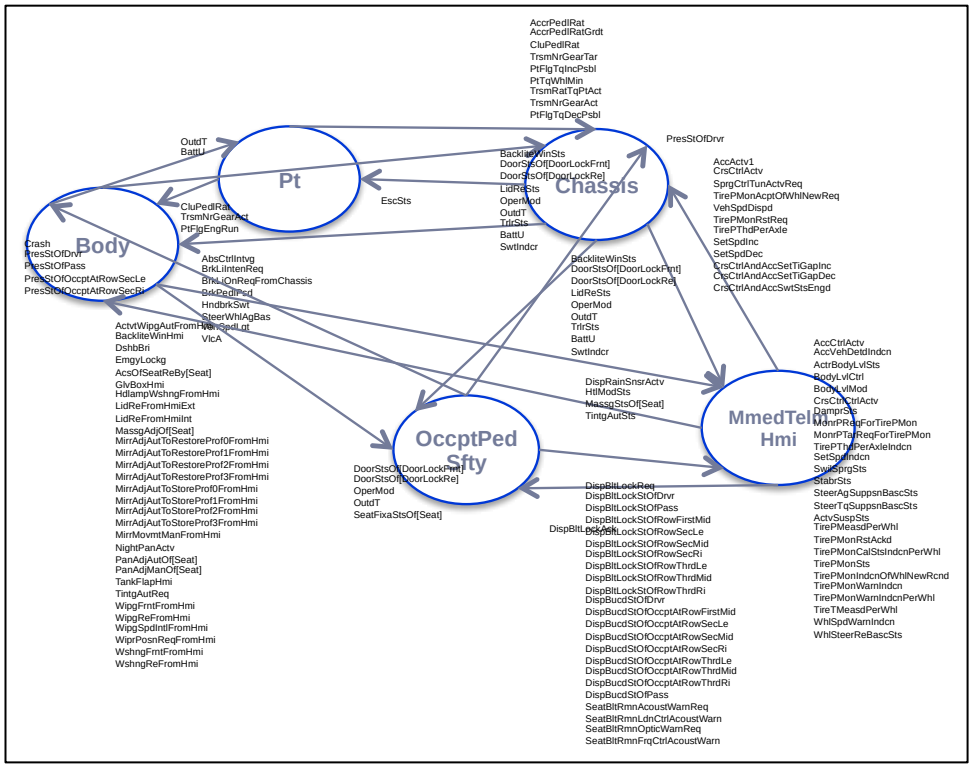


Figure 12 - Visualization of interconnection between vehicle domains

This interconnection and the involved complexity for the integration of new functions is becoming a serious challenge for further vehicle generations. The current state of automotive software development was not introduced with such a high interaction of the functions in mind, as development styles are very different in the respective domains, according to the differences in extra-functional requirements such as reliability.

This means that continuous control systems have to interact with discrete state systems and safety-critical systems will have to interact with less critical comfort systems.

Software engineering techniques will have to be adapted to the increase in relevance of software-based functionality and to the introduction of interconnectivity between functions of different domains and properties. Currently, there are no standardized modeling or development approaches. Also a large variety of different, often incompatible development tools make interaction in development complicated already on the technical level.

3.1.3 Relationship between OEM and Supplier

On the organizational level of development, the automotive industry highly relies on relationships between suppliers, OEMs and engineering consultants. While the overall product with all its distinct characteristics and features is designed and planned by the OEM, most components are developed and produced by external parties. Today, usually only central parts of the combustion engine and the car body are specified, developed and produced in-house.

3.1.3.1 Division of Labor

Next to cost-efficiency, it is beneficial for the OEM to limit their activities to specification and integration and leave development details to suppliers who are specialized on the types of aggregates they are to develop. Often, suppliers develop a standardized product offered to multiple OEMs and only customize details to match the specifications. This is more cost efficient than developing multiple independent solutions for similar tasks.

This division of development tasks has proven to be very successful with functionality being organized in a vertical manner, meaning independent functions realized by independent subsystems. Also, suppliers have many degrees of freedom to choose a suitable implementation solution, as long as all specifications and requirements defined by the contract are met.

As already mentioned in the preceding sections, the role of automotive software becomes more and more prominent. This also has strong impact on the future relationship between suppliers and OEMs:

- A well-coordinated and efficient distributed development process between OEM and the suppliers is crucial. With higher interconnectivity of functions, different suppliers with their individual implementation strategies have to be coordinated and integrated. It is therefore necessary to introduce additional levels of interaction on a technical level.
- Today, the item that is specified and delivered is usually an ECU with the included software functionality. Suppliers carry out this integration, without tight integration in a holistic system integration strategy that covers all other components of the car. Problems arise if the system developed and integrated in isolation behaves differently when integrated into the vehicle architecture. Because of the “black box” nature of the delivered component, it is hard for the OEM to localize bugs or other deviations from the specification that cannot be tested in an isolated environment.
- During the implementation of the control software, the suppliers create new intellectual property that lies beyond the area of influence of the suppliers. Therefore, new ways of dealing with these new types of values that are generated have to be introduced.

3.1.3.2 Unit-based Cost Model

Traditionally, the cost model between supplier and OEM is unit-based. A unit price is calculated to cover development, production and maintenance cost. The supplier’s margin is generated from a high number of shipped items and from an optimized development process.

This led to a radically cost-driven business model where all activities are focused on reducing per-unit costs. Hence, software development optimizations are focused on reducing the CPU and memory footprint of an implementation, at the expense of activities to optimize quality, enhance functionality or modularity to enable reuse. The code created due to these con-

straints is often more complex than necessary, less flexible and due to the high degree of hardware-specific adaptations, most likely not reusable on future hardware generations.

The costs for developing software-intensive systems include

- Development cost
- Maintenance cost
- Different forms of opportunity cost
- Brand-related cost.

While production costs are declining due to technical advancements and the move from hardware- to software-based functions, development costs increase, driven by more complex software and a more elaborate testing and integration process. Also, licensing aspects come into play as some functionality that is already created is protected intellectual property by the supplier or a third party.

A future challenge with respect to pricing models is to honor efforts related to software development costs explicitly, as economies of scale work completely different from traditional hardware production. This can enable a priority shift towards more powerful hardware that can greatly reduce development and maintenance efforts.

3.1.4 Product Lines and Variant Management

A very prominent topic in the automotive industry is the necessary support for product lines and variants. Traditionally, manufacturers offer several vehicle models in different classes (lower-, middle-, upper class vehicles), multiple body configurations (sedan, coupe, station wagon, convertible) and several combinations of engines and transmission systems (manual, automatic, double clutch). Apart from variability in the basic features of a vehicle, variability of auxiliary vehicle equipment has expanded. Customers can chose from a variety of comfort equipment such as seating, air conditioning or infotainment systems. Also, it is possible to order combinations of advanced driver assistance systems (ADAS) as well as partially autonomous functions.

It is driven by three main reasons:

- **Driver 1.** Customizing. This is especially relevant for European premium automakers. The customer is able to create his individual vehicle configuration from a multitude of available positions.
- **Driver 2.** Adaptations to regional needs and regulations. Depending on local situations, such as climate conditions, fuel and street quality as well as regulations, country-specific variants (e.g. right-hand drive in GB, US SAE vs. European ECE norms)
- **Driver 3.** Technological advancement and spare part supply guarantees up to 15 years. Most vehicle manufacturers are obligated to guarantee the availability of spare parts for up to 15 years after the model is discontinued, based on different

regulations. Even while the model is still in production, technological advancements can render common parts, such as specific CPU models, unavailable.

As variability on a mechanical level is well established by using standardized couplings and plugin-modules, variability on software level becomes more and more important.

A building block to enable variability in software-intensive systems is the definition of clean, stable logical and technical syntactic as well as semantic interfaces, similar to standardized mechanical couplings. However, the software included in an ECU consists of multiple components, such as 3rd party libraries, generic components and ECU-specific implementations. Leveraging this would also enable reuse of generic software components in different models or variants, while only a small specific part has to be rewritten.

Currently, reuse is performed on ECU level, not on software level. Also, automotive software development is currently focused on enabling cost-efficient hardware and not on software reuse. Usually, functionality between models differs only about 10%, but much more than 10% is currently rewritten. Also, short hardware lifecycles may require frequent re-implementations. As the amount of automotive software increases, the current development method causes higher development efforts and more complex testing, increasing costs and delays in time-to-market.

As the importance of software is highly increasing, proper development methodologies can leverage the advantages of software based features such as a high degree of reuse and portability, which will overall reduce costs and development efforts while providing more flexibility and quality as tests can also be reused. Hence, a switch from pure focus on hardware optimization to more generous hardware platforms in combination with well-engineered middleware software can greatly improve the value creation by software while keeping overall costs at a reasonable level, provided a suitable cost model.

3.1.5 Testing and Maintenance

In a software-intensive system, it is not always easy to distinguish a hardware problem from a fault caused by ill-designed or simply wrong software implementations. Therefore, frequently healthy aggregates are changed during maintenance to solve an error that is actually caused by software. As vehicles inherit characteristics of complex IT systems, development as well as testing and maintenance strategies should be reworked and adapted to the new and upcoming challenges.

3.2 Focus Topics for Automotive Programming Paradigms

A programming paradigm covers the programming language, a development environment and the development process it is embedded in. Depending on the type of system, various programming paradigms exist. Often, multiple paradigms are applied for one product, depending on the subdomain that is under development. To develop and evaluate programming paradigms, domain-specific topics have to be defined that provide requirements and/or propose language, IDE or process features. For the automotive domain, we defined 12 focus topics.

3.2.1 Models

This focus topic captures general questions regarding model driven development. We separately focus on the problem of practical round trip development, when the generated code has to be modified, and integrated into the model, so that the model is not out of the process after the modification. UML is one of the modeling languages used often in practice. Companies using UML have specific problems with it, which we put into this focus topic as well.

3.2.2 Cross-Domain Development

Modern cars represent a complex system, which has subsystems with different kinds of software and requirements for them, which have to collaborate, to provide the system-wide features. For example, the software for the engine control usually represents control software and has real time constraints. Infotainment software usually is focused on processing complex data structures, and does not have any special requirements. The problem of software development across the domains when features of different subsystems are to collaborate involves software engineering corresponding to the requirements of both domains. Orchestration languages, universal multi-paradigm languages might be of interest here.

3.2.3 Reuse and Portability

This focus topic covers a broad topic of reuse. Over generations cars retain many of the key features but changing often the hardware platform. Thus reuse of the existing artifacts and portability come into play. In this focus topic we investigate various problems, which prevent extensive reuse and portability due to the nature of the programming languages used.

3.2.4 Requirements Management

As implementation code is produced from requirements, requirements stand close to the implementation. Thus the way requirements are formulated, processed, passed and stored can influence the implementation quality. In particular requirements can be formulated in a special language, which can be used for generating implementation, or in connection with implementation. In this focus topics we investigate the problems in the industry resulting from the way requirements are processed.

3.2.5 Architecture

In a similar way the requirements are passed to the system engineer, the architecture designed by him has to be propagated to the implementers. It thus becomes important in which way it is communicated, whether the architecture designed might be a basis for implementation via generation, or it might be incorporated into the implementation via special language constructs supporting the architecture. The problems of software generality and portability are often induced by the software architecture *predefined by* the hardware archi-

tecture. We investigate the implementation language features allowing avoiding this connection.

3.2.6 Traceability

Establishing connection between requirements, architecture, and implementation is called traceability. It is important to have traceability, as it allows checking that implementation corresponds to the requirements, or that the architectural principles are followed by the implementation. Traceability comes into play when changes happen. Once the code is changed, it is possible to see whether the corresponding requirements still hold. On the other hand changing requirements via traceability indicate code fragments, which have to change accordingly. In this focus topic we investigate the role of implementation languages and their features in establishing and maintaining traceability.

We call the process of establishing traceability *tracing* often.

3.2.7 Functional Safety

The safety standards govern the use of the programming languages when implementing a safety critical system. In this focus topic we look on the safety certification from the implementation languages perspective. We investigate how various features of a programming language make a program in it easier or harder to certify.

3.2.8 Security

Security is a largely underestimated topic in the automobile software engineering, which is gaining importance rapidly. Connected vehicles become often a subject to malicious attacks. When developing a secure system new processes and tools, as well as programming languages are to be used. The programming languages should allow for higher correctness, and should make the processes connected with security easier. For example security audits can be supported and made more productive by higher modularity support of a language, as well as integrated analysis possibilities.

3.2.9 Runtime Behavior

Runtime properties of the automotive software are often the core properties of it: real-time requirements, deployment to the distributed ECUs, and parallel execution of software components. Parallel execution is especially important topic, as it is the modern way to make the programs run faster, which brings in turn the need to develop new distributed algorithms, and deal with problems typical for concurrency: data races and locks, timing constraints for a parallel program. In this topic we investigate which implementation language features are required to express and make use of the runtime aspects of the software.

3.2.10 Migration

We highlight this topic separately, as it is an important part of portability specific for the automotive domain. As hardware systems upgrade once in a generation, the software has to keep up. From the programming languages stand point modularity and independence from hardware on the one side, and the ability to capture certain hardware specific properties on the other side would allow for higher portability and easier migration from existing hardware platforms to the new one. In this focus topic we go deeper in the programming language features allowing for easier transition to new hardware platforms.

3.2.11 Testability and Verifiability

Once the properties of an implementation are captured by requirements, they could be formulated in some form and either verified or tested. Implementation language might be supporting the verification of the properties, formulation of the properties, and either more or less suitable for doing so with external means. Integrated testing facilities, verification, higher modularity are all the preconditions for a more robust software development.

3.2.12 Complexity

Cars represent modern software systems of high complexity. They support many features and interactions between them. The software starts from low abstraction layers controlling the hardware components of the vehicle and ends up with high abstraction level software in infotainment and driver support, needing to run complex data processing algorithms and user interfaces. The describe complexity is traditionally solved with modularity in software engineering. The functionality is split horizontally, in different domains and features of the car, and vertically from the top most abstractions in user interface to hardware control down the abstraction. Implementation languages can support this broad range of requirements by multiple paradigms in them, and modularity support in forms of structuring the code base in modules, classes, etc.

3.3 Challenges in Software Engineering in the Automotive Domain

Apart from the challenges and trends already given in the domain overview, some challenges that have direct impact on the choice of a proper development and implementation paradigm are highlighted in this section. These challenges are derived from the different types of systems that have to be implemented and regulations that force manufacturers to account for the functional safety of the vehicles.

Here, we describe the basic paradigms and the influence of modern model-based development techniques.

3.3.1 Continuous Control Systems

Most driving- and engine-related control tasks have a continuous nature. These processes are modeled using methods that originate from control theory and use differential equations. As digital control systems have to work in a discrete way, several methods to deal with continuous processes in the digital world have been established. This means that capturing the process output using sensors, signal processing and the generation of output reference values have to comply with the laws and restrictions of control theory such as the sampling theorem or the Nyquist stability criterion.

Usually, continuous control tasks on the ECUs are executed in a time-triggered manner to ensure the aforementioned criteria. For that, multiple tasks, sometimes across multiple ECUs have to be coordinated and synchronized. Apart from pure handwritten C code, model-based development methodologies such as provided by Matlab/Simulink are used to model and design the control algorithms. The block-based graphical modeling language allows developers to intuitively design control applications based on the data flow between the components. It also provides several methods for synchronization and discretization. Based on the fully tested model, target-specific c-code can be generated, compiled and directly deployed onto the ECU.

This well-established paradigm is widely accepted and applied in the automotive industry and has proven its reliability while providing a rather high level of development comfort. However, taking into account the rising complexity of automotive software, these paradigms have some shortcomings:

- **Traceability.** While code generation and deployment from the model is a seamless process, it is currently hard to trace directly from a requirement to a specific aspect of the model. Also, if requirements or specifications change during the development process, the developer's changes to the model cannot be traced back to requirements easily.
- **Non-functional Requirements.** While this model-based approach already covers certain timing-aspects, other aspects such as degradation or failure handling are not yet covered properly in modeling and code generation, but often have to be added manually.
- **Testing and test case generation.** Similar to the executable model code, test cases as well should be generated from the model. This way, tests would also be adapted to changes to the model during development activities. Also, model-in-the-loop tests would be beneficial prior to integration testing.
- **Integration Testing.** As non-functional properties such as proper timing are crucial for the correct behavior of control applications, these properties also have to be verified in a "real-life" environment with the component integrated in a vehicle network. The respective tests could also be generated from a more sophisticated model to ensure the performance of real-time applications.

While the current approach looks promising, there is still room for optimizations, especially with a higher number of mixed continuous/discrete applications emerging and many traditionally isolated control tasks becoming part of a networked system function.

3.3.2 Discrete State Systems

Discrete state systems are very prominent in the body and infotainment domain of the vehicle. Requirements with respect to functional safety and timing are usually not as strict as it is the case with critical control tasks. Because these systems mostly process state changes, an event-triggered model of execution is widely established.

3.3.2.1 Infotainment Domain

The implementation style in the infotainment domain is more similar to multi-purpose software applications and therefore not standardized and very heterogeneous, and heavily relies on frameworks for graphical user interfaces and user interaction. Also, the control of the audio system and access to networked services requires platforms that have more in common with consumer devices such as smartphones. Therefore, programming paradigms such as object-oriented programming and also languages such as java that require a comprehensive runtime environment are frequently used.

While these paradigms enable more efficient implementation and allow better methods of reuse and architecture engineering to reduce time-to-market, testing and verifying such applications becomes more complex. As advanced driver assistance systems as well as autonomous driving functions will heavily rely on functionality implemented in high level languages, similar requirements with respect to testability, traceability and reliability will become effective in this domain and therefore will also have to be supported by the paradigm. Also, a higher amount of standardization with stable semantic and syntactic interfaces would ease integration and continuous updating.

3.3.2.2 Body Domain

Discrete state based control systems are used to control the central locking, lights or windshield wipers of the vehicle. The implementation of such behavior can be modeled using an extension of the aforementioned Simulink package Stateflow. Stateflow supports model-based development of discrete-state event-triggered systems, including the evaluation and generation of C code.

While more uniform than with the infotainment domain, software development for state based control systems is not as standardized as with continuous control. Also, while unit-testing such components is rather easy, provided a sufficiently complete and non-ambiguous specification, the extremely high number of possible interactions and interdependencies with other functions forces elaborate manual testing and integration efforts. Such issues should be mitigated on architecture level as well as with optimizations to the current programming paradigm.

3.3.3 Mixed Systems

Applications that contain critical real-time control tasks as well as an elaborate number of states and state transitions are called mixed, or hybrid systems. This category especially contains modern ADAS as well as adaptive control functionality. While such systems can be modeled and implemented with model-based tools such as Simulink in combination with the Stateflow package, there is no established standardized approach. Also, as such systems typically consist of multiple components (software and hardware) that are provided by different suppliers. This creates additional challenges with respect to verification, testing and integration.

A rigorous approach to specify and model such systems would greatly support the efficient development of such functionality and helps to ensure reliable and safe behavior. Also, flexibility would improve, as clear and standardized interfaces would allow a wider range of reuse and modularity.

3.3.4 Safety-Critical Systems[14]

Learning from mistakes, some of them quite costly, military and space programs were the first domains to collect techniques, procedures and activities into best-practices, guides and development manuals, which evolved into binding documents for the involved industries. Such standards were not only useful as gauges for product safety, but served as guidelines for the evolution of technologies and techniques. For instance, the U.S. Navy's conformance standards for testing computer systems and components, introduced in the early 1970's and used most significantly for early programming languages such as FORTRAN and COBOL, helped shape the development of high-level programming languages[15] by identifying the challenges facing then current technologies, and led to significant convergence among the programming language dialects of the major computer vendors at the time.

In the ensuing years, various industries adopted their own guidelines and standards, often adapting those applied in neighboring domains to their specific use-cases. With origins in the process control industry, where Electric, Electronic and Programmable Electronic (E/E/PE) systems had steadily been taking over control functions from traditional hydraulic and mechanical systems, and targeting the consolidation of the many common aspects of safety engineering for E/E/PE systems, work began in the early 1990's on an international standard intended to cover basic functional safety for those systems and be applicable to all kinds of industry. The result was the IEC 61508[16], first published in 1998, and titled "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE)".

[14] Excerpts from: M. Khalil. „A pattern-based approach for the guided reuse of safety mechanisms in the automotive domain." Doctoral Thesis. TU Munich. 2015. UNPUBLISHED.

[15] U.S. Navy Archives. http://www.public.navy.mil/surfor/ddg70/Pages/graceHopper.aspx. (Retrieved March 5th, 2015)

[16] International Electrotechnical Commission – IEC61508. "Functional safety for electrical / electronic / programmable electronic safety-related systems", 1st Edition. International Electrotechnical Commission (IEC). Geneva, Switzerland. 1998.

3.3.4.1 IEC61508

The IEC61508 covers the entire safety lifecycle – from concept through development and onwards till decommissioning – and targets the reduction of risks, calculated by combining the probability of a hazard occurring with its severity, to a tolerable level. While other technologies may be employed in reducing the risk, only those safety functions relying on E/E/PE systems are covered by the detailed requirements of IEC 61508.

Despite, or perhaps due to, its general applicability, the IEC61508 foresaw the provision of being further interpreted and tailored to suit specific projects or address sector specific issues. Where sufficient need and momentum was reached, this manifested into standalone industry-specific standards, ultimately creating a landscape of IEC61508-driven or related industry-specific standards.

These standards share some common views on safety engineering:

- Zero risk is unachievable.
- Non-tolerable risks must be reduced.
- Safety has to be considered and planned from the onset.

An overview of the IEC61508 and its related family of standards is given in Figure 13.

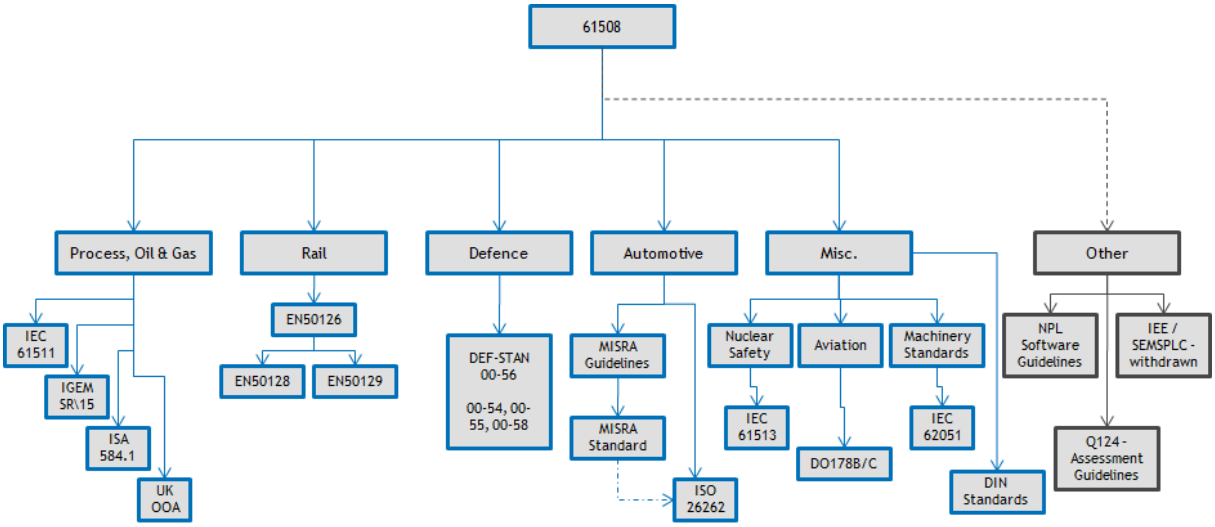


Figure 13 - Overview of the IEC61508-Family of Safety Standards (Source M. Khalil)

3.3.4.2 ISO26262 – Functional Safety for the Automotive Domain

Prior to the introduction of an automotive-specific safety standard, software development for safety-related systems in the automotive industry was primarily covered by the Motor Industry Software Reliability Association (MISRA) guidelines, such as those for C programming. Providing the first automotive industry-specific interpretation of the principles of the, then still emerging, IEC 61508, the MISRA guidelines, introduced in 1994, were conceived to provide recommendations for the development of embedded software in road vehicle electronic systems. Work on an automotive-specific tailoring of the IEC61508 continued, and

would culminate in the publication of the ISO26262[17] standard, titled "Road Vehicles – Functional Safety", in late 2011.

The ISO26262 standard includes ten parts covering all activities and defining roles and responsibilities during the entire safety life-cycle, as well as recommending techniques and methods for development, and warning against known pitfalls. It addresses electrical, electronic, and software elements performing safety-related functions, as well as hardware requirements emanating from these components and the specification of the hardware-software-interface (HSI). Similar to other standards, it provides requirements for confirmation measures to ensure an acceptable level of safety is achieved.

3.3.4.3 Development and Design Techniques for Safety

It is important to note the difference between defensive programming, which is an implementation technique; modular architectures, which is a design trait; and redundant deployment, which is an architectural pattern. The first two belong in this section, the latter in the following section.

Many of the development and design techniques included in this category can be found in "best-practices" or standardized development guidelines, and are also listed inside the respective sections of the safety standards. For instance, the ISO26262 recommends the use of language subsets, strong typing, defensive implementation and unambiguous graphical representations, as well as the use of style guides and naming conventions and the enforcement of low complexity in its implementation guidelines (Part 6 – 5.4.7). It further specifically warns against using dynamic objects, global variables, unconditional jumps, implicit type conversions, recursions and hidden data or control flow (Part 6 – 8.4.4). If one is implementing in the C programming language, then such recommendations are covered by the MISRA C guidelines[18], which can be applied by code checkers to ensure conformance to these rules; a fact highlighted in the ISO26262 standard.

On the other hand, the IEC61508 states "*From a safety viewpoint, the software architecture is where the basic safety strategy is developed in the software*" (Part 3). The ISO26262 – (Part 4) expounds on this, stating that first and foremost, architectural design should be modular; display and adequate level of granularity; and be simple.

[17] International Standards Organization – ISO 26262. "Road Vehicles – Functional Safety." www.iso.org, 2011.
[18] MISRA-C:2004. "Guidelines for the use of the C language in critical systems", ISBN 978-0-9524156-2-6, MIRA. October 2004.

It goes on to specify which properties of modular design are necessary for each ASIL, as seen in the following table. (+) denotes recommended actions, while (++) denotes highly recommended.

Properties		ASIL			
		A	B	C	D
1	Hierarchical design	+	+	++	++
2	Precisely defined interfaces	+	+	+	+
3	Avoidance of unnecessary complexity of hardware components and software components	+	+	+	+
4	Avoidance of unnecessary complexity of interfaces	+	+	+	+
5	Maintainability during service	+	+	+	+
6	Testability during development and operation	+	+	++	++

Table 1: Properties of modular system design for ASIL – ISO26262 recommendations (Source ISO26262)

4 Examples from Other Industries

Highly interconnected functionality, robust operation and high performance while processing large amounts of data – those challenges already arose within other domains such as business information systems or the avionics domain. While the developed solutions to tackle those challenges are highly specific and cannot be naively transferred to the automotive domain, some aspects can prove helpful when developing automotive software development methodologies further.

4.1 Business Information Systems

Business information systems are large software systems that support all business processes of a company, including accounting, sourcing, sales and distribution as well as reporting. Depending on the complexity of the processes and the size of the company, these systems have to serve tens of thousands of concurrent users, ensuring reliable execution of all transactions and maintain high availability. Traditionally, this domain is dominated by middleware platforms such as Java Enterprise, Microsoft .NET or SAP and consulting companies that provide company specific solutions for the individual process requirements. These companies also develop and control the quasi-standards driven by their products.

A main trend in this domain is the transition from static product development towards more flexible, “agile” development processes, enabling shorter development cycles and product iterations. This is supported by test-driven development methodologies that focuses on test-driven specification of functionality. This way, it can be automatically ensured that all parts of the product still meet all defined requirements after changes. Globalized, highly competitive market situations lead the companies to a state of permanent organizational changes to adapt to new challenges. The companies’ information systems have to reflect those changes in adapted functionality. In many environments, thinking in fixed product cycles has been replaced by fast product iterations that are continuously delivered and deployed. Technologies such as test-driven development ensure that there is no trade-off between quality and implementation speed.

For a longer period of time, the market for business information systems was controlled by a few vendors and consultancies specialized on the vendor-specific platforms. Recently, with the introduction of new data-driven applications or settings that have to support a high number of mostly passive clients, new flexible approaches and technologies such as server-side JavaScript were introduced that drive the heterogeneity of this domain which is expected to increase further in the future.

We now present two developments in the automotive domain that have similarities with developments that already occurred with business information systems, namely processing

and analyzing high amount of data and the development of fail-operational software-based functionality.

4.1.1 „Big Data“ processing and analytics

Typical data exchanged in vehicles used to be limited to simple time series of control values and state information. That was sufficient to control the different functions as long as complexity as well as the level of interaction between the functions was limited. Today, the amount of data produced by a modern vehicle is enormous. Future connected vehicles are expected to transmit up to 25 Gigabytes per hour[19]. In the next years, numbers will grow from 4 Megabytes to 5 Gigabytes per Month[20]. Today, the amount of information exchanged within the car easily exceeds this number. Analyzing this data could not only provide valuable information about the vehicle's performance or upcoming maintenance tasks, but can also be used to evaluate the behavior of vehicle functionality and find new ways for optimization. Especially autonomous behavior cannot be evaluated and tested using the same methods as for typical vehicle control applications, as the individual reaction of an artificial intelligence depends on all aspects of a driving situation (e.g. entering a roundabout, turning on a busy intersection).

This "Data-Driven" way of developing and improving a product is new to the automotive sector, but very common in business information systems where business intelligence solutions are employed to continuously monitor and evaluate existing processes to check for compliance to company policies and to discover potential for optimizations.

4.1.2 „Fail-Operational“ Behavior and Redundancy

Although business information systems are nowhere as safety-critical as automotive applications, availability and reliability of services are of extremely high importance as the financial impact of errors can be immense. Therefore, multiple mechanisms were established to provide redundancy and scalability such as efficient replication of databases, computing clusters with transaction safety and centralized control over large datacenters providing elastic computing capabilities. Services and hardware nodes can be added or removed from the system while the system is online as most components are designed to deal with such a dynamic environment.

While "fail-operational" in the automotive sector is dedicated to provide reliability of safety-critical applications such as the braking system which is currently achieved by mechanical measures (physical redundancy, separated brake circuits), upcoming software based functionality, especially for autonomous control of the vehicle, can benefit from methods developed for business information systems as these are proven to provide reliability and perfor-

[19] Guilherme Miguel Taveira Pinto: The Internet on Wheels and Hitachi, Ltd., 2014

[20] Deutsche Telekom: Connected cars get big data rolling, 2013

mance. However, adaptions have to be made in order to properly address the specific needs in the automotive domain.

4.1.3 Summary

Unlike the automotive and avionics domain, software development for business information systems focuses on fast, efficient implementation of new functionality and maintainability. Because business processes are subject to frequent changes and adaptions, software systems supporting those processes need to provide equal flexibility when it comes to modifications. On the software architecture level, this is achieved by a high level of standardized components and interfaces.

However, new requirements in the automotive domain are similar to those that drive business information systems, so the methods established here can be a valuable source for solutions.

4.2 Aerospace Industry

Just like for the automotive industry, many techniques exist in the avionic industry depending on the considered subdomain (e.g., control, infotainment) and on the company. Some might program all the code manually while others use Model-based development heavily. In the first case, the paradigm does not particularly differ from other domains (except when it comes to taking into account the certification processes, see next section), with the notable exception of Ada usage. We will therefore more focus on Model-based development.

4.2.1 Ada

Ada is famous for emphasizing many safety and maintainability aspects directly in the language: it is strongly typed and is very strict about the rights in a program (the notion of module is unavoidable and provides a good abstraction between the various parts and concerns of the code). In practice however, the language does not seem to encounter a big usage: industry practitioners complain about the lack of trained workforce, about the cost of Ada compilers, as well as the difficulty to deal with two languages in a same company (since C is still unavoidable for legacy reasons). Two lessons can be taken out of this case to ensure a successful introduction of a new programming language/paradigm:

- The new language shall introduce a smooth transition with existing languages (typically C),
- The new language shall deeply integrate interoperability with other languages.

4.2.2 Matlab/Simulink

As in the automotive industry, Matlab/Simulink is the tool par excellence for MBD: the architecture of the software is designed using Simulink and refined until a level of detail where code can be used. Code is generated out of the Simulink architecture and the low-level implementation is done manually. The common threat of modifying generated code and therefore using model-based development only for prototyping is avoided since the low level behavior is implemented manually (and not generated) and since the generated, "architectural", code is stored separately from the manual implementation.

This approach actually covers different methodologies depending on the level at which one decides to switch from the Simulink models to the manually implemented S-functions: depending on the company processes or on the will of the developer (familiarity with modeling or programming) one can prefer to delay or, on the contrary, to switch earlier to the manual programming.

4.2.3 SCADE

Famous for being the only tool whose code generation has been qualified (i.e., the code generated by SCADE is automatically considered as certified) for DO-178B and recently (Jan. 8th 2015) for ISO26262, SCADE[21] is used only by big companies (due to its cost) in a way similar to Simulink: a SYSML-like model-based design approach is used and code is generated out of it. In this context however all the code is generated using SCADE: there is no "switch" from generated to manual code. This does not mean that all the software is done using SCADE but that if a software component is designed using SCADE then it should be designed completely with SCADE. This entails that the SCADE method seems generally not to be used as a whole for a complete software.

The approach imposed by SCADE is questioned by many practitioners: the cost of development and of qualification, and therefore of the final product, is extremely high – maybe too high to be worth it. Practitioners advocate more flexible solutions: instead of imposing a tool like SCADE during the development process, a possibility would be to use any tool (e.g., Matlab/Simulink) during development and using separate tools to check the generated code. Such tools would have the advantage to be usable in many more contexts thus amortizing their qualification cost.

Though appealing because of its pragmatism, this solution is however much more restricted: not controlling the development process also entails that a lot of information is not available to the "checking tool". This is actually precisely the point of model-based development in general (and of SCADE in particular) to make use of some higher-level information in order to provide a better insurance that the code is correct.

[21] http://www.ansys.com/Products/Simulation+Technology/Systems+&+Embedded+Software

4.2.4 Impact of Functional Safety and Certification

In all cases, when it comes to development techniques, a major distinction is the impact of safety and consequently of the certification process. The domain standard DO-178B (or its recent successor DO-178C)[22] is more constraining than the ISO26262: independently of their many structural differences, the main impact of DO-178B on the development process is the necessity that every artifact should be *requirements-based*: this imposes *traceability* from the requirements to the design artifacts and to the code. This impacts the development process as follows:

- when using code only: comments with references to requirements have to be added (potentially significantly increasing the size of the source code files). This shows how a process constraint actually imposes changes at the programming level.
- when using Matlab/Simulink: the DO178 toolbox of Simulink allows to add traceability information to the models. This information is then also exported when generating code. Consequently, the process constraint imposes changes at the modeling level as well.
- when using SCADE: the traceability information can also be added to the models. However there is no need to export this information in the generated code since the generated code is automatically certified and therefore is not part of the safety case.

Still, in all cases, the object code itself must be tested (with traceability to the requirements).

By itself the DO-178C standard provides a clear structuring of the following phases : high-level requirements, low-level requirements, design, and implementation. However, just like in the automotive industry, these phases are essentially assigned to the safety case artifacts only once the complete development has been done in order to provide the certification authority with the necessary information. It seems however that more and more companies are really trying to put these phases to practice through processes: for instance, every phase (e.g., high-level requirements) must be internally validated before proceeding to the next one (e.g., low-level requirements).

4.2.5 Testability and Verifiability

As for code, tests must be requirements-based, therefore traceability is also essential there. It is implemented in the same way as for normal code. In a model-based environment like Matlab/Simulink tests are also “written” in Simulink for software-in-the-loop testing. Traceability of tests can therefore be applied also just as mentioned above using the DO-178 toolbox, i.e., by attaching requirements information to the models.

[22] DO-178C, Software Considerations in Airborne Systems and Equipment Certification (2011) by Special C. of RTCA

The DO178 toolbox of Matlab/Simulink as well as SCADE also claim to provide formal verification. It seems, however, that they are not used in practice due either to complexity of usage or to performance[23].

4.2.6 Security

Just like with the increase of connectivity in the automotive industry, the increase of software in the avionic domain raises the necessity to tackle security threats. The need for connectivity is however not as high as in the automotive industry: the utter need for connectivity in the automotive industry arises essentially from the will of going more and more in the direction of completely autonomous driving. On the other hand, in the avionics domain, piloting functions are and will remain probably for a while, if not forever, in control of professionals. Auto-pilot functions exist, but they have less need for connectivity than in the automotive domain: the fleet of planes is in no way comparable with the ones of cars driving on a highway, the "road" is much less restricted thus reducing the need for remote knowledge, the only possible congestions are at the arrival in an airport where the piloting remains between the hands of the pilot. More generally, the higher risk in case of accidents makes the avionics industry much further away from the automotive industry when it comes to security.

Still, researchers are starting to attract attention towards this problematic in the avionic domain as well: recently (Apr. 19, 2015) Chris Roberts, a security researcher, managed to connect, through the infotainment system, to the Internet (which he proved by publishing a message on Twitter) thus demonstrating the possibility to access to functions which were normally disallowed for passengers[24]. Previously, even though not through the infotainment system but through a physical, disallowed, connection to the software system, he even had briefly taken control of another airplane engine[25]. Such events being triggered only by security researchers, the risk induced by security on an airplane is for now still not considered as high. In particular, the measures taken were simply to forbid Chris Roberts from flying but have not had, so far, any consequence on regulation and/or on programming practices.

4.2.7 Programming Paradigms in the Aerospace Industry

By many aspects programming in the aerospace industry is similar to the avionics industry: model-based development through Matlab/Simulink is also heavily used in conjunction with manual implementation. However the certification process is very different here: there are

[23] Basold, H., Günther, H., Huhn, M., & Milius, S. (2014). An Open Alternative for SMT-Based Verification of Scade Models. In *Formal Methods for Industrial Critical Systems* (pp. 124-139). Springer International Publishing.

[24] http://arstechnica.com/security/2015/04/researcher-who-joked-about-hacking-a-jet-plane-barred-from-united-flight/

[25] http://www.wired.com/2015/05/feds-say-banned-researcher-commandeered-plane/

no “passengers” in a satellite, and the passengers of a shuttle are professionals or military and not customers. This yields much lower restrictions when it comes to safety and certification. Therefore traceability is not that important and, consequently, the usage of tools like SCADE is of no value. This also means that all sorts of fancy techniques are theoretically available to the programmer (e.g., dynamic allocation, reflectivity). On the other hand, the cost of failed mission is so high that the correctness of the design is as essential. The only difference is that the criticality of the system is not determined by a certification process and/or authority but simply by the necessity to design a system correctly. At the programming level, NASA has for instance defined programming rules meant to ensure the correctness of critical code[26]. At a high level of abstraction, these rules can be summarized as follows[27]:

1. All code shall be restricted to simple control flow: e.g., recursion should not be used.
2. Every loop must have a trivially-provable upper bound.
3. Dynamic memory allocation should not be used after initialization.
4. Functions should be small, typically 60 lines of code per function.
5. Assertions must be used at a minimum of 2 per function (on average).
6. Data objects must be declared at the smallest possible scope.
7. The parameters of each function shall be checked at the entrance of the function and the returned value shall be checked by the caller function.
8. Pre-processor must be restricted to trivial usages (file inclusion, simple macros).
9. Pointers must be avoided as much as possible (no function pointers).
10. All compiler warnings must be turned on (and no of course warning should display when compiling)

These are however only recommendations but nothing makes them mandatory in the complete industry. Actually, many practitioners use C++ which, per se, can be seen as contradicting rule 1: as in all object-oriented languages, inheritance makes the control flow sometimes very hard to know statically since it depends on the class of the current object under consideration. For this reason, one might consider a further rule for object-oriented languages: every class should be final by default, thus preventing control flows which were not wanted by the original developer of the class to be implemented. Similarly, the mechanism of overloading which automatically comes with inheritance is generally implemented through systematic and heavy usage of function pointers which are forbidden by rule 9. However as the

[26] Gerard J. Holzmann. The Power of Ten – Rules for Developing Safety Critical Code. *NASA/JPL Laboratory for Reliable Software Pasadena, CA 91109*

[27] Adapted from: https://jaxenter.com/power-ten-nasas-coding-commandments-114124.html

author of the above rules suggests, these rules should not be seen as absolute and have to be reasonably taken into account depending on the criticality of the targeted functions of the software under development.

5 Survey

To verify our intuition about the most prominent obstacles on the way of modern automotive software projects we conveyed an on-line survey among the participants of the project. The survey was hosted at our own premises and was based on the free and open source LimeSurvey software. You can find instructions in the Appendix II on how to use the whole survey data set which we ship with this report in a number of different formats.

In this chapter we highlight the most significant findings of the survey and interpret some of the answers based on our experience.

5.1 Structure and Questions

The majority of the questions in the survey were optional to answer multiple choice questions. This is how any of the options might have received up to 87 votes. The sum of all votes might be greater than 87, as multiple choices are possible. And on the contrary the total of votes might be below 87 as well, when people decide not to answer on a particular question. To see the exact numbers it is possible to review the numbers behind the diagrams presented here by navigating to the Excel tables embedded into them.

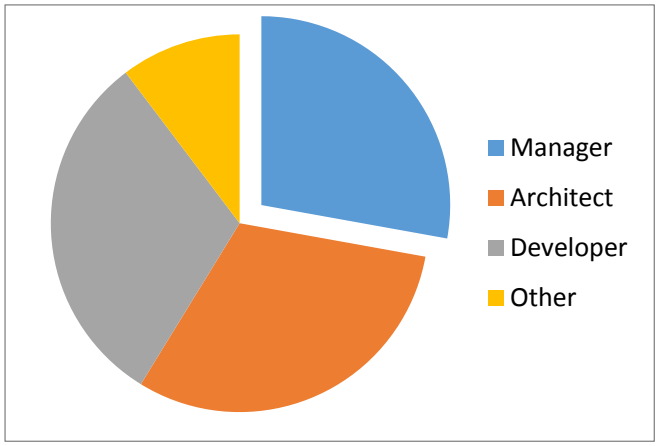


Figure 14 – Roles of the surveyed specialists

5.2 General statistics

In this section we provide the statistics common to the all focus topics listed below. This is the general information about survey participants including the fields they are working in.

5.2.1 Demographics

In the online survey held we got a total of 87 individual responses from employees of 9 companies. To protect privacy of the participants and help them provide the information in an unbiased fashion we did not include in the survey any personal data at all but the roles in the companies and the employment duration. We did not ask for names, contact data, employer and similar so that the people personalities stay hard to identify.

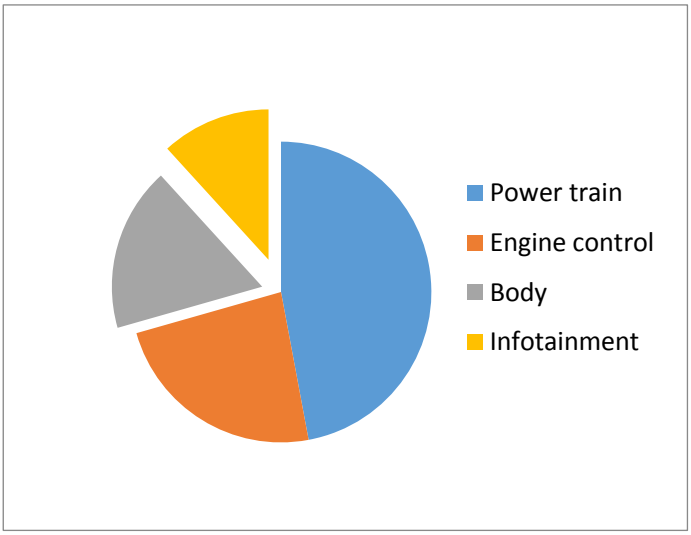


Figure 15 – Domains the specialists work on

We got people close to code to answer the questions, they work however in different roles, see Figure 14. Arguably only the pure managerial roles do not interact with code directly, although being development project managers even people in these roles can judge on the programming languages and their usage in the project.

5.2.2 Domains

Inside the automotive industry various subdomains vary greatly in the requirements they present to the systems developed. The power train and engine control domains predominate others in this survey, Figure 15. Thus the accent generally is put on the real-time systems.

Two other domains are body and infotainment. It is common that one system of a car belongs to many domains at the same time.

The diagram on the right shows the answers on the question whether the last project under development was a cross-domain project. The answers vary from 1 meaning that the project is strictly isolated to one domain and do not interact with others, to 5 meaning that the system surely interacts with other domains and cooperates with other systems extensively. The absolute majority of the systems heavily interact with other. There are no projects without cross-domain interaction.

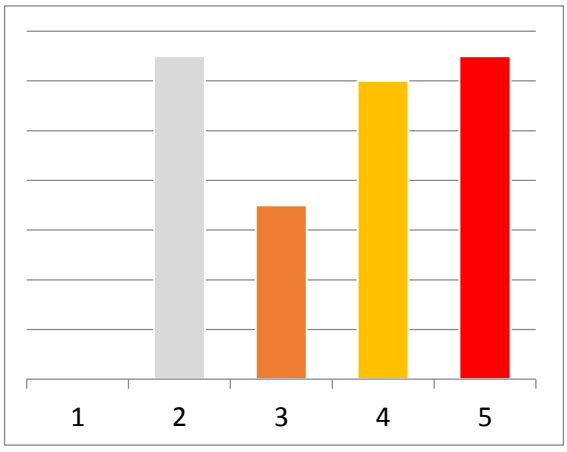


Figure 16 - Is it a cross-domain project? (from 1 = isolated, no interaction with other domains, to 5 = surely interacts with other domain)

5.2.3 Main Challenges

Before going to describe the technologies used in the projects we show the survey results when answering the question: What was the main challenge while working on the project?

Figure 18 provides the answers. 1/3rd of the activities have to do with the source code and programming languages directly.

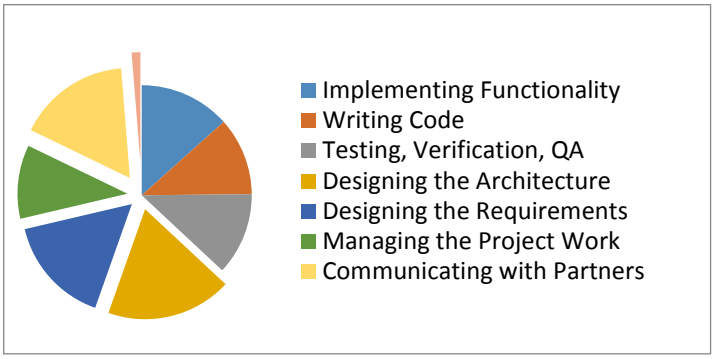


Figure 18 – What were the main challenges in the project?

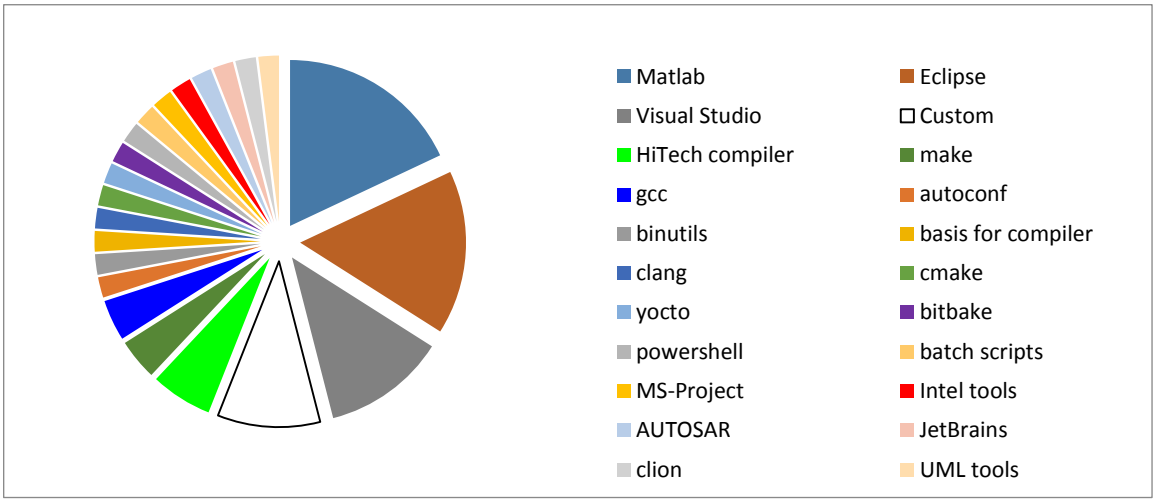


Figure 17 – Which tools and technologies do you use?

5.2.4 Tools and Technologies

It is remarkable, that the companies use a large variety of tools and technologies, Figure 17. They have proprietor and open-source origins as well. Matlab and Eclipse-based tools start the list being the most popular followed by Microsoft Visual Studio.

The fourth place in popularity is taken by custom, in-house-built tools. This might mean that the companies could already at least partially have the resources in-house, necessary to build custom programming languages and infrastructures for them.

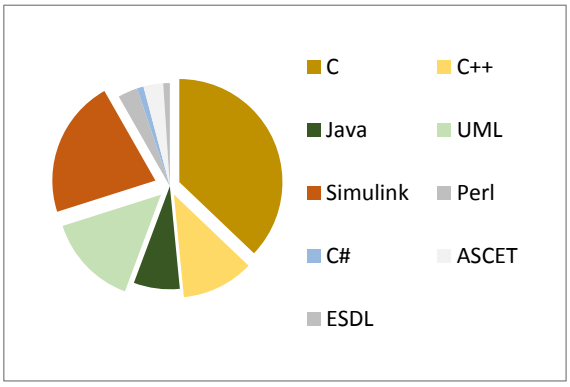


Figure 19 – Languages in automotive industry

Speaking of programming and other languages used in the software development in automotive we have a strict domination of C, C++ and Simulink, Figure 19. Perl and Java are often used for tools. The overall portion of modern languages is extremely low.

C, a technology from 60s, is the absolute champion-language in modern automotive industry.

5.3 Answers Accumulated per Focus Topic

In this section we talk about specific answers related to focus topics. As a reminder: the focus topics are the main concerns of the automotive software manufacturers who have taken part in the survey.

The survey asked to prioritize the focus topics. We assigned the scores for the places in priority list like this: 5 points for the 1st place + … + 1 point for the 5th place, 0 for less. We pay more attention in this report to higher prioritized topics. Results are in Figure 20.

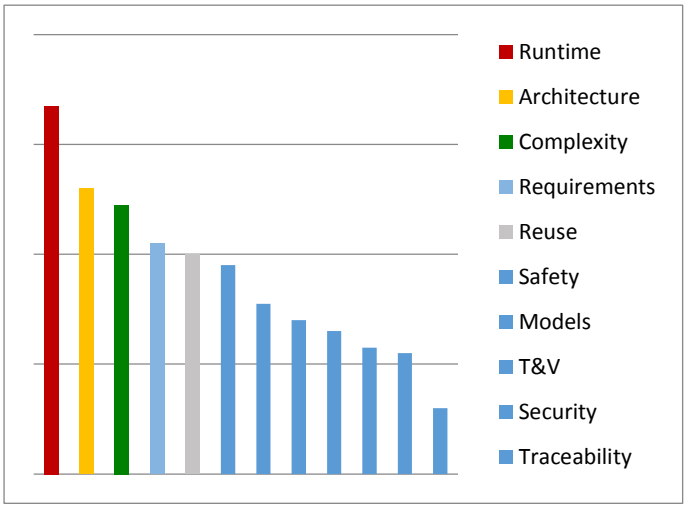


Figure 20 – Focus topics sorted by importance

Below survey details are discussed per focus topic.

5.3.1 Runtime Properties

Parallelism is widely demanded but rarely supported in the automotive programming languages:

- 80% of specialists deal with parallelism and consider it important.
- However, up to 82% have no special constructs for it in the language

Real-time constraints suffer from the similar situation:

- At least 77% of specialist have to deal with real-time constraints
- And up to 72% have no special constructs for it in the language

Overall 39% are unhappy with the used languages for these reasons. And 0% are completely satisfied with the existing solutions.

Based on this information we conclude that the area of run-time properties is the one demanding innovations the most.

5.3.2 Architecture Enforcing

Before the architecture could be enforced, potentially with the help of tools, it has to be expressed in some format and managed, Figure 21.

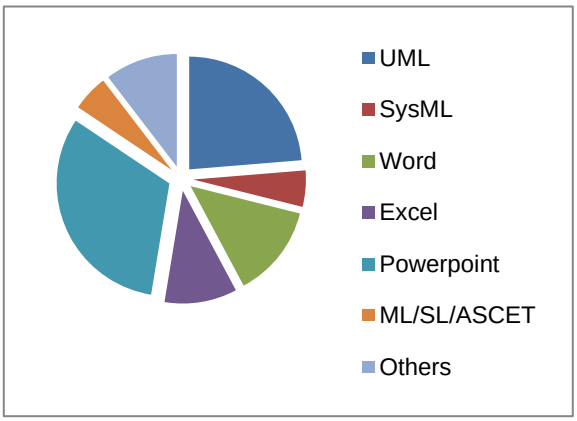


Figure 21 – Architecture formats and tools

Unfortinately architecture information is stored in unstructured formats disabling precise interpretations and tracing. Another problem is the absence of a unified tool for the situation when several teams collaborate.

As a result the architecture is produced, but not enforced very well: 53% answer on the scale from 1 to 3 (1 - poorly, 5 - very well enforced).

Tracing the architecture also suffers: 65% from 1 to 3, same scale as before.

5.3.3 Complexity

From the survey we are able to derive the reasons for high complexity and diagrammatically present them here. General complexity reasons are the issues, which are hard in all the other industries in software development. The methods to cope with them must not be new and unique for the automotive domain.

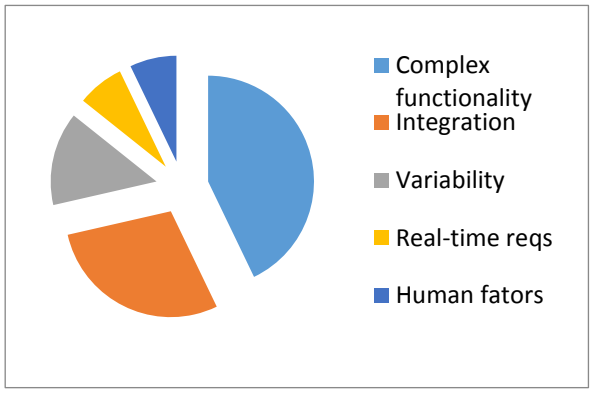


Figure 22 – The reasons for complexity

Figure 22 shows the main complexity reasons and variability is probably the only one quite special for automotive industry in particular.

Next we classify complexity into two kinds:

- Hardware-induced and
- Language induced.

The hardware-induced complexity, Figure 23, arrives from the specific hardware choices and could potentially be improved upon when tackled from the hardware side.

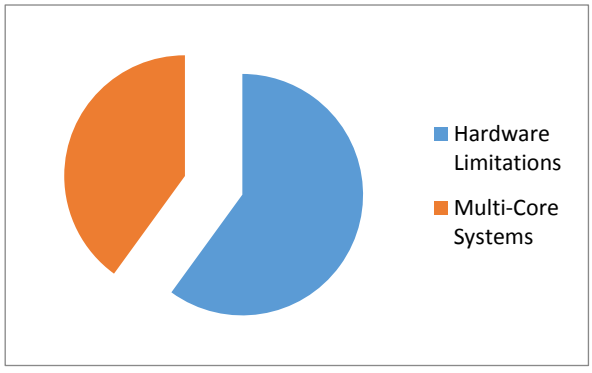


Figure 23 – Hardware-induced complexity

We suggest that the hardware-induced prob-

lems are to be solved on the company-wide basis, when the hardware costs and software costs are estimated together including the estimations for hardware maintenance when and growing complexity due to the limited resources.

The next class of problems is purely implementation-language induced. They are the main focus of this project. The lack of support for encapsulation, architecture and mode-based programming are the main obstacles the popular languages in automotive deliver.

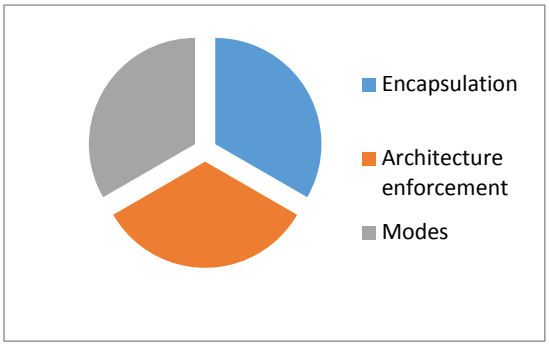


Figure 24 – Software-induced complexity

The software complexity is very high, and we support this by more facts from the survey.

Answering the question:

- "How much time is needed by a new team member to fully join the development?"

68% of the responders assume 6 months or more!

Additional facts:

- 36% do not have documentation built-in in the code
- 65% can't say that the language chosen helps to describe modularity
- 27% tell there is no way to describe interfaces or blocks
- Only 13% are happy with the way the complexity is dealt with

This happens despite the fact that platforms are used extensively. For instance, AUTOSAR is used in 82% of projects and OSEK in 55%.

We proceed to ask the specialists to review the language features:

Which language features support you in dealing with complexity? Which language features are lacking?

- *We use pre-processors*
- *Matlab - variant management and C - pre-processor/compiler directives*
- *Function and classes in C++, C++ templates*
- *To reduce complexity, you should use proper data structures but these aren't a language feature.*
- *Lacking: object orientation*
- *For tooling, the language is OK (Java). But I wouldn't use or trust Java for anything safety or security critical. You need Ada for this.*

These answers basically lead us to the thoughts of multiple domains and different ways to reduce complexity. The need for modularization is encapsulation is prominent. Tool support

and language extensibility are low and poor-man's dangerous language engineering is used (macros).

5.3.4 Requirements

Projects have various kinds of requirements in the automotive.

Functional requirements prevail among the requirements types, Figure 25, especially if we count the real-time requirements for functional ones. Quality, Certification and Robustness complete the picture.

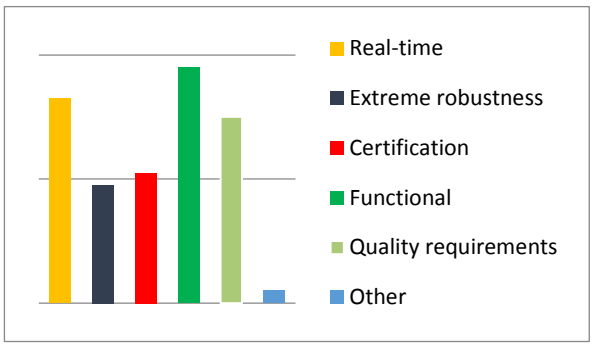


Figure 25 – Types of requirements

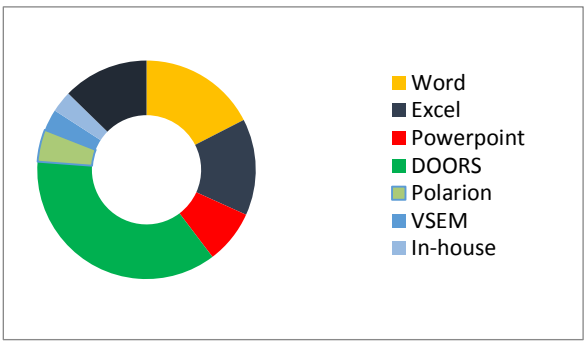


Figure 26 – Tools used for requirements management

Various tools are used in order to engineer and manage requirements, Figure 26. The Microsoft tools prevail here not being special tools for requirements engineering. They are followed by IBM DOORS. Tracing is a not resolved to the full satisfaction issue common to these tools.

We ask a general question first:

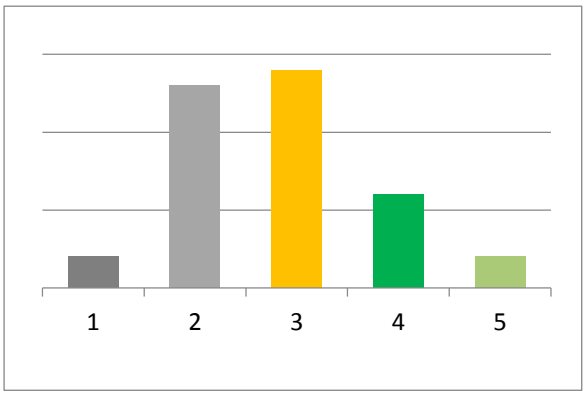


Figure 27 – Satisfaction with tools for requirements

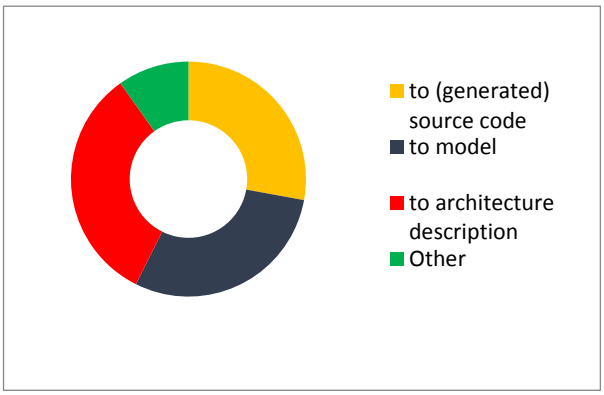


Figure 28 – Tracing requirements

- "How well do these tools work for you?"

The answer is given on Figure 27. This might explain the reason why in-house tools are built. In the end of the chapter on language traits we discuss some issues with language infrastructure development. It partially applies to other home-grown tools as well.

Tracing requirements is the major use case for requirements, Figure 28. Here one could see that tracing is used in multiple direction and some tool unification might be needed in order to support the all the needs harmonically.

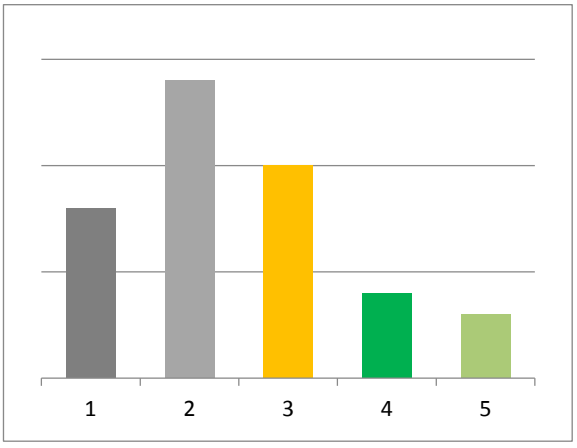


Figure 30 – Tracing requirements to source code

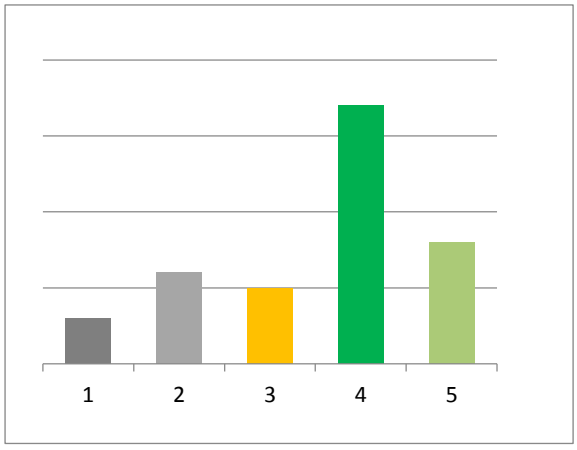


Figure 29 – How well is source code actually reused?

Unsurprisingly, as the tools not supporting tracing are mostly used we get negative response on the following question:

- "How well can you trace the requirements to and from the source code?"

This situation is not optimal, Figure 30. Not being able to trace requirements it is hard to ensure that the existing code is in fact required and that all the requirements are satisfied. Requirements management and tracing are mainly tool problems. Still, in the later chapters we approach this problem from the implementation language perspective as well.

5.3.5 Reuse

Reuse is mainly happening because

- many hardware and software platforms are used
- family of related products are developed and
- migration is happening time to time to new platforms

No matter the specialist claim reuse to be well-supported by the programming language used (vast majority votes in favor of positive options), many features are missing from the programming languages on which reused could be based. Still, specialist indicate good reuse factor, Figure 29. We would be sceptic about this opinion and would recommend an external study/audit to determing actual reuse quality.

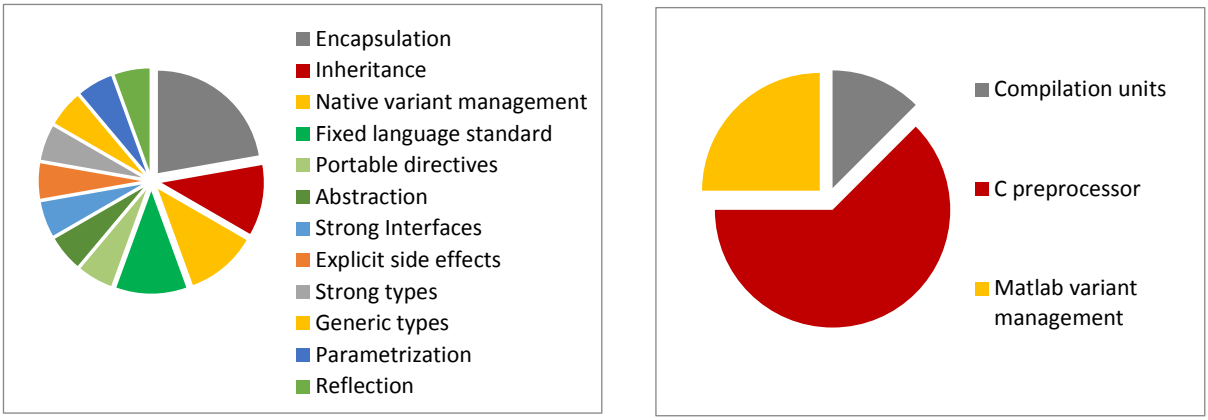


Figure 31 – Left: missing features, right: available features – to support reuse

A quick look on Figure 31 suggests that the great demand for the language features to improve reuse opens up a large innovation field, but also indicates potential great reuse problems in existing code bases.

Modularity, including OOP, and generic programming are the most missing features.

5.3.6 Safety

Figure 33 tells us that the majority of the projects/products are safety-critical.

In contrast to this Figure 32 shows us that the language support in the implementation languages is very weak.

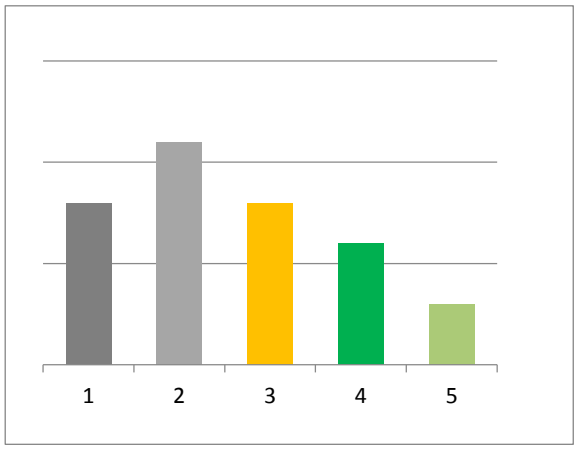


Figure 32 – How well does your language support the safety development?

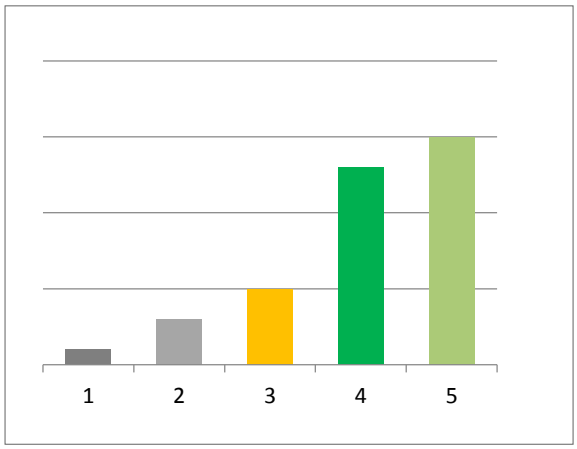


Figure 33 – Safety requirements in a project: "Does your project/product deal with safety requirements? ", 1 - not at all, 5 - all the time.

To compensate on the weakness of the language other methods are employed:

- Tools are used heavily
- Static analysis is performed
- Lint-like tools are standard
- Coding guidelines are written
- Programming standards required

It is important to understand that under formal verification sometimes a formal process of verification is meant. It rarely has to do with formal methods struggling still do break out of the university doors.

Later topics we try to cover only briefly due to their lower prioritization. Please, refer to appendix and addendum if you would like to get the full review data and answers to questions.

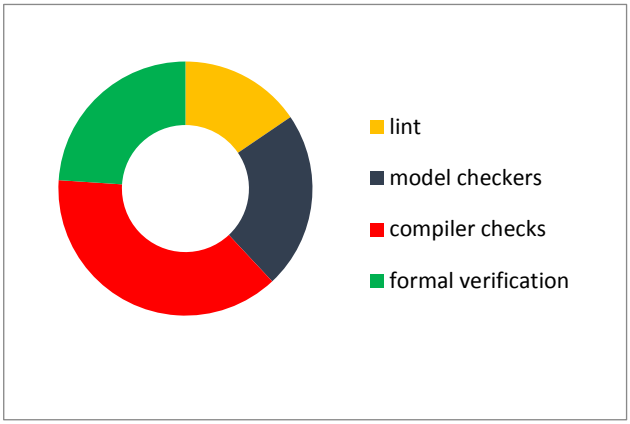


Figure 34 – Tools for safety

5.3.7 Testability and Verifiability

As automotive is a domain with the demand for highly reliable software testing and verification are happening on a large scale. Quality checks are performed in every project, and 22% of projects consider quality checks of paramount importance.

Manually written tests (40%) and applying light-weight standard and syntax controlling code checkers (36%) are the most common practice. Timing analysis and similar heavy verification techniques are performed rarely. Mostly people are moderately satisfied with the state of the practice in testing.

Under formal verification automotive specialist, dislike academics, understand often bug finding techniques like Polyspace analyzer or manual review with a formalized process. Usual obstacles on the way of real formal verification resource and time consumption, user-friendliness and availability of experts are present in the automotive industry as well. Languages used make it generally speaking harder to perform testing and verification.

5.3.8 Security

Security is considered to be important to a great number of projects (33%). 17% of systems are connected to the Internet. It is claimed that 29% of the systems could become a subject for an attack indirectly, even not being connected themselves.

In 20% of projects there are no security audit practices employed, which is very alerting to our opinion.

The degree of toolification and automation for security is extremely low as well.

5.3.9 Traceability

Once a change is happening to the architecture description or to the requirements less than 6% of all the projects can trace very well down to the source code the change. 18% are of projects are practically unable to perform such tracing. We considered tracing in the context of requirements and architecture enforcement above as well.

5.3.10 Cross-Domain Development

We have considered the domains of the participants and the need for cross-domain development above in the demographics and projects part. 30% of all projects are cross-domain, which means according to the results of this study, that:

- Many projects should be implemented in more than one language at once
- Multi-layered cross-domain programming and orchestration are very important topics for the future research

Mechanisms to cope with cross-domain development while providing domain-specific development paradigms are focused in future activities proposed in the conclusion.

5.3.11 Migration

Migration to a new platform happens not very often (30% of projects claim it to be rare in fact). A software re-write is happening however in 30% of cases which is very often but has to be measure with the scope of the rewrite as well. 32% of projects migrating have to only reimplement a small portion of the software from scratch. 6% reimplement everything. The reuse focus topic and our recommendation might improve on this situation. Right now 11% of participants are not happy with the way migration is performed.

5.4 Summary

The survey highlights many important issues in the automotive software development. The main outcome for us is the need to update technologies. The new tools are built intensively in house. Suboptimal tools are used often which are not specialized enough for the purpose.

We think it could be beneficial to perform the technologies update in a unified and potentially open-source fashion to enable cross-manufacturer compatibility, competitiveness and maximal freedom.

6 Potential Improvements to the Languages

In this chapter we are trying to suggest some features for the programming languages which might be of use in the automotive in the future. We shortly discuss compatibility of these features with each other in Chapter 6.2 and features appropriate for a given subdomain in Chapter 6.3. Finally, we briefly point out that a move towards a more appropriate language could actually happen in practice only after the infrastructure for the new language has been developed. We describe which obstacles could appear on the migration path in Chapter 6.4.

Please note that the statements in this chapter, whilst developed corresponding to the state of the art to the best of our knowledge and our best intents, are our attempts to predict a better future of the automotive PLs. Therefore these recommendations bear a speculative nature. Before any of the suggestions given here are taken into consideration in the real world development, a plan to update the languages and tools has to be separately developed. This plan has to include estimation on costs and risks of such a move. In this report we focus only on the nature of implementation languages and do not claim to give any business advice.

6.1 Structured Feature Suggestions

Often problems arise from the inability of the programming language to *directly* express some notions which developers operate with. These notions then stay implicit in the code and end up in the documentation in the best case. Generally speaking, we argue here for direct and explicit support for the notions used by developers in the programming languages. Language constructs are to be introduced for such notions.

To be as practical as possible we structure the content of this section based on the survey results. We order features per focus topic and sort focus topics by their importance based on the survey results and their relationships. Next we use the answers in the survey to identify the most critical spots and address them in the subsections below. We illustrate our ideas with examples.

6.1.1 Architecture

We start here from one of the most critical topic in the domain. Calling it generally “architecture” what we actually mean is the ability to express, enforce, preserve and trace the architecture planned initially on the level of a programming language. We do not discuss here any properties of the automotive architectures themselves, but rather the properties of implementation languages necessary to support explicit expression of the architectural decisions.

There are multiple kinds of architecture. Static component architecture might describe components and the dependencies between them. Dynamic architectures might describe the way software components are instantiated, deployed and communicate to each other.

6.1.1.1 Software Modules

Component architecture focuses on structural software entities. Components could be software parts on the highest granularity somewhat similar to modules. Clear support for modules and dependencies between them in the implementation language is crucial to achieve the necessary degree of granularity, enable reuse, and explicitly describe dependencies between software parts.

Let's take an example of the C programming language[28]. It is not supporting modules and implementation .c files are replacing them. Their interfaces are declared in header .h files.

Example 1. C files instead of modules.

module1.h

```
#include <some_header>
int f()
int data; // declaration
```

module1.c

```
#include <module1>
#include <another_header> //dependency to module declared
int data; // definition
int f1(){
      return f2(); // f2 is the actual dependency
}
```

There are a number of weaknesses the C language cares along its history, which could be improved in a better language. Let's consider first the split into two files itself. It has a number of disadvantages compared to one-file-per-module way.

- The files have to be synchronized (declarations have to match definitions)
- Some of the dependencies appear only in the .c file
- The inclusion is happening textually
- Exact inclusion sources are build settings dependent

Once a module has to be changed, the modification has to be reflected in two places. This is an unnecessary redundancy.

Next, the header only defines the dependencies, necessary to use it. The whole picture of dependencies is more complicated: the source is using some another module in order to

[28] We simplify the syntax here keeping only the relevant parts

function. The exact dependency nature, that `f2()` is used in our example, can only be extracted if the whole source is considered. Depending on the build settings the actual inclusion of modules may happen from various folders, so the implementation files do not capture the dependency precisely enough. The declaration and definition split for the shared data is also confusing, it might end up with multiple or zero instantiations in the run-time if inclusions are done wrongly. These problems could be avoided once a clear notion of module and its interface, together with dependencies, is introduced into the language.

Example 2. Introducing modules in the implementation language.

Architecture description:

Module1.architecture

```
provides int f(), int data;
requires module2.f2(); //clear definition of the dependency
```

Module1.implementation

```
exported int f() {
      return module2.f2();
}
exported int data;
void f3() {
      f404() //not declared dependency blocked
}
```

Above we present another way to describe similar information in a new language[29]. At first in the architecture part we explicitly declare what our module is to provide and require. Next the implementation declares elements to be exported and thus links them to the architecture provides statement. Next an IDE could control whether the implementation corresponds to the architecture planned.

No textual inclusion is necessary. Information is made explicit and IDE could check it. E.g. that enough exported elements is supplied to match the declarations, and that nothing is used which was not declared, see `f404()` in this example.

6.1.1.2 Design Patterns

Often architecture of implementation is described with the use of design patterns[30]. Design patterns can be seen like a missing language features. The books on design patterns provide

[29] Inspiration is taken from mbeddr.modules language.
[30] Gamma et. al., “Design Patterns. Elements of Reusable Object-Oriented Software.”, Prentice Hall, 1994.

reference implementations for particular patterns in a given language. There are a number of issues with them:

- A given implementation might be deviating from the reference (and be wrong also)
- Implementing a reference implementation still means code duplication
- Information on the pattern used get lost (gets implicit in the shape of realization)
- Automotive domain might have special patterns, not commonly known

These aspects could all be improved once the patterns are moved inside the language itself.

Example 3. Singleton pattern versions.

Singleton.clike

```
shared static foo = 0;
get_foo(){
 if foo == 0 then foo = new … ;
 return foo;
}
```

Singleton.xnew

```
singleton foo = new foo;
```

Above we show how an implementation of the singleton pattern could be moved into the language. It is first demonstrated in a C-like language and then in a new language containing singleton keyword and notion inside. Singleton pattern is used to make sure that only one object of a class exists. It could be needed to express only one single instance of a physical device for example.

The second solution has the following advantages:

- Implementation of a pattern is not duplicated
- The architectural decision is retained explicit (singleton keyword)
- IDE might take advantage of the definition (e.g. filter out assignments to foo)

We have taken an example of one pattern, but obviously the same technique could be used to introduce more patterns directly into the implementation language. As design patterns are general enough in their nature, we could try speculating: typical architectural decisions in automotive could be expressed in a number of patterns, which in turn could find direct support in the implementation languages used.

6.1.1.3 Tracing and Enforcing

Once the architectural notions get explicitly expressed in the implementation the questions of tracing and enforcing become more technically implementable. The architecture *itself* however should be expressed in a way which allows referencing it for tracing, or checking the rules in it which are to be enforced. Above we give the examples, mentioning the desired

IDE capabilities, on how the architecture could be enforced, once expressed. For instance, the singleton variable is forbidden in the left part of assignments, or undeclared dependency is forbidden from the use.

Enforcing dynamical architecture is more complicated, as it involves reasoning about the program's behavior, but could be, generally speaking, attempted in a similar fashion. We describe tracing separately as a Focus Topic in the corresponding section.

6.1.2 Complexity

Very high complexity of automotive software is one of the unique features for the automotive domain. The main reasons behind the complexity boom are the rapidly growing portion of tasks a vehicle is solving using software, thus also the numerous functional and strong non-functional[31] requirements applying to the same software code, and the use of technologies, not ready to tackle with the interplay of requirements.

In the section where we discuss the survey results we split the complexity reasons into three categories:

- General Complexity
- Hardware-induced Complexity
- Language-induced Complexity

The categories are defined in that section as well and statistics is given diagrammatically on the percentage of surveyed specialists naming one or another reason for complexity.

Several reasons for complexity are dealt with in adjacent sections as they are of great importance on their own for the automotive domain and have strong influence on the implementation language features necessary. Here we deal primarily with the language-induced complexity, i.e. the complexity software might have or must have due to the nature of the programming language used.

6.1.2.1 Lack of means to express architecture

No matter we touched this matter in the previous section in detail we return here once again to the language features necessary to express the architecture, as the lack of them leads to the complexity growth. Let us bring up an example of a pattern use, which induces complexity for the lack of the feature in the language indeed (C++-similar language is used as example). In the example we define class Brick which as to be produced by a factory class `BrickFactory`. For this a typical C++ pattern implementation is used.

[31] The edge between functional and non-functional requirements is indeed blurred: e.g. real-time requirements as performance requirements are usually considered to be non-functional, they however of the primary importance for the functioning of a number of automotive subsystems.

Example 4. C++-similar implementation **o**f the Factory pattern

```
class BrickFactory; // forward declaration
class Brick {
      private:
            Brick();
      public:
      friend class BrickFactory;
}

class BrickFactory {
      Brick * makeBrick() {
            return new Brick();
      }
}
```

In the example above, we use abstractions of C++ to express the Factory pattern. First the usual construction of the Brick class has to be forbidden. For this the constructor of the Brick is placed into private section of the class in C++. Next, in order to access it from the `Brick-Factory` class, we have to declare `BrickFactory` as a friend class of the Brick. Before that, we have to mention, that such class exists, which requires a forward declaration. And finally we could define a class which constructs Brick class objects on the heap and returns a pointer to them.

The complexity is immediately growing, as we need several lines of code, with the only two truly meaningful for this use case:

Example 5. Factory pattern supported by the language.

```
class Brick manufactured by BrickFactory;
factory BrickFactory { return new Brick() }
```

The code above is concise, but it's not the only benefit. The explicit information about the factory-entity kind of relationship between the two classes is not only a direct architectural constraint expressed. A developer reading this code immediately grasps the idea, without the necessity to lift the level of abstraction from the C++ access definitions first. An IDE could be built, which would not simply forbid the use of the `Brick()` constructor, but would suggest ways to construct the object of the class. Finally, purely syntactical compiler helpers like forward declarations are gone in the newly suggested language. Two lines of code instead of eight in the first example represents 75% reduction of code volume to express a pattern.

Overall we gain:

- Explicit notion of architecture in the language
- IDE support
- Code clarity and maintainability
- Reduction of LOCs

With this example we intended to show that supporting architectural constructs on the layer of implementation language is one of the means to reduce *complexity*. We used here the example of the factory pattern. It is just a popular design pattern, but also it could be useful in automotive itself: factories could provide advantage when creating configurable Plug and Play software.

6.1.2.2 Complex control flow

Supporting complex control flow when a number of decisions are to be made based on available or unavailable information (e.g. passed through signals), modes of operation, exceptional situations, and, potentially including the choices related to product variability can easily get complex, if the language used for it just knows the `if` statement to express them.

Here are the activities at which programmers could get additional support from a modern tool and the language used in it:

- Creating code handling a large variety of operation modes
- Efficiently manipulate the code designed for several variants
- Testing and verifying such states-rich applications
- Supporting handling of error conditions, and its correctness

Below we demonstrate a number of examples, where advanced languages could play role in reducing complexity based on complex control flow.

In the following example we use a new language to define structural parts of a system. The definition includes the description of signals which parts might send or receive. Additionally, modes are described which are relevant for the operation of the system as whole. In this example control application, we want to handle the situation, when an engine has to accelerate in respond to an air conditioner requiring more power to operate.

Example 6. Mode-based control flow. Subsystems and Signals.

```
Subsystems:
 Engine
      states: on/off,
      signal in accelerate                valid when on;
 AC
      states: on/ off
      signal out power_needed
Modes:
 modeAC: Engine on, AC on
 modeNoAC: Engine on, AC off
 modeOff: Engine off
```

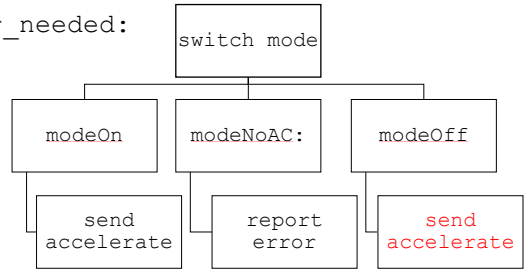
The mode information stays often implicit in the code. Instead, all the subsystem states take part in decision making process. These states altogether represent a large set.

Notice, that the cross-product of all mode values has 4 values: (Engine on, AC on), (Engine on, AC off), (Engine off, AC on) and (Engine off, AC off). The mode `modeOff` aggregates the last two values in it. This is often the case that the modes relevant for the system operation are subsets of the cross-product mode value set. Being able to explicitly describe only the relevant modes in the first place reduces the complexity by explicifying the relevant modes, and then gives an IDE a change to verify the mode description for completeness and consistency.

In the following example we show how the definitions from the previous example could be used when creating control code.

Example 7. Control code example.

```
Control Code:
Control_Engine_AC (in AC.power_needed,
      out Engine.accelerate) {
On AC.power_needed:
```




```
}
```

The control code has to react on the `AC.power_needed` signal coming. The reaction depends on mode. We use a graphical[32] switch statement to discriminate over the mode. In the `modeOff` the implementer decides (wrongly) to send `accelerate` signal to the engine. It is however not valid in the mode (see the signal definition above), and IDE takes advantage of this fact by reporting an error immediately while composing the code (before compilation even).

We switch now to another example of complex control flow induced by the need to handle exceptions.

- Why exception handling is hard?
- How can it get inefficient?
- How re-raised exceptions appear?
- What could be done?
- Error handling standard?

Example 8. Exception handling built-in into the language.

```
try_sequentially{ // efficient exceptions without stack
      step1;
      step2;
      step3;
} on failure {    // error handling supported by the language
      quit_gracefully();
}
```

Example 9. Exception handling translated in plain C.

```
if (step1() != 0) goto fail;
if (step2() != 0) goto fail; goto fail;
if (step3() != 0) goto fail;
fail: quit();
```

[32] Graphical switch is not strictly necessary here, but we use it to show, that a modern language can have both graphical and textual notations. When making a decision on more than one variable a graphical switch could be advantageous. It can get handy in practice when we take variability into account.

6.1.3 Run-Time Properties

Run-time properties include various aspects of the program's behavior at the run-time. Included are:

- Execution time properties: WCET
- Parallelism: concurrency, communication
- Deployment: multi-core or multiprocessor systems, multiple ECUs
- Dynamic architecture: which objects are created, how they behave
- Resources: occupied memory, needed hardware, etc.

We will continue with these aspects one by one, describing the relevant language features and infrastructural requirements.

6.1.3.1 Execution time

We consider the execution time problem as a concern of the tool *more* than a concern of a language. Execution time depends on hardware factors and hardware-software interactions like: CPU microarchitecture including caching and pipelining, deployment of software modules on hardware, networking between hardware modules and others. As a result, it is not enough to just have a language expressing the necessary software properties: the tools are needed to support these properties to come true. Such tools might be WCET analyzers, time-aware compilers[33], compilers for synchronous languages[34].

The problem with WCET analyzers and time-aware compilers for the common languages is that they are extremely complex. They require creation of tools, which model the processor's microarchitecture, which is very labor intensive. Moreover, such tools are not existing for complex computers which would include multiprocessors, multicore systems and networking. In this way the switch to multicore systems which have to provide strong real-time guaranties is very questionable.

Synchronous languages, on the other hand, are very limited in their applicability. They are not suitable for expressing complex algorithms manipulating non-trivial data structures. The language comparison chapter of this report is suggesting it clearly. Adding to this problem is the need to build very complex compilers for the synchronous languages, and this is compilers, not the languages themselves, what ensures execution properties.

Another problem with synchronous languages and with common languages too is that they do not mention time explicitly. This is for the reasons, that there are no straightforward ways to ensure the time properties as of now. *We suggest to change this, and explain why we do so.*

[33] Absint set of analyzers and a time-aware compiler: http://absint.com.
[34] Synchronous languages are Esterel, Lustre, Signal and so forth, see for example: http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/

Example 10. Time in a language.

```
@takes <= 100 ms on CPU_model_a1
function f_compute(){
	//…
}

@takes <= 20 ms on CPU_model_a1
function f_compute_print(){
	x = f_compute();
	print (x);
}
```

In the example above we add time annotations for a sample language. The annotation format is straightforward. Noteworthy are the use of measurement units for time and the information about deployment. Support for measurement units and conversion between them is a standard feature in many languages, e.g. in mbeddr. Deployment information is necessary, because the execution time obviously depends on deployment.

Given that the language contains the information captured in the time annotation we can discuss potential tools to make use of them. No matter ensuring exact WCET is a complex problem, *generating tests* for the annotated piece of code, and running them on the target CPU in order to verify that the average execution time still realistically falls into the provided specifications is a meaningful opportunity. Next, watching the control flow we can *find problems* internal for the code. In the example above, `f_compute_print` has to run in 20 ms, but it calls `f_compute` which potentially takes up to 100 ms. There is a chance that this is a mistake, not accounted for in advance. Therefore, the annotation of `f_compute_print` is highlighted red by the tool.

That said we continue to the next kind of run-time properties.

6.1.3.2 Parallelism

Concurrent execution is a widespread need of the automotive software. Inherent problems related to higher complexity of distributed systems programming can be supported on the level of languages by introducing appropriate constructs for:

- Communication: message passing, channels, queues, etc.
- Seamless way to account multiple concurrent threads
- Expressing dynamic architecture – who communicates with whom with proofs in implementation

We will not bring extensive examples here and just refer the reader to the prototype we have built for this project, see the corresponding appendix, and to known languages, like Google's Go[35].

The prototype demonstrates dynamic communication and simple analysis for it. The Go language demonstrates more sophisticated way to spawn threads and communicate through channels.

6.1.4 Requirements

No matter requirements themselves are not a part of implementation, it is crucial that implementation corresponds to requirements in a needed way. In particular this correspondence is established through the ability to trace from and to the requirements. Implementation languages might either make this task easier or harder.

The following features of implementation languages help to support tracing:

- Modularity
- Encapsulation and clear borders
- Capturing of matters described in requirements (e.g. if requirements talk about time, implementation language should support time directly, see Example 10.)
- An implementation language might include support for creating tests, which link to scenarios in requirements

Apart of modularity, which is typical for object-oriented programming languages, and languages which support modules and procedures functions the best way to help requirements be traced better is a higher level of abstraction making the programs closer to the requirements syntactically. This can either be achieved by modelling in the language (classes) or language being domain-specific itself.

Example 11. Domain-specific language

```
Requirement:
Execute the Transition 1 successfully or report a failure.

Implementation:
execute Transition 1
      on failure:
            report "Transition 1 failed"
```

[35] Golang website: https://golang.org/

In the example above the domain-specific language resembles closely the language of requirements. Tracing and validation is made much easier in this way.

Tracing itself is supported next by a tool. The tool needs to be able to reference various programming language parts and particularities of requirements as well. Managing the integrity of links is essential. Tools which cope with these tasks well already exist[36]. We offer you to have a look on such a tool in the prototype as well, please, see the corresponding appendix.

6.1.5 Reuse

Reuse is happening in automotive primarily in the context of related products, versions of products, related product families and various target platforms for the same code base. A programming language could support such reuse by:

- Object-oriented Programming (OOP) – as a mean of modularization
- Software modules – modularization on a larger than OOP scale
- Software variability – explicit support for product lines, code parameterization
- Higher correctness: stronger typing, well-defined semantics, etc.

As OOP is a widely understood notion, and modules were discussed above, we jump right into software variability below.

In C variants are often expressed by macros and conditional compilation. As macros lack defined semantics understood by compilers and IDEs they can introduce mistakes[37]. A step further on a fine-grain implementation code could be feature modules and support for conditional compilation in the language itself[38]. Variants become more interesting when considered in a coarse-grain fashion. The principles stay the same however. An example of the code parameterization is given below:

Example 12. Code parameterization, C++

```
      template<class ForwardIterator1, class ForwardIterator2, class Swap-
Strategy>

       ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardItera-
tor1 last1,

                                ForwardIterator2 first2) {

        SwapStrategy s;

        while (first1!=last1) {

          s.swap(*first1, *first2);

          ++first1; ++first2;
```

[36] E.g. mbeddr: http://mbeddr.com/

[37] See for example chapter 3.10 „Macro Pitfalls“ in https://gcc.gnu.org/onlinedocs/cpp.pdf , and: H. Spencer “#ifdef considered harmful, or portability experience with c news,” in Proc. of the Summer 1992 USENIX Conf., San Antionio, Texas, 1992, pp. 185–198.

[38] M. Voelter and E. Visser, “Product line engineering using domain-specific languages,” in Software Product Line Conference (SPLC), 15th Int., 2011, pp. 70 –79

```
        }
        return first2;
    }
```

The `swap_ranges` function is parameterized to be independent of containers the ranges are stored in and even from the way the elements are swapped. Higher correctness and higher code quality influences portability greatly. Thus sticking to the language with higher internal correctness (see the corresponding part in the language characteristics chapter) shall be a good way to go searching for reuse.

Another important aspect to consider lies out of the language scope directly but can be influenced by the language: the process for the actual reuse to happen. It should reflect on the following practical tasks in software development:

- How is the situation for reuse identified?
- How is the search for existing code performed?
- How code clones are avoided?
- How well is the reuse documented, reflected and maintained?
- How to measure the reuse?

It is possible to get acquainted with some of these aspects on practice by e.g. having a look no how it is done at Google[39].

6.1.6 Safety

Safety for the automotive software could be informally split into two categories: the need to go through the certification and follow the processes, and the actual operational safety which follows from the software correctness and fault-tolerance.

Speaking of software development lifecycle as according to the ISO 26262, the requirements put on the development are in their majority process requirements and as such are to be supported mainly by tools and processes. The development according to ISO 26262 could be supported better by the implementation language and tooling for it, once they are supportive in these individual steps: processing requirements, architectural design with tracing, modularity on the implementation stage + correctness, and the testability.

The majority of these aspects are focus topics in this report and are discussed separately. The prototype for this report has a simple non-interference analysis in a message passing setup. It gives an example of how an implementation language might reflect a safety-relevant aspect directly. When developing up to some given coding standard, like MISRA C, a restricted language might be needed which does not require syntactical adherence required by standards and coding guidelines to be clean.

[39] http://www4.in.tum.de/~bauerv/docs/bauer2014exploratory.pdf

Example 13. Getting syntax out of the way

```
for i in range(0, 9):
  print(“Hello”)
```

Consider the example code above. A subset of Python allows the only one syntactical way to implement a for loop. Compare it to C. Forbidding some of the language parts, also known as language subsetting, might be used to produce “safer” code. This is a feature of tools, and not languages themselves.

6.1.7 Model-driven Development

While model-driven development is an established way to produce higher quality software, some problems come along with it in automotive software engineering practice. Usually models are used for the code generation purposes. After the code has been generated there might be a need to modify it: once it does not comply with needed performance, platform or standard requirements. Once the generated code has been modified, the consistency of the model and code group of artifacts is challenged: once regenerated, code has to be modified again. It is desirable to be able to modify the code generated and the model interchangeably and keep them always in sync. This is also called a round-trip programming.

In fact, the round trip programming is a tool problem and not a language problem per-se: the need to support bidirectionality of a transformation can be seen as a transformation property and a tool task. We still suggest a number of approaches to tackling this problem.

1. A hybrid approach towards modelling,
2. Modifiable generators and
3. Variability for generators

Modifiable generators assume that the developer has access to the generator code and can modify the generator in order to achieve the necessary code modifications. Best practices of code transformations are to be taken into account once this way is taken. Modifiable generators are a workaround and not a real solution to the bidirectional transformation problem: the transformation is supposed to stay unidirectional in this case.

Variability for generators is another way to modify generators, but not intrusively. As a part of input generators could take information about the way to generate a specific instance of a model. This information might vary: for generation targets, for specific changing properties of the generated code.

A hybrid approach assumes tools which allow low-level code inclusion in the higher-abstraction-level models. Generally speaking, it involves interoperability of the languages with different abstraction levels and goes into the research topic of the next proposed project. Once the abstraction levels are not drastically different, the code of both abstraction levels might be mixed freely. A trivial example would be adding a single paradigm in the example language and making this paradigm model something from the domain.

Example 14. Hybrid approach, state machine in C

```
state machine Trafficlight receives tick{
      state:
            Red
                  On tick { time++; if (time > 20) { go Green } }
            Green
                  On tick { go Red; time=0;}
}
```

Here, we see C mixed with a simple definition of a state machine. Two languages coexist and are mixed. Like this it is modelling and "regular" programming language hybrid. See e.g. mbeddr.com for more such example in a real-world tool.

6.1.8 Testability and Verifiability

Testability and verifiability are not only very important by themselves but are a must in the safety development life cycle. Testing and verification are hard and very time consuming. Formal methods are not always easily accessible for all of the tasks. Instrumentation, observation and ability to check some on some properties of the (embedded) software are prominent and important problems. From the language perspective there are a number of ways to go to assist the developer with these tasks.

Modularity. Before testing a software module, its border hast to be defined well. For this the programming language has to be highly modular. Testing can start with the same scope as the language module or granular part has. Clean interfaces conveying as much information as possible are highly desirable for complete and targeted verification.

More language than implementation language. Testing and verification always contain of several parts:

- What is to be verified? – Software module
- Which properties are to be shown?
- In which environment?
- In which way the verification has to happen?

It is a wrong way to go, in our opinion, to express all these 4 components of verification or test in one language, implementation language, which is (hopefully) good only for the first component. As an example, we consider a language describing a test case.

Example 15. Hybrid approach, state machine in C

```
Unit Test Speed_Handling for Module m1:
Allocate 1MB of Memory
Use 32mhz tact
Test: terminates in 10ms
Environment:
      Sensor Velocity: int v, non-determinism in range [0..350]
Connect Velocity.v to m1.v
Start Test
End
```

Note that the properties to be tested are connected to the environment: we test execution time here given a frequency of the CPU. The test case reasons about the environment, including physical environment and its properties like non-deterministic nature of the sensor input. A method is described finally to connect the system under test with the environment and perform the actual test. The language to describe the test case has many new features which are not necessarily present (and even needed or possible: non-determinism!) in the implementation language. Special constructs in the test language (or languages) make it easier to *validate* the tests, which is of paramount importance for the final results.

Observability, testability, instrumentation. Once a software system has been designed and deployed testing it could be made more complicated by the requirement to observe it without modifying the original behavior. This task appears not to be trivial. The programming language could come in handy here as well.

Reflecting the need for observability in the language could be used in the generation phase to create a necessary instrumentation for debugging infrastructure.

Example 16. Observability

```
class Processor {
      private:
            observable int mCount;
      /*... */
}
```

Declaring a state member of a class as observable allows to:

- generate setters publishing modifications to the state
- keep the need to observe a state orthogonal to the access class (private)
- automatically remove observation machinery once compiled for deployment

Under publishing modifications to the state we understand special signals generated for debugger, which would reflect the state change during the test for the testing infrastructure.

Correctness. Programs in a language with higher internal correctness, as described in language characteristics, are easier to test and verify. Multiple factors contribute to this statement, but mainly: clear and defined semantics of the language, higher modularity, safety and strong type systems. The idea is simple, when it is known based on the program what it should do, then this semantics might be a basis for testing.

6.1.9 Security

Security is as relevant as it is hard in the modern software world. In automotive, according to our review up to 45% of systems are in some or another way connected, and in 60% of the systems there are no security audits performed.

Tools and processes. supported by the tools are, from our perspective, the right way to go. Education is a corner stone for security as well. Adapting and controlling SDLC alongside with safety development should be a standard practice. Once it is a programming language could play a role contributing to security or weakening it.

Correctness. Same as above, an inclining to be correct language is a necessary base for a secure implementation. Once there are doubts in what a program in some language is doing, there for no doubts will be security issues in it.

DSLs for security. Embedding the notions necessary for security in languages and tools could be a nice hand in a robust code creation. This is possible to do not only by picking an entirely new language, but also by introducing features into existing ones in a more transition-friendly fashion.

There are a number of languages and language traits designed for higher security already[40]. We recommend to have a look on these materials for inspiration when creating a better secure language. However, the problem still stays in the domains of education, processes and tools.

6.1.10 Traceability

Traceability is required to be able to map the implementation code fragments back and forth to the architecture description and to the requirements for the system implemented. We see this problem as a tooling problem mainly. Let's have a look on the way it is solved in mbeddr/MPS.

[40] Security enhancements for programming languages:
https://distrinet.cs.kuleuven.be/research/taskforces/showTaskforce.do?taskforceID=seclang or
http://www.voelter.de/data/pub/mbeddr_security_voelter_2014_roadmap.pdf

Example 17. Tracing.

```
function process( int sig) {
if ( sig > 100 ) {
      report error;
} else {
      continue;
}
}
```

```
function <id 0> process( int sig<id 1>) {
if ( sig > 100 <id 2> ) {
      report error <id 3>;
} else {
      continue;
}
}
```

The function on the left hand side is stored as an abstract syntax tree inside the editor. The editor can assign unique identifiers to the tree nodes, see the left hand side of the example. In a similar way requirements could be referenced. Next once the elements are uniquely identified establishing the mapping between them and processing it is a trivial data base management task. Once the editor stores the code in a tree format, it is not a problem to display it or some parts of it graphically. This solves the problem of tracing graphical parts of requirements and architecture to the implementation as well. You are welcome to have a look on the video demonstrating the use of mbeddr requirements[41].

Implementing a similar book keeping functionality could be possible when textual editors are used (vs. projectional editors). The book keeping appears to be more cumbersome though. In any case, we see tracing as a tooling problem rather than a programming language concern. Meaningful tracing granularity however requires language to be modular enough thus making modularity the only one recommendation for a good-to-trace language from our side.

6.1.11 Cross-Domain Development

Inside the automotive industry there are multiple domains: infotainment, control, driver assistance, etc. We call them subdomains of the automotive industry domain usually. Software development for automotive might involve systems which have to function in several subdomains at the same time. The problem is that these subdomains put different (sometimes conflicting) requirements on the programming language best suitable for the task in the domain. For example, infotainment applications work with complex data structures and profit from languages whose run time environment supports polymorphic types. Control domain in contrast would potentially lose when a language with complicated run time is used, because run-time properties get to be hard to predict.

We believe the problem of cross-domain development boils down to using appropriate language in different parts of the software and then orchestrate the collaboration. Orchestrating requires among others type translation ant interfacing between software with different characteristics: a real-time software might collaborate with a non-real-time, the result would

[41] Screencast: mbeddr, requirements and tracing, http://mbeddr.com/screencasts.html#req

be non-real-time system, unless certain precautions are met when the collaboration is organized.

Platforms and interoperability solutions come into play. They however do not go inside the languages and individual software implementations. This results to situations when orchestrating platform (e.g. AUTOSAR) puts some restrictions on a software subsystem, which are not followed by these subsystems internally. Thus what we propose here is to adapt platform limitations and limitation induced from orchestration in the implementation languages and create tools which will ensure that the limitations are followed. As an example, a time limitation can be taken into consideration. Please, see Example 10 and Example 15 above.

6.1.12 Migration

The usual reason to perform migration of code is an update of hardware in the new line of systems. The practice shows that in this process parts or whole subsystems have to be reimplemented. What could be done inside the implementation language to reduce the migration-induced work?

At first, language modularity and clear interfaces shall be used to separate machine-dependent code from machine-independent. The more code is contained in the machine-independent part of a system, the easier it is to migrate it. Machine-independency is achieved by a language having strong and defined machine-independent semantics. Strong typing would be one of the examples. Generally speaking, languages with higher internal correctness tend to be more machine-independent as well. Portability is the main characteristic however we demand for the language to assist migration the best.

Machine-dependent parts are unavoidable still. Important is that they stay as slick as possible. Besides lower migration volumes it ensures the ease of migration: smaller software is easier to encapsulate behind a well-defined interface. And here we come to interfaces, contracts and modularity again, this time from the bottom up.

6.2 Compatibility of Features

It's impossible to make a good dish adding all the ingredients which seem attractive as they have to pass to each other. In the same way it is impossible and unnecessary to combine all of the language features in one language. Among others this is another argument for the "no silver bullet" outcome.

Let's consider the table below.

Features	Patterns	Modules	Modes	Error Handling	Message Passing	Synchrony	OOP		
Patterns	Yes	Yes	Partially	Yes	Partially	Partially	Yes		
Modules		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Modes			Yes	Yes	Partially	Yes	Yes		
Error Handling				Yes	Yes	Yes	Yes		
Message Passing					Yes	Yes	Yes		
Synchrony						Yes	Partially		
OOP							Yes		
								Yes	
									Yes

This table gives intuition about features compatibility. We take as an example a number of features and investigate their compatibility with each other. It is important to tell, that many of the features are hard to combine in one language *implementation*, while others are combinable poorly on a language architecture design phase.

Rarely features are combinable with everything. A good example of such feature is modules.

Let's investigate compatibility of various programming patterns with "Modes". Under "Modes" we mean a set of language features supporting programming of state-based software like state machines, or state-based decision diagrams (see Example 7 above). Some of the patterns explicitly assume stateless operation of components, as for example REST[42] programming pattern.

As there are patterns assuming shared memory or direct access to objects a language implementing message passing internally will not harmonically embed such patterns as well. Other patterns require polymorphism which is hard to embed in a synchronous language: time calculation for truly run-time polymorphic operations is not possible generally speaking. Stateful programming and message passing might also come in a clash, as states of an object define which messages are to be received, while message passing usually does not limit it. Moreover, mechanism beyond message passing usually introduces complicated states be-

[42] See https://en.wikipedia.org/wiki/Representational_state_transfer for the definition of REST

hind them. Thus a wish to have a full deterministic control of the system state comes in a logical conflict with a freeing nature of unrestricted message passing.

These few example should give the intuition why it is hard and unnecessary to combine all the needed features. Beware that advertisement for the "*new best language for automotive industry*" always means some feature combination: either narrow and elegant, and passing to some subdomain, or broad, and, probably, hard to use. We continue to a short overview on which language features are good for a give subdomain.

There are as well few more obstacles on the way of seamlessly combining various language features:

- Two features should fit into one programming paradigm well together
  - Example: mode-oriented programming and message passing are not easily combinable as message passing introduces modes implicitly
- Semantical combination of two feature is not always straightforward:
  - Example: measurement units are easy to support in the statically-typed program fragments, but are harder to support in dynamically typed programs because information has to be passed in the dynamic type systems: in object, in exceptions, etc.
- Language engineering enables low-cost syntactical feature combinations, semantical feature combination however increases complexity and can make language unlearnable/unpredictable/incorrect:
  - Examples: Many features are syntactically combined in mbeddr, it has 70 languages combinable with each other. Languages with many feature combined are hard to learn and error-prone in practice: consider C++ or Scala. What are the average definitive guide book sizes? How easy it is to find professionals who deeply understand how the boost library is implemented in C++?

6.3 Features per Domain

Various domains have different requirements on the languages best suited for them. Even in one domain several languages might be recommended. Below we list domains and what language *characteristics* might be maximized in order to achieve an adequate choice for the domain. The same judgment can be used both for picking an existing language and for developing a new one for a domain or for a project/set of projects.

To track down to the individual language features one has to scroll up to the definition of the characteristics and when building a language take into account the new traits we propose above.

6.3.1 Drive Train, Engine, Suspension and Breaks

We call this domain also control- or real-time domain. In this domain programs are controlling the physical events happening in the real-time. The programs must be predictable, safe, fast, compact and observable. There is usually no need to process large and heavily structured data sets.

These characteristics will be of use:

- Correctness
- Simplicity
- Performance
- Low Resource Consumption

Other characteristics are clearly of secondary importance in this case.

6.3.2 Infotainment and Comfort

Systems inside the infotainment domain as the name suggests inform and entertain the driver and the passengers. They are not necessarily and even rarely safety-critical[43]. Software in this domain has more resources to rely on and processes large sets of structured data. Implementing user interfaces is one of the areas software is used in this domain for.

The characteristics for a good language should be:

- Portability
- Flexibility
- Concurrency Readiness
- Applied Practicality

Security comes in to be very important for this domain as well[44]. Generally speaking, in this domain we want a highly flexible and mighty high-level language.

6.3.3 Driving Assistance

This is the most complex and modern domain, as it is combining the features and the needs of both infotainment and control domains. The diversity of this domain is great and dependent on the project kind and closeness to either control or infotainment a set of features for the best language is to be selected on the per-project basis.

As one of the ways to go we could consider splitting a software system for driving assistance in a several parts making individual parts closer to infotainment or to the control. Language orchestration and layering will be a necessary requirement in such a project. One of our future research proposals goes exactly in this direction.

[43] Future and interaction with control might well change it though!
[44] Example car hacking without modification: http://illmatics.com/Remote%20Car%20Hacking.pdf

6.3.4 Research and Prototyping

Depending on the domain where research and prototyping is happening we can have a starting point on what kind of languages could do the job. However, as prototype is not a real product some of the requirements might be dropped for the sake of development speed.

We recommend thus:

- Practicality
- Applicability
- Domain-specific recommendations

Finally, a good distinction has to be made between the prototype and the final product. Reworking a prototype to meet the quality standards required in production can assume a change of the language for a more robust one. This transition might take a lot of time and resources but is absolutely necessary and cannot be missed: the prototype should not be confused with the real product.

6.4 A New Language: Migration Path

Once a new language is selected or is planned to be built, the infrastructure for it has to be planned carefully. A language without the infrastructure will have to fail or cause additional costs – potentially much larger costs than the costs not spent when not building the infrastructure.

Here are some tools which are required to make use of a language in production:

- Editor
- Debugger
- Versioning supporting tools - comes especially important for non-textual languages)
- Review supporting tools - integrating a model part in a review might be hard, tracing)
- Compiler – by far not the one and the only tool, but potentially expensive one
- Additional tools by demand – given that the project has special requirements: certification, coding guidelines, verification demands, etc.

Designing and building supreme-quality tools for the language is essential:

1. It is hard-to-impossible to build robust code using low-quality tools
2. Software engineer's performance depends on tools drastically

Tool certification might come into consideration which complicates the tool building process and makes it much longer and expensive. It has to be considered in advance.

Building language infrastructure requires special expertise different from one needed to build automotive software: it is risky and maybe wrong to build in-house tools on the large scale. Once the infrastructure is built there are still steps to perform before the language switch might happen:

- Developer education
- Legacy code migration, if necessary
- Ensuring compatibility with existing infrastructure in the migration period

Overall migrating to a new language is a task which requires careful planning. This report highlights language traits and what they could bring for the automotive software development. The actual switch to another language requires additional careful work and research on the migration paths. This section has just briefly outlined the potential tasks and pitfalls of such migration.

7 Conclusion

7.1 Results

In the scope of this research we investigated what new programming languages and tools have to offer for the automotive industry. We started with comparing existing widespread languages to each other pursuing an illustration by example for our main thesis in this study: We do not see that there is a “silver bullet”, one single language capable of covering all requirements of the automotive industry’s subdomains. We overviewed the state of the art in other industries showing that similar problems appear in them and are partially getting solved. One of the important take-aways there is that other industries have specialized languages and also *build* languages to their demand. We continued to important topics coming out of the survey and spotted, which language traits not used now in the automotive industry are to be of use once a new language is considered. Languages, tools and processes *together* could improve situation for the better, thus, where possible, we point out to processes and tools and how they could be improved.

7.2 Open Questions

There are some important questions which we left out of scope for this research either because of time limitations or because the answer would imply a need for a full-weight research on the topic itself:

1. Given there are perfect languages for each of the automotive subdomains, what is the best way to orchestrate them?
2. Business models for an IT evolutionary step up: how to migrate to a new language and a new infrastructure for it?

We consider these two questions the most important ones for the upcoming future of automotive software.

7.3 Next Steps

In the current study, we collected main future challenges for the development of automotive software. An important result is that there will be no one-fits-all solution based on a single programming language. Instead, we propose to create a consistent set of programming paradigms optimized for specific types of applications and show how software components from different types are combined.

There might be two approaches which are to be researched in the future towards better integrated software development in the automotive industry. First, one highly customizable language allowing domain-specific *restrictions* might be implemented. Depending on the task, the use of this language should be restricted towards appropriate paradigms only. Second, an approach to build multiple but *compatible* programming languages might be researched. The compatibility of the languages is the most essential cornerstone of the whole approach: For two programs in different languages to "collaborate", one hast to ensure that types can be converted safely, and the semantics of interactions is clear (patterns can be used here, e.g. delegation). Features like modes and error handling as well as run-time properties are to be made compatible in some way explicitly, unlike black box Simulink-to-C delegation.

Advanced Driver Assistance Systems (ADAS) introduce multiple challenges to the software development task. Prominent challenges in implementing ADAS are the combination of heterogeneous functionality, such as computationally intensive calculations and real-time control, the assurance of functional safety and realization of fail-operational behavior, and the integration and testing of interacting ADAS functions. As the current project is showing, the paradigms currently in use do not support those challenges very well and necessary paradigms cannot be easily integrated in a single language

Figure 34 shows a possible structure of the ADAS domain. We propose a layered architecture consisting of four layers with different degrees of abstraction. On the lowest layer, local control loops are executed in hard real time and data is preprocessed. The dynamic control layer performs distributed control tasks and orchestrates the execution of tasks across multiple ECUs. Higher layers are dedicated to deciding and executing strategies.

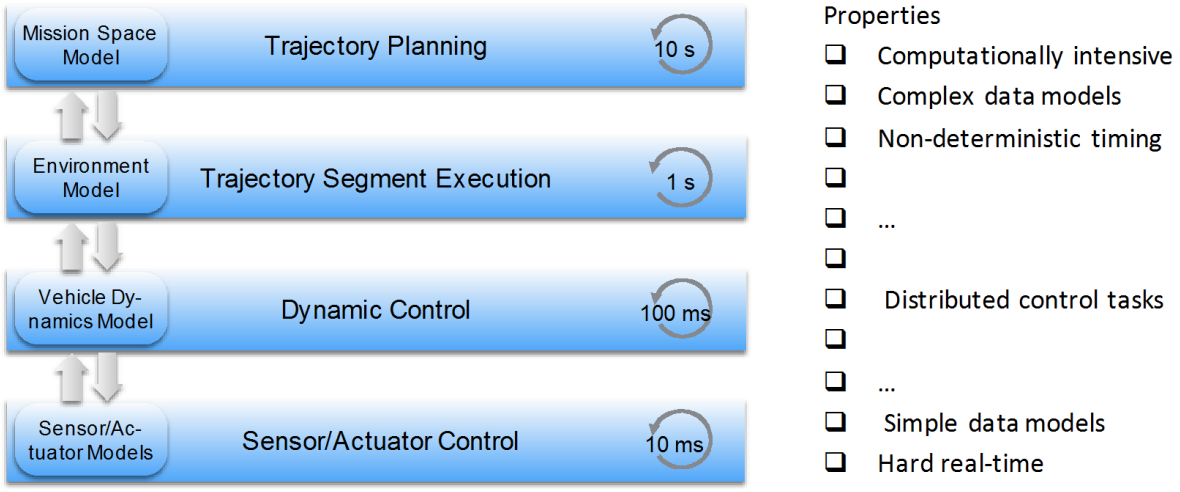


Figure 35 Structure of the ADAS Domain

In future activities, we propose to establish a consistent definition of the ADAS domain structure and define key properties of the architectural layers. Based on the results of this study, we then describe programming paradigms that can support the implementation on

the level of the respective architectural layer. Also, we provide a preview on how an ADAS can be developed using optimized programming paradigms. As it is unlikely to develop a language paradigm that will fit all needs, we emphasize the need for interoperability mechanisms between the specialized layer specific paradigms.

Appendix I. Tool Prototype

In this appendix we describe the tool prototype we have built to illustrate the idea of combining language features. We accompany the prototype with a screencast[45] describing the internals of the prototype. In this appendix we focus only on the installation and usage aspects.

Installation

The prototype itself does not require any installation per se. It is an MPS project dependent on mbeddr. It is located in the `FAT_VDA_Sample.zip` file.

1. Install (MPS and) mbeddr as described on http://mbeddr.com/download.html
2. Run mbeddr and open the `mbeddr.core` project
3. Extract the prototype from the zip archive
4. Open the prototype in a new window *without closing the mbedd.core window*

The prototype should open and you should be able to experiment with it as on the screencast.

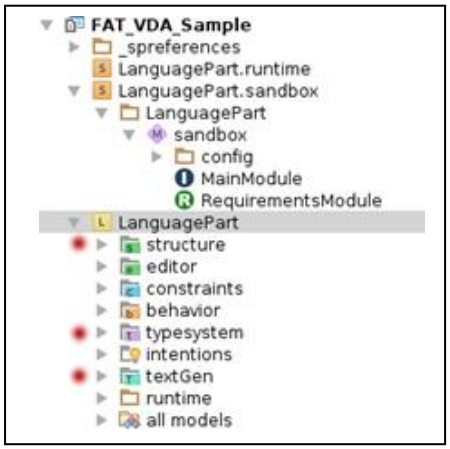


Figure 1 – Protoype's Logical Structure

Usage

At first you could watch the screencast and try reproducing the same experience in your MPS: use the languages, reproduce the generation part, inspect the code generated.

Next, to get into language implementation details, we recommend you to start by learning more about MPS and mbeddr. These resources might be helpful:

- http://mbeddr.com/screencasts.html
- https://www.youtube.com/playlist?list=PLQ176FUIyIUY9rAcAH6MNOxJqGfau0Jb1
- https://confluence.jetbrains.com/display/MPSD32/MPS+User's+Guide
- https://www.jetbrains.com/mps/documentation/

After getting acquainted with MPS internals you are ready to inspect in detail the language created in the prototype and the languages included in mbeddr as well.

The `LanguagePart` in the solution corresponds to the language elements created for the prototype. Inspect its structure first.

Continue to the type system rules and the generator, Figure 1.

[45] It is in the `FAT_VDA_Sample_Screencast.wmv` and `FAT_VDA_Sample_Screencast.mp4` files, content is the same, encoding differs. The later one is of better quality, play it with VLC player or another player supporting a rich set of decoders.

Appendix II. Survey Data

We ship the raw survey data with this report as well for completeness.

Instructions

In the survey_data.zip file you could find the following structure:

```
survey_data /sps/survey.sps.zip
survey_data/csv/survey.csv.zip
survey_data/sql/limesurvey.db.zip
survey_data/html/survey.html.zip
survey_data/doc/survey.doc.zip
```

The data was exported in various formats for you to make use of it in the best way. Below we describe shortly how to use these formats.

CSV. It is the simplest format with comma-separated values. It is compatible with Excel is however not human-readable. We recommend this format to be used for custom processing tools.

SPS. This is the format of IBM's SPSS tool[46]. This is a proprietary tool for statistical analysis of survey data.

SQL. This is the most complete file – a database dump of LimeSurvey 2.0.6+[47] survey software. This is an open-source survey software. Once installed, one could use our database dumped and interact with the survey in the way a survey administrator would. Once the database is restored[48] from the dump and pointed to from LimeSurvey, you'll have to login as `admin` and use the password `survey`.

HTML. This is a user-readable format. You could open it with any browser and view the responses one-by-one.

DOC. This is a regular Microsoft Word file format. We only recommend you to use it if you plan to edit the data or copy a part of it. Use HTML format for viewing purposes otherwise. The reason why we do not recommend this format is the file size and difficulties when displaying it in MS Word software which might occur.

Additionally we ship LimeSurvey statistic dump-outs in PDF grouped by focus topic in the statistics.zip file!

[46] https://en.wikipedia.org/wiki/SPSS
[47] It could be obtained from here: https://www.limesurvey.org/downloads/category/24-archived-releases
[48] This is usually done like this with MySQL: `mysql -u root -p[root_password] [database_name] < dumpfilename.sql`

Appendix III. Language Comparison

While working on the chapter on language comparison and characteristics we have built a semi-automated language comparison system. It can be enhanced and extended by demand. New languages can be added as well. Here we describe how to work with it.

Instructions

The artifacts needed to repeat the language analysis are located in 3 archives:

`drawing_scripts.zip` – scripts to generate images for comparison

`keywords.zip` – lists of keywords in some languages, needed for reference only

`language_comparison_tables.zip` – Excel tables with the comparison system

Language comparison tables are in ODS and CSV formats. The ODS document is the original one and is to be edited. It contains definitions of language characteristics as formulas. The CSV is an export from it needed for the drawing scripts to work.

Drawing scripts are Python automations which produce images to compare languages based on the characteristics. They take the exported CSV file as input. In order to use them simply run `drawAll.py` script making sure the CSV file and drawing `Template.svg` are in the working folder. The Python scripts are open-source and to be modified on demand freely.

The images you see in the language comparison chapter are manually modified after the generation to increase their readability. We recommend to use Inkscape[49] to perform such modifications if images are regenerated.

[49] Inkscape graphics editor web site: https://inkscape.org/de/

Appendix IV. Glossary

Architecture pattern	A pattern is a guideline/best-practice how to implement a certain functionality.
Central computing unit	A central computer in a vehicle, which can typically execute several functions in parallel. May consists of several computing entities, which form one logical central computing unit.
Characteristic	See language characteristic below and Chapter 2.2.
Cloud	Processing and/or storage capabilities which can be accessed via an internet connection.
Complexity	Either a focus topic as complexity of software projects, or language's internal complexity, a language characteristic as defined in Chapter 2.2.2.
Concurrency readiness	A language characteristic as in Chapter 2.2.7.
Correctness	In this report under correctness we understand mostly language's internal correctness, a characteristic introduced in Chapter 2.2.3.
CPS (cyber-physical system)	Cyber-physical systems can be seen as smart and dynamic embedded systems. They are especially characterized by three dimensions of complexity[50]: "cross"-dimension (e.g. cross-technologies), "self"-dimension (e.g. self-adapting) and "live"-dimension (e.g. live-configuration).
Domain	A subject area for applying programming, for example automotive domain.
E/E architecture	The electric and electronic architecture of a

[50] Bernhard S. "The Role of Models in Engineering of Cyber-Physical Systems–Challenges and Possibilities." Tagungsband des Dagstuhl-Workshops. 2014.

	vehicle. In a wide sense, the E/E architecture also includes the software components.
ECU (electronic control unit)	An electronic control unit is a computing platform in a vehicle. Usually, several ECU's are deployed in a modern car.
Fail-operational	A system with fail-operational behavior can tolerate at least one failure and is still operational.
Fail-safe	A system with fail-safe behavior is stopped and brought into a safe state if a failure happens.
Feature	A language feature, usually an atomic feature of a language, which is objectively identifiable, like for instance support for OOP. See Chapter 2.1.
Flexibility	Language flexibility, a language characteristic defined in Chapter 2.2.6.
IDE	Integrated development environment
Language characteristic	Cumulative informal measure consisting out of language features and intending to show suitability of a language for some purpose, e.g. correctness or concurrency-readiness.
Language feature (trait)	Individual language feature like strong typing or structural programming
Logical architecture	The logical architecture defines the properties of a system on a logical level. This includes the software components and their virtual connections with each other. The mapping of the logical architecture to the technical architecture is done in the deployment step.
Migration	Moving software to a new hardware or software platform.
Modularity	Organizing a program into separate units

	which can be reused separately: usually modules, classes, functions, etc.
MPS	JetBrains MPS – Meta Programming System, see https://www.jetbrains.com/mps/
OOP	Object-oriented Programming: encapsulation, polymorphism, virtualization.
Performance	In this report often language-induced performance, a language characteristic introduced in Chapter 2.2.4.
PL	Programming language
Portability	In this report a language characteristic as in Chapter 2.2.5.
Practicality	Applied practicality, a language characteristic introduced in 2.2.8.
Resource consumption	A language characteristic showing how much computational resources a program in a language might consume, usually displayed in an inverted fashion on charts: the higher the better. See Chapter 2.2.9.
Separation	A technology used to guarantee the interference-free execution of several activities on one device. In a more detailed sense, this term usually compromises separation in space (e.g. memory) and time (e.g. time-triggered execution). Virtualization combined with separation enables the conflict-free integration of several functions on one host platform.
Simplicity	Language simplicity, an invert of language internal complexity, see Chapter 2.2.2.
Smart aggregate	A smart aggregate is a compound of one or multiple sensors and/or one or multiple actuators with local processing capability to preprocess data, execute abstract commands and to run local control loops.

SOA (service-oriented architecture)	SOA is an architectural pattern for the implementation of loosely coupled, distributed systems. Functions are implemented as services which can be used by multiple other functions, usually in a client-server relation.
Subdomain	A part of a domain, for instance control subdomain of the automotive domain.
Technical architecture	The technical architecture represents the properties of the physical systems forming the vehicle structure. The technical architecture usually includes the electronic control units, the set of aggregates, the physical communication links and network infrastructure.
Trait	A language trait, the same as a language feature in this report.
TS	Type System of a programming language
Virtualization	A technical feature for the emulation of resources. E.g. virtualization enables the parallel execution of several operating systems on one core.

Bisher in der FAT-Schriftenreihe erschienen (ab 2010)

Nr.	Titel
227	Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Dünnbleche aus Stahl, 2010
228	Systemmodellierung für Komponenten von Hybridfahrzeugen unter Berücksichtigung von Funktions- und EMV-Gesichtspunkten, 2010
229	Methodische und technische Aspekte einer Naturalistic Driving Study, 2010
230	Analyse der sekundären Gewichtseinsparung, 2010
231	Zuverlässigkeit von automotive embedded Systems, 2011
232	Erweiterung von Prozessgrenzen der Bonded Blank Technologie durch hydromechanische Umformung, 2011
233	Spezifische Anforderungen an das Heiz-Klimasystem elektromotorisch angetriebener Fahrzeuge, 2011
234	Konsistentes Materialmodell für Umwandlung und mechanische Eigenschaften beim Schweißen hochfester Mehrphasen-Stähle, 2011
235	Makrostrukturelle Änderungen des Straßenverkehrslärms, Auswirkung auf Lästigkeit und Leistung, 2011
236	Verbesserung der Crashsimulation von Kunststoffbauteilen durch Einbinden von Morphologiedaten aus der Spritzgießsimulation, 2011
237	Verbrauchsreduktion an Nutzfahrzeugkombinationen durch aerodynamische Maßnahmen, 2011
238	Wechselwirkungen zwischen Dieselmotortechnik und -emissionen mit dem Schwerpunkt auf Partikeln, 2012
239	Überlasten und ihre Auswirkungen auf die Betriebsfestigkeit widerstandspunktgeschweißter Feinblech-strukturen, 2012
240	Einsatz- und Marktpotenzial neuer verbrauchseffizienter Fahrzeugkonzepte, 2012
241	Aerodynamik von schweren Nutzfahrzeugen - Stand des Wissens, 2012
242	Nutzung des Leichtbaupotentials von höchstfesten Stahlfeinblechen durch die Berücksichtigung von Fertigungseinflüssen auf die Festigkeitseigenschaften, 2012
243	Aluminiumschaum für den Automobileinsatz, 2012
244	Beitrag zum Fortschritt im Automobilleichtbau durch belastungsgerechte Gestaltung und innovative Lösungen für lokale Verstärkungen von Fahrzeugstrukturen in Mischbauweise, 2012
245	Verkehrssicherheit von schwächeren Verkehrsteilnehmern im Zusammenhang mit dem geringen Geräuschniveau von Fahrzeugen mit alternativen Antrieben, 2012
246	Beitrag zum Fortschritt im Automobilleichtbau durch die Entwicklung von Crashabsorbern aus textil-verstärkten Kunststoffen auf Basis geflochtener Preforms und deren Abbildung in der Simulation, 2013
247	Zuverlässige Wiederverwendung und abgesicherte Integration von Softwarekomponenten im Automobil, 2013
248	Modellierung des dynamischen Verhaltens von Komponenten im Bordnetz unter Berücksichtigung des EMV-Verhaltens im Hochvoltbereich, 2013
249	Hochspannungsverkopplung in elektronischen Komponenten und Steuergeräten, 2013
250	Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Feinbleche aus Stahl unter Schubbeanspruchung, 2013

251 Parametrischer Bauraum – synchronisierter Fahrzeugentwurf, 2013

252 Reifenentwicklung unter aerodynamischen Aspekten, 2013

253 Einsatz- und Marktpotenzial neuer verbrauchseffizienter Fahrzeugkonzepte – Phase 2, 2013

254 Qualifizierung von Aluminiumwerkstoffen für korrosiv beanspruchte Fahrwerksbauteile unter zyklischer Belastung (Salzkorrosion), 2013

255 Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf echten Fahrbahnen, 2013

256 Naturalistic Driving Data, Re-Analyse von Daten aus dem EU-Projekt euroFOT, 2013

257 Ableitung eines messbaren Klimasummenmaßes für den Vergleich des Fahrzeugklimas konventioneller und elektrischer Fahrzeuge, 2013

258 Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen, Teile 1 und 2, 2013

259 Erweiterung des Kerbspannungskonzepts auf Nahtübergänge von Linienschweißnähten an dünnen Blechen, 2013

260 Numerische Untersuchungen zur Aerodynamik von Nutzfahrzeugkombinationen bei realitätsnahen Fahrbedingungen unter Seitenwindeinfluss, 2013

261 Rechnerische und probandengestützte Untersuchung des Einflusses der Kontaktwärmeübertragung in Fahrzeugsitzen auf die thermische Behaglichkeit, 2013

262 Modellierung der Auswirkungen verkehrsbedingter Partikelanzahl-Emissionen auf die Luftqualität für eine typische Hauptverkehrsstraße, 2013

263 Laserstrahlschweißen von Stahl an Aluminium mittels spektroskopischer Kontrolle der Einschweißtiefe und erhöhter Anbindungsbreite durch zweidimensional ausgeprägte Schweißnähte, 2014

264 Entwicklung von Methoden zur zuverlässigen Metamodellierung von CAE Simulations-Modellen, 2014

265 Auswirkungen alternativer Antriebskonzepte auf die Fahrdynamik von PKW, 2014

266 Entwicklung einer numerischen Methode zur Berücksichtigung stochastischer Effekte für die Crash-simulation von Punktschweißverbindungen, 2014

267 Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 1, 2014

268 Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 2, 2014

269 Schwingfestigkeitsbewertung von Schweißnahtenden MSG-geschweißter Feinblechstrukturen aus Aluminium, 2014

270 Physiologische Effekte bei PWM-gesteuerter LED-Beleuchtung im Automobil, 2015

271 Auskunft über verfügbare Parkplätze in Städten, 2015

272 Zusammenhang zwischen lokalem und globalem Behaglichkeitsempfinden: Untersuchung des Kombinationseffektes von Sitzheizung und Strahlungswärmeübertragung zur energieeffizienten Fahrzeugklimatisierung, 2015

273 UmCra - Werkstoffmodelle und Kennwertermittlung für die industrielle Anwendung der Umform- und Crash-Simulation unter Berücksichtigung der mechanischen und thermischen Vorgeschichte bei hochfesten Stählen, 2015

274 Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components, 2015

275 Hochrechnung von GIDAS auf das Unfallgeschehen in Deutschland, 2015

276 Literaturanalyse und Methodenauswahl zur Gestaltung von Systemen zum hochautomatisierten Fahren, 2015

277 Modellierung der Einflüsse von Porenmorphologie auf das Versagensverhalten von Al-Druckgussteilen mit stochastischem Aspekt für durchgängige Simulation von Gießen bis Crash, 2015

278 Wahrnehmung und Bewertung von Fahrzeugaußengeräuschen durch Fußgänger in verschiedenen Verkehrssituationen und unterschiedlichen Betriebszuständen, 2015

279 Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen – Teil 3, 2015

280 PCM from iGLAD database, 2015

281 Schwere Nutzfahrzeugkonfigurationen unter Einfluss realitätsnaher Anströmbedingungen, 2015

282 Studie zur Wirkung niederfrequenter magnetischer Felder in der Umwelt auf medizinische Implantate, 2015

283 Verformungs- und Versagensverhalten von Stählen für den Automobilbau unter crashartiger mehrachsiger Belastung, 2016

284 Entwicklung einer Methode zur Crashsimulation von langfaserverstärkten Thermoplast (LFT) Bauteilen auf Basis der Faserorientierung aus der Formfüllsimulation, 2016

285 Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf realer Fahrbahn, 2016

286 χMCF - A Standard for Describing Connections and Joints in the Automotive Industry, 2016

287 Future Programming Paradigms in the Automotive Industry, 2016

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863

VDA | Verband der Automobilindustrie

FAT | Forschungsvereinigung Automobiltechnik

Behrenstraße 35
10117 Berlin
www.vda.de
www.vda-fat.de