Processes and considerations for non-standardized languages in functional safety

Nefeli Tavoulari
nefeli.tavoulari@student.uva.nl

July 4, 2024, 51 pages

Academic supervisor: Andrés Goens, a.goens@uva.nl
Daily supervisor 1: Remi van Veen, remi@solidsands.nl
Daily supervisor 2: Vladislav Yaglamunov, vlad@solidsands.nl
Host organisation: Solid Sands, https://solidsands.nl/

University of Amsterdam
Faculty of Science
Master Software Engineering

Abstract

The growing interest in developing safety-critical applications in industries such as automotive emphasizes the necessity for reliable software to prevent malfunctions that could pose risks to users and the environment. For this reason, international standards organizations, such as the International Organization for Standardization (ISO), have set functional safety standards for systems, including software tools, such as compilers. Established languages like C, C++, and Ada are standardized, which means that they have stable specifications and test suites can be created to validate that the compilers correctly implement what is written in these language specifications. However, newer languages like Rust and Zig lack complete specifications, blocking the development of stable test suites and their adoption in safety-critical domains. The goal of this project is to facilitate the adoption of emerging, non-standardized languages, such as Rust, in safety-critical applications by developing a qualification process that emphasizes thorough validation. Our approach proposes practical guidelines on developing a language specification and a compiler validation test suite. This results from an extensive review of existing work on the topic, an analysis and comparison of language specifications and compiler test suites, and a proof-of-concept that includes the creation of a specification and test suite with over 80 tests targeting selected features of Rust. Our method led us to identify areas for improvement and successfully contributed four merge requests to the Ferrocene Specification and the Rust Reference.

Contents

Chapter 1

Introduction

Nowadays, there is a growing interest in the development of safety-critical applications across different industries such as automotive, railways, and aerospace. The primary concern is ensuring the safety and reliability of software to prevent any malfunctions that could endanger users and the environment. As a result, international standards organizations, such as the International Organization for Standardization (ISO) [1], have set standards for the functional safety of electrical and electronic systems, including software. Programming languages are equally crucial in safety-critical applications. That is why functional safety standards require that confidence is created in the correct operation of software tools, such as compilers.

When even perfectly safe code gets incorrectly translated by the compiler, the entire application's functional safety is at risk. For instance, a bug in Java 7 caused many popular non-critical Apache applications to crash [2], which was inconvenient, but a similar issue in a safety-critical application could have been fatal. Compiler bugs not only result in unexpected behavior with potentially severe consequences but also make software debugging more complicated. Developers may waste significant time to understand whether a software failure is because of their defective code or because of compiler bugs. Due to the critical role of compilers, there is significant interest in ensuring their correct implementation. However, this is a challenging task due to the complexity of compilers [3].

The typical way of obtaining confidence in the correct operation of a compiler is by validating the compiler using a test suite that is created based on the programming language specification [4]. Established languages like C [5], C++ [6], and Ada [7] are standardized. This means that they have stable language specifications and test suites can be created to validate that the compilers correctly implement what is written in these language specifications. SuperTest, developed by the company Solid Sands [4], is an example of such a test suite. It contains a large collection of tests for C and C++ compilers, ensuring adherence to the C and C++ standards. In this way, confidence can be ensured in the correct implementation of these languages. However, newer languages like Rust and Zig lack stable and complete language specifications. And so, a stable test suite cannot be implemented. For deploying these languages in safety-critical applications, it is essential not to assume that compilers are error-free.

For instance, Rust is gaining ground for its performance and safety features, which align with the goals of safety-critical applications. But the obstacle is in the absence of validation of the Rust project's compiler, particularly for industries like automotive, which follows ISO 26262. Rust's focus on compiler-based checks and safe memory management are significant advantages, but without a complete language specification, as well as validation of these compiler-based checks, full confidence is lacking [8].

To address this gap, the Rust community has already taken some measures. The Rust Project Language Specification Team [9] has been established to create a language specification for Rust [10], though this effort has not yet fully started. Furthermore, recently, the Safety-critical Rust Consortium [11] was established, in an effort to integrate Rust into safety-critical applications. A significant milestone is the qualification of Rust for use in the automotive through the Ferrocene toolchain [12].

The goal of this project is to contribute to programming language safety, focusing on emerging languages such as Rust, by establishing a method to allow their adoption in safety-critical domains.

1.1 Problem statement

This project has been conducted in collaboration with Solid Sands [4], a company specializing in compiler and library testing and validation on C and C++. With extensive experience in compiler qualification

for safety-critical domains, Solid Sands develops compiler test suites that adhere to language standards, ensuring confidence in the compilers' reliability. The company is now exploring Rust, due to its growing popularity and adoption as an alternative to C and C++. However, unlike C and C++, Rust lacks a standardized or stable specification, which makes the qualification process more complex.

1.1.1 Research questions

To tackle these issues, we investigate the following research questions:

- **RQ1**: What systematic process can be established to validate the compiler of a non-standardized programming language, ensuring compliance with functional safety standards?
- **RQ2**: How can a robust programming language specification be developed for a documented and implemented language that lacks a formal specification, ensuring compliance with industry standards?
- **RQ3**: How can a compiler validation test suite be developed for a non-standardized programming language to ensure compiler reliability in functional safety?

1.1.2 Research method

The research method used in this project is Exploratory Case Study. It involves gathering, analyzing, and comparing information from existing research and activities related to programming language specifications, compiler validation testing, and compiler qualification, regarding Rust but also other languages. Next, with a Proof-of-Concept, the project synthesizes and applies all this knowledge in the development of a language specification and compiler validation test suite for selected features of Rust. These outcomes will serve as research artifacts. All in all, a qualification process for non-standardized languages is proposed. The case study's goal is to be the starting point for a large-scale research project.

1.2 Contributions

Our research makes the following contributions:

1. Proposing a structured compiler qualification process for non-standardized languages.
2. Providing guidelines and a proof-of-concept for developing a robust language specification.
3. Providing guidelines and a proof-of-concept for developing a compiler validation test suite.

1.3 Outline

In Chapter 2 we delve into the background of this thesis, explaining key concepts. Chapter 3 describes important principles of the compiler qualification process and introduces examples of compiler qualification in practice, including considerations specifically for Rust. In Chapter 4, we explore the importance of a language specification and present best practices derived from comparing different language specifications. Similarly, Chapter 5 compares different compiler test suites and draws conclusions on best practices. Next, in Chapter 6, we integrate these insights into the proposed qualification process, where we develop a language specification and test suite for Rust. Chapter 7 addresses potential threats to validity, which is followed by Chapter 8 that reviews related work relevant to this thesis. Finally, Chapter 9 presents our conclusions and directions for future work.

Chapter 2

Background

In this chapter, we provide necessary background information for this thesis. Firstly, we define safety-critical applications and discuss the ISO 26262 safety standards, for the automotive industry, which detail how software tools are validated and qualified to ensure reliability. Then, we explore the concept of language standardization and its components. Next, we introduce the Rust programming language, highlighting the factors that have contributed to its recent popularity and we discuss the role and relevance of the Rust Reference. Finally, we introduce Solid Sands, explaining its expertise in compiler and library testing, which is useful to our research efforts.

2.1 Safety-critical Applications

A safety-critical or safety system is responsible for taking safety measures against potential failures that could be harmful to human life, property, or the environment. These systems are designed to prevent accidents by taking proactive measures and to reduce the consequences of potential accidents. For this purpose, there is a need for clear software development, verification and validation processes to ensure confidence and satisfy compliance requirements [13].

Functional safety, as defined by the IEC Standard 61508 [14], is a subset of overall safety that relies on a system or equipment functioning correctly in response to its inputs. For instance, in an automobile system, the automatic activation of airbags during an accident is an example of functional safety, which aims to prevent risks to human life. Safety standards, including IEC 61508 [14], provide guidelines for designers, developers, and manufacturers to ensure that products are compatible, cost-effective, functionally safe, and user-friendly. Compliance with safety standards is obligatory for systems performing safety functions. These standards categorize systems according to the required level of reliability and specify ways to achieve such reliability. They accomplish that by defining management and development processes, as well as documentation and verification requirements [13].

Even though the fundamental safety principles remain the same across industries, specific standards have been created for each sector, to address the special characteristics of each domain. For example, fail-safe designs vary depending on the industry, such as automatic reactor shutdown in the nuclear industry compared to the generation of alarms in the aviation industry. Certification of suitability by regulatory authorities is essential before operating systems with safety considerations [13].

2.2 ISO 26262

The ISO 26262 series of standards is a collection of guidelines from the IEC 61508 [14] series, which specializes in the safety of electrical and electronic systems within road vehicles. These standards are crucial in ensuring functional safety, particularly with the complexity of automotive technology. They provide a structured approach to the automotive safety lifecycle, covering processes regarding the development, production, operation, service, and decommissioning phases. Also, they provide a process to determine integrity levels (Automotive Safety Integrity Levels or ASILs), requirements for functional safety management, design, implementation, verification, validation, confirmation measures, and requirements for relations between customers and suppliers [1].

2.2.1 Confidence in the use of software tools

An important part of ISO 26262 are the requirements for ensuring confidence in the use of software tools, such as compilers. The goals of these requirements are to establish criteria to determine confidence levels in software tools and to provide ways to qualify these tools for supporting ISO 26262 activities. To achieve these goals, the standard emphasizes minimizing the risk of systematic faults in products resulting from software tool malfunctions and ensuring compliance with ISO 26262 standards in product development. Confidence in software tools consists of usage and qualification aspects. Tool usage involves evaluating the software tool based on its functions and properties, determining the Tool Confidence Level (TCL) through analysis of Tool Impact (TI) and Tool error Detection (TD) (Figure 2.1), integrating the tool into the user's environment, verifying its operation, and ensuring proper usage for development or verification tasks. Tool qualification involves validating the tool based on provided or assumed information regarding its usage, including use cases, user requirements, TCL, and Automotive Safety Integrity Levels (ASILs) (Figure 2.2) [1].

The Tool Impact (TI) classes categorize the likelihood of a software tool causing or failing to detect errors in safety-related items. TI1 is chosen when evidence shows no possibility of errors, while TI2 is selected for all other cases. The Tool error Detection (TD) classes indicate confidence in measures preventing or detecting software tool malfunctions. TD1 signifies high confidence, TD2 medium confidence, and TD3 is used for all other cases. The Tool Confidence Level (TCL) classes represent the current confidence level in a tool. TCL1 indicates high confidence with no further qualification needed, while TCL2 and TCL3 indicate medium and low confidence, both requiring additional qualification methods. Automotive Safety Integrity Level (ASIL) is a risk classification system defined by ISO 26262 for the functional safety of road vehicles [1].

Compilers are usually classified as TI2 (High Impact) because errors in the compilation process can lead to critical failures in the software. Incorrect machine code generation can cause software crashes and unpredictability, even if the original source code is perfectly correct. We could say that compilers are the "heart" of the software. Also, compilers are typically classified as TD3 (Low Detection Capability), since the likelihood of detecting these potential errors is very low. Compilers are complex systems that translate high-level programming languages to machine code, making error detection challenging. They handle various programming constructs, optimizations, and platform-specific instructions. Even with thorough testing, it is almost impossible to cover every code path and input combination, leading to some errors only presenting under specific, untested conditions. So, usually the Tool Confidence Level of a compiler is TCL3, which means that there is Low Level of Confidence and further qualification methods are required [1].

		Tool error detection		
		TD1	**TD2**	**TD3**
Tool impact	**TI1**	TCL1	TCL1	TCL1
	TI2	TCL1	TCL2	TCL3

Figure 2.1: Determination of the Tool Confidence Level (TCL) [1]

2.2.2 Qualification of a software tool

The qualification of the software tool shall be documented including the unique identification and version number of the tool, the maximum Tool Confidence Levels, the pre-determined ASILs, the configurations and the environment for which the tool is qualified and the organization that performed the qualification. Furthermore, the used qualification methods, the results and any identified usage constraints or malfunctions should also be documented [1]. For the qualification of software tools classified at TCL3 or TCL2, one of the methods listed below shall be applied. A software tool classified at TCL1 needs no qualification methods [1].

Increased confidence from use

A tool can be considered to have increased confidence from use if specific evidence is provided. Firstly, the tool must have been previously used for the same purpose under similar conditions, including comparable use cases, operating environments, and functional constraints. Additionally, the justification for increased

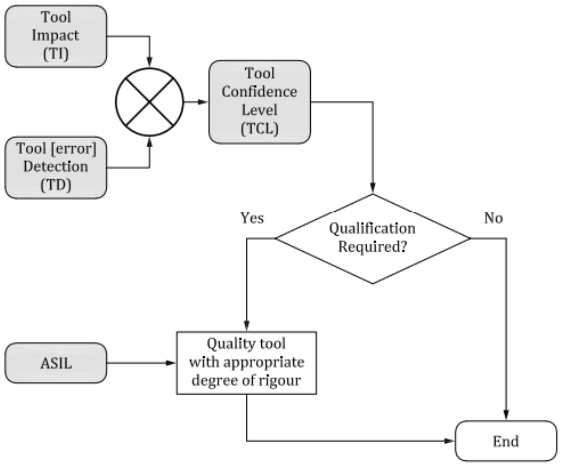


Figure 2.2: Tool evaluation and qualification flow [1]

confidence must be supported by sufficient data, documenting malfunctions and their outputs from previous development phases. Furthermore, the specification of the software tool must remain unchanged, and any malfunctions or errors encountered during previous usage must be systematically documented. However, new, non-standardized languages, like Rust, lack the extensive data necessary for this method. Furthermore, as far as Rust is concerned, the language specification is neither stable nor complete, as highlighted by the Rust team themselves [15].

Evaluation of the tool development process

The tool development process can be evaluated by checking that it complies with an appropriate standard. This evaluation should be based on national or international standards, providing evidence of the application of a suitable software development process. It should cover an adequate and relevant subset of the tool's features. Rust, being an open-source project, does not conform to these strict development process standards.

Validation of the software tool

Validation measures should provide evidence that the tool complies with specified requirements for its intended purpose. This validation can be achieved through customized test suites developed by the user or provided by the tool vendor. Moreover, any malfunctions and corresponding errors during validation must be analyzed, along with measures to avoid or detect them. Lastly, the software tool's response to anomalous operating conditions, such as misuse or incomplete input data, should be examined. However, in order to create a validation test suite, a stable language specification should be in place, which is currently lacking for Rust.

Development in accordance with a safety standard

Rust has not been developed in accordance with a safety standard such as ISO 26262. This is evident from the absence of a stable language specification.

It is evident that only the validation of the software tool qualification method is feasible for our case. However, it is crucial to first create a stable language specification.

2.3 Language Standardization

As programming languages have grown more complex, the need for standardization has become more critical to ensure reliable and efficient implementations. Language standardization, typically conducted by standards committees comprising experts and industry representatives, defines the syntax and semantics of a programming language, enabling multiple independent compiler implementations. This allows developers to create compilers designed for their specific needs while ensuring compatibility and

interoperability with the standardized language, promoting innovation and flexibility. Furthermore, language standardization offers significant benefits, including saving time, effort, and money [16], since a standardized programming language allows for portability of programs between different computers.

According to the "ISO/IEC Guidelines for the preparation of programming language standards", a language standard should contain at least the specification of the syntax of the language, including rules for conformity of programs and processors, the specification of the semantics of the language, including rules for conformity of programs and processors, the specification of all further requirements on standard-conforming programs and rules for conformity, the specification of all further requirements on standard-conforming processors, such as error and exception detection, reporting and processing, provision of processor options to the user, documentation, and validation, along with rules for conformity, and some informative annexes [17].

2.4 Rust Programming Language

Rust, developed by Mozilla and cited as the most beloved language [18] in the past eight years according to the Stack Overflow Surveys, stands out as a language with multiple advantages that appeal to developers across various domains. Rust offers fast speed and efficient memory management [8], without a runtime or garbage collector. This enables Rust to support performance-critical services, operate on embedded devices, and seamlessly integrate with other languages, like C and C++. Rust achieves a balance between safety and control and prioritizes reliability through its strict type system and ownership-borrowing model, ensuring memory and thread safety [19]. In contrast to languages like C++, Rust's memory safety model detects bugs and data races during compilation, preventing runtime errors like accessing deallocated memory and enables resilience in development. Overall, Rust's combination of performance, reliability, and productivity makes it a unique programming language.

2.4.1 Rust Reference

The Rust Reference [20], is one of many documentation resources provided from the Rust team and it is the main manual of the language, explaining its key concepts and syntax. However, as noted by the Rust team, it is not a stable and complete language specification. Its primary purpose is to help developers learn Rust [21].

2.5 Solid Sands

Solid Sands is a dominant provider of compiler and library testing and qualification technology, with a strong focus on improving the quality of C and C++ compilers, libraries, and analysis tools to ensure their safe and reliable use. The company serves a diverse range of industries, including semiconductor, IP, security, automotive, robotics, railway, and medical sectors. They collaborate with companies to help them meet ISO compliance and functional safety standard requirements. Solid Sands offers two main products. The first is the SuperTest Compiler Test and Validation Suite, which provides a comprehensive validation environment designed to help customers achieve the high software quality level required by ISO language and functional safety standards. It ensures the reliability and correctness of compilers through rigorous testing and validation processes. The second product is the SuperGuard Library Safety Qualification Suite, which focuses on testing and qualifying standard libraries using a requirements-based approach. It offers full traceability between the requirements derived from language specifications and individual library tests, ensuring that libraries meet the necessary safety standards. Given that this project is focused on compiler qualification and validation, the expertise and experience from Solid Sands, particularly their SuperTest product, will be useful for our research. Solid Sands has been a leader in testing and validating C and C++ compilers and we aim to leverage their methodologies to achieve similar success with non-standardized programming languages [4] .

Chapter 3

Compiler Qualification Process

This chapter explores fundamental principles of compiler qualification, and introduces examples of compiler qualification in practice. More specifically, it introduces the Solid Sands SuperTest product and the Ferrocene Rust toolchain and its relevance within the broader Rust development context. In this way, we understand the difference between the two approaches.

3.1 Compiler Qualification Principles

In the "ISO/IEC Guidelines for the preparation of programming language standards", it is noted that a programming language's standard can serve as a foundation for defining requirements to validate the corresponding compiler. This is reasonable, since by standardizing a language, it is established that the specification is stable, accurate, and comprehensive, having undergone this structured process by a standards committee. Consequently, converting this specification into a compiler validation test suite ensures confidence in the compiler's reliability in a straightforward manner. On the other hand, for a non-standardized language, the process is quite different. That is mainly because there is a lack of a complete, accurate, and stable language specification, which necessitates its creation. Subsequently, this specification is fundamental in defining requirements to develop a test suite for validating the compiler's adherence to the language's expected behavior [17].

3.1.1 Language Specification

A robust language specification, similarly with a language standard (Section 2.3), provides a precise definition of syntax and semantics, essential for defining the language's structure and meaning. The development of the language specification involves specifying requirements for error and exception detection, reporting, and handling. Through this process, the language specification development ensures a comprehensive, accurate, and stable foundation for programming languages [17].

A clear example to illustrate the meaning of a specification, is the specification of a method, which should contain a precondition, which is related to the required input of the method and a postcondition, which is related to the return value, output or item modification of the method. A specification functions as a black box and in the context of a method, the return value and the parameters should be specified, but not the local variables or the private fields of the method's class [22]. More details about language specifications can be found in Chapter 4.

Language Specification VS Language Standardization

Even though language specification and language standardization are closely related, they have different objectives and roles. As explained above, a language specification is a complete, accurate and clear formal description of the syntax and semantics of a programming language. It defines how the language should behave, which can guide developers and compiler developers to correctly implement the language. Whereas, language standardization is the process through which a language specification is formally approved by a recognized standards organization, such as the International Organization for Standardization (ISO). The goal of standardization is to ensure that the language can be utilized across different platforms and environments, promoting interoperability and reliability in software development. The

standardization process requires the collaboration of stakeholders to decide upon the language specification and produce an official standard, that can be useful for compiler developers and others.

In the context of compiler qualification, language standardization significantly encourages trust in the language specification, since it involves a language that has been accepted by a standards committee. However, a language does not have to be standardized, so that a corresponding compiler gets qualified. According to ISO 26262, a detailed, accurate and complete language specification is enough for validating that the compiler behaves as expected [1]. Even so, following the requirements of language standards (Section 2.3), can be a great guidance to create a quality language specification.

3.1.2 Compiler Test Suite

Compilers translate programs from high-level languages into executable forms. Errors in the compiler can cause the original program to be translated into an executable that behaves differently from the intended semantics of the original program. Therefore, compiler correctness and validation is fundamental to ensure reliable operation of any software developed with that compiler.

Testing is one of the most crucial methods for quality assurance and error detection in compilers. It can be done either in a custom test suite, developed by the user or by the tool vendor [1]. Traditionally, testing methods are divided into "white-box" and "black-box" methods. White-box testing creates tests based on the implementation source code, while black-box testing generates tests based only on the specification. White-box testing checks that the source program is correct, but does not necessarily ensure that the desired functionality is implemented, according to the language. Black-box methods, on the other hand, are designed to check the implementation's adherence to the language specification. That is why black-box testing often requires additional effort to develop a product specification, which includes informal syntax and semantics descriptions and formal syntax descriptions [23]. Requirements-based testing aims to prove that the program's behavior is consistent with its requirements. This includes equivalence class testing, which partitions variables' ranges into sub-classes, including normal and boundary values, and boundary value testing, which extends equivalence class testing to include values outside the normal range, enhancing robustness [24].

Typically black-box testing is used for compiler qualification because, compilers are often delivered in binary, so it is unknown how they look internally. Also, black-box testing allows qualification of multiple compilers, while targeting the same language. These tests are requirements-based because they are created based on the language specification [25]. More details about the compiler validation test suite implementation can be found in Chapter 5.

3.1.3 Traceability

Traceability in compiler qualification involves the creation of connections between sections of the language specification and the corresponding tests in the compiler validation test suite. It is required to ensure the completeness and the coverage of the tests and identify gaps that should be filled. Traceability is straightforward when a test corresponds to one specific language specification rule and requirement. However, there is often a many-to-many relationship between the tests and the specification, which makes it harder to calculate the completeness of the test suite [24]. More details about traceability can be found in Chapters 4 and 5.

3.1.4 Documentation

Thorough documentation is vital for the compiler qualification process. This includes several elements, such as a Traceability Matrix, which visualizes the connections between the language specification and the corresponding tests that were developed during validation. Moreover, detailed test reports are essential, including the tests, with their outcomes and any detected issues. It is important to be aware of the degree to which the compiler is compliant to the specification, have insight into the identified errors and exceptions, and how the compiler handles them. A clear and precise report is vital to ensure reliability and transparency throughout the qualification process. Besides, the goal of this process is not to prove that the compiler is flawless, but that there is an awareness of the compiler's errors [1].

3.1.5 Version Diversity - Compatibility

Programming languages constantly evolve to address new needs, fix issues and make improvements. These modifications can introduce compatibility issues with the older language versions and create con-

fusion in existing programs. Prioritizing backward compatibility is important to ensure that new versions do not break old version code. When an element is deprecated, its use should get gradually discouraged. Also, sometimes backward incompatibilities do not generate diagnostics, which can be misleading [26].

This emphasis on backward compatibility extends beyond languages themselves to encompass their corresponding tests and specifications. It is necessary to ensure that the compiler maintains its adherence to the specification. This can be done using version control to track language specification changes and ensure the test suite is updated accordingly. Additionally, regression testing is vital, requiring the re-execution of the test suite against new compiler versions to ensure backward compatibility. As the language specification evolves or new compiler versions are released, it is important to ensure that error and exception handling requirements are consistently met. By adopting these practices, compilers can maintain their reliability and compatibility across different versions. More details about versioning can be found in Chapters 4 and 5.

3.2 Compiler Qualification in Practice

In this section, we present two examples of compiler qualification, one for C and C++ compilers, the SuperTest product, from the company Solid Sands [4], and one for the Rust compiler, Ferrocene, from the company Ferrous Systems [27]. This analysis is important to understand the different needs and requirements in each case, due to the existence or lack of a language standard.

3.2.1 SuperTest

The SuperTest test and validation suite for C and C++ compilers and libraries, made from Solid Sands, is used for qualification of C and C++ compilers. It utilizes the ISO standards of the C and C++ languages and validates that the compilers behave accordingly. Since, C and C++ are standardized languages, no extra effort had to be made, to make a specification for these languages [4].

3.2.2 Ferrocene

Ferrocene is the first qualification of Rust achieved by the German company Ferrous Systems [27]. The Ferrocene qualification includes the assessment of the Ferrocene compiler's functionality, on the Rust versions 1.68.2 and 1.76.0, which can now be used in safety-critical environments, especially in the automotive [12].

Specifically, Ferrocene has been qualified for use in systems up to ASIL D, which is the highest classification of initial hazard as defined by the Automotive Safety Integrity Level (ASIL) standard [1]. In line with the common practice described in Section 2.2.1 the "Validation of the software tool" qualification method was chosen for Ferrocene [12].

Each Ferrocene release corresponds to a specific Rust version. The initial release includes Rust 1.68.2, while the second major release will include Rust 1.76.0. To validate the compiler, the Ferrocene team had to create the first Rust specification. The Ferrocene Language Specification [28] is currently the only existing Rust specification. It was inspired by the Ada Reference Manual, as Ferrous Systems collaborated with AdaCore [29] during its creation. Consequently, the two specifications share a similar format. More details about the Ferrocene Specification can be found in Section 4.2. Ferrocene incorporates the Rust test suite without including additional tests of their own [30]. However, the existing tests have been "organized" to demonstrate the completeness of their validation process. The specification was linked to the already existing test suite, by the addition of a traceability mechanism. In the Traceability Matrix [31], as can be seen in Figure 3.1, the specification is linked to the tests, to ensure that all the documented requirements are covered by the validation test suite. It provides visibility into the tests written for each section of the specification and the test coverage percentage. More information on how this was achieved can be found in Section 4.2.4 and 5.2.3.

3.2.3 Rust Development Process

Since Ferrocene is built upon the Rust project, serving as a fork that utilizes Rust's test suites and adheres to its release plan and versioning schema, understanding the Ferrocene qualification process requires an analysis of the Rust development process. This analysis also helps to understand the steps taken and the level of quality achieved in Rust's development, providing a foundation for assessing and ensuring compiler reliability [1].

Figure 3.1: Ferrocene Traceability Matrix [31]

Continuous Integration - Test-focused Development

As an open-source project, Rust emphasizes testing when introducing new features [32]. Each new feature should undergo thorough testing, and the project includes a comprehensive suite of tests to ensure quality and reliability. Test-focused development and Continuous Integration (CI) are essential parts of the development process, helping to capture errors early before review. When changes are pushed, the compiler is automatically built and tested against the test suite, including tests for compliance with Rust's style guidelines. Writing comprehensive tests for new features is mandatory, and pull requests lacking tests are not accepted. Finally, pull requests undergo peer review, with reviewers rotating regularly to ensure thorough examination. This rigorous development process ensures the stability and reliability of Rust's features [33].

Request for Comments

Rust also employs the Request for Comments (RFC) [34] mechanism to propose and discuss new features in a documented and consistent way. This mechanism involves drafting an RFC document, which is then reviewed and discussed within the community. After incorporating feedback, the relevant sub-team reviews the RFC, and if considered ready, a Final Comment Period (FCP) starts for final objections or comments. Following the FCP, the RFC is either merged or closed based on the consensus.

Overall, this mechanism offers several benefits to the language's development process and success. By providing a structured way for proposing and discussing new features, the RFC process encourages active participation from the Rust community and promotes diversity of perspectives. Through thorough review and discussion, potential issues can be identified, leading to better-quality proposals that improve the stability and reliability of Rust while ensuring changes are carefully considered and aligned with the project's goals. Moreover, all stages of this process are documented, adding transparency in the decision-making process.

Rust Versioning

According to [35], Rust releases a new version every 6 weeks, ensuring a rapid pace of development and iteration. Additionally, a new edition is introduced approximately every 3 years [36]. This versioning strategy maintains a balance between frequent updates and long-term stability.

Backward compatibility is a key principle in Rust's versioning approach. New language features are designed to be supported in all future versions, ensuring that code written with newer features remains compatible with older code. The extensive test suite also contributes to maintaining compatibility across versions. However, when changes are proposed that would introduce backward incompatibility, a new edition is created. For example, in the 2015 edition, `‘async‘` might be usable as a variable name, but in the 2018 edition, it becomes a reserved keyword. Editions are not isolated versions of the language or compilers frozen in time, they evolve alongside the language, staying in sync with the latest Rust versions.

In summary, Rust's versioning strategy combines regular updates with long-term stability, prioritizing backward compatibility and interoperability across editions. This approach enables developers to benefit from new language features while maintaining compatibility with existing code.

Rust Release Trains

Additionally, Rust's versioning process undergoes rigorous checks to ensure stability. New features follow a sequence through different release channels. The Nightly channel follows the master branch of the repository and includes unstable, incomplete, or experimental features that are available with feature gates. The Beta channel contains the upcoming release, which is expected to become stable within six

weeks. The Stable channel features the latest stable release, intended for general usage. This release process ensures that new features are thoroughly tested before reaching general use. Issues in the Nightly and Beta stages are identi ed and xed early, improving the reliability of the Stable release (Figure 3.2) [37].

Figure 3.2: Rust Release trains [12]

Language VS Compiler

The language and the compiler of a language are two terms closely related. The language includes the syntax, semantics, and rules that de ne how a program written in this language should be structured and function. The compiler is the tool that translates code written in this language into machine code that can be executed. It implements the rules de ned by the language.

For example, in the case of C and C++, popular compilers include GCC [38] and Clang [39]. These languages have standards, maintained by organizations like ISO, which de ne them and ensure that C and C++ code is portable across compliant compilers. This multiplicity of standards and compilers provides more choices and can help nd issues of the language. However, it can also lead to inconsistencies, as no single compiler can serve as the de nitive implementation of the language.

In contrast, Rust has a single o cial compiler, rustc [40], alongside a few alternative ones, such as Mrustc [41], and Rust-GCC [42], which are not fully usable yet. Rust and rustc are developed together, in the same repository, by the same team [43]. Rustc de ned the Rust language. This close integration ensures that language features and compiler behaviors are synchronized, providing consistency across platforms. Furthermore, Rust has a uni ed toolchain, including cargo, the Rust tool for package management and building, which simpli es the development process and reduces the e ort required to ensure consistency.

Overall, Rust bene ts from the simultaneous evolution of the language and its compiler, whereas C/C++ have multiple compilers developed independently from the language standards. This makes it easier for Rust to maintain consistency and reliability across di erent environments.

All in all, from these facts, we can make several conclusions. The Rust team has put a lot of e ort in testing the compiler, in parallel of developing new features. If there is a high test coverage, it can really make the quali cation process quicker and ensure con dence faster. With the Request for Comments mechanism, but mostly with the release management the Rust team uses, they ensure stability of the features. This is positive because it indicates that no random pull requests are merged in the stable branch, but that new features have to undergo reviewing from the Rust team, which ensures their quality. Therefore, having stable features means having a stable compiler, which again makes the con dence stronger. The Rust versioning and the editions mechanism proves that backward compatibility is a concern and priority of the Rust team. This is positive for the compiler quali cation process, since the tests must also respect this principle. However, of course, forward compatibility is still a concern. From this, we can also understand why Ferrocene includes the quali cation of a speci c version. Rustc adheres closely to the Rust language, since they have been developed together, by the same team and in parallel. This stability of Rust is the argumentation used for Ferrocene to not add any new tests to the Rust test suite. However, the language is not de ned yet entirely and o cially and this is the gap Ferrocene tries to cover.

Chapter 4

Language Specification

Language specifications help avoid misunderstandings and make it easier to detect errors in the compiler, preventing wasted time on locating bugs in irrelevant areas. A language specification serves as a definitive guide for the language and acts as a firewall between the client and the implementer. This specification allows the client to use the language as a black box, focusing solely on the behavior of its components without needing to understand their inner implementation. Similarly, the implementer can develop and maintain the language according to the specification without needing to consider every possible client use case. By respecting the specification, both parties can use and implement the language independently and effectively [22].

The Rust Reference, while useful, is not a complete language specification. Many parts of the language are missing or are not explained adequately and in the right way. This can be seen by this simple example of a small syntax and semantics defect we found and contributed to the Rust Reference[1]. The Rust team is interested in developing a comprehensive specification to define valid Rust programs, aiming for both prescriptive and descriptive clarity across versions. They acknowledge this is a challenging task, initially focusing on the current Rust version and recommending a subset for the beginning. They emphasize completeness, correctness, and accessibility, with references in natural language and hyperlinks, and highlighting differences between versions. While they aim for specifications to accompany the Rust releases, initial delays are acceptable to avoid disrupting Rust's development [15].

An RFC explains the benefits of a Rust specification for various users, particularly authors of unsafe code and safety-critical software developers, who need clear definitions of the language behavior. It would also help in discussions of new features. Standardization relies on having an accurate specification, a process for language evolution, and stability, of which only the specification is currently missing. The team has concerns about the scope, formatting, structure, starting point, depth, and use of formal language in the specification [21].

Furthermore, the Rust Foundation recently announced about creating a Safety-Critical Rust Consortium, including big automotive companies to support the responsible use of Rust in safety-critical applications. It is mentioned that there will be collaboration with the Rust Project Language Specification Team [9] and the Formal Methods team to write a language specification for Rust [11].

In this chapter, we present some further requirements on language specifications, found from ISO documents and other sources. Next, we compare different language specifications of C, C++, Ada, Ferrocene and the Rust Reference and we come up with best practices for developing a language specification. Then, we specify how the Ferrocene specification is an improvement of the Rust Reference.

4.1 Guidelines for Language Specifications

In this section, we combine guidelines from ISO documents on how a language specification should be.

4.1.1 Quality

The quality of a language specification is based on a few criteria. One criterion is the ease with which a third party can create an implementation that is compliant with the specification and can handle existing programs, using only the specification. Another criterion is the number of defects found in the specification, as reported in ISO Defect Reports. For example, 215 defects were reported for the Ada95

[1] union syntax x

Standard, leading to 116 changes in Ada99, and 178 defects for the C90 Standard, with about half leading to changes in C99. However, it is important to note that the discovery of defects can be a ected by secondary factors like the number of people reading and paying attention to the speci cation. Moreover, the e ort involved in developing the speci cation can indicate commercial interest within the language's development, as well as attention to detail. Finally, the ISO language vulnerability working group (Open Web Governance & Veri cation - OWGV) considers the ease of extracting reliable information from the speci cation for creating generic guidelines, the most crucial quality attribute of a speci cation [44].

4.1.2 Methods

Language speci cations can be structured either through machine-executable forms or prose descriptions, each with di erent advantages and disadvantages. Machine-executable speci cations can be further divided into model implementations, which prioritize clear language de nition over runtime e ciency (e.g., Pascal, C), and production use implementations, which focus on execution in production environments, often sacri cing readability (e.g., PERL). Prose speci cations also have two approaches, one that speci es implementation behavior, requiring readers to derive source code behavior (e.g., C++), and another that speci es source code properties, from which implementation behavior is derived (e.g., C, Fortran, Java). The OWGV (Open Web Governance & Veri cation) nds prose speci cations easier to manage due to their clarity and abstraction of requirements. However, existing languages often come with predetermined speci cation methods. For instance, languages like PERL and PHP are mainly de ned through implementation. [44]

4.1.3 Metalanguages

Formal methods and metalanguages can help provide precision and eliminate ambiguity, which is a common problem with natural language. However, the standard committee also evaluates factors such as feasibility, accessibility to non-specialist readers, and compatibility with existing standards or practices, when deciding to use formal methods. If formal methods are included in the language speci cation, they should be analyzed in detail, either by citing external standards or providing de nitions within the standard itself, potentially in an annex. Informal descriptions of formal methods should be included to help comprehension. This combination of formal and informal explanations ensures that the speci cation is both precise and accessible, facilitating widespread use [17].

A syntactic metalanguage is crucial for de ning the syntax of languages formally, through rules that describe non-terminal symbols and their forms. There is a plethora of notations available, which leads to confusion and consequently, under-appreciation of formal de nitions. Extended BNF, which is based on BNF (Backus-Naur Form), aims to standardize these de nitions and address this issue. A standard metalanguage helps express ideas into unambiguous de nitions and it should be concise, precise, formal, natural, general, simple in character set, self-describing, and linear. This makes it easier to understand, process by computers, and suitable for many purposes, including de ning di erent languages and processing syntax e ciently [45].

4.1.4 Content

Extracting information from both implementation-based and prose-based speci cations demands signi cant e ort. In implementation-based speci cations, di erentiating between algorithmic details and language requirements is crucial, while in prose speci cations, understanding the terminology and its application is important. Language speci cations typically adhere to a compile/link/execute sequence, ensuring that all phases are completed sequentially before program execution. Languages like Ada, C, C++, Fortran, and Java necessitate source code conformance checks prior to execution, facilitating the resolution of violations without relying on program ow. Additionally, linking processes in most languages involve minimal requirements, primarily ensuring that symbols used in the code have corresponding de nitions available during linking. Furthermore, certain language constructs may exhibit undesirable behavior only under speci c conditions, such as division by zero or arithmetic over ow. Detecting such behaviors statically can pose challenges, which indicates the importance of tools and detailed, accurate speci cations of language semantics. Overall, this is an e cient way to gather requirements for a language speci cation [44].

4.1.5 Verbal Forms

To avoid misinterpretations of natural language in language speci cations, ISO/IEC has de ned rules for the use of verbal forms, to clearly di erentiate between requirements, recommendations, permissions, possibilities and constraints (Table 4.1). Di erent verbal forms should not be used [46].

Category	Preferred Verbal Form	Alternatives	Notes
Requirement	shall	is to, is required to, it is required that, has to, only ... is permitted, it is necessary	Avoid using "must" as an alternative for "shall".
Recommendation	should	it is recommended that, ought to	
Permission	may	is permitted, is allowed, is permissible	Avoid using "possible" or "impossible" in this context. Negative permissions are ambiguous; rewrite to state what is permitted instead. Avoid using "might" or "can" instead of "may".
Possibility	can	be able to, there is a possibility of, it is possible to	Do not use "may" instead of "can" in this context.
External Constraint	must		Avoid using "must" as an alternative for "shall".

Table 4.1: Verbal Forms in Language Speci cations [46]

Indeed, there can be a lot of ambiguity in natural language [47] and speci c words should be used with care. Some examples to illustrate this, can be found in table 4.2.

4.2 Comparison of Language Speci cations

In this section we compare di erent language speci cations, standards and language manuals, regarding the scope, structure, and how they handle traceability and versioning. More speci cally, we compare the C[5] and C++[6] standards, with the Ferrocene Speci cation[28] and the Rust Reference[20]. We also analyze the di erences between the Ferrocene Speci cation and the Rust Reference and why the former is more suitable as a language speci cation. By this comparison, we can make some conclusions on good practices in the development of a language speci cation.

4.2.1 Scope

The scope of the speci cations is about their content with regards to the language they are trying to de ne.

C Standard

The standard de nes the form and interpretation of programs written in the C programming language, aiming to improve the portability of C programs across di erent data-processing systems. It is intended for both implementers and programmers. The standard covers the structure of C programs, the syntax, the semantic rules for interpreting C programs, the format of input and output data, and the constraints required by a conforming implementation of C. However, it does not specify the means by which a C program is compiled or executed by a data-processing system, the means by which input data are used or output data are handled after being produced by a C program, the size or complexity of a program

Ambiguity	Considerations for Clarity
"Any"	Explain if it means "singular" or "plural".
"Include"	Explain if it means "consists of" or "contains as a subset".
"Minimum" and "Maximum"	Explicitly state intervals (open or closed).
"Or"	Explicitly specify whether inclusive (one or both) or exclusive (one or the other, but not both).
"Only" and "Also"	Placement a ects sentence meaning.
"This"	Ensure "this" refers to a noun, not a whole idea.
"Otherwise"	Explicitly show which condition "otherwise" refers to.
"Not"	Placement is important to avoid unintended negation.
Logical Operators	Explicitly clarify precedence and associativity with "and" and "or."

Table 4.2: Ambiguities in Natural Language [47]

and its data that will exceed the capacity of any particular data-processing system or processor, or all minimal requirements of a data-processing system capable of supporting a conforming implementation.

C++ Standard

This document de nes C++ and speci es requirements for implementations of the C++ programming language. It contains the same information as the C standards.

Ada Speci cation

The Ada Reference Manual speci es the structure of an Ada program, the e ect of compiling and running such a program, and the method for combining program units to create Ada programs. It also details the libraries that a conforming Ada implementation must provide, the allowed variations in conformance to the rules and how they should be documented, the violations that a compliant tool must detect and the consequences of attempting to compile or run a program with such violations, and the violations that a compliant tool is not obligated to detect. However, the Ada Reference Manual does not specify the process of transforming an Ada program into object code executable by a processor, the methods for invoking the compilation or execution of Ada programs and controlling the execution units, the size or speed of the generated object code, or the relative execution speed of various language constructs. Additionally, it does not cover the format or content of any listings produced by a tool, including error or warning messages, the e ects of unde ned behavior, or the program or program unit size that exceeds the capacity of a compliant tool.

Rust Reference

The Rust Reference serves as a reference for stable Rust, assuming readers already have background familiarity with the language. It encourages treating the compiled program as a black box, where behavior conforms to what is speci ed in the reference. The Reference is not normative and may include details speci c to rustc, not serving as a speci cation for the Rust language. However, it describes language constructs, memory and concurrency models, runtime services, linkage model, and debugging facilities. It is incomplete, and documenting everything takes time. It includes syntax of the grammar, using a custom notation and containing links to relevant sections.

Ferrocene Speci cation

This document speci es the structure of a Rust program, the e ect of compiling and running such a program, and the methods for combining crates and modules to create Rust programs. It also outlines the libraries that a conforming Rust implementation must provide, the violations that a compliant tool must

detect and the consequences of attempting to compile or run a program with such violations, as well as the violations that a compliant tool is not obligated to detect. However, the document does not specify the process of transforming a Rust program into object code executable by a processor, the methods for invoking the compilation or execution of Rust programs and controlling the execution units, the size or speed of the generated object code, or the relative execution speed of various language constructs. Additionally, it does not cover the format or content of any listings produced by a tool, including error or warning messages, the e ects of unde ned behavior, or the program or program unit size that exceeds the capacity of a compliant tool.

Synthesizing the information gained from these speci cations and from our knowledge about the language standards (Section 2.3), we make the following conclusion:

> Finding 1: The language speci cation should include the syntax and semantics of the language, potential constraints, accepted input and output, the behavior during compilation and execution, and the violations that a compliant compiler is required to detect. It should exclude details of intermediate data processing stages.

The language speci cation should give us the right information to understand rules of writing code in this the language, the meaning of this code, the behavior, during compilation and during execution, with valid and invalid input, we should know what the expected output is, potential constraints in its use and what errors should be detected and handled. It is important that a language speci cation remains abstract, independent of any particular implementation, so that it can be used across di erent platforms and compilers. In this way, also, freedom and innovation are encouraged, so more and more optimized compilers are created according to the speci c needs.

4.2.2 Structure

The structure of the speci cations is about how their content is organized in subsections for each feature.

C Standard

The C standard speci cation de nes the features of the language by providing detailed information in several key sections. It includes the syntax of the grammar using a custom notation, constraints that limit how features can be used, and the semantics which explain the behavior and meaning of features. Additionally, it o ers recommended practices for best usage, speci es implementation limits for various aspects of the language, and includes forward references to related sections for further information. The description section gives a detailed explanation of the feature's purpose and usage. Examples illustrate usage of the constructs.

C++ Standard

It contains syntax of the grammar, as can be seen in Figure 4.1, using a custom notation and examples to illustrate how the language behaves. Other than that, the document is not separated into subsections belonging to a feature, such as semantics, rules, constraints.

Ada Speci cation

The speci cation contains several key sections. Syntax speci es the syntax rules of the grammar using a simpli ed variant of Backus-Naur Form, including links to speci ed features. Name Resolution Rules de ne compile-time rules for name resolution and overload resolution. Legality Rules describe compile-time rules, indicating that a construct is legal if it adheres to all Legality Rules. Static Semantics detail the compile-time e ects of each construct. Post-Compilation Rules specify rules enforced before running a partition, stating that a partition is legal if its compilation units adhere to all Post-Compilation Rules. Dynamic Semantics explain the run-time e ects of each construct. Bounded (Run-Time) Errors identify situations resulting in bounded run-time errors. Erroneous Execution describes situations leading to erroneous execution. Implementation Requirements list additional requirements for conforming implementations. Documentation Requirements specify documentation needs for conforming implementations. Metrics de ne time and space properties for executing certain language constructs. Implementation Permissions grant additional permissions to the implementer. Implementation Advice provides optional

Figure 4.1: C++ Syntax Example [6]

advice with "should" indicating a recommendation, not a requirement and Examples illustrate possible forms of the described constructs.

One could argue that the Ada speci cation has many sections, making it a bit hard to di erentiate between them. For simplicity, Compile-Time Rules can contain the Legality Rules, Name Resolution Rules, Static Semantics and Post-compilation Rules. Run-time Rules can contain the Dynamic Semantics, Bounded Errors and Erroneous Execution and the Metrics. And Implementation Requirements can contain the Implementation Guidance and Advice.

Rust Reference

It contains syntax of the grammar, using a custom notation and examples to illustrate how the language behaves. Other than that, the speci cation of speci c features do not contain subsections.

Ferrocene Speci cation

The document contains several subsections for its features: Syntax, which represents the syntax of a construct using a simple variant of Backus-Naur form and includes links to related features as shown in Figure 4.2, Legality Rules, specifying compile-time rules for each construct, Dynamic Semantics, specifying the run-time e ects of each construct, Unde ned Behavior, identifying situations leading to unde ned behavior or unbounded errors, Implementation Requirements, specifying additional requirements for tools to be conforming and Examples, providing illustrations of possible forms of a construct.

After combining all these ndings and grouping overlapping sections, to end up with a more simple, clear and concise result, we can come up with the following conclusion:

> Finding 2: The language speci cation should include the syntax in a metalanguage form (which should be explained), compile-time rules and runtime rules. Also, unde ned behavior should be speci ed and examples should be illustrated to showcase the rules. Implementation-speci c requirements and description of the features should also be stated. Optionally, recommended practices that explain also how to avoid e.g. unde ned behaviors, can be used. And references to relevant sections with links can be convenient.

Separating this information in sections, helps ensure maintainability and testability of the speci cation, since it is easier for the compiler tester to understand what should be tested and in what way,

Figure 4.2: Ferrocene Syntax Example [28]

whether it is expected that this program will compile, run or fail.

4.2.3 Versioning

This section examines how different standards and specifications handle multiple versions. Including many versions in one language specification, can offer maintainability and portability.

C Standard

There are revisions of the C standard: C89, C99, C11, C17, C23. These revisions introduce new features and improvements, in addition to the previous standards. They contain a section, explaining major changes from the previous editions.

C++ Standard

There are revisions of the C++ standard: C++98, C++03, C++11, C++14, C++17, C++20, C++23. These revisions introduce new features and improvements, in addition to the previous standards. They also contain an annex explaining compatibility features (Figure 4.3).

Ada Specification

There are revisions of the Ada standard: Ada 83, Ada 95, Ada 2005, Ada 2012, Ada 2022. These revisions introduce new features and improvements, in addition to the previous standards. Major changes are specified in separate documents.

Rust Reference

The latest release of the Rust Reference, matching the latest Rust version, can always be found at https://doc.rust-lang.org/reference/. Prior versions can be found by adding the Rust version before

Figure 4.3: C++ Annex: Compatibility Features [6]

the "reference" directory. For example, the Reference for Rust 1.49.0 is located at https://doc.rust-lang.org/1.49.0/reference/.

However, there are some parts in the Reference, where differences of features between different versions and editions are specified, as can be seen in Figure 4.4.

Figure 4.4: Rust Reference: differences between versions [20]

Also, in other places, there are warnings that some features will not be valid in future versions, as can be seen in Figure 4.5. In this way, again, the Rust team achieves backward compatibility.

Figure 4.5: Rust Reference: warning about future version [20]

Ferrocene Specification

Currently, there are two versions of the language specification, referring to two Rust compiler versions. Also, the specification only includes the 2021 Rust edition (Figure 4.6). That is why the document does not refer to specific version or edition behavior.

Synthesizing the information gained from these specifications, we make the following conclusion:

Figure 4.6: Ferrocene Edition [12]

> Finding 3: The language speci cation should explicitly detail any di erences between versions of a language or future intentions to deprecate a feature. Prioritizing backward compatibility is crucial for implementations to ensure that existing code continues to function as expected across di erent language versions.

Because languages improve, it is hard to maintain complete backward compatibility, that is why these improvements should be known. By clearly outlining them, developers can understand the impact on their code and make necessary adjustments to maintain compatibility and functionality.

4.2.4 Traceability

It is useful to see how the di erent speci cations are structured, in a way that can be utilized to link a possible test suite to the speci cation.

C Speci cation

The C standard contains numbers in each section and paragraph, as can be seen in Figure 4.7, which can be utilized from the quali cation procedure for traceability, when creating a test suite.

Figure 4.7: C Standard Traceability [5]

C++ Specification

The C++ standard contains numbers in each section and paragraph, which can be utilized from the qualification procedure for traceability, when creating a test suite. Also, tags are provided, such as [namespace.alias] , so that if the section number changes in a future section, it can still be retrieved based on the tag.

Ada Specification

The Ada standard contains numbers in each section and paragraph, which can be utilized from the qualification procedure for traceability, when creating a test suite.

Rust Reference

The Rust Reference only contains numbers for sections, such as "Types", but not for clauses.

Ferrocene Specification

The Ferrocene specification contains numbers in each section and paragraph, for easy reference to the specification, as can be seen in Figure 4.8. But other than the numbers we see, a hidden, unique id is assigned to each clause. This practice can add maintainability and re-usability to the whole process. If a clause of the specification gets removed in a new version of the specification, the corresponding tests will just be deleted. If the clauses are reordered because of the addition of a new clause or for some other reason, nothing will change for the clauses, since they have been assigned a permanent unique id. Only the clause numbers change in the specification, which happens automatically, the way the specification has been developed.

Synthesizing the information gained from these specifications, we make the following conclusion:

> Finding 4: Assigning a unique id to each clause can facilitate traceability and help ensure test suite completeness, while also ensuring maintainability and easy development of the specification. Section numbering is good for reference to the specification but can be hard to maintain as a traceability mechanism.

4.2.5 Rust Reference VS Ferrocene

We investigated why relying only on the Rust Reference is not enough and how the Ferrocene Specification can cover that gap. The Rust Reference is mostly made to help developers learn the language. So, in some cases it can be more practical, but at the same time does not go in depth in the constructs. For example, in the section of "unions", there are subsections, such as "Initialization of a union", or "Reading and writing union fields". Indeed, this might be convenient for a developer that is looking for quick answers. However, in a formal specification these subsections would be part of different sections, that would define in detail the syntax and semantics of these constructs, that would be named Field access expressions and Union initialization expressions, as is done in Ferrocene. Furthermore, in the Rust Reference, the specifications are presented as a whole, without any more explanation. On the contrary, in the Ferrocene Specification the specifications are organized in sections, such as Legality Rules or Dynamic Semantics. This makes it much easier and faster to create a corresponding test suite, since one can know whether the test should not compile or should fail. Moreover, in the Rust Reference, there is a lack of Traceability. On the contrary, in the Ferrocene Specification, sections and paragraphs are numbered, so that they can be referenced in the compiler validation test suite. It is the only safe and fast way to ensure completeness of the validation. In the Ferrocene Specification, the syntax given for the grammar of the language is often clearer and more user-friendly, since definitions are split in more parts, making it a more versatile grammar. Also, in the Ferrocene Specification, the semantics are more organized and simply expressed. For example, in Figure 4.8, it can be seen how the specification is written almost in a test requirements form.

Figure 4.8: Ferrocene Semantics Example [28]

Chapter 5

Test Suite Implementation

In this chapter, we discuss compiler testing methods and we compare di erent test suites, SuperTest, the Rust test suite and the Ada Conformity Assessment Test Suite, to come up with some best practices on compiler testing.

5.1 Compiler Testing Methods

Developing a robust test suite is essential for validating that a compiler adheres to the language speci - cation. The test suite must cover all aspects of the language, including syntax and semantics, to verify that the compiler accurately interprets the language's grammar and meaning. Additionally, it should include edge cases, with unusual or extreme values, to assess the compiler's ability in handling di erent scenarios. By creating test cases targeting identi ed errors and exceptions, and ensuring high coverage of various cases, the test suite can e ectively validate the compiler's error and exception handling capabilities. Also, tool validation typically involves black-box testing [22]. This approach ensures that the compiler aligns with the language speci cation and performs reliably.

5.1.1 Syntax Testing

The syntax of a programming language is formally de ned by a grammar. Positive test cases validate syntactically correct programs, con rming the compiler's ability to recognize valid syntax. Negative test cases, on the other hand, assess the compiler's capability to detect syntax errors in programs [23].

5.1.2 Static Semantics Testing

Static semantics include rules for computing program properties like variable and expression types, applicable only to syntactically correct programs. Positive test cases validate programs that satisfy context conditions, while negative test cases identify static errors in programs that violate context conditions [23].

5.1.3 Dynamic Semantics Testing

Dynamic semantics de ne the behavior of programs during execution. Testing dynamic semantics includes compiling statically correct programs, executing them, and comparing their behavior against expected outputs de ned by the language's dynamic semantics. Positive test cases validate programs that run and produce the expected output, while negative test cases identify dynamic errors in programs that do not produce the expected output [23].

5.2 Comparison of Compiler Test Suites

In this section, we compare di erent test suites, SuperTest [4], the Rust test suite [43] and the Ada Conformity Assessment Test Suite (ACATS) [48], regarding the types of tests they contain, the way the compare the test output to the expected one, the way they handle traceability and multiple compiler versions, and the testing methodology they use. SuperTest and ACATS are mature products that can

teach us some signi cant lessons on compiler validation. By comparing these test suites, we can make some conclusions on good practices in the development of a compiler test suite.

5.2.1 Types of Tests

There are positive and negative tests, there are also tests that check the syntax, the compiler output, or the execution output. We investigated what types of tests these test suites include.

Rust Test Suite

The o cial Rust source code repository [43] includes various test suites, each serving di erent purposes. Among these, the ui test suite, located in the tests/ui directory of the Rust repository, stands out for validating the compiler's behavior, focusing primarily on compilation output. By default, UI tests in Rust aim to detect compile errors, focusing on assessing invalid input and error diagnostics. However, it is also possible to create UI tests that compile successfully and then execute the compiled programs. For this purpose, several header commands are available. Some of them are the following [49]:

Pass Headers

```
//@ check-pass | Compilation should succeed but skip codegen.
//@ build-pass | Compilation and linking should succeed but not run the binary.
//@ run-pass | Compilation and running the binary should both succeed.
```

Fail Headers

```
//@ check-fail | Compilation should fail (with skipped codegen).
//@ build-fail | Compilation should fail during codegen phase.
//@ run-fail | Compilation succeeds, but running the binary fails.
```

Solid Sands SuperTest

There are two types of tests in the test suite, positive tests and negative tests, each corresponding to a di erent le name pattern. Positive tests are correct C/C++ programs for which the behavior is de ned by the language speci cation. These tests are self-testing and should compile and run successfully in order to pass. For negative tests a diagnostic is expected by the compiler, and compilation is expected to fail. So a negative test passes when its compilation fails, and it fails when the compiler compiles without an error. No attempt is made to run a negative test, since it has no de ned semantics. Furthermore, Solid Sands has ensured that unde ned behavior is not existent in the test suite.

Ada Conformity Assessment Test Suite

The test suite includes various types of tests to evaluate a compiler's reliability. These tests cover different aspects, such as ensuring that valid program constructs compile and execute correctly, detecting and handling illegal code to prevent unintended behaviors, verifying the accurate execution of executable parts of the code with expected outcomes, and testing the handling of large numerical values to ensure precise arithmetic operations. Additionally, there are tests that involve manual inspection to con rm correctness under speci c conditions, and checks for dependencies and pragmas within library units to ensure proper processing.

Synthesizing the information gained from these test suites, we make the following conclusion:

> Finding 5: A robust validation process should include multiple test types, that evaluate di erent aspects of language behavior. This should include both positive tests and negative tests, to expose compiler issues. Additionally, testing should include the syntax, as well as the compilation and execution of a construct, to ensure the reliability of the compiler. Unde ned behavior should not be included in the test suite, since it does not o er any valuable conclusion about the compiler.

5.2.2 Output Comparison

Checking if the output of a test program is the expected one, is a challenging task. We investigated how these test suites handle it.

Rust Test Suite

Rust's UI tests store expected compiler output in .stderr and .stdout files, mirroring the test file names. These files are manually inspected to ensure they match expected output. Output normalization overlooks platform-specific differences such as file paths and standardizes output. For instance, directories are replaced with placeholders like $DIR, as can be seen in Listing 5.1.

```
error[E0606]: cannot cast `usize` to a pointer that is wide
  --> $DIR/cast-macro-lhs.rs:8:23
   |
LL |     let x = foo!() as *const [u8];
   |             ^^^^^^^^^^^^^^^^^^^^^ creating a `*const [u8]` requires both an address and a length
   |
   = consider casting this expression to `*const ()`, then using `core::ptr::from_raw_parts`

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0606`.
```

Listing 5.1: cast-macro-lhs.stderr [43]

Additionally, substrings of the errors are annotated within the source code as well. This redundancy serves multiple purposes. Firstly, it helps prevent errors, as .stderr files are often auto-generated. Secondly, it provides a clear indication of where error spans are expected, simplifying error analysis compared to comparing the .stderr file with the source. Lastly, it ensures that no additional unexpected errors occur. The error annotations of the program that generated the above-mentioned stderr can be seen in Listing 5.2.

```
// Test to make sure we suggest "consider casting" on the right span

macro_rules! foo {
    () => { 0 }
}

fn main() {
    let x = foo!() as *const [u8];
    //~^ ERROR cannot cast `usize` to a pointer that is wide
    //~| NOTE creating a `*const [u8]` requires both an address and a length
    //~| NOTE consider casting this expression to `*const ()`, then using `core::ptr
        ::from_raw_parts`
}
```

Listing 5.2: cast-macro-lhs.rs [43]

This approach is rather unstable, because, even though normalization is performed, it is impossible to predict what error message every compiler would give, especially since this information is not part of the language specification. Of course, it makes sense that this works for Rust, which only has one usable implementation at the moment.

Solid Sands SuperTest

A case-analysis is done for four cases:

- A negative test where the output contains the expected diagnostic. This is correct.
- A positive test that does not generate a diagnostic. This is also correct.
- A negative test that does not generate the expected diagnostic. This indicates an error of the compiler.
- A positive test that generates a diagnostic. This also indicates an error of the compiler.

To verify that the correct diagnostics are produced for negative tests, the test driver parses the errors generated by the compiler and returns the line number for which a diagnostic was given. The lines

which contain diagnostic are marked with /* Diagnostic expected */ , manually by the developers that wrote the tests.

Solid Sands previously did not verify the speci c line where errors occurred, which caused instability and risked misidentifying the cause of program failures. By con rming that errors occur on the expected line, Solid Sands now ensures that tests fail for the anticipated reasons. This enhancement signi cantly improves accuracy, since it is improbable that an error would occur on the speci c line, for a di erent reason than the expected one.

Ada Conformity Assessment Test Suite

Lines that contain errors are marked { ERROR: and generally include a brief description of the illegality on the same or following line. Some tests also mark some lines as { OK, indicating that the line must not be agged as an error. An implementation passes a class B test if each indicated error in the test is detected and reported, and no other errors are reported. The test fails if one or more of the indicated errors are not reported, or if an error is reported that cannot be associated with one of the indicated errors. The error message is not taken into consideration for the detection of an error, only the location of the error, similarly with SuperTest.

Synthesizing the information gained from these test suites, we make the following conclusion:

> Finding 6: To enhance the compiler validation test suite, a mechanism should be implemented to compare the compiler output of a program to the expected output. One of the most stable and reliable approaches is to identify the line where the error occurred and check if it was the expected line. This approach ensures that errors are detected and reported reliably, even without explicitly validating the error message content.

5.2.3 Traceability

We investigated how the test suites are connected to the speci cation, in order to provide traceability.

Rust Test Suite

Comprehensive naming for test les and corresponding output les based on functionality tested contribute to the overall e ectiveness of Rust's UI testing framework.

Enhancing Traceability with Ferrocene

The Ferrocene toolchain, a fork from the Rust repository on GitHub, closely parallels its parent repository, frequently pulling updates from it. Ferrocene enriches its test suite with traceability features, enhancing its adherence to speci cations. Each section and paragraph in the speci cation is assigned a unique identi er, which is then included as annotations within relevant test les, establishing a clear link between tests and the speci cation (Figure 5.1). A test can be linked to multiple sections of the speci- cation. The resulting Traceability Matrix [31], as can be seen in Figure 3.1, provides a comprehensive overview, enabling easy identi cation of test coverage for speci c sections. This systematic approach ensures thorough testing alignment with the de ned speci cations, encouraging transparency and con- dence in the Ferrocene toolchain's reliability. However, the traceability mechanism is not used to its full potential yet, because the tests written are only matched with speci cation sections and not each speci c clause. In this way, there can not be full con dence of completeness of the test suite.

Solid Sands SuperTest

The organization of tests within SuperTest follows a logical folder structure, with directories named after the versions or the sections of the standards they correspond to. This structure facilitates easy navigation and management of the extensive collection of tests, ensuring e cient testing coverage across di erent language speci cations. An illustration of the SuperTest folder structure is provided in Figure 5.2.

In the standards of C and C++ there is no unique id for the clauses, only section and clause numbers. This does not help much the re-usability of SuperTest, because when, for example new clauses are added, in the improved standards, the clause numbers are changed and should be updated in the folder names

Figure 5.1: Ferrocene Traceability Annotations [50]

Figure 5.2: SuperTest folder structure

of the test suite. To handle this small obstacle, Solid Sands manually matches in a csv le the old with the new clause numbers and in a semi-automatic way, updates the test suite.

Ada Conformity Assessment Test Suite

As can be seen in Figure 5.3, a strict naming convention is used for the test le naming, indicating the speci cation section that every test refers to, o ering in that way traceability.

Synthesizing the information gained from these test suites, we make the following conclusion:

> Finding 7: A traceability mechanism should be established to connect the tests to the speci- cation. This connection can be achieved by either associating the test le name with the cor- responding speci cation section or by embedding a unique identi er, assigned to a speci cation clause and section, within a comment in the test le. The latter approach o ers greater exibility, enabling a many-to-many relationship between tests and speci cation clauses. Additionally, it allows for more user-friendly test le names, which can signi cantly facilitate the development of the validation test suite.

5.2.4 Versioning

We investigated the way the test suites handle many di erent versions of the language.

Rust Test Suite

An important header used in the Rust tests is //@ edition , which is by default set to 2015 and it sets the Rust edition the test should be compiled with [20]. It ensures compatibility across di erent Rust editions simultaneously, allowing tests to address all editions e ectively and it simpli es testing by accommodating various Rust language versions within a single framework.

Solid Sands SuperTest

SuperTest includes a Core test suite designed to cover various versions of the C and C++ standards. This suite includes standards such as:

- ISO/IEC 9899:1990 (C90)
- ISO/IEC 9899:1999 (C99)

Figure 5.3: Ada le naming convention [48]

- ISO/IEC 9899:2011 (C11)
- ISO/IEC 9899:2018 (C18)
- ISO/IEC TR 18037:2008 (Embedded C extension)
- ISO/IEC 14882:2003 (C++03)
- ISO/IEC 14882:2011 (C++11)
- ISO/IEC 14882:2014 (C++14)
- ISO/IEC 14882:2017 (C++17)
- ISO/IEC 14882:2020 (C++20)

This ensures that the SuperTest suite is comprehensive across di erent standards, enabling thorough testing and validation of compilers against these standards. Additionally, it includes test suites created for individual versions. To prevent redundancy, SuperTest includes tests only for the new features introduced in each version. SuperTest covers multiple versions simultaneously, ensuring both backward and forward compatibility.

Ada Conformity Assessment Test Suite

The Ada Conformity Assessment Test Suite (ACATS) is available in versions for each major version of Ada:

- ACATS 4.1 for Ada 2012
- ACATS 3.1 for Ada 2005
- ACATS 2.6 for Ada 95
- ACATS 1.11 for Ada 83

There is not a test suite that covers multiple versions simultaneously, which could be considered a defect of ACATS. The reason for this is that it complicates the process of verifying compliance across di erent versions of Ada.

Synthesizing the information gained from these test suites, we make the following conclusion:

Finding 8: The compiler validation test suite should be able to accommodate multiple versions of the language. The core functionality of the language should correspond to tests that are common for every version, but there should also be tests only for speci c versions of the language. Backward compatibility should be prioritized, so that the test suite does not fail for di erent versions.

5.2.5 Testing methodology

We investigated the testing methodologies being used from the di erent test suites.

Rust Test Suite

The tests that are developed for the Rust compiler are required to be added with every new feature that is introduced. These tests are not requirements-based, as there is no language speci cation guiding their creation. Instead, they are black-box tests, focusing on functionality rather than the implementation. Plus, all the tests currently pass and should pass, so that a pull request is merged. This shows that the Rust team uses test-focused development and has managed to have a robust compiler.

Solid Sands SuperTest

The testing methodology used in SuperTest is requirements-based testing. This means that Solid Sands rst wrote test requirements on the C and C++ standards and then created a test suite that is guided by these requirements. Of course, this is black-box testing, since the Solid Sands testers do not need any knowledge on the implementation source code, they solely use the standards.

Ada Conformity Assessment Test Suite

The ACATS was created following the language speci cation, with the objective for Ada compilers to get quali ed. Therefore, the testing methodology used was requirements-based testing and black-box testing.

Synthesizing the information gained from these test suites, we make the following conclusion:

Finding 9: While white-box testing helps ensure compiler reliability and correctness, it is implementation-dependent, which means that such a test suite cannot be reused to validate other compilers, in contrast to black-box testing. Also, it does not help ensure that a compiler has the expected behavior as de ned in a language speci cation. A black-box, requirements-based test suite ensures certainty and reliability, since it is designed around prede ned requirements and is also compiler-independent. Thus, we consider it more suitable for compiler quali cation.

Chapter 6

Quali cation process application

To apply all the best practices identi ed in the previous sections, we developed our own language spec-i cation for selected features of Rust, along with corresponding tests for the compiler. This approach allowed us to identify several challenges in the quali cation process, as well as potential additions to Ferrocene, the Rust Reference, and the Rust project. The speci cation and the test suite were implemented in parallel, since they are co-dependent.

6.1 Language Speci cation

The language speci cation[1] we created was inspired by the Rust Reference and the Ferrocene Speci-cation. Initially, however, we approached the task as if the Ferrocene Speci cation did not exist. This was done to ensure an unbiased outcome. Our focus is on a non-standardized language that, while lacking a formal speci cation, has both documentation and implementation. This is a crucial aspect of our approach. The situation would have been vastly di erent if there were no documentation or no compiler to test the language against. In this section, we present the reasoning process behind creating this speci cation, including the considerations and decisions we made. We detail how we organized the speci cation, the sections it comprises, and the scope we chose. Additionally, we discuss what we included, the additions we propose compared to the Ferrocene Speci cation and the Rust Reference, and the challenges we encountered during this process. The speci cation was written in Markdown[2] for simplicity, but it can be improved by using a di erent format, such as one that supports automatic section numbering or automatic linking of sections.

6.1.1 Scope

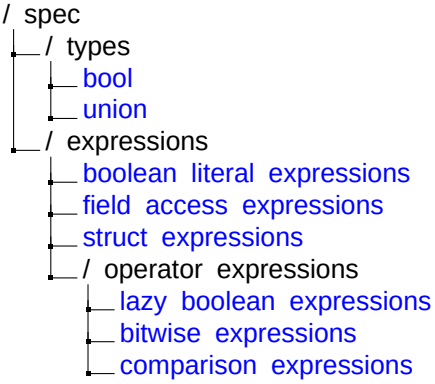
First of all, our research did not require an in-depth exploration of Rust or the creation of a complete language speci cation. While it would be ideal to comprehensively showcase potential challenges in the process and suggest ways of addressing them, the limited timeline of this project made that infeasible. Therefore, we focused on simple features of the language, such as the Bool Type and Union Type, as well as some simple operations related to them, such as Boolean Literal Expression, Bitwise Expression, Lazy Boolean Expression, Comparison Expression, Field Access Expression and Struct Expression.

There are several reasons why the speci c features were chosen. First of all, both the Bool Type and Union Type can introduce unde ned behavior, which requires careful consideration in their implementation. Furthermore, Union Type was not initially present in Rust, which makes it interesting to investigate how to accommodate di erent Rust versions in the speci cation. Another reason we chose these features is because we identi ed minor syntax and semantics gaps in the Union Type of the Ferrocene Speci cation and the Rust Reference, where we proposed improvements. This, also, motivated us to dig deeper into these features, to think of ways that these small issues could have been avoided. As non-experts in Rust, focusing on widely known features helps us contribute e ectively towards a quali cation process for all non-standardized languages. It also enabled us to use the C and C++ standards for comparison. Specifying a complicated Rust feature would not add value to our research. Finally, organizing the specifications into clear sections was challenging. Choosing these straightforward examples helped us de ne

[1] language speci cation
[2] Markdown

the scope and ensure thorough testing.

The features can be visualized more clearly with the following tree:



6.1.2 Structure

The structure used for the specification of a feature is organized as follows:

- Description : Provides informative details essential for understanding the specific feature. It does not require testing.
- Syntax : Written in an E-BNF variant to precisely define the grammar of the feature.
- Legality Rules : The name is inspired by the Ada Reference and the Ferrocene Specification. Legilty rules clearly define the rules necessary for a Rust program to compile successfully.
- Runtime Semantics : Also inspired by the Ada Reference and the Ferrocene Specification, clearly explains the behavior of the program during execution.
- Undefined Behavior : Addresses cases crucial to avoiding erroneous conclusions during testing. This should never be included in the tests.
- Examples : Includes illustrative examples to demonstrate the defined rules. These would ideally be tested with a CI pipeline, something that the Rust team already does.
- References : Provides citations to specifications of closely related features. There are also links to these sections.

This structured approach ensures clarity, minimizes confusion, and specifies clear testing requirements. In this way, we satisfy our conclusions in Finding 1 and Finding 2.

Numbering and Unique Identification

Clauses and sections in the specification are numbered for easier navigation and maintainability. Also, each testable clause is assigned a unique id, which is annotated as a comment in the tests corresponding to this specification. This unique id system ensures traceability and helps identify which clauses have been tested and which have not. The numbering of the clauses and sections alone is not sufficient for this purpose because the numbers may change when adding, reordering, or removing clauses. Consequently, maintaining unique, unchangeable ids for each clause is essential to avoid inconsistency between the specification and the corresponding tests. In this way, we satisfy our conclusions in Finding 4. This can be seen in Figure 6.1.

Versions - Editions

Since Rust is backward compatible, we did not have to take any further measures to support many versions in the language specification. For example, unions did not exist before Rust version 1.19.0[3] However, unions were added in such a way to the language, that it did not cause any incompatibilities with previous Rust versions. That is because 'union' is a reserved word only in the context of a union[4]. If unions were not backward compatible, then they would have been added in a Rust Edition. In that case, in the specification, we would have to specify that this feature is only valid in this Rust Edition. In this way, we satisfy our conclusions in Finding 3.

[3] Rust version 1.19.0
[4] union backward compatibility
[5] Union Type Spec

Figure 6.1: Union Type Speci cation [5]

Figure 6.2: Field Access Expression Specification [6]

Figure 6.3: Runtime Semantics for Lazy boolean expressions [7]

6.1.3 Reasoning Process

In this section, we describe the reasoning process in choosing the clauses of the speci cation. At rst, we did not consult the Ferrocene speci cation to stay unbiased and objective and create a speci cation from scratch. First, we thought of some considerations that guided the scope and the content of the speci cation. We used trusted sources, such as the Rust Reference and the C and C++ standards and tested edge cases or checked if C/C++ rules apply also for Rust. Then, investigating the tests we wrote and the already existing ones, we found some more elements that should be added in the speci cation. Finally, a retrospective review of the Ferrocene Speci cation allowed us to identify potential gaps in our approach. The initial considerations we thought of are grouped in two categories, those for types and those for expressions.

Types

When considering types we started by de ning their purpose and explaining their syntax. We explored the potential values they could hold and the allowed types for their elds. We examined the operations typically associated with each type and how the types are depicted in memory. Also, we explored their runtime behavior and established clear rules for their valid and invalid use, identifying any potential unde ned behavior. We tried to think of edge cases, and researched for changes in the types across di erent versions or editions. Finally, we created examples to illustrate these concepts, and we added relevant references with links to other sections. A speci cation on Union Type can be seen in Figure 6.1.

Expressions

In the case of expressions, our approach was similar with that for types. We rst de ned their intended purpose and explained their syntax and the various types of expressions they include, such as logical AND and OR. Then, we explored the symbols used in these expressions, and their runtime behavior, such as how operations within expressions are evaluated. We established clear rules for the valid and invalid use of the expressions and we determined the type and value of the expressions and its operands. We identi ed any potential unde ned behavior, examined edge cases and researched for changes in the expressions across di erent versions or editions. Finally, we created examples to illustrate these concepts, and we added relevant references with links to other sections. Detailed tables were also added, such as those demonstrating results for logical operations like logical AND. A speci cation on eld access expressions be seen in Figure 6.2.

Out of scope

For the speci cations at hand, certain topics were intentionally excluded from consideration to maintain clarity. Regarding the Bool Type speci cation, discussions on casting, while expressions, if expressions, bitwise expressions, comparison expressions, lazy boolean expressions, and boolean literal expressions

[6] Field Access Expression Spec
[7] Lazy Boolean Expression Spec

were deemed out of scope. Similarly, for the Union Type specification, topics such as pattern matching, field access expressions (read and write), struct expressions for initialization, generics, type representations, recursive types, implementations, and unsafety were intentionally excluded. In Bitwise Expressions, Comparison Expressions, and Lazy Boolean Expressions, the specification of Bool Type was not provided, but there was a reference to it. Similarly, in Field Access Expressions and Struct Expressions, the specification of Union Type was not provided, but there was a reference to it.

6.1.4 Contribution

We made six merge requests to the Rust Reference and to the Ferrocene Specification, to propose fixes and improvements.

Feature	GitHub contribution
Union Spec	Syntax Fix, Legality Rule Addition
Field Access Expressions Spec	Leaglity Rule Addition, Runtime Semantics Addition (not merged)
Struct Expressions Spec	Legality Rule Fix
While Loop Expressions	Syntax Fix

In the Union specification, we identified inconsistencies between the Rust Reference and Ferrocene syntax regarding unions with empty field lists. Initially, based on a test [8], we concluded that unions could not be empty. However, upon discovering the use of macros in Rust, as shown by testing (Listing 6.1), we found out that unions can indeed be empty and we realized that it is an edge case that needs to be clarified. Consequently, we introduced a legality rule in the Ferrocene Specification and updated the Rust Reference to accommodate empty unions. This process highlighted issues in the Rust Reference's completeness and accuracy, which indicates the need for a formal specification.

For the Field Access Expression specification, a test we wrote (Listing 6.3) demonstrated that field selectors differ across different data structures, such as tuples, unions and structs, a detail crucial for accurate semantics. Our pull request was not merged because it was suggested that this rule was implied in the Field Resolution section. However, we did not find such a rule and we strongly believe that because it is a fundamental rule that explains the syntax, it would be beneficial to specify it explicitly, and in the section, where the corresponding syntax is presented. Another approach would be to link the two specifications with a reference, to ensure that it can be retrieved easily. Additionally, we explained the behavior of unions, where writing to one field overwrites others due to shared memory into the Runtime Semantics section, since it is an important characteristic of unions. Our pull request was not accepted as well, because it was suggested that this rule is written in the Type Layout section, with the clause "For a union type, the memory layout is undefined, unless the union type is subject to attribute repr. All union fields share a common storage." [28]. However, this clause works as a Legality Rule, explaining the memory layout of unions, and does not explicitly explain this particular runtime behavior of unions. Also, in the union specification in the C++ standard, this behavior of unions is the first rule that is explained, since it is the essence of the union functionality. We do not intend to be absolute about our approach, as there are many valid perspectives. However, we aim to clarify our approach, which emphasizes that critical details are explicitly stated in the appropriate sections.

In the context of Struct Expression specification, we identified a small inaccuracy regarding base initializers being used with unions. This was proven by a test we created (Listing 6.5), where we tried to use a union value as base initializer and failed. This is reasonable, since unions can only be initialized with one field, making such statements pointless.

Regarding the While Loop Expression specification, although we did not extensively explore it, we were interested in a quick investigation because of its interaction with boolean expressions. We specified further the syntax, so that it becomes more clear and strict.

6.1.5 Further Suggestions

Regarding the Bool Type specification, we made an addition, compared to Ferrocene by specifying the size and alignment of bool, since we considered it a useful information to have in the Bool specification. Also, while Ferrocene has tables showing the results of boolean expressions, we chose to keep them separately in different specifications, for the Bitwise, Lazy Boolean and Comparison Expressions, for more clarity and detailed coverage.

[8] union-empty.rs

Concerning the Union Type specification, we clarified that unions lack a specified memory layout unless type representations are utilized. we considered this information useful to understand unions. Furthermore, we provided explanations on the use of generic parameters and where clauses in unions. We believe that the syntax should be further explained, so that it can be converted into testable requirements. The Union Type specification can be seen in Figure 6.1.

Concerning the Comparison Expressions specification, we added tables illustrating the operation of comparison expressions with boolean operands and grouped some clauses for improved readability. We explained the usage of comparison expressions with different primitive types, such as characters, strings, numbers, and booleans. Additionally, we explained the necessity of parentheses when chaining comparison operators, an important clarification to avoid confusion and errors (Listing 6.4).

Concerning the Bitwise Expressions, we introduced tables demonstrating the operation of bitwise expressions with boolean operands and included also the negation bitwise expression. We explained the syntax, specifying that bitwise AND, OR, NOT and XOR expressions involve integer and boolean operands, while bitwise shift left and shift right operations involve integer operands. Also, we clarified that shift left and right operations can work between integers of different type, but the rest operations cannot. We deemed these clarifications important to avoid confusion and explain the syntax, which we cannot prevent from being generic.

Lastly, in the Lazy Boolean Expressions specification, we clarified the distinctions from bitwise expressions and included tables illustrating their operation on boolean values.

We hope that these suggestions improve the clarity, readability, accuracy and completeness of the specification.

6.2 Compiler Test Suite

For the compiler validation test suite, we leveraged the existing Rust test suite, since it is a mature project with many useful features. We wrote more than 80 tests, some of them are similar to existing ones, while others are not. But all of the tests were written as if there was no tests already there, to have an unbiased outcome. The tests we wrote can be found on GitHub[9], in directory tests/ui and can be tested with ./x test tests/ui/ --bless . In this section we present the reasoning process behind creating this test suite and the scope we chose.

6.2.1 Scope

We tried to cover all the clauses of the specification and have at least the basic testing for them, with at least one test for each clause. But our main focus was bools and unions. So, even though a specification might have been inclusive also for structs and enums, the tests might not include them, for simplicity reasons. We performed black-box requirements-based testing and in this way, we satisfy our conclusions in Finding 9. This can be seen, for example in Listing 6.2.

6.2.2 Structure

The tests were organized in directories, in the same way the features are in the specification, for easy navigation. The files have names that explain their functionality, as is already done in the Rust test suite. In this way, it is more user-friendly and can be maintained with more ease.

Test File Naming and Comments

Test files are named in a way that clearly indicates their purpose, which is useful for maintainability and development of the test suite. This naming convention reduces the need for additional comments in the tests, contributing to cleaner and more maintainable code. Comments are included only in cases of complicated code.

Traceability

As explained in Section 6.1.2, each testable clause is assigned a unique id in the language specification, which is annotated as a comment in the tests corresponding to this specification. This unique id system ensures traceability and helps identify which clauses have been tested and which have not. This can be

[9] rust test suite

seen, for example in Listing 6.2. In this way, we satisfy our conclusions in Finding 7 and we can create a traceability matrix visualizing all the tests and all the clauses they correspond to (Table 6.1).

Types of Tests

Furthermore, we leveraged the test headers of the Rust test suite, to showcase when a test should fail to compile or should compile and run successfully. In this way, we wrote both positive and negative tests, and tested the syntax, the compilation and the execution of the constructs and so, we satisfy our conclusions in Finding 5. This can be seen in Listings 6.2 and 6.3, which are positive and negative tests, that correspond to the Legality Rule 2.2.3., shown in Figure 6.2. Also, Listing 6.1 shows a syntax test and Listing 6.6 shows a test on the Runtime Semantics rules, which can be seen in Figure 6.3. From the last test, we can see how Lazy Boolean Expressions behave during runtime.

Output Comparison

Moreover, the output comparison method from the Rust test suite was used to ensure that the output taken from a test is the expected one. For that reason .stderr files were created containing the compilation errors produced. We used substrings of these errors in annotated comments in the test files to show in which lines the errors were. In this way, we satisfy our conclusions in Finding 6.

Versions - Editions

None of the features we tested is different across Rust Editions. However, for example, 'unions' were introduced in Rust version 1.19.0, as explained in 6.1.2. This means that if the test suite ran with a compiler version older than this, the tests on unions would not compile, because they would not be recognized. This is hard to handle for Rust, because it has a new release every six weeks, but a way would be to have version-specific tests, and with an annotation show that specific tests should be run only with a version and after. In this way, we satisfy our conclusions in Finding 8.

6.2.3 Reasoning Process

As explained above, we tried to see every clause written in the specification, as a test case and to create at least one test for it. Also, we tried to split the tests as much as possible for maintainability reasons and so that more and more tests can be created for each clause. For example, in the Listing 6.2, we can see a positive test on Field Access of tuples, which corresponds to the 2.2.3. clause of the Field Access Expression Legality Rules (Figure 6.2) and runs successfully. It is a simple test that checks that all fields in tuples are accessed using the index. In the Listing 6.3, we can see a negative test on Field Access of tuples, which corresponds to the same clause of the specification and is a test that should fail to compile and give us errors. We made an effort to include edge cases. In this test, we tried to access a field using a name instead of an index, using an out-of-bounds index and using an empty tuple. These tests were created both to confirm the specification, but also defined the specification. In this way, we proposed some additions to the Ferrocene Specification, such as the clause 2.2.3., that we added[10].

```
//@ check-pass
// Union Types: 7f345296-9cec-4bb7-a0f1-7abf0b45bd3c
#[cfg(any())]
union U {}

fn main() {}
```

Listing 6.1: Empty union with macros

```
//@ run-pass
// Field Access Expressions: 7fc7d0a9-9066-4c99-81a5-37bd9ca6b223

fn main() {
    let my_tuple = (3, "hi", 1.2);

    let first = my_tuple.0;
    let second = my_tuple.1;
    let third = my_tuple.2;
```

[10] field-access-expressions.md

```
    assert_eq!(first, 3);
    assert_eq!(second, "hi");
    assert_eq!(third, 1.2);
}
```

Listing 6.2: Field selector for tuples

```
// Field Access Expressions: 7fc7d0a9-9066-4c99-81a5-37bd9ca6b223

fn main() {
    let my_tuple = (3, "hi", 1.2);
    let _ = my_tuple.first;
    //~^ ERROR no field `first` on type `({integer}, &str, {float})`
    let _ = my_tuple.3;
    //~^ ERROR no field `3` on type `({integer}, &str, {float})`
    let empty_tuple = ();
    let _ = empty_tuple.0;
    //~^ ERROR no field `0` on type `()`
}
```

Listing 6.3: Invalid field selector for tuples

```
// Comparison expressions: 4eff8002-c02a-43d4-8773-ac28a98e0218

pub fn main() {
    let a = true;
    let b = true;
    let c = true;

    let invalid = a == b == c;
    //~^ ERROR comparison operators cannot be chained
    let result = (a == b) == c;
}
```

Listing 6.4: Invalid comparison expression with parentheses

```
// Struct Expressions: 367c244b-0ac1-42d0-be03-3ad108ee89a3
union MyUnion {
    x: i32,
    y: f64,
}

fn main() {
    let base_named_union = MyUnion { x: 5 };
    let mut new_union = unsafe { MyUnion { ..base_named_union } };
    //~^ ERROR union expressions should have exactly one field
    //~| ERROR functional record update syntax requires a struct
}
```

Listing 6.5: Invalid use of base initializer for union

```
//@ run-pass
// Lazy Boolean Opearations: 266ee141-7c6b-4738-bc33-24a72e8e3d99
// Lazy Boolean Opearations: 0735bf32-a30d-4a64-8ca1-8ad873bc9e04

pub fn main() {
    assert!(!(false && panic!()));
    assert!(true || panic!());
}
```

Listing 6.6: Lazy Boolean Evaluation

6.2.4 Contribution

We created some additional tests that are not originally included in the Rust project. This process was the result of our efforts to rigorously question what should be included in the specification and to ensure that each clause had at least one corresponding test. We focused on designing both positive and negative tests, as well as covering edge cases. Through this process, we realized that further testing is always possible, even for seemingly simple language features. Developers, when writing tests, often operate from

a perspective of trust in their own implementation, which can lead to less detailed testing. In contrast, those qualifying a compiler must adopt an abstract and meticulous approach, examining every minor detail closely. Some examples of tests we developed that could be an addition to the Rust test suite are the following:

- union-empty-macros.rs
- bool-bit-pattern.rs, bool-size-alignment.rs
- lazy-boolean-evaluation.rs, lazy-boolean-operations.rs, lazy-boolean-type.rs, lazy-boolean-types-2.rs, eld-selector-tuples-2.rs
- bitwise-bool-and.rs, bitwise-bool-not.rs, bitwise-bool-or.rs, bitwise-bool-xor.rs, bitwise-expressions-types-1.rs, bitwise-expressions-types-2.rs, bitwise-expressions-types-3.rs, bitwise-expressions-types-4.rs
- comparison-bool-operations-2.rs, comparison-primitive-char.rs, comparison-primitive-string.rs, comparison-types-1.rs, comparison-types-2.rs, equality.rs, greater-than-or-equals.rs, greater-than.rs, inequality.rs, less-than-or-equals.rs, less-than.rs

6.3 Challenges

Creating these speci cations and tests presented several challenges. One of them was organizing content into appropriate sections. Ensuring completeness and thorough testing necessitates careful consideration of where each part should be placed. Abstract thinking is crucial to avoid having overlapping sections. Additionally, structuring the speci cation logically, from the perspective of a developer learning the language, enhances clarity and usability. For instance, separating union initialization and read-write operations from the union type itself helps maintain organization and provides easy navigation through references. An argument for this is also that this is how it is separated in the standards I examined, for C and C++.

In some cases, such as comparison expressions, it is challenging to specify syntax strictly due to the variability of operands. Clearly explaining the syntax is essential. For example, stating that in a shift-left expression, the involved values cannot be of bool type. This clarity simpli es the creation of corresponding tests. Writing syntax without an existing documentation is particularly di cult, but once established, it allows for testing edge cases and thorough examination.

Unde ned behavior poses another signi cant challenge in the absence of initial language documentation. By de nition, unde ned behavior cannot be tested through the implementation, so it should be de ned from the language. It is crucial to distinguish between unspeci ed behavior, where some speci cation freedom exists, and unde ned behavior, which can lead to unpredictability and potential issues like memory corruption. Understanding runtime semantics, which explain the execution's background processes, is also di cult without insight into the language's development.

Ensuring completeness of both the speci cation and the test suite is a very big task. Languages are complex constructs, and covering all aspects requires extensive e ort. For instance, Solid Sands SuperTest, is being actively developed for more than 40 years. Even in our project, we wrote over 80 tests for simple features, uncovering issues and potential improvements in both the Rust Reference and the Ferrocene Speci cation.

In conclusion, creating a robust speci cation demands a stable, well-documented language, even if the documentation is informal, as well as a reliable implementation. Some degree of trust in the documentation should be there, but rigorous testing against the implementation is equally important to ensure the validity of the speci cation. Then, to determine whether the language or the implementation is correct, knowledge of common practices in other languages, and guidance from the language development team can help. All in all, utilizing the traceability mechanism we created, by trying to test every single clause from the speci cation, we managed to create many tests, that could also be an addition to the Rust test suite. Additionally, detailed explanations and clari cations of syntax were also helpful for the same purpose. Finally, putting e ort in thinking of edge cases and analysing every subpart of a construct, helps identify even deeper the language behavior and thus generate even more compiler tests.

Section	Clause ID	Number of Tests	Tests
Union Type		17	
	7f345296-9cec-4bb7-a0f1-7abf0b45bd3c	2	union-empty-macros.rs, union-empty.rs
	e1c2f91e-7a68-48bd-b103-66749e82703c	1	union-generic-params.rs
	93059842-a3be-4dd1-92c7-1b79f40e252f	1	union-generic-params-where-clause.rs
	3e3ac7d8-8bac-427d-8898-6ae9fc6277d4	1	union-memory-layout.rs
	26ad2e4a- 73-4eb4-b16f-d33a6e5d7e7f	2	union-types-copy.rs, union-types-copy-2.rs
	847acf71-84b6-4ace-92d8-9e127ba0911e	2	union-types-drop.rs, union-types-drop-2.rs
	11a3041f-f307-4 4-acf3-fb256baf9f49	1	union-types-mut-ref.rs
	d1b5850a-f09d-4785-9d56-6ec53d7cfccf	1	union-types-array.rs, union-types-array-2.rs
	218f449a-7973-4157-8a92-87645b9ceedc	1	union-types-tuple.rs, union-types-tuple-2.rs
	6356434c-6cdb-47cd-8e99-c 31c5bef14	1	union-unique- eld-names.rs
...			
Total		85	

Table 6.1: Traceability Matrix

Chapter 7

Threats to validity

Our proof-of-concept focused on a small fragment of Rust due to our time constraints and specific objectives, which might have limited the width of our experience. Also, our findings were applied only on Rust, which is a language with some specific characteristics. It is a language that has only one usable implementation, it is documented, it is thoroughly tested and it is backward compatible. For example, for a language that is backward incompatible, our focus would have shifted more towards handling multiple versions in the specification and test suite, providing us with further significant insights. Our research is biased from that perspective. Consequently, our research may be biased in this regard. Moreover, although we tried to base conclusions on logical arguments, the process includes subjective elements and personal opinions.

Chapter 8

Related work

The Ferrocene Rust toolchain [12], developed by the German company Ferrous Systems includes the quali cation of Rust. In this thesis, we focused a lot on Ferrocene's approach, analyzing the followed process and trying to nd areas for improvement. From our research on language speci cation and compiler validation, we propose a framework for compiler quali cation of non-standardized languages, suggesting ways to achieve completeness and accuracy in this challenging process. For instance, we propose an emphasis on traceability and including Rust editions into the language speci cation, suggestions that could also contribute to Ferrocene.

The "RustHornBelt: A Semantic Foundation for Functional Veri cation of Rust Programs with Unsafe Code" paper [51] presents a framework that o ers veri cation of Rust programs, including those using unsafe code. It combines RustHorn [52], which uses Rust's type system and rst-order logic to verify that Rust programs behave as expected and RustBelt [53], which provides a semantic model that proves that the methods used in RustHorn are sound, even for complex programs and unsafe code. RustHornBelt introduces also a way to better handle mutable borrows in the veri cation process. While these projects contribute signi cantly to validating Rust and propose formal speci cations, these methods are not yet widely used in industry for compiler quali cation. In our project, we focused on a more conventional approach by creating a language speci cation in natural language rather than formal mathematical forms. This decision was because of time constraints and our lack of expertise in formal methods. However, integrating these methodologies with our approach could o er improved compiler validation.

The paper "RustSmith: Random Di erential Compiler Testing for Rust" [54] introduces a tool focused on random di erential testing of Rust compilers to detect and address compiler bugs through output comparison of randomly generated programs. It generates correct Rust programs, which are used to stress-test the o cial Rust compiler, rustc, against other Rust compilers or at di erent optimizations. It is inspired by CSmith [55] that has similar functionality for C compilers. RustSmith improves the reliability of Rust compilers by exposing potential issues and edge cases. This work is signi cant for compiler validation. However, this method uses a di erent perspective, since it does not check if the compilers adhere to a speci cation and that they have the expected behavior. Still, they provide a very useful way to nd compiler bugs and vulnerabilities. Using such tools to further test a compiler, while qualifying it, would surely increase its reliability.

The CompCert [56] compiler is a formally veri ed compiler for a big subset of C99. It uses machine-veri ed mathematical proofs to generate executable code that is proven to behave according to the speci cation of the semantics of the source C program. It o ers compiler reliability and correctness and so, it aims to be used for safety-critical applications. Nonetheless, this e ort required building a new compiler from scratch, which is di erent from our case, where we want to qualify an existing compiler, like rustc.

Chapter 9

Conclusion

In this thesis, we conducted research on compiler quali cation, analyzing relevant ISO documents and existing e orts related to Rust and other languages. Our investigation included various language speci - cations and standards, including those of C, C++, Ada, the Ferrocene Speci cation, as well as the Rust Reference. We compared these speci cations based on their scope, structure, handling of traceability, and management of multiple language versions, deriving conclusions on best practices. Similarly, we evaluated compiler validation test suites such as SuperTest, the Ada Conformity Test Suite, and the Rust test suite. Our comparative analysis focused on the types of tests included, methods for comparing output to expected results, traceability mechanisms, support for multiple language versions, and testing methodology, leading to the establishment of best practices. Following this, we applied this knowledge to develop a Rust language speci cation and a compiler validation test suite focusing on some basic fea- tures such as Bool Type, Union Type, Field Access Expression, Struct Expression, Bitwise Expression, Comparison Expression, and Lazy Boolean Expression. Leveraging the Rust test suite, we developed more than 80 compiler tests. Our contributions extended also to the Ferrocene Speci cation and the Rust Reference, where we proposed improvements and xes, by opening four merge requests. Finally, we address our research questions:

RQ1: What systematic process can be established to validate the compiler of a non-standardized programming language, ensuring compliance with functional safety stan- dards? To answer this question, we had to analyze the ISO 26262 to understand how the compiler quali cation process is for standardized languages. Then, we had to generalize it for non-standardized languages, which have a lack of a stable speci cation. We analyzed the main compiler quali cation prin- ciples, such as a stable and accurate language speci cation, a compiler validation test suite that adheres to the speci cation, a traceability mechanism to connect these two, thorough documentation, reporting compiler issues, tests reports and a traceability matrix, as well as, a mechanism to accommodate multi- ple versions and ensure backward compatibility. As far as Rust is concerned, we investigated the Rust development process and the quali cation method of Ferrocene. We also examined SuperTest, the test validation suite for C and C++ compilers, developed by Solid Sands, in order to see the clear di erence between these approaches.

RQ2: How can a robust programming language speci cation be developed for a docu-mented and implemented language that lacks a formal speci cation, ensuring compliance with industry standards? To answer this question, we organized ndings from ISO documents and research, we compared di erent language speci cations and standards and we provided both guidelines, as well as a proof-of-concept to illustrate these guidelines. The language speci cation should include the syntax and semantics of the language, potential constraints, accepted input and output, behavior during compilation and execution, and the violations that a compliant compiler is required to detect, excluding details of intermediate data processing stages. It should present syntax in a metalanguage form (with explanations), compile-time and runtime rules, and specify unde ned behavior, all with il- lustrative examples. Implementation-speci c requirements and feature descriptions should be stated, and recommended practices to avoid issues like unde ned behaviors can be included. Also, references to relevant sections can be convenient for navigation. The speci cation should explicitly detail di erences between language versions or future plans to deprecate features, prioritizing backward compatibility to ensure existing code functions across di erent versions. Assigning a unique id to each clause is essential

for traceability and ensuring test completeness.

RQ3: How can a compiler validation test suite be developed for a non-standardized programming language to ensure compiler reliability in functional safety? To answer this question, we organized ndings from research, we compared compiler validation test suites and provided both guidelines, as well as a proof-of-concept to illustrate these guidelines. A robust validation process should include diverse test types, categorized according to the aspects of language behavior they evaluate, including both positive and negative tests to expose compiler issues. Additionally, testing should cover the syntax and the compilation and execution phases of a construct to ensure the correctness and reliability of the language implementation. Unde ned behavior should not be included in the test suite, since it cannot o er any conclusion about the implementation. To enhance the compiler validation test suite, a mechanism should be implemented to compare program output to the expected output, with one reliable approach being to identify the line where an error occurred and verify if it was the expected one, ensuring reliable error detection and reporting without explicitly validating error message content. A traceability mechanism should connect tests to the speci cation by associating test le names with corresponding speci cation sections or embedding unique identi ers within test le comments, allowing exibility and user-friendly test le names to streamline validation suite development. The test suite should accommodate multiple language versions, including common tests for all versions and speci c tests for individual versions, with a priority on backward compatibility to avoid failures across di erent versions. The development of a requirements-based, black-box test suite ensures that a compiler satises the requirements described in a language speci cation. This approach is also compiler-independent, allowing the same test suite to be used to validate di erent compilers.

This thesis makes several signi cant contributions to the eld of compiler quali cation for safety-critical applications, by providing a framework for the adoption of non-standardized languages in safety-critical domains, o ering practical solutions for language speci cation and compiler validation.

9.1 Future work

This research is hopefully just the beginning of e orts to ensure the safe use of non-standardized programming languages in functional safety. Our proof-of-concept focused on a small fragment of Rust due to our time constraints and speci c goals. However, developing a robust language speci cation and validation test suite for the entire Rust language could provide signi cant experience regarding the quali cation process and its challenges, which is greatly required by both the Rust community and the safety-critical domain. Additionally, applying our ndings to other non-standardized programming languages, like Zig, which may have been developed in a di erent way, and might have multiple implementations, potential backward compatibility issues, and less thorough testing, could validate our results and lead to further realizations. Furthermore, integrating static analysis tools to complement the test suite and exploring the use of formal methods to validate critical aspects of the Rust language and its compiler would enhance con dence. Lastly, it is worth investigating ways to ensure test coverage of a compiler against the language speci cation and how to identify uncovered edge cases, possibly with the use of a random test generator.

Acknowledgements

I would like to express my sincere gratitude to my academic supervisor, Professor Andreas Goens, and my daily supervisors from Solid Sands, Remi van Veen and Vladislav Yaglamunov, for their support throughout this challenging process. Professor Goens guided me in maintaining the academic character and formality required for this thesis, while encouraging my critical thinking and allowing me freedom in decision-making. My daily supervisors provided me with the industry perspective and shared their professional expertise with me. The wisdom and guidance of Professor Goens and my daily supervisors, pushed me to delve deeply and critically analyze the information I gathered, which I believe is the whole essence of this thesis. I am also deeply thankful to my colleagues at Solid Sands, whose support created a motivating work environment and made each day at the office enjoyable. This project has been an enriching journey, providing me with learning experiences that have surely contributed to my growth as a software engineer.

Bibliography

[1] ISO Central Secretary, \Road vehicles | functional safety," en, International Organization for Standardization, Standard ISO 26262-1:2018, 2018. [Online]. Available: https://www.iso.org/standard/68383.html .

[2] Java 7 breaks apache projects, https://www.developerfusion.com/news/123346/java-7-breaks-apache-projects/ .

[3] J. Chen et al., \A survey of compiler testing," ACM Comput. Surv., vol. 53, no. 1, Feb. 2020, issn: 0360-0300. doi: 10.1145/3363562. [Online]. Available: https://doi.org/10.1145/3363562 .

[4] Solid Sands, https://solidsands.com/ .

[5] C Speci cation, https://www.open-std.org/jtc1/sc22/wg14/www/projects.html .

[6] C++ Speci cation , https://open-std.org/jtc1/sc22/wg21/docs/standards .

[7] Ada Speci cation, http://www.ada-auth.org/standards/22rm/html/RM-TOC.html .

[8] Rust Documentation, https://www.rust-lang.org/ .

[9] The rust language speci cation team, https://www.rust-lang.org/governance/teams/lang#team-spec.

[10] The rust speci cation, https://github.com/rust-lang/spec/tree/main .

[11] Rust foundation: Announcing the safety-critical rust consortium, https://foundation.rust-lang.org/news/announcing-the-safety-critical-rust-consortium/ .

[12] Ferrocene documentation, https://public-docs.ferrocene.dev/main/index.html .

[13] G. Karmakar, A. Wakankar, A. Kabra, and P. Pandya, Development of Safety-Critical Systems: Architecture and Software Springer Nature Switzerland, 2023, isbn: 9783031279003. [Online]. Available: https://books.google.gr/books?id=NdOnzwEACAAJ .

[14] \Functional safety of electrical/electronic/programmable electronic safety-related systems - part 1: General requirements," en, International Electrotechnical Commission, Standard IEC 61508-1:2010, 2010. [Online]. Available: https://webstore.iec.ch/publication/5515 .

[15] Inside Rust Blog: Our Vision for the Rust Speci cation, https://blog.rust-lang.org/inside-rust/2023/11/15/spec-vision.html .

[16] C. A. R. Hoare, \Notes on the standardisation of programming languages," Typed manuscript, Feb. 1975.

[17] ISO, IEC, \Guidelines for the preparation of programming language standards," en, International Organization for Standardization, International Electrotechnical Commission, Standard ISO/IEC TR 10176, 2003. [Online]. Available: https://www.iso.org/standard/37765.html .

[18] Stack over ow developer survey 2023, https://survey.stackoverflow.co/2023 .

[19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, \Safe systems programming in rust," Commun. ACM, vol. 64, no. 4, pp. 144{152, Mar. 2021, issn: 0001-0782. doi: 10.1145/3418295. [Online]. Available: https://doi.org/10.1145/3418295 .

[20] Rust Reference, https://doc.rust-lang.org/stable/reference/ .

[21] RFC: Start working on a Rust speci cation, https://rust-lang.github.io/rfcs/3355-rust-spec.html .

[22] MIT: Software Construction - Speci cations, https://ocw.mit.edu/ans7870/6/6.005/s16/classes/06-specifications/specs/ .

[23] A. Kossatchev and M. Posypkin, "Survey of compiler testing methods," *Programming and Computer Software*, vol. 31, pp. 10–19, Jan. 2005. DOI: 10.1007/s11086-005-0008-6.

[24] B. A. Wichmann, "Guidance for the use of the ada programming language in high integrity systems," *Ada Lett.*, vol. XVIII, no. 4, pp. 47–94, Jul. 1998, ISSN: 1094-3641. DOI: 10.1145/290214.290222. [Online]. Available: https://doi.org/10.1145/290214.290222.

[25] *Black-box and white-box testing with supertest*, https://solidsands.com/black-box-and-white-box-testing-with-supertest.

[26] T. BRUNNER, N. PATAKI, and Z. Porkoláb, "Backward compatibility violations and their detection in c++ legacy code using static analysis," *Acta Electrotechnica et Informatica*, vol. 16, pp. 12–19, Jun. 2016. DOI: 10.15546/aeei-2016-0009.

[27] *Ferrous systems*, https://ferrous-systems.com/.

[28] *Ferrocene Specification*, https://spec.ferrocene.dev/.

[29] *Adacore*, https://www.adacore.com/.

[30] *Ferrocene: Qualification method*, https://public-docs.ferrocene.dev/main/qualification/evaluation-report/rustc/method.html.

[31] *Traceability Matrix*, https://public-docs.ferrocene.dev/main/qualification/traceability-matrix.html.

[32] *Rust compiler development guide: Contribution procedures*, https://rustc-dev-guide.rust-lang.org/contributing.html.

[33] *Rust Compiler Development Guide: Implementing new language features*, https://rustc-dev-guide.rust-lang.org/implementing_new_features.html.

[34] *Rust request for comments process*, https://rust-lang.github.io/rfcs/.

[35] *Rust Releases*, https://github.com/rust-lang/rust/tags.

[36] *Rust Editions*, https://doc.rust-lang.org/edition-guide/editions/.

[37] *Rust release trains*, https://rust-lang.github.io/rustup/concepts/channels.html.

[38] *Gcc compiler*, https://gcc.gnu.org/.

[39] *Clang compiler*, https://clang.llvm.org/.

[40] *Rustc compiler*, https://doc.rust-lang.org/rustc/.

[41] *Mrustc compiler*, https://github.com/thepowersgang/mrustc.

[42] *Rust-gcc compiler*, https://github.com/Rust-GCC/gccrs.

[43] *Rust GitHub Repository*, https://github.com/rust-lang/rust.

[44] D. Jones, "Forms of language specification: Examples from commonly used computer languages," Sep. 2010.

[45] ISO, IEC, "Syntactic metalanguage extended bnf," en, International Organization for Standardization, International Electrotechnical Commission, Standard ISO/IEC 14977:1996, 1996. [Online]. Available: https://www.iso.org/standard/26153.html.

[46] ISO, IEC, "Principles and rules for the structure and drafting of iso and iec documents," en, International Organization for Standardization, International Electrotechnical Commission, Standard ISO/IEC 2021, 2021. [Online]. Available: https://www.iso.org/sites/directives/current/part2/index.xhtml.

[47] D. M. Berry, E. Kamsties, and M. Krieger, "From contract drafting to software specification: Linguistic sources of ambiguity," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:18255573.

[48] *ADA CONFORMITY ASSESSMENT TEST SUITE*, http://www.ada-auth.org/acats-files/4.1/docs/ACATS-UG.PDF.

[49] *Rust UI Tests*, https://rustc-dev-guide.rust-lang.org/tests/ui.html.

[50] *Ferrocene GitHub Repository*, https://github.com/ferrocene/ferrocene/tree/main.

[51] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer, "Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 841–856, ISBN: 9781450392655. DOI: 10.1145/3519939.3523704. [Online]. Available: https://doi.org/10.1145/3519939.3523704.

[52] *Rusthorn: A chc-based automated verifer for rust*, https://github.com/hopv/rust-horn.

[53] *The rustbelt project*, https://plv.mpi-sws.org/rustbelt/#project.

[54] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1483–1486, ISBN: 9798400702211. DOI: 10.1145/3597926.3604919. [Online]. Available: https://doi.org/10.1145/3597926.3604919.

[55] *Csmith, a random generator of c programs*, https://github.com/csmith-project/csmith.

[56] *The compcert c compiler*, https://compcert.org/compcert-C.html.