HTWD Hochschule für Technik und Wirtschaft Dresden
University of Applied Sciences

Master Thesis

Verification in Rust

By:	Kevin Slijepcevic
Field of study:	Applied Informatics, M. Sc.
University:	University of Applied Sciences Dresden
Matriculation number:	50962
First evaluator:	Prof. Dr.-Ing. Andreas Westfeld University of Applied Sciences Dresden
Second evaluator:	Dr. rer. nat. Tobias Denkinger secunet Security Networks AG
Due date:	18.09.2023

Abstract

The systems programming language Rust is the "most admired" programming language, according to the 2023 Stack Overflow Developer Survey. Organizations like Discord and the Linux Kernel Organization or projects like Chromium are embracing the advantages of Rust, including memory safety, and steadily integrate code written in Rust into their codebase. Memory safety is an aspect that can also be proven with formal verification. Safe Rust guarantees memory safety, therefore the language does not need formal verification in this domain. However, formal verification can be used to prove further properties, outside the memory safety domain. There are several research communities in the formal verification field for Rust, which provide a platform for showcasing and discussing different tools.

In this thesis, we present an overview of the current state of formal verification in Rust. We provide an overview for the tools Prusti, Creusot, Verus, Aeneas and MIRAI, based on eight parameters and an example factorial implementation, we propose for this overview. Furthermore, we consider Creusot and Prusti as two interesting candidates and demonstrate the practical application of formal verification on the Bubblesort sorting algorithm.

Contents

List of Abbreviations

MIR Mid-level Intermediate Representation: Represents a simplified form of Rust code [1]. It is based on the control-flow graph, produced by the compiler and used for checks like the borrow-checker [1]. Acts often as the basis for verification tools. Requires the use of the nightly version of the Rust compiler.

HIR High-level Intermediate Representation: Represents a compiler-friendly form of the abstract syntax tree after parsing [2]. Requires the use of the nightly version of the Rust compiler.

1 Introduction

The systems programming language Rust is the "most admired" programming language, according to the 2023 Stack Overflow Developer Survey [3]. Organizations like Discord [4] and the Linux Kernel Organization [5] or projects like Chromium [6] are embracing the advantages of Rust and steadily integrate code written in Rust into their codebase. The motivation behind this transition are the promises Rust pledges regarding memory safety: while using *safe* Rust, it is impossible to produce code which generates *undefined behavior* and therefore can cause memory safety problems, like using dangling pointers or use-after-free [7]. For example, around 70 percent of the vulnerabilities found in Google Chrome and at Microsoft are related to memory handling errors [8]. Safe Rust prevents users from using operations like dereferencing raw pointers or calling unsafe functions, for example functions written in the programming language C [9].

Memory safety is an aspect that can also be proven with *formal verification*. Formal verification is the use of mathematical methods, to ensure that a specification is being adhered to by an implementation [10]. That means, in terms of software engineering, that source code is checked and verified against constraints, describing its desired properties. Safe Rust guarantees memory safety, therefore the language does not need formal verification in this domain. However, formal verification can be used to prove further properties, outside the memory safety domain.

There are several research communities in the formal verification field for Rust: for example, the *Rust Verification Workshop* [11], whose third occurrence took place in April 2023, with 15 talks about formal verification in Rust. Another example is the *Rust Formal Methods Interest Group* [12], with its last meeting being held in October 2022. A third example is from Google Research, though discontinued since September 2021, which provided a collection of tools and libraries called *Rust Verification Tools (RVT)* [13]. These communities provide a platform where different formal verification tools are being showcased and discussed.

We present an overview of the current state of formal verification in Rust:

- We propose eight parameters to evaluate the tools: Usability, Technology, Feedback, Language and Annotation Support, Development, Activity, Scientific Activity and Documentation. Furthermore, we provide a brief example of the application of formal verification for each tool, verifying an implementation for calculating the factorial value of a non-negative integer.
- We evaluate the tools Prusti, Creusot, Verus, Aeneas and MIRAI on these parameters, based on scientific literature and user manuals.

- Among the previously mentioned tools, we consider Creusot and Prusti as two interesting candidates and demonstrate the practical application of formal verification on the Bubblesort sorting algorithm.

Outline. The thesis is structured as follows: The following section (Section 2) provides an introduction to the terms used in the formal verification domain. The next section (Section 3) discusses the used evaluation parameters for the overview. Subsequently, Section 4 provides the overview for formal verification tools: Section 4.1 covers the tool Prusti, Section 4.2 Creusot, Section 4.3 Verus, Section 4.4 Aeneas and Section 4.5 MIRAI. Section 4.6 provides a discussion for the tool overview. Section 5 demonstrates the practical application of formal verification for Bubblesort. Section 6 concludes the thesis and provides an outlook for further research.

2 Introduction to Formal Verification

This section provides a brief overview of formal verification and the used terms.

Formal verification is the application of mathematical techniques to prove functional correctness of a design [14]. That means, that an implementation is verified against a set of *specifications*, which describe its desired behavior. These specifications can be encoded as *annotations* in the to-be-verified source code for a formal verification tool. Additionally, these specifications can be implicit, for example the absence of integer overflows or can be provided by the type system, based on a function signature.

The annotations can be designed with different intentions, for example as pre- and post-conditions: *Preconditions* have to be fulfilled before a function can start its execution. *Postconditions* have to be fulfilled after a function has returned. According to the *Floyd-Hoare* triple $\{p\}f\{q\}$, a function f is partially correct, if for every input that satisfies the preconditions p, and, in case f terminates, the result of the function satisfies the postconditions q [15]. Another example are *invariants*, which are used to state a fixed property and can be used to act as an inductive proof for proving postconditions, for example *loop invariants*: these invariants have to be true before and after a loop iteration, but may be violated during an iteration. *Variants* represent the corresponding counterpart and are used to state a changing property of a design, for example a decreasing input parameter for a recursive function call to prove its termination. *Assumptions* are used to assume that a certain property is true. *Assertions* are used to prove additional properties.

Annotations can consist of different logical statements, for example an *implication*, which represents a logical consequence in the form of *if x, then y*.

Quantifiers can be used, for example, to apply a condition over a sequence, in the form of *forall*, where the condition must be true for every member of the sequence, and *exists*, where at least one member in the sequence must meet the condition.

Proof assistants are interactive theorem provers, which are used to define mathematical theories and properties, and to perform logical reasoning [16]. Additionally, there are also automatic theorem provers, for example in the form of *Satisfiability Modulo Theory solvers* (in short *SMT solvers*). These solvers are used to check, if a *formula* is satisfiable by adhering to certain logical theories, for example equality theory [17]. The formulas can be derived from *verification conditions*, which are generated by a formal verification tool.

Ghost code and *ghost variables* are code constructs, used for verification purposes only.

3 Parameters for the Overview of Verification Tools

In this section, we propose eight parameters for evaluating the formal verification tools. Furthermore, we discuss an example implementation for calculating the factorial value of a non-negative integer. This implementation will be verified with each tool.

Usability

Does the tool operate fully autonomous, or does it require manual steps? How to apply the tool to Rust source code?

Technology

What technology does the tool utilize for generating and validating the proofs? Can it be customized to use specific tools, for example a special SMT solver?

Feedback

How does the tool report success or errors? Can the tool generate counterexamples?

Language and Annotation Support

Does the tool support all language features of Rust? What are its limitations? What kinds of annotations does the tool support?

Development

Who has developed the tool? Is it backed by a company?

Activity

Is the tool in active development or abandoned? Is it feature complete? Is the tool used in production?

Scientific activity

Is the tool often quoted in scientific publications, workshops, or presentations?

Documentation

Is there documentation available for the tool? Does the tool provide examples of usage?

Example

The example implementation in Listing 1 calculates the factorial value of an input n without recursion. The function *factorial* returns an *Option<u64>*. The return value is *None*, if an integer overflow occurred during the multiplication operation in line 8 of

Listing 1: Rust example of a function calculating the factorial value of n.

```
fn factorial(n: u64) -> Option<u64> {
    if n == 0 { return Some(1); }
    let mut result: u64 = 1;
    let mut k: u64 = 1;

    while k < n {
        k = k + 1;
        result = match result.checked_mul(k) {
            Some(res) => res,
            None => return None,
        };
    }
    Some(result)
}
```

Listing 2: Unsupported feature error for iterators in Prusti.

```
error: [Prusti: unsupported feature] iterators are not fully supported yet
  --> src/main.rs:68:14
   |
68 |     for k in 1..=n {
   |              ^^^^^
```

Listing 1. The return value is *Some(x)*, if no overflow occurred, with x containing the factorial value of n.

Please note that we use while loops instead of for loops in this thesis. At the time of writing and to the best of the author's knowledge, Prusti does not support iterators, as can be seen in Listing 2. For loops in Rust require the use of iterators [18]. To ensure comparability, we decided to use while loops for every tool.

4 Overview of Formal Verification Tools for Rust

This section provides an overview over five tools for formal verification in Rust: Prusti in Section 4.1, Creusot in Section 4.2, Verus in Section 4.3, Aeneas in Section 4.4 and MIRAI in Section 4.5. The overview will be discussed in Section 4.6. All tools are designed for static analysis. The parameters for the evaluation have been discussed in Section 3.

4.1 Prusti

Prusti is a general-purpose verifier for Rust [19]. Its development started in December 2017, according to the tool's GitHub log [20]. It leverages Rust's type system while providing a specification language, which can be used to describe properties of a design in order to verify them [21]. This verification process, visualized in Figure 4.1, is automated by translating the Rust code including specifications into the Viper intermediate language [22], which is used for reasoning [21]. According to the authors, there were three design goals for Prusti [19]:

- provide functional correctness properties for Rust programs, without depending on Rust's safety guarantees,
- reduce the amount and complexity of annotations by leveraging Rust's design, and
- effortlessly become a part of a Rust developer workflow.

Figure 4.1: The verification process in Prusti.

4.1.1 Usability

Prusti integrates itself into the Rust compiler and relies on both the generated HIR and MIR [19]. Additionally, it relies on information generated by Rust's borrow-checker [19]. There is an extension called *Prusti Assistant* [23] for the code editor *Visual Studio Code*. It offers the ability to automatically install Prusti, and it integrates itself into the editor for inline error highlighting of messages produced by Prusti.

The user can add annotations to the source code. These annotations are processed fully automatic by Prusti. The provided information from the Rust compiler is translated [21] into the Viper intermediate language, which is a part of the Viper infrastructure [22]. The Viper infrastructure then produces the verification results and returns them to Prusti, which consequently returns them to the user [21].

According to the authors, Prusti proposes an incremental verification workflow [19]. The tool makes it easy to retrofit an existing codebase with Prusti annotations and incrementally add more specifications to acquire more guarantees [19]. Prusti can prove by default that a program will not terminate after reaching an unrecoverable state, for example a *panic* or *unreachable* macro call, without annotations in the Rust source code [19].

The Prusti syntax for designing specifications is based on the default Rust syntax and enhanced by special keywords. For example, for creating postconditions, it uses the keyword *result* to refer to the result of a given function [19]. Another keyword is *old*, which refers to the state of a variable before the function has been executed [19]. The Prusti specifications are Rust specific attributes [24], distinguished with a Prusti keyword like *requires* or *ensures*. It is possible to use pure functions in the specifications [19]. The return value of a pure function only depends on the values of its parameters. Additionally, a call to a pure function does not have any side effects, for example modifying its input parameters [25].

4.1.2 Technology

The backbone of the verification process in Prusti is the Viper infrastructure, as shown in Figure 4.1. This infrastructure is developed at the ETH Zurich [22]. The core of this infrastructure is the Viper intermediate language [22]. According to the authors, the key advantage of the Viper intermediate language is, compared to other intermediate languages like Boogie or Why, the support for permission logic, in particular *implicit dynamic frames* [19, 21, 22]. Permission logic is used to reason about data in the heap of a program [19, 21, 22].

The Viper infrastructure consists of three parts [26]:

- the Viper intermediate language, in short *Viper*,
- *Silicon*, which is used for symbolic execution, and
- *Carbon*, which generates verification conditions in the intermediate language *Boogie*.

Boogie is both an intermediate verification language and the name of a tool, which generates verification conditions for the SMT solver Z3 [27].

Both Silicon and Carbon utilize the SMT solver Z3 for their reasoning. However, Lasse F. Wolff Anthony has shown that it is feasible to swap the default SMT solver Z3 in Silicon with the SMT solver cvc5 [26]. According to the author, the motivation for choosing cvc5 was the ability to solve more SMT-LIB benchmarks than with Z3 [26].

Prusti automatically generates *core proofs* in Viper, which represent the guarantees the standard Rust compiler provides, for example for borrowing [21]. That means, that Prusti in combination with the Viper infrastructure does not rely on these guarantees but verifies independently. These core proofs provide a foundation for verifying further properties, for example the absence of overflows [21].

Prusti acts as a frontend for the Viper infrastructure. There are several other Viper infrastructure frontends for languages like Go, Python, Java, or OpenCL [22, 28]. The Viper infrastructure is also used in different research projects, for example the *VerCors* project[1] and for teaching purposes at several universities, for example the University of Minnesota [28].

4.1.3 Feedback

Due to the integration of Prusti in the Rust compiler, verification error messages are displayed like regular Rust compiler error messages. Prusti highlights the relevant specification where the error originates from. To illustrate this, see Listing 3: the postcondition requires that the function increments the value of a given variable by two, though the function body states that the function only increments by one. Prusti catches this violation of the specification and reports that the postcondition might not hold with the given function implementation. The *might* is important here as Prusti is a conservative verifier, which means that it reports an error, if it is unable to prove that a certain condition holds [29].

Prusti can create counterexamples, which can help in understanding why, for example, a certain postcondition fails [30].

4.1.4 Language and Annotation Support

Prusti proves partial correctness [31]. The user has to ensure that the Rust code terminates, otherwise the guarantees made by Prusti may not hold [31]. Gijsberts has demon-

[1] https://vercors.ewi.utwente.nl/

Listing 3: Example output of a verification error in Prusti.

```
error: [Prusti: verification error] postcondition might not hold.
 --> src/main.rs:4:11
  |
4 | #[ensures(*val == *old(val) + 2)]
  |           ^^^^^^^^^^^^^^^^^^^^
  |
note: the error originates here
 --> src/main.rs:5:1
  |
5 | / fn incr(val: &mut u32) {
6 | |     *val += 1;
7 | | }
  | |_^

Verification failed
```

strated that a non-terminating pure function can result in false positive verification results [32]. This property makes Prusti not well suited for intentionally non-terminating programs like OS kernels or servers.

Prusti currently supports safe Rust. According to the authors, support for unsafe Rust is planned [21]. At the time of writing, it is possible to encapsulate unsafe Rust code and exclude it from the verifying process, marking it as a *trusted* function [33]. A trusted function encapsulation can also be used for wrapping Rust language features, which are not supported by Prusti.

Prusti supports annotating pre- and postconditions for functions [19, 21]. It is possible to call pure Rust functions in those annotations, as well as calling predicates [34], which have been defined with the *predicate* macro [19]. Additionally, Prusti supports the usage of the quantifiers *forall* and *exists* [19].

According to the authors, an advanced Prusti feature is the support of *pledges* [21]. Pledges are used to design specifications for variables, which are being returned as a *reborrow* from a function call. A function which uses the concept of reborrowing takes mutable references and returns mutable references [19]. With the keywords *before_expiry* and *after_expiry*, it is possible to define specifications which have to be adhered, before or after a borrow ends, respectively [35].

Prusti has support for external specifications. This is necessary for function calls to external libraries such as the Rust standard library, because they do not include any annotations for Prusti. With external specifications, it is possible to retrofit those functions with pre- and postconditions and enable Prusti to reason about those [19, 36]. It is also possible to mark them as pure. All functions that are annotated with external specifications will be implicitly marked as trusted [36]. That means, that Prusti assumes that all pre- and postconditions for that function are correct.

At the time of writing, designing specifications for closures is not supported by Prusti [37]. Closures in Rust are anonymous functions, that can be saved in variables or be passed to other functions as input parameters [38]. Additionally, closures can capture variables from the scope in which they are defined [38]. Wolff et al. have shown that it is feasible to add support for closures into Prusti [39].

Additionally, Prusti has support for

- recursive function calls [19],
- assertions, refutations, and assumptions (refutations are used to prove a specification, which might hold in some but not all cases) [40],
- type and loop invariants [19],
- type-conditional spec refinements (these refinements can be used to add specifications for generic functions, which only apply to a certain data type) [41] and
- implications [42].

4.1.5 Development

Prusti is developed at the ETH Zurich and is introduced in the publication "Leveraging Rust Types for Modular Specification and Verification" [21]. The Swiss National Science Foundation, Facebook and Amazon Web Services are funding the project [43]. The current projects members from the ETH Zurich are Vytautas Astrauskas, Aurel Bílý, Jonas Fiala, Peter Müller and Federico Poli [43]. Further members are Christoph Matheja from the DTU Copenhagen and Alex Summers from the UBC in Vancouver, Canada [43].

4.1.6 Activity

Prusti is being actively developed, with over 7,000 commits in the main branch since 2017 [44]. Its latest stable release was in January 2023, the latest nightly pre-release in May 2023 [45].

Prusti does not mention any feature-completion or the lack of it in its complementary literature and GitHub project page. There are about 200 issues in GitHub currently open [46]. Many of them are proposals for new features. About 200 issues have been closed already [46].

The authors have evaluated the tool in several ways: first, they successfully tested the automated core proof generation on 11,791 functions of the top 500 Rust crates of the

year 2018 [21]. Second, they identified 519 possible functions in those crates where an overflow or division by zero could happen, showing that 467 functions have some sort of problem, with the vast majority having an overflow or division by zero problem [21]. Third, they took 16 examples for verifying panic and overflow freedom, while designing and applying specifications on 7 of them [21]. Those 16 examples had a total amount of 471 lines of code, with a total investment of 389 lines of code for Prusti annotations.

Gijsberts has demonstrated that it is feasible to verify a key-value store with Prusti [32]. Schar has used Prusti to verify a refinement between a TLA^+ model and a Rust implementation [47].

4.1.7 Scientific activity

Prusti has been presented at the first occurrence of the Rust Verification Workshop in 2021 [48]. This presentation is also featured on the official Rust YouTube channel [49]. Additionally, it has also been mentioned by the Rust Formal Methods Interest Group [50].

The first paper [21] that introduced Prusti has been cited around 150 times, according to Google Scholar, at the time of writing. Its successor [19] has been cited 17 times at the time of writing.

4.1.8 Documentation

Prusti has a dedicated user's guide to demonstrate its capabilities, and it provides a guided tour, to teach new users the usage of Prusti [51]. Prusti also provides code examples of its usage in the two complementary papers [19, 21] and the GitHub project page [44].

Prusti ships with over 300 test cases [21]. They include correct and incorrect examples [52] of Rust code and annotations.

4.1.9 Example

Prusti is able to successfully verify the factorial example. As shown in Listing 4, the implementation has been equipped with a postcondition in lines 21–26, two loop invariants in lines 34 and 35 and an assertion in line 32. Additionally, to enable Prusti to reason about the *checked_mul* function for the *u64* datatype, the function has been retrofitted with two postconditions, one for defining the result if an overflow occurs and one for

Listing 4: Rust example of a function calculating the factorial value of n with annotations for Prusti.

```
#[extern_spec]
impl<T> std::option::Option<T> {
    #[pure]
    #[ensures(result == matches!(self, None))]
    pub const fn is_none(&self) -> bool;
}

#[extern_spec]
impl u64 {
    #[ensures(self * rhs > u64::MAX ==> result.is_none())]
    #[ensures(self * rhs <= u64::MAX ==> result == Some((self * rhs)))]
    pub fn checked_mul(self, rhs: u64) -> Option<u64>;
}

predicate! {
    fn fac(n: u64) -> u64 {
        if n <= 0 { 1 } else { n * fac(n-1) }
    }
}

#[ensures(
    match result {
        Some(res) => res == fac(n),
        None => true,
    }
)]
fn factorial(n: u64) -> Option<u64> {
    if n == 0 { return Some(1); }
    let mut result: u64 = 1;
    let mut k: u64 = 1;

    prusti_assert!(fac(0) == 1);
    while k < n {
        body_invariant!(result == fac(k));
        body_invariant!(k <= n);
        k = k + 1;
        result = match result.checked_mul(k) {
            Some(res) => res,
            None => return None,
        };
    }
    Some(result)
}
```

defining the result if no overflow occurs. These postconditions are based on the Verus implementation of the *checked_mul* function [53], see also Section 4.3.9. Furthermore, the *is_none* function for the *Option<T>* data type has to be marked as pure and equipped with a postcondition, specifying its behavior. These annotations are taken from the Prusti user guide [54]. The postcondition for *factorial* ensures that if no overflow has occurred, the returned value for *n* has to be the same as defined in the predicate *fac*. This predicate represents the recursive definition of the factorial value of *n*.

4.2 Creusot

Creusot is a verification tool for Rust. Its development has started in October 2020, according to the tool's GitHub log [55]. The verification process, shown in Figure 4.2, is semi-automatic: Creusot automatically translates Rust source code, including annotations for specifications, into Why3 source code [56]. Why3 is then used to prove the annotations with one or more SMT solvers [56]. Creusot uses the concept of *prophecies*, which have been first introduced to Rust by Matsushita et al., in a project called *RustHorn* [56, 57]. Prophecies enable Creusot to reason about a variable's *final* value, for example after a mutable borrow ends [56].

Creusot uses a language called *Pearlite* for designing its specifications, which represents a pure subset of the Rust programming language extended with logical expressions [56, 58].

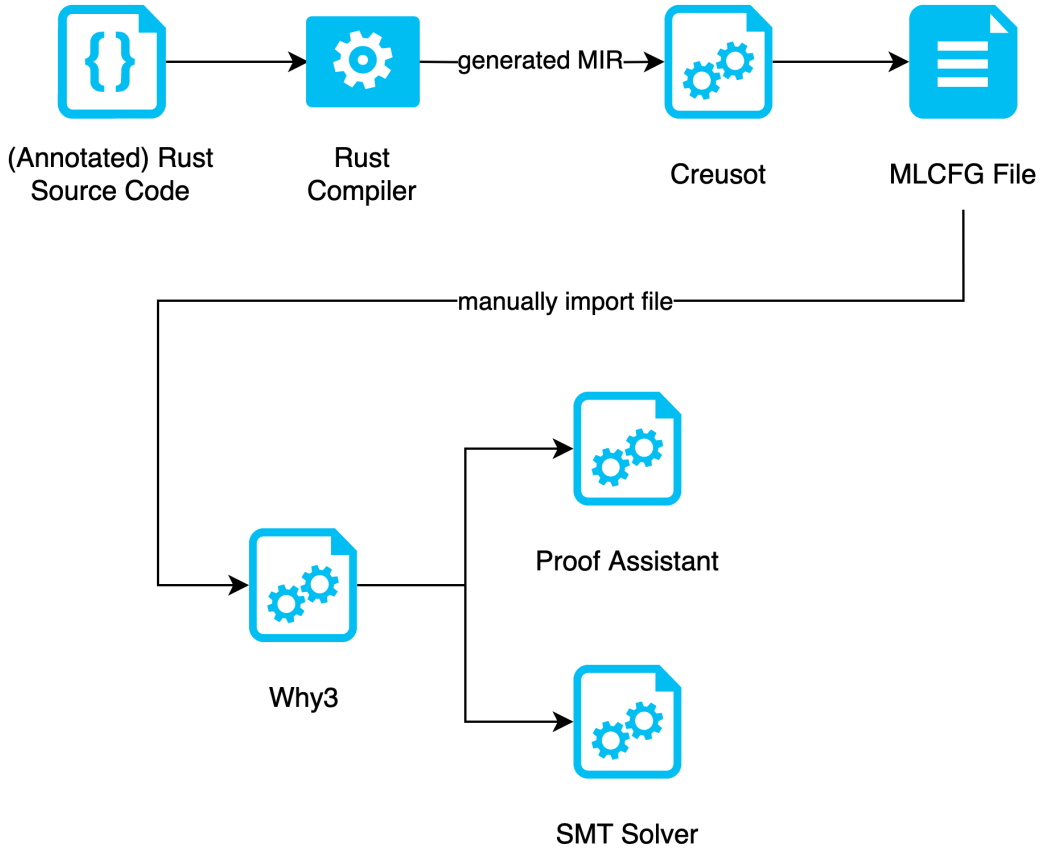

Figure 4.2: The verification process in Creusot.

4.2.1 Usability

Creusot integrates itself into the Rust compiler and uses the generated MIR for its code generation for Why3 [56]. If no annotations are present in the source code, Creusot creates code for Why3 based on implicit specifications, for example the absence of integer overflows. The user can annotate the source code with annotations, based on further specifications. The generated code has to be manually imported into Why3. Creusot provides a script for that purpose [56, 58]. According to the developers, it is planned to improve this part of the process, in the form of a Visual Studio Code plugin which acts as a frontend for Why3 [59]. The user can then prove in Why3 that the implementation satisfies the specifications, with different strategies and SMT solvers.

The annotations designed in Pearlite are similar to those in Prusti, using the same Rust-specific attribute style.

It is possible to call functions inside of specifications, marked with the *predicate*, *logic* or *ghost* annotation. Functions marked with *predicate* must return a boolean value [58]. Functions marked with *logic* can contain any logical operations such as quantifiers and can only be used during verification [58]. Furthermore, logic functions have no restriction regarding the returned data types. The *ghost* annotations marks ghost functions, which can return a *Ghost<T>* data type [58].

4.2.2 Technology

Creusot translates annotated Rust code into the *WhyML* language, which is designed to be executed by Why3 [56]. To represent the unstructured nature of the generated MIR, a WhyML language extension called *MLCFG* is used by Creusot [56, 60].

Creusot relies on Rust's borrow-checker, as it does not create core proofs like Prusti (see Section 4.1.2) [56]. According to the authors, omitting this procedure significantly reduces verification time effort, compared to Prusti [56].

With the use of Why3 for proving the annotations, the tool relies on a well-known and well-tested verification platform [56]. Why3 supports a wide variety of automatic SMT solvers, like CVC3, CVC4, Z3, Alt-Ergo and proof assistants like Coq or Isabelle [61].

4.2.3 Feedback

Due to Creusot's integration into the Rust compiler, error messages regarding the Pearlite language are presented like regular Rust errors. Because Creusot only translates the

source code and does not perform reasoning, non-provable specifications are not reported through the Rust compiler. Succeeding or failing annotations have to be observed in Why3. The graphical interface for Why3 can display and mark the original source code, which has been translated by Creusot, to highlight which generated goal proves which part of the source code.

4.2.4 Language and Annotation Support

Creusot can process safe Rust code [56]. Unsafe Rust code can be wrapped by a safe abstraction layer, which can be marked as trusted [56]. Creusot makes use of this technique by providing a subset of Rust's standard library, retrofitted with annotations [62].

Creusot supports a wide variety of the Rust syntax like traits, structs, enumeration, borrowing, loops and recursive function calls [62]. Hayeß has encountered some missing Rust language features in Creusot, for example closures [63].

Creusot supports access to the mathematical models of data structures with the @ operator [62]. For example, the *vector* data structure of the Rust standard library can be viewed as a mathematical sequence of values [62]. It is possible to design models for custom data structures [58, 62].

Creusot leverages prophecies into modelling mutable borrows. It enables the tool to access the final value of a mutable borrow. This functionality can be compared to Prusti's pledges (see Chapter 4.1.4) [56, 62]. The distinction of both techniques is based on the consideration of time: Prusti's pledges are checked at the end of the mutable borrows' lifetime, while Creusot specifies the future value of a mutable borrow at the start of its lifetime [62].

Creusot supports ghost variables, which only exist during verification [56]. They can be used to store the state of a variable before reaching a loop, for example [56].

Additionally, Creusot has support for

- pre- and postconditions [56],
- predicates, quantifiers, loop invariants and variants, implications [56],
- user specific lemmas [56] and
- assertions.

4.2.5 Development

Creusot is developed as a research software [58] and is introduced in the publication "The CREUSOT Environment for the Deductive Verification of Rust Programs" [62]. The three authors of both publications [56, 62] for Creusot, Xavier Denis, Jacques-Henri Jourdan and Claude Marché are from the Université Paris-Saclay in France. There is no mention of any funding in the complementary literature [56, 62] and GitHub project page.

4.2.6 Activity

Creusot is being actively developed, with 1,800 commits in the main branch since 2020 [58].

Creusot doesn't mention any feature-completion or the lack of it in its complementary literature and GitHub project page. There are currently about 80 issues open in GitHub, about 10 of them marked as enhancements, and about 190 issues have been closed [64].

The developers have evaluated Creusot with 16 example programs [56]. Those examples had a total amount of 336 lines of code, with a total investment of 534 lines of code for Creusot annotations [56].

Creusot has been used to verify CreuSAT, a SAT solver written in Rust [65]. Additionally, it has been used to verify parts of the Rust runtime for Lingua Franca, a polyglot coordination language [63].

4.2.7 Scientific activity

Creusot has been presented at the 2021 and 2022 occurrence of the Rust Verification Workshop [48, 66]. It has also been featured on the official Rust YouTube channel [67]. Furthermore, Creusot has been featured by the Rust Formal Methods Interest Group in 2021 [50] and a presentation has also been held on their official YouTube channel in 2022 [68].

The first paper [62] that introduced Creusot has been cited 14 times, according to Google Scholar. Its successor [56] has been cited 11 times at the time of writing.

4.2.8 Documentation

Creusot does not feature a full documentation of its capabilities or a dedicated user's guide. However, it provides a basic overview of common Creusot annotations and example Rust code with annotations on the GitHub project page [58].

Creusot ships with test code, which contains functional and non-functional code examples [69].

4.2.9 Example

Creusot in combination with Why3 and the SMT-Solver Z3 is able to successfully verify the factorial example. As seen in Listing 5, the implementation has been equipped with a postcondition in lines 9–13 and two loop invariants in lines 21 and 22. The postcondition for *factorial* ensures, that if no overflow has occurred, the returned value for *n* is the same as defined in the logic function *fac*. This logic function represents the recursive definition of the factorial value of *n*. The variant attribute in line 2 ensures, that *n* decreases with each recursive function call of *fac*, therefore proving the functions' termination. The @ operator accesses the mathematical model for a data type, as previously discussed in Section 4.2.4.

Listing 5: Rust example of a function calculating the factorial value of n with annotations for Creusot.

```
#[logic]
#[variant(n)]
fn fac(n: Int) -> Int {
    pearlite! {
        if n <= 0 { 1 } else { n * fac(n-1) }
    }
}

#[ensures(
    match result {
        Some(res) => res@ == fac(n@)
        None => true,
})]
fn factorial(n: u64) -> Option<u64> {
    if n == 0 { return Some(1); }
    let mut result: u64 = 1;
    let mut k: u64 = 1;

    proof_assert!(fac(0) == 1);

    #[invariant(result@ == fac(k@))]
    #[invariant(k <= n)]
    while k < n {
        k = k + 1;
        result = match result.checked_mul(k) {
            Some(res) => res,
            None => return None,
        };
    }
    Some(result)
}
```

4.3 Verus

Verus is a verification tool for *Rust-like* code [70, 71]. Its development started in April 2021, according to the tool's GitHub log [72]. The program and specification code has to be implemented inside the *verus* macro [73]. Verus uses an altered syntax, based on the default Rust syntax, inside that macro. The development is mainly performed at the ETH Zurich and Carnegie Mellon University, USA [70].

The tool features automatic verification and translates the program and specification code into instructions for the underlying SMT solver Z3, as shown in Figure 4.3. The tool is inspired by projects like Dafny, Boogie, Prusti (see Section 4.1), Coq and more [71]. According to the authors, there were several goals [71] for designing Verus, including:

- provide a mathematical language for designing specifications and proofs,
- provide an imperative language for designing executable code, and
- generate small and simple verification conditions for the SMT solver.

Note that the syntax for Verus-specific annotations mentioned in the complementary paper [70] is outdated, as it has been revised in the meantime [74].

Figure 4.3: The verification process in Verus.

4.3.1 Usability

Verus integrates itself into the Rust compiler. In contrast to Prusti and Creusot, Verus utilizes the HIR, as its foundation for reasoning [75]. In a previous version of Verus, the tool made use of a forked version of the compiler [70]. This has been rectified recently, and Verus now integrates itself into the default Rust compiler [76].

The user has to design the source code and annotations inside the verus macro, following Verus syntax rules [73]. The verification process in Verus is automatic. Verus converts the HIR into instructions for the underlying SMT solver, as seen in Figure 4.3.

The SMT solver then evaluates the instructions and returns the results to Verus, which consequently returns them to the user.

The source code Verus processes is split into three modes: *specification* mode, in which specification functions are declared. Those are not borrow-checked and represent functionally designed ghost code [70]. The second mode is the *proof* mode, in which proof functions are stated, for example user-specific lemmas [70]. Those functions are borrow-checked and also ghost code; they can additionally be equipped with pre- and postconditions [70]. The third mode is the *exec* mode, which contains functions that are compiled to machine code by the Rust compiler, hence including the guarantees standard Rust promises [70]. In this mode, functions implemented in the specification and proof mode can be called [70]. Furthermore, those functions can also be equipped with pre- and postconditions [70].

Because the verus macro is foreign to most IDEs, syntax highlighting and suggestions are not functional. To resolve this limitation, Verus syntax support inside the macro can be added to the *rust-analyzer* library [73]. This library represents a language-server protocol implementation for Rust, which can be used in various IDEs and code editors like Visual Studio Code or Emacs [77].

4.3.2 Technology

Verus translates source code into instructions for the SMT solver Z3 [70]. Functions written in specification mode are directly translated to SMT function calls, as Verus aims to provide lightweight encoding of verification conditions [70]. As a result, specification functions cannot be equipped with pre- and postconditions [70]. To support the user while encountering a failing specification, Verus allows to equip those with *soft* preconditions, referenced by the *recommends* keyword, which Verus only invokes if it encounters an error [70].

Verus relies on Rust's borrow-checker [70], like Creusot (see Chapter 4.2.2). The tool does not generate additional verification conditions for Z3, to prove the guarantees standard Rust promises. This is apparent in the process of how Verus encodes mutable references: they are encoded to Z3 as immutable values, one assignment per mutation [70], thereby fully trusting the Rust compiler with correct memory aliasing.

4.3.3 Feedback

Due to the integration of Verus in the Rust compiler, verification error messages and errors regarding the Verus-specific syntax are displayed like regular Rust compiler er-

Listing 6: Example output of a verification error in Verus.

```
error: postcondition not satisfied
  --> src/main.rs:9:5
   |
8  |               *val == *old(val) + 2
   |               - failed this postcondition
9  | /     {
10 | |         *val = *val + 1;
11 | |     }
   | |_____^ at the end of the function body

error: aborting due to previous error

verification results:: verified: 1 errors: 1
```

ror messages, similar to Prusti's feedback (see Section 4.1.3). As shown for example in Listing 6, Verus marks the postcondition which cannot be proven.

4.3.4 Language and Annotation Support

Verus supports recursive function calls in all three code modes [70]. The user has to ensure that recursive function calls in specification and proof mode terminate [70]. Verus enforces this with a required *decreases* clause, which marks the parameter that is decreased in the recursive function call [70]. Code written in exec mode is not affected by this requirement [70].

Verus does not support unsafe Rust [70]. According to the authors, it aims to replace some unsafe operations with core primitives [70]. For example, Verus offers *permission pointers* and *permission plus data* types as an alternative for raw heap pointers in unsafe Rust. Verus proofs the correctness of those replacements with a logic called *linear ghost state* [70]. In the complementary literature [70, 78], the authors of Verus prove the capabilities of their alternative by implementing a doubly-linked list. Theoretically, this can be done in safe Rust as well, but does not represent a feasible solution [79].

Verus supports *interior mutability* [70]. Interior mutability is a method to alter data behind a shared reference indirectly, which is considered unsafe [80]. The tool provides two strategies to safely use this technique: first, it offers an alternative to the *UnsafeCell<T>* data type [70], which safe wrappers like *Rc<T>* or *Cell<T>* depend on. Its behavior is analogous to the alternative raw heap pointers Verus offers [70]. Second, the tool provides a type called *InvCell<T>*, which does not keep track of its interior value [70]. It uses invariants to restrict the valid values and enforces those invariants at the time a value is written into the cell [70]. For the latter, the authors provide a function call example, which uses memoization across multiple clients while using interior mutability for the cache [70, 78].

Verus supports reasoning for concurrent data structures [70]. It uses *user-defined ghost values* which, in summary, represent a thread-local view of a global program state, enhanced with invariants and ghost variables [70]. The authors provide four examples demonstrating that technique, including a concurrent FIFO queue and a thread-safe reader-writer lock [70, 78].

Additionally, Verus has support for

- pre- and postconditions [70],
- assertions and assumptions [70, 81],
- quantifiers [70, 82],
- loop invariants [70, 83] and
- overflow/underflow checks [84].

4.3.5 Development

Verus is developed as a research project at ETH Zurich, Carnegie Mellon University in the USA, UNSW Sydney, VMWare Research and Microsoft Research [70]. It was first introduced at the 2022 occurrence of the Rust Verification Workshop [66]. The project members involved in the development process were Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno and Chris Hawblitzel [70]. The project is in part funded by VMWare, the Alfred P. Sloan Foundation, Intel Corporation and the Future Enterprise Security Initiative at Carnegie Mellon CyLab [70].

4.3.6 Activity

Verus is being actively developed, with over 1,750 commits in the main branch since 2021 [85].

Verus is not feature complete yet [85]. This lack of feature-completeness is also indicated by the several grayed-out chapters in the complementary user's guide [73]. There are currently about 40 issues open in GitHub, and about 150 have been closed [86].

Verus has been evaluated with 12 different example programs, which can be found, among other examples, in the complementary material [78]. Those examples had a total amount of 834 lines of code in specification mode, 487 lines of code in proof mode and 811 lines of code in exec mode.

Brun has used Verus to verify paging functions written in Rust for the research operating system NrOS, targeting x86_64 architecture [87].

4.3.7 Scientific activity

Verus has been presented at the 2022 occurrence of the Rust Verification Workshop [66]. Furthermore, it has been featured by the Rust Formal Methods Interest Group [50] and presented on their official YouTube channel in 2023 [88]. It has also been featured on the official Rust YouTube channel in 2023 [89].

The first scientific publication mentioning Verus [70] has been cited three times in total, with one citation for the extended version and two for the normal version, according to Google Scholar at the time of writing.

4.3.8 Documentation

Verus provides a dedicated user's guide including code examples for various features of the tool [90]. The guide is a work-in-progress, as can be seen by the several grayed-out chapters inside of it.

The tool provides 14 example programs in its complementary material [78], which demonstrate the practical appliance of Verus.

4.3.9 Example

We were not able to verify the factorial example with the annotations provided in Listing 7. The annotations are based on the examples for Prusti (see Section 4.1.9) and Creusot (see Section 4.2.9). Verus provides an implementation of the *checked_mul* function for the *u*32 data type, which has been retrofitted with postconditions, specifying its behavior [53]. Therefore, the data type for *n* and the result inside the *Option* data type has been changed to *u*32 for this example. As can be seen in Listing 8, the loop invariant in line 22 of Listing 7 cannot be proven by Verus.

Listing 7: Rust example of a function calculating the factorial value of n with annotations for Verus.

```
verus! {
    spec fn fac(n: int) -> int
        decreases n
    {
        if n <= 0 { 1 } else { n * fac(n-1) }
    }

    exec fn factorial(n: u32) -> (result: Option<u32>)
        ensures
            match result {
                Some(res) => res as int == fac(n as int),
                None => true
            }
    {
        if n == 0 { return Some(1); }
        let mut result: u32 = 1;
        let mut k: u32 = 1;

        assert(fac(0) == 1 as int);
        while k < n
            invariant
                result as int == fac(k as int),
                k <= n,
        {
            k = k + 1;
            result = match result.checked_mul(k) {
                Some(res) => res,
                None => return None,
            };
        }
        Some(result)
    }
}
```

Listing 8: Failing invariant for the factorial example in Verus.

```
error: invariant not satisfied at end of loop body
  --> src/main.rs:22:17
   |
22 |                 result as int == fac(k as int),
   |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: aborting due to previous error

verification results:: 3 verified, 1 errors
```

4.4 Aeneas

Aeneas is a verification toolchain for translating Rust into a functional model [91], as shown in Figure 4.4. Its development has started in November 2021, according to the tool's GitHub log [92].

The Aeneas toolchain consists of two modules: the first module is *Charon* [93], which is a standalone Rust compiler plugin and is used to extract a *Low-level Borrow Calculus* (LLBC) representation from Rust source code. The second module is *Aeneas* [94], which performs the functional model translation based on the LLBC representation [91]. The verification of this generated functional model has to be performed by the user in an external proof assistant.

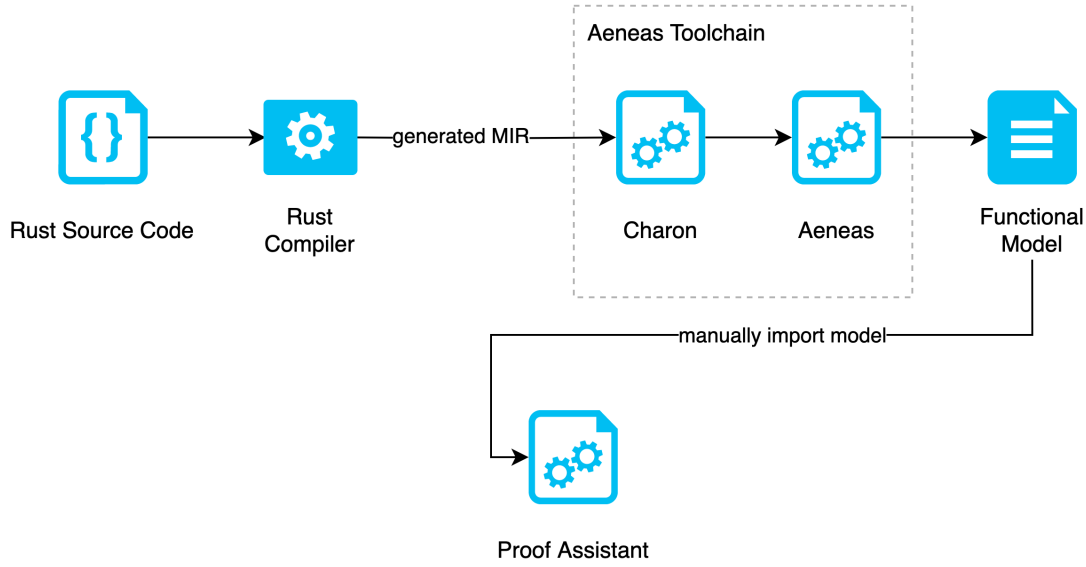

Figure 4.4: The extraction and translation process in Aeneas.

4.4.1 Usability

Charon integrates itself into the Rust compiler, using the generated MIR for its extraction process [91].

The translation process, shown in Figure 4.4, from Rust source code to a functional model is semi-automatic: first, the user has to invoke the Charon binary, which takes the source code and extracts the LLBC representation from it. Second, the extracted LLBC representation has to be parsed and processed by the Aeneas binary, which produces the functional model for the selected proof assistant. This model then has to be imported

Listing 9: Example error message for an unsupported Rust feature in Charon.

```
Compiling dictionary v0.1.0 (/Users/kevin/IdeaProjects/dictionary)
thread 'rustc' panicked at 'not implemented', src/register.rs:670:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: could not compile `dictionary`
```

into the proof assistant and verified there. Aeneas currently supports F* and Coq [91, 94]. According to the authors, support for HOL4, Lean and others is planned [91, 94].

The Aeneas toolchain does not require any annotations in the Rust source code [91]. That subsequently means, that the Rust source code and the annotations for the selected proof assistant are kept separated.

4.4.2 Technology

A key contribution of the Aeneas toolchain is the LLBC generation from Rust source code. According to the authors, the LLBC represents a new approach of borrow-checking and allows for fine-grained tracking of value aliasing [91]. Therefore, Aeneas does not rely on Rust's default borrow-checker. The LLBC has been inspired from Rust's MIR, but adds some high-level techniques like an intact control-flow and a limited form of pattern matching [91]. The authors provide fully-featured semantics for the LLBC in their complementary literature and a description of the extraction and translation process [91]. The authors also state that their provided technique surpasses the capabilities of Rust's default borrow-checker and even the experimental improved version of it, named *Polonius* [95] [91].

According to the authors, the application of the LLBC semantics is checked with the utilization of tightly-designed invariants, to ensure the correct extraction and translation of the Rust source code [91].

4.4.3 Feedback

The Aeneas toolchain provides feedback for translating the Rust source code into the functional model. Verification error messages are up to the used proof assistants.

Charon is the main source of feedback. If Charon is unable to extract the LLBC from the Rust source code, it issues an error. An error source is the use of unsupported Rust features. The generated error message is vague, as it does not state what exactly is causing the error, as can be seen in Listing 9.

4.4.4 Language and Annotation Support

According to the authors, Aeneas supports and focuses on a subset of safe Rust, which is functional in its *essence* [91]. It is planned to extend the scope of support for Rust and to be on par with other tools like Prusti (see Chapter 4.1) and Creusot (see Chapter 4.2) [91], so some of the following limitations could be solved in the future.

The tool can translate almost all variants of borrowing, which includes mutable and shared borrowing, two-phase borrows, reborrowing, nested borrows and returning borrows from function calls [91]. One exception is the handling of nested borrows in function headers, which is excluded to prevent an especially difficult case for translation from happening: nested mutable borrows with different lifetimes like *&'a mut &'b mut T* [91].

At the time of writing, loops are not supported, but recursive function calls are [91]. For recursive function calls, Aeneas adds auto-generated invariants to the generated model [91]. For example, for the F* programming language, Aeneas adds a decrease clause which refers to a user-supplied lemma which has to prove, that the recursive function call terminates [91].

Aeneas can also handle external function calls to libraries and code which interacts with external resources, for example I/O [91]. It is possible to mark those as *opaque*, which lets Aeneas assume them as correct [91]. The user can then add lemmas inside the generated model to those marked crates and functions, to further refine their assumed behavior [91].

At the time of writing, Aeneas does not support traits, closures, functions pointers, interior mutability, concurrent execution and type parameters containing a borrow, for example *Option<&mut T>* [91, 94].

Support for annotations is not evaluated, due to the outsourcing of the verification process.

4.4.5 Development

Aeneas is developed by Son Ho from the National Institute for Research in Digital Science and Technology in France and Jonathan Protzenko from Microsoft Research, USA [91]. The complementary paper [91] does not mention any funding.

4.4.6 Activity

The Aeneas toolchain is being actively developed. Aeneas [94] has about 1,500 commits in its main branch, Charon about 500 [93]. The authors of the toolchain are dedicated to improve its capabilities and limitations, as mentioned in Section 4.4.4. At the time of writing, there are 11 issues open for Aeneas and zero have been closed [96]. Charon has six issues open and two closed [97].

The authors have evaluated the Aeneas toolchain with an example implementation of a resizable hash table in Rust, which has been translated into a functional model for the F* programming language [91]. The authors have invested 201 lines of code in F* for verifying this implementation [91].

4.4.7 Scientific activity

Aeneas has been presented at the 2022 and 2023 occurrences of the Rust Verification Workshop [11, 66]. Additionally, it has been presented on the official YouTube channel of the Rust Format Methods Interest Group in 2022 [98].

The complementary publication regarding Aeneas [91] has been cited 15 times, according to Google Scholar.

4.4.8 Documentation

The tool provides a user's guide [94] on how to use the binaries for extracting the LLBC and generating the functional model. Additionally, the tool provides automatically generated developer documentation for the Charon [93] and Aeneas [94] modules.

The authors provide a detailed documentation of their LLBC semantics and example translations from Rust source code into a functional model for the F* proof assistant in their complementary paper [91].

4.4.9 Example

Due to the outsourcing of the verification process, the factorial example will be omitted. Performing formal verification with a proof assistant does not fit into the scope of this thesis.

4.5 MIRAI

MIRAI is an abstract interpreter for Rust for performing static analysis [99, 100]. Its development has started in November 2018, according to the tool's GitHub log [101].

MIRAI provides four key features:

- locate potential runtime errors without annotations [100],
- verify correctness properties with annotations [100] (visualized in Figure 4.5),
- tag analysis with annotations, especially taint analysis [99, 100], and
- call graph generation [102].

According to the authors, the goal for MIRAI is to become widely adopted for performing static analysis in Rust [100]. Also, according to the authors, MIRAI is a production-ready alternative to research projects like Prusti or Creusot, that is efficient and easy to use [99].

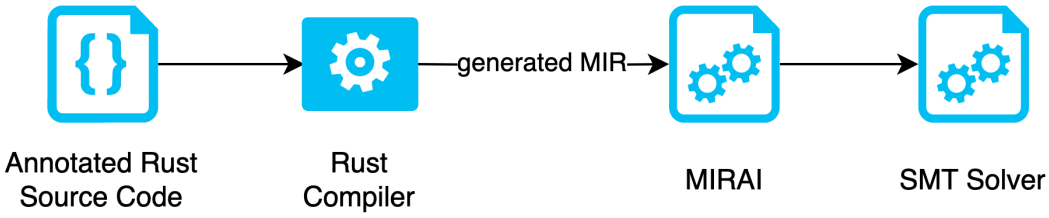

Figure 4.5: The verification process in MIRAI.

4.5.1 Usability

MIRAI integrates itself into the Rust compiler, using the generated MIR for its static analysis [103].

The tool can be used without annotations to prove the absence of runtime errors. According to the authors, it performs a *reachability analysis*, to prove that a program does not contain an execution path which may cause a runtime error [99]. This automatic approach can result in false positives and false negatives, if the MIR is missing for a function, for example a function call in another programming language [99]. False negatives can also occur, if MIRAI analyzes functions with a great number of loops and functions calls [99, 104]. MIRAI requires *entry points* for its analysis, such as the main function, tests or any other public functions [99]. The analysis is limited to functions

Listing 10: Example Rust code of an increase function with a pre- and postcondition from the contracts crate for MIRAI.

```
#[requires(*val <= u32::MAX - 1)]
#[ensures(*val == old(*val) + 1)]
fn incr(val: &mut u32) {
    *val += 1;
}
```

Listing 11: Example Rust code of an increase function with a pre- and postcondition from the mirai-annotations crate for MIRAI.

```
fn incr(val: &mut u32) {
    precondition!(*val <= u32::MAX - 1);
    let _old = *val;
    *val += 1;
    postcondition!(*val == _old + 1);
}
```

that are reachable from the entry points, therefore libraries require a broad coverage of unit tests to enable MIRAI to fully analyze the entire code base [99].

Annotations are required inside the Rust source code to prove further properties of an implementation. MIRAI supports two crates[1] for importing the annotations: first, the *contracts* crate [105], which is, according to the authors, more light-weight and accessible to beginners, but does not support the more advanced features of MIRAI, like tag analysis [99]. The annotations provided by this crate can be compared to annotations used by Prusti or Creusot, as can be seen in Listing 10. The second crate MIRAI supports is the *mirai-annotations* crate [106], which provides macros that can be used to annotate the code with specifications, as shown in Listing 11. It is possible to use both crates simultaneously.

4.5.2 Technology

MIRAI is implemented as an interpreter for the generated MIR, which can handle abstract and concrete values [99]. It uses the SMT solver Z3 for its reasoning [107].

The tool creates *summaries* for functions [99]. A summary consists of a generated list of preconditions for a function, an abstract return value, a list of side effects and a postcondition [99]. These summaries are generated and cached while MIRAI analyzes the code paths a particular function contains [99]. The preconditions are derived from paths which might trigger a runtime error and must be fulfilled by the caller of the function to not enter the error causing path [99].

[1] An external library is called *crate* in Rust

Listing 12: Example output of a verification error in MIRAI with the mirai-annotations crate.

```
warning: provably false postcondition
  --> src/main.rs:14:5
   |
14 |     postcondition!(*val == _old + 2);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |

warning: `mirai` (bin "mirai") generated 1 warning
```

4.5.3 Feedback

The integration of MIRAI into the Rust compiler enables error messages to be displayed as regular Rust errors, shown in Listing 12. Furthermore, due to the usage of Rust's crate system, IDEs are capable of performing syntax checking and error highlighting for the used annotations.

4.5.4 Language and Annotation Support

The complementary documentation for MIRAI does not state any lack of support for certain Rust features. Furthermore, it does not mention whether it does or does not support unsafe Rust code. However, the included test suite [108] contains tests which use unsafe Rust.

Using invariants for structs is possible via the contracts crate [99].

Quantifiers are not supported by MIRAI [99]. According to the authors, this deficiency is not a drawback due to possible workarounds with assumptions [99].

MIRAI supports *tag analysis* [107]. Tag analysis is a technique to determine the flow of information inside a program [107]. In detail, MIRAI enables the user to track untrusted input, constant time analysis to prevent timing side-channel attacks and combining verification results to their corresponding values [99, 107].

MIRAI is able to construct a precise call graph from Rust source code [102]. It can generate graphs for *Graphviz*, an open source graph visualization software [109], or convert them into the *Datalog* programming language [102].

MIRAI can handle foreign function calls, which are not present in the generated MIR, for example system calls or functions written in other programming languages [99]. These functions require a *model*, that enables MIRAI to reason about them [99]. MIRAI ships with models for some foreign functions from the Rust standard library [99]. Additionally, the user can create and supply MIRAI with custom models for additional foreign functions [99].

Furthermore, MIRAI supports assumptions, pre- and postconditions.

4.5.5 Development

MIRAI is developed by Facebook Research to address the lack of static analysis tools for Rust [99]. This absence of static analysis tools was discovered while researching and choosing Rust for the development of *Diem* [99]. Diem is a decentralized and programmable distributed ledger for providing a financial structure [110]. The source code for Diem (formerly known as Libra) has first been published in June 2019, according to its GitHub log [111].

4.5.6 Activity

MIRAI is being actively developed, with over 1,100 commits in the main branch since 2018 [100]. The latest release at the time of writing, version 1.1.8, has been released in May 2023 [112] and is considered stable [99]. According to the authors, there are subtle additions planed for MIRAI like providing a counterexample if the absence of runtime errors cannot be proven or improved support for parameterized unit tests [99]. Therefore, MIRAI can be considered nearly feature complete. There are currently 9 issues open for MIRAI and 99 have been closed [113].

Furthermore, MIRAI has been used to verify the Diem project [110].

4.5.7 Scientific activity

MIRAI has been presented at the 2021 occurrence of the Rust Verification Workshop [48]. Additionally, it has been featured by the Rust Formal Verification Group [50] and presented on their official YouTube channel [114].

MIRAI does not offer any complementary scientific publication. It does, however, offer a discussion for some of its design decisions in its GitHub project page [115].

4.5.8 Documentation

MIRAI provides an overview [99] of its capabilities and provides a user's guide for its tag analysis [107] and graph generator feature [102]. Additionally, both the contract [105] and mirai-annotations [106] crate offer a user's guide.

Listing 13: Rust example of a function calculating the factorial value of n with annotations for MIRAI.

```
1  fn fac(n: u64) -> u64 {
2      if n == 0 { 1 } else { n * fac(n-1) }
3  }
4
5  fn factorial(n: u64) -> Option<u64> {
6      if n == 0 { return Some(1); }
7      let mut result: u64 = 1;
8      let mut k: u64 = 1;
9
10     checked_verify!(fac(0) == 1);
11     while k < n {
12         checked_verify!(result == fac(k));
13         checked_verify!(k <= n);
14         k = k + 1;
15         result = match result.checked_mul(k) {
16             Some(res) => res,
17             None => return None,
18         };
19         checked_verify!(result == fac(k));
20         checked_verify!(k <= n);
21     }
22     postcondition!(result == fac(n));
23     Some(result)
24 }
```

MIRAI provides six code examples on how to use its verification and tag analysis features [116]. The tool also includes a test suite, where safe and unsafe Rust code examples are analyzed by MIRAI [108]. Furthermore, the Diem project [110] offers practical examples for the appliance of MIRAI in combination with the mirai-annotations crate.

4.5.9 Example

We were not able to verify the factorial example with the annotations provided in Listing 13. The example uses the mirai-annotations crate. The annotations are based on the examples for Prusti (see Section 4.1.9) and Creusot (see Section 4.2.9). Because MIRAI does not support ghost functions, the *fac* function for calculating the factorial value recursively is a default Rust function. Furthermore, because MIRAI does not support loop invariants, the *checked_verify* macro calls in lines 12, 13, 19 and 20 are used to replace their behavior. As shown in Listing 14, the generated verification conditions for the annotated *checked_verify* macro calls and the postcondition cannot be proven by MIRAI.

Listing 14: Failing verification conditions for the factorial example in MIRAI.

```
warning: possible false verification condition
  --> src/main.rs:10:5
   |
10 |     checked_verify!(fac(0) == 1);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: possible false verification condition
  --> src/main.rs:12:9
   |
12 |         checked_verify!(result == fac(k));
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: possible false verification condition
  --> src/main.rs:19:9
   |
19 |         checked_verify!(result == fac(k));
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: possible unsatisfied postcondition
  --> src/main.rs:22:5
   |
22 |     postcondition!(result == fac(n));
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

4.6 Discussion of the Overview

A common feature of Creusot, Aeneas and MIRAI is the use of the MIR for their reasoning. Prusti relies on both, MIR and HIR, whereas Verus relies exclusively on the HIR. The Rust foundation supports a project called *Stable MIR*, which aims to provide a stable API for accessing the MIR in the Rust compiler [117]. At the time of writing, this API is considered unstable and can therefore only be used in the nightly version of the compiler.

Each of the discussed tools has its focus on different goals and therefore its individual pros and cons. We provide no general recommendation, the user has to analyze its requirements the tool has to fulfill and choose based on these requirements.

Prusti offers an easy operation and an extensive documentation of its capabilities. It is also the most popular tool, measured on the amount of citations of the complementary scientific literature. The guided tour is helpful for both beginners and experts in the domain of formal verification.

Creusot is a versatile tool, due to the usage of Why3 for its verification. It enables the user to apply different solving strategies and quickly switch between SMT solvers for its reasoning.

Verus also offers an easy operation and extensive documentation, like Prusti. It is the only tool which provides safe replacements for interior mutability and raw heap pointers and supports concurrent data structures. Due to the use of a different syntax, legacy Rust source code has to be converted first.

The Aeneas toolchain is flexible, as it decouples the verification process from the toolchain itself, comparable to the approach Creusot follows. The user can choose from a variety of supported proof assistants and apply different solving strategies.

MIRAI is a good addition to exhaustive unit testing, due to the need for entry points. It offers unique features like tag analysis and call graph visualization. Compared to the other tools mentioned in the overview, its verification capabilities are basic.

Table 1 provides a brief overview of a chosen variety of annotations supported by Prusti, Creusot, Verus and MIRAI. Aeneas is excluded, as the supported annotations depend on the chosen language, in which the functional model has been extracted. Note that ghost variables can be found in the source code of Prusti and its test suite, but this feature is not documented in the user guide or scientific literature. This is also discussed in Section 5.3.3.

Table 1: Overview of supported annotations for the tools Prusti, Creusot, Verus and MIRAI.

Annotations	Prusti	Creusot	Verus	MIRAI
Pre-/Postconditions	✓	✓	✓	✓
Assertions	✓	✓	✓	✓
Assumptions	✓		✓	✓
Trusted functions	✓	✓		
Loop invariants	✓	✓	✓	
Ghost variables	(✓)	✓	✓	
Quantifiers	✓	✓	✓	

5 Practical Application of Formal Verification in Rust

In this section, we consider Creusot and Prusti as interesting tools for the practical application of formal verification. Furthermore, we consider the sorting algorithm Bubblesort as a viable example for demonstrating the practical application of formal verification.

Section 5.1 describes the experimental setup for Creusot and Prusti. Section 5.2 presents the specifications for the sorting algorithm. Section 5.3 discusses the verification process for Bubblesort.

5.1 Setup

Creusot is based on the latest commit[1] from 25th July 2023. Why3 has been used in version 1.6.0+git, including the SMT solvers Z3 in version 4.12.2 and Alt-Ergo in version 2.4.3. All goals in Why3 were executed with the strategy *Auto Level 2*.

Prusti is based on the latest commit[2] from 14th August 2023. The used ViperTools toolchain for Prusti is based on the release[3] from 5th August 2023. The included SMT solver Z3 inside the ViperTools toolchain has been replaced with version 4.12.2.

5.2 Specifications for Sorting Algorithms

We use the notation $\mathbb{N}_0$ to refer to all natural numbers, including zero. Furthermore, we specify that sequences start at index zero and end at index $n-1$, with n as the length of the sequence.

All sorting algorithms share the same desired behavior: *the output of the algorithm is sorted, without adding or removing any element.* To specify this behavior, two specifications are necessary. The first specification specifies that *the output of the algorithm is sorted*:

Given a partially-ordered set X and a finite sequence $A = (x_i \mid x_i \in X \text{and } 0 \leq i < n)$ with length n, we say that A is ordered in ascending order, if the following holds:

$$\text{for every } k, l \in \mathbb{N}_0 \text{ it holds that: If } k \leq l < n, \text{ then } x_k \leq x_l\,. \tag{1}$$

[1] https://github.com/xldenis/creusot/commit/cf5c886dc36295f7804c56546f1568617767a77f

[2] https://github.com/viperproject/prusti-dev/commit/cb6849485887d99f04f127fe2fdb046c545046b5

[3] https://github.com/viperproject/viper-ide/releases/tag/v-2023-08-05-122

For example, the finite sequence $(1, 2, 3)$ is sorted in ascending order, $(3, 2, 1)$ is not.

Second, we specify the *without adding or removing any element* behavior. This specification is based on the permutation definition for Why3 [118]:

Let $occ(x, s)$ count the occurrences of x in a sequence s. Given a set X and finite sequences $A = (x_i \mid x_i \in X \text{and } 0 \leq i < n)$ and $B = (y_i \mid y_i \in X \text{and } 0 \leq i < n)$, both with length n. We call B a permutation of A if

$$\text{for every } k \in X \text{ it holds that: } occ(k, A) = occ(k, B) \tag{2}$$

holds. That means, that the number of occurrences of element x in A must be equal to the number of occurrences of element x in B, for every element of X.

For example, $(2, 1, 3)$ is a permutation of $(1, 2, 3)$ but $(1, 2)$ or $(1, 2, 3, 4)$ are no permutations of $(1, 2, 3)$, according to (2).

Finally, the postconditions for a sorting algorithm, taking a sequence A over a partially-ordered set X and producing a sequence A' over X, are:

1. A' is sorted in ascending order and
2. A' is a permutation of A.

5.3 Bubblesort

Bubblesort is a simple and inefficient sorting algorithm, which repeatedly swaps adjacent elements that are out-of-order [119, 120]. The implementation covers many features of the Rust programming language like loops, mutable borrowing, using the standard library for storing vectors and performing swap operations. The simplicity of Bubblesort makes it a good example for demonstrating the practical application of formal verification.

Please note that the annotations used in this demonstration are based on a Bubblesort implementation from a lecture for the programming language Dafny [121]. The annotations have been adapted, as the control variable i for the outer while loop of the Rust implementation starts at index zero, as can be seen in Listing 15. The Dafny example uses the total length of the list minus one as the starting point for i. Furthermore, an annotation for proving a permutation according to (2) of the vector has been added.

The implementation in Listing 15 takes a mutable reference pointing to a vector containing *i32* values and applies the Bubblesort algorithm. Vectors in Rust are defined in the

Listing 15: Rust implementation of Bubblesort.

```
fn bubblesort(arr: &mut Vec<i32>) {
    let mut i: usize = 0;
    while i < arr.len() {
        let mut j: usize = 0;
        while j < arr.len() - i - 1 {
            if arr[j] > arr[j+1] {
                arr.swap(j, j+1);
            }
            j += 1;
        }
        i += 1;
    }
}
```

standard library and represent a dynamically allocated array [122]. The swap function is defined for the vector data type and accepts indices as *usize* data types. The len function is also part of the vector data type and returns the length of the vector as a *usize* data type. The *usize* data type represents an unsigned integer in pointer-size, dependent on the platform the implementation has been compiled for [123]. The *i32* data type represents a signed integer with 32 bit width. The mutable reference pointing to the vector permits the function to alter the content of the vector, while having exclusive memory access to it.

5.3.1 Annotations for Bubblesort

We derive the postconditions for Bubblesort based on the discussed specifications in Section 5.2. The finite sequences are implemented as vectors in Rust. This results in two postconditions for Bubblesort: first, the vector on which the mutable reference *arr* points to, must be sorted according to specification (1). Second, the resulting vector must be a permutation of the original vector, according to specification (2). Preconditions are not required for this implementation.

To prove the postconditions for Bubblesort, it is necessary to provide loop invariants. There are two groups of loop invariants required for this implementation: for the outer while loop and the inner while loop.

Please note that $V[x]$ denotes the xth element of the vector V.

The first loop invariant for the outer while loop asserts that *control variable i is within bounds*. Annotation (3) must hold, with n as the length of the vector:

$$0 \leq i \leq n \tag{3}$$

Control variable i is not allowed to be less than zero, as it would cause an invalid vector access at the inner loop. This requirement is enforced by Rust's type system, which does not allow the *usize* data type of i to be negative. Furthermore, i cannot be greater than n, as it would cause the inner loop to never execute. Additionally, this annotation ensures that i does not overflow during the increment operation on line 11 of Listing 15, because the standard library ensures that n is an in-bounds value for *usize* and i cannot be incremented beyond n. Note that i can be equal to n after the last iteration of the outer while loop. The loop guard for the outer while loop does not allow the control flow to enter the while loop with i equal n, therefore preventing an invalid vector access.

The second loop invariant for the outer while loop asserts that the *vector is a permutation of the original vector*, according to (2). If this is true for all iterations of the outer loop and due to the transitive property of permutation, it is valid to state that the annotation is true for the entire Bubblesort implementation and therefore the postcondition is true.

The third loop invariant for the outer while loop asserts that the *sorted area grows from the end of the vector*. Control variable i behaves as the counter for the amount of sorted elements inside this area. Annotation (4) must hold, with vector V and n as the length of the vector:

$$\text{for every } k, l \in \mathbb{N}_0 \text{ it holds that: If } n - i \leq k < l < n, \text{ then } V[k] \leq V[l]\,. \tag{4}$$

The sorted area starts at index $n - i$ and ends at index $n - 1$.

The fourth loop invariant for the outer while loop asserts that *control variable i partitions the vector*. Annotation (5) must hold with vector V and n as the length of the vector:

$$\text{for every } k, l \in \mathbb{N}_0 \text{ it holds that: If } 0 \leq k < i \leq l < n, \text{ then } V[k] \leq V[l]\,. \tag{5}$$

The annotation ensures that the vector V is partioned: That means, that every element in the left part of the vector, separated by i, has to be less or equal compared to every element in the right part. The element indexed by i belongs to the right part of the vector.

The first loop invariant for the inner while loop asserts that *control variable j is within bounds and outside the already sorted area.* Annotation (6) must hold with control variable j for the inner and i for the outer while loop and n as the length of the vector:

$$0 \leq j \leq n - i - 1 \tag{6}$$

As previously stated in annotation (4), $n - i$ defines the start index of the already sorted area, starting from the right end of the vector. The control variable j is only allowed to

be located in the left unsorted part of the vector. This implies that j can be incremented with every iteration from zero to $n - i - 1$. The decrease by one operation is mandatory, as otherwise the if statement and swap operation on line 6 and 7 in Listing 15 would access the already sorted part of the vector. Furthermore, the annotation (4) ensures that j cannot overflow during the increment operation, comparable to annotation (3).

The second loop invariant for the inner while loop asserts that the *vector is a permutation of the original vector*, according to (2). To ensure that the invariant for the outer while loop permutation invariant is guaranteed, it is mandatory to ensure that the invariant for the inner loop is guaranteed as well.

The third loop invariant for the inner while loop asserts that the *left part of the vector is equal or smaller than the jth element.* It ensures that the swap operation pushes the highest element of the unsorted area to the right end of this area. Annotation (7) must hold:

$$\text{for every } k \in \mathbb{N}_0 \text{ it holds that: If } 0 \leq k < j, \text{ then } V[k] \leq V[j]\,. \tag{7}$$

The annotation states that the current element indexed by control variable j must be greater or equal compared to all elements on the left side of the vector, separated by j. This subsequently means, that after the loop has terminated, the highest element is located at the right end of the unsorted area and can be added to the sorted area.

The fourth invariant for the inner while loop asserts that the *sorted area grows from the end of the vector.* This invariant is mandatory to prove the corresponding outer while loop invariant for annotation (4). It also ensures that the inner while loop does not tamper the already sorted area of vector.

The fifth invariant for the inner while loop asserts that the *control variable i partitions the vector.* Like the fourth invariant for the inner while loop, this invariant is mandatory for proving the corresponding outer while loop invariant for annotation (5).

5.3.2 Implementation of the Annotations in Creusot

This section explains and discusses the implementation of the annotations introduced in Section 5.3.1 for the tool Creusot.

There are two omnipresent operators used in this section: The ^ operator refers to the *final* value of a parameter, using Creusot's *prophecy* technique (see also Section 4.2.4). For example, ^*arr* refers to the state the parameter has after a function mutating it has terminated. The second operator is the @ operator, which refers to the mathematical model for a data type (see also Section 4.2.4). For example, a vector *Vec<T>* can be modelled as a mathematical sequence *Seq<T>*; an *usize* can be modelled as an integer *Int*.

The first postcondition for proving the sorted specification (1) for the vector *arr* is located on line 13 of Listing 16. The complementary paper for Creusot [56] provides two examples for a *sorted_range* and *sorted* predicate. These examples correspond to the sorted specification (1). The sorted predicate acts as a wrapper for calling the sorted_range predicate on the complete sequence.

The second postcondition for proving the permutation according to (2) of the vector *arr* is located on line 14 of Listing 16. The complementary paper for Creusot [56] mentions an included *permutation_of* predicate. This predicate is implemented for the *Seq<T>* data type and also expects a *Seq<T>* as the predicate's parameter. To the best of the author's knowledge, this predicate adheres to the permutation specification (2).

Listing 16: Postconditions for Bubblesort in Creusot including sorted and sorted_range predicates.

```
1  #[predicate]
2  fn sorted_range(v: Seq<i32>, start: Int, end: Int) -> bool {
3      pearlite! {
4          forall<k : Int, l : Int> start <= k && k < l && l < end ==> v[k] <= v[l]
5      }
6  }
7
8  #[predicate]
9  fn sorted(v: Seq<i32>) -> bool {
10     sorted_range(v, 0, v.len())
11 }
12
13 #[ensures(sorted((^arr)@))]
14 #[ensures((^arr)@.permutation_of(arr@))]
15 fn bubblesort(arr: &mut Vec<i32>) { /* ... */ }
```

The first loop invariant for the outer while loop is located on line 11 of Listing 17. It ensures that *control variable i is within bounds* according to annotation (3). Creusot provides an included *len* function, which is implemented for the *Seq<T>* data type [56]. It returns the amount of elements stored in the sequence as an *Int* data type.

The second loop invariant for the outer while loop is located on line 12 of Listing 17. It ensures that *arr is a permutation of old_arr*, according to (2). The permutation_of predicate requires accessing the state of *arr* at the beginning of the execution of Bubblesort. Creusot provides the *gh* macro for that purpose, as shown on line 8 of Listing 17. It creates a ghost variable for storing the current value of a parameter at a specific point of the execution. This ghost variable can be used inside of invariants and assertions.

The third loop invariant for the outer while loop is located on line 13 of Listing 17. It ensures that the *sorted area grows from the end of the vector arr*, according to annotation (4).

The fourth loop invariant for the outer while loop is located on line 14 of Listing 17. It ensures that the *control variable i partitions the vector arr*, according to annotation (5).

The annotation has been implemented in the *partioned* predicate.

Listing 17: Outer loop invariants including partioned predicate for Bubblesort in Creusot.

```
1  #[predicate]
2  fn partioned(v: Seq<i32>, i: Int) -> bool {
3          forall<k : Int, l : Int> 0 <= k && k < i && i <= l && l < v.len() ==> v[k] <= v[l]
4      }
5  }
6
7  fn bubblesort(arr: &mut Vec<i32>) {
8      let old_arr = gh! { arr };
9      let mut i: usize = 0;
10
11     #[invariant(0 <= i@ && i@ <= arr@.len())]
12     #[invariant(arr@.permutation_of(old_arr@))]
13     #[invariant(sorted_range(arr@, arr@.len() - i@, arr@.len()))]
14     #[invariant(partioned(arr@, arr@.len() - i@))]
15     while i < arr.len() { /* ... */ }
16 }
```

The first loop invariant for the inner while loop is located on line 8 of Listing 18. It ensures that the *control variable j is within bounds and outside the already sorted area* of *arr*, according to annotation (6).

The second loop invariant for the inner while loop is located on line 9 of Listing 18. It ensures that *arr is a permutation of old_arr*, according to (2).

The third loop invariant for the inner while loop is located on line 10 of Listing 18. It ensures that *all elements in the left part of the vector arr are smaller or equal to the jth element*, according to annotation (7). Creusot allows defining quantifiers inside of invariants without calling a dedicated predicate.

The fourth loop invariant for the inner while loop is located on line 11 of Listing 18. It ensures that the *sorted area grows from the end of the vector arr*, according to annotation (4). It also ensures that the inner while loop does not modify the already sorted area of *arr*.

The fifth loop invariant for the inner while loop is located on line 12 of Listing 18. It ensures that the *control variable i partitions the vector arr*, according to annotation (5).

Listing 18: Inner loop invariants for Bubblesort in Creusot.

```
fn bubblesort(arr: &mut Vec<i32>) {
    let old_arr = gh! { arr };
    let mut i: usize = 0;

    while i < arr.len() {
        let mut j: usize = 0;

        #[invariant(0 <= j@ && j@ <= arr@.len() - i@ - 1)]
        #[invariant(arr@.permutation_of(old_arr@))]
        #[invariant(forall<k : Int> 0 <= k && k < j@ ==> arr[k] <= arr[j@])]
        #[invariant(sorted_range(arr@, arr@.len() - i@, arr@.len()))]
        #[invariant(partioned(arr@, arr@.len() - i@))]
        while j < arr.len() - i - 1 { /* ... */ }
    }
}
```

5.3.3 Implementation of the Annotations in Prusti

This section explains and discusses the implementation of the annotations introduced in Section 5.3.1 for the tool Prusti.

To the best of the author's knowledge, Prusti has a limitation for using non-slice types inside of comparison statements at the time of writing, as can be seen in Listing 19. To address this limitation, the unsorted vector is being converted into a mutable slice *&mut* [*i*32], prior to calling the *bubblesort* function. A slice in Rust represents a view into a contiguous sequence of elements of a collection, which can either be mutable or shared [124]. Additionally, the use of slices is encouraged by a second limitation of Prusti: to the best of the author's knowledge, the use of the *usize* data type in predicates to access elements inside the vector is not supported at the time of writing, as can be seen in Listing 20. Predicates are pure functions by design in Prusti [34].

Listing 19: Unsupported feature error for using a non-slice data type inside an comparison statement in Prusti.

```
error: [Prusti: unsupported feature] Non-slice LHS type '&i32' not supported yet
  --> src/main.rs:79:16
   |
79 |             if arr[j] > arr[j+1] {
   |                ^^^^^^
```

Listing 20: Unsupported feature error for using the usize data type to access elements inside a mutable vector reference in Prusti.

```
error: [Prusti: unsupported feature] Using usize as index/range type for &std::vec::Vec<i32> is not currently
       supported in pure functions
--> src/main.rs:32:86
|
32 |             forall(|i: usize, j: usize| (start <= i && i < j && j < end) ==> v[i] <= v[j])
```

The first postcondition for proving the sorted specification (1) for the mutable slice *arr*

is located on line 19 of Listing 21. The sorted and sorted_range predicates have been adapted from Creusot.

To the best of the author's knowledge, Prusti does not provide an included permutation predicate at the time of writing. Instead, we guarantee that the length of the mutable slice does not change during execution. This annotation does not represent as strong a guarantee as the permutation specification (2), because it allows the Bubblesort implementation to remove and add elements at arbitrary indices. For example, it would be a valid result if the algorithm takes a mutable slice $[3, 2, 1]$ and produces $[1, 1, 1]$. This result is valid for the sorted specification (1) and has an equal length compared to the input slice, but does not contain the same elements as before. The equal length postcondition can be found on line 20 of Listing 21. The keyword *old* refers to the state of *arr* prior to the execution of the function. To use the standard library function *len* for the slice data type inside the postcondition, it is necessary to provide an external specification for Prusti. In this external specification, the len function has to be marked as *pure*, as only pure functions and predicates are allowed to be called in Prusti annotations (see also Section 4.1.1).

Listing 21: Postconditions for Bubblesort in Prusti including sorted and sorted_range predicate and external specifications for standard library function len.

```
1  #[extern_spec]
2  impl<T> [T] {
3      #[pure]
4      pub fn len(&self) -> usize;
5  }
6
7  predicate! {
8      fn sorted_range(v: &[i32], start: usize, end: usize) -> bool {
9              forall(|k: usize, l: usize| (start <= k && k < l && l < end) ==> v[k] <= v[l])
10     }
11 }
12
13 predicate! {
14     fn sorted(v: &[i32]) -> bool {
15         sorted_range(v, 0, v.len())
16     }
17 }
18
19 #[ensures(sorted(arr))]
20 #[ensures(old(arr.len()) == arr.len())]
21 fn bubblesort(arr: &mut [i32]) { /* ... */ }
```

The first loop invariant for the outer while loop is located on line 12 of Listing 23. It ensures that *control variable i is within bounds*, according to annotation (3).

The second loop invariant for the outer while loop is located on line 13 of Listing 23. It ensures that the *length of the mutable slice does not change* inside the outer while loop. To be able to compare, it is necessary to save the length of *arr* prior to executing Bubblesort. Prusti can create ghost variables with the *ghost* macro, which encapsulates a value inside a *Ghost<T>* data type. At the time of writing, this feature is not described

in the documentation. However, examples of the macro appear in the test suite of Prusti. These examples are not reusable for the desired use case of this invariant. This is because the examples only create the ghost variable, but do not access the value inside of it. Trying to access the value inside the ghost variable results in a Prusti error, as shown in Listing 22. Therefore, the length of the mutable slice is stored at the beginning of the execution in a regular Rust *usize* value called *old_len*.

Listing 22: Invalid specification error for dereferencing a Ghost<T> value in Prusti.

```
error: [Prusti: invalid specification] use of impure function "std::ops::Deref::deref" in pure code is not
     allowed
  --> src/main.rs:85:25
   |
85 |         body_invariant!(*old_len == arr.len());
   |                         ^^^^^^^^
```

The third loop invariant for the outer while loop is located on line 14 of Listing 23. It ensures that the *sorted area grows from the end of the mutable slice arr*, according to annotation (4).

The fourth loop invariant for the outer while loop is located on line 15 of Listing 23. It ensures that the *control variable i partitions the mutable slice arr*, according to annotation (5). The annotation has been implemented in the *partioned* predicate.

Listing 23: Outer loop body invariants including partioned predicate for Bubblesort in Prusti.

```
1  predicate! {
2      fn partioned(v: &[i32], i: usize) -> bool {
3          forall(|k: usize, l: usize| (0 <= k && k <= i && i < l && l < v.len()) ==> v[k] <= v[l])
4      }
5  }
6
7  fn bubblesort(arr: &mut [i32]) {
8      let old_len = arr.len();
9      let mut i: usize = 0;
10
11     while i < arr.len() {
12         body_invariant!(0 <= i && i <= arr.len());
13         body_invariant!(old_len == arr.len());
14         body_invariant!(sorted_range(arr, arr.len() - i, arr.len()));
15         body_invariant!(partioned(arr, arr.len() - i));
16         /* ... */
17 }
```

The first loop invariant for the inner while loop is located on line 16 of Listing 24. It ensures that the *control variable j is within bounds and outside the already sorted area* of *arr*, according to annotation (6).

The second loop invariant for the inner while loop is located on line 17 of Listing 24. It ensures that the *length of the mutable slice arr does not change*.

The third loop invariant for the inner while loop is located on line 18 of Listing 24. It ensures that the *left part of the mutable slice arr is equal or smaller than the jth element* according to annotation (7). Prusti allows defining quantifiers inside of invariants without calling a dedicated predicate.

The fourth loop invariant for the inner while loop is located on line 19 of Listing 24. It ensures that the *sorted area grows from the end of the mutable slice arr*, according to annotation (4). It also ensures that the inner while loop does not modify the already sorted area of *arr*.

The fifth loop invariant for the inner while loop is located on line 20 of Listing 24. It ensures that the *control variable i partitions the vector arr*, according to annotation (5).

Prusti requires designated specifications for external functions like swap for the slice data type. These enable Prusti to reason about the properties of this function (see also Section 4.1.4). The specifications consist of two preconditions and four postconditions, as can be seen in Listing 24: both preconditions require the function parameters *a* and *b* to be valid indices for the slice, referenced by *self*. The postcondition on line 5 states, that swap does not change the length of the vector. The postconditions on line 6 and 7 ensure that the element at index *a* is located at index *b* and vice versa, after swap has terminated. The *snap* function is used to take a copy of the value referenced by *self* and to bypass Rust's ownership rules [42], which is necessary for both statements. The *old* function borrows *self* and prevents it to be used in the comparison at the same time. The postcondition on line 8 ensures, that all elements inside the mutable slice, which have not been affected by the swap function, are positioned at the same index as prior to the execution of the function.

Listing 24: Inner loop body invariants for Bubblesort including external specificaions for the swap function in Prusti.

```
#[extern_spec]
impl<T: PartialEq> [T] {
    #[requires(0 <= a && a < self.len())]
    #[requires(0 <= b && b < self.len())]
    #[ensures(self.len() == old(self.len()))]
    #[ensures(snap(&self[a]) == old(snap(&self[b])))]
    #[ensures(snap(&self[b]) == old(snap(&self[a])))]
    #[ensures(forall(|i: usize| (0 <= i && i < self.len() && i != a && i != b) ==> snap(&self[i]) == old(snap
        (&self[i]))))]
    pub fn swap(&mut self, a: usize, b: usize);
}

fn bubblesort(arr: &mut [i32]) {
        /* ... */
        let mut j: usize = 0;
        while j < arr.len() - i - 1 {
            body_invariant!(0 <= j && j <= arr.len() - i - 1);
            body_invariant!(old_len == arr.len());
            body_invariant!(forall(|k: usize| (0 <= k && k < j) ==> arr[k] <= arr[j]));
            body_invariant!(sorted_range(arr, arr.len() - i, arr.len()));
            body_invariant!(partioned(arr, arr.len() - i));
            if arr[j] > arr[j+1] {
                arr.swap(j, j+1);
            }
            /* ... */
}
```

5.3.4 Discussion of Bubblesort Verification with Creusot and Prusti

The verification of the postconditions and invariants described in Section 5.3.2 was successful with Creusot and Why3. Creusot creates a total of 30 goals for Why3, which are derived from the supplied annotations: six prove the absence of integer overflows, four prove that all indices accessing the vector are valid, 18 prove that the invariants are guaranteed before and after each loop iteration and two prove the postconditions for Bubblesort.

The verification of the postconditions and invariants described in Section 5.3.3 was partly successful for Prusti. There were two adjustments necessary to prove all postconditions and invariants: first, the *sorted_range* predicate for proving specification (1) had to be extended. The author was not able to prove the previously discussed implementation of this predicate with Prusti, as shown in Listing 21. To examine this behavior, the predicate has been verified in Creusot and Why3 with only the Z3 SMT solver activated. Z3 was also not able to find a solution, comparable to the behavior observed in Prusti. To further investigate this behavior, the Alt-Ergo SMT solver had been activated again in Why3. Alt-Ergo was capable of proving the annotation. This discovery may be due to that each SMT solver is designed differently and has different strengths [125]. To solve this limitation in Prusti, the sorted_range predicate has been extended with two additional constraints inside the *forall* quantifier, as can be seen in Listing 25: lines 4 and 5 ensure that the parameters *start* and *end* are valid indices for the shared slice

v. These two additional constraints enable Z3 to verify this annotation in both Prusti and Creusot/Why3 with Z3 activated only. The second adjustment was to omit the *partioned* annotation in Prusti. The author was not able to prove the partioned predicate (5) with Prusti for the Bubblesort implementation. This behavior has also been observed for Creusot and Why3 with only Z3 activated. While researching, it turned out that the partioned annotation, which is based on a lecture for the programming language Dafny (see also Section 5.3), is not required to prove the postconditions. Omitting this annotation in combination with the extended sorted_range predicate results in a positive verifying result in Prusti. This discovery can also be observed with Creusot and Why3, with both Alt-Ergo and Z3.

Listing 25: Enhanced sorted_range predicate for Prusti.

```
predicate! {
    fn sorted_range(v: &[i32], start: usize, end: usize) -> bool {
            forall(|k : usize, l : usize| (start <= k && k < l && l < end
            && 0 <= start && start < v.len()
            && 0 <= end && end < v.len())
            ==> v[k] <= v[l])
    }
}
```

6 Conclusion and Outlook

This work provides an overview over the tools Prusti, Creusot, Verus, Aeneas and MIRAI for formal verification in Rust. We evaluated each tool with a set of parameters and briefly demonstrated the practical application of each on a factorial example implementation. We chose Creusot and Prusti as interesting candidates for performing formal verification on the Bubblesort sorting algorithm. We introduced the desired specifications of the sorting algorithm and demonstrated the implementation of the resulting annotations for each tool. We discussed the resulting behavior and limitations of both tools. To the best of the author's knowledge, Bubblesort has not been verified in Rust prior to this thesis.

The use of Prusti is beginner-friendly. Its extensive documentation and guided tour provide a solid basis for performing formal verification. However, its limitations at the time of writing can be challenging. Support for different SMT solvers could make the tool more versatile.

Creusot in combination with Why3 provides a good verification experience. Why3's ability to verify goals with different SMT solvers can be helpful for debugging failing annotations. However, the lack of a full documentation for Creusot makes the use challenging.

The approach Verus follows is interesting, especially the support for interior mutability and concurrent data structures. The documentation is also extensive and makes the use beginner-friendly. Like Prusti, support for different SMT solvers could make the tool more versatile.

The Aeneas toolchain offers a unique technique with the Low-level Borrow Calculus. The separation of Rust source code and the annotations for the selected proof assistant can make an iterative development and verification process cumbersome. The error reporting mechanism in Charon for encountering an unsupported Rust feature could be improved.

MIRAI provides a stable experience. The verification capabilities could be extended. Like Prusti and Verus, support for different SMT solvers could make the tool more versatile.

Outlook. This thesis considered tools for static analysis. There exists a wide variety of tools which provide dynamic analysis, for example Kani[1], Loom[2], Miri[3] and more. These tools could also be analyzed with the parameters and the factorial example introduced in this thesis. Furthermore, the failing annotations for the factorial example for Verus in Section 4.3.9 and MIRAI in Section 4.5.9 could be investigated further. Additionally, this example could also be verified with Aeneas. For future work in Prusti, it could be interesting to swap the SMT solver inside the Viper toolchain, according to previous work accomplished by Anthony [26]. Additionally, further investigation into the *ghost* macro for Prusti may yield a viable result regarding the use of ghost variables. Furthermore, for proving the same specifications as Creusot, it could be interesting to design a permutation predicate according to specification (2). In addition, the implementation for Bubblesort could be converted to a generic implementation, sorting not only *i*32 values. This could demonstrate the handling capabilities of the tools regarding generic data types. At last, it could be interesting to use the remaining tools Verus, Aeneas and MIRAI to verify Bubblesort.

[1] https://github.com/model-checking/kani
[2] https://github.com/tokio-rs/loom
[3] https://github.com/rust-lang/miri

Declaration

The author declares that he has prepared the present work independently, without outside help and without using any aids other than those specified. Thoughts taken directly or indirectly from external sources (including electronic sources) are without exception marked as such. The thesis has not yet been submitted in the same or a similar form or in extracts in the context of another examination. In addition, the author confirms that he has read and understood the chair guidelines in the currently valid version and is therefore clear about the requirements and assessment criteria.

Date & Place

Kevin Slijepcevic

Listings

List of Figures

List of Tables

References

[1] Rust Foundation. "The MIR (mid-level IR) - rust compiler development guide," [Online]. Available: https://rustc-dev-guide.rust-lang.org/mir/index.html (visited on May 29, 2023).

[2] Rust Foundation. "The HIR (high-level IR) - rust compiler development guide," [Online]. Available: https://rustc-dev-guide.rust-lang.org/hir.html (visited on Jun. 14, 2023).

[3] "Stack overflow developer survey 2023," Stack Overflow, [Online]. Available: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023 (visited on Aug. 21, 2023).

[4] Jesse Howarth. "Why discord is switching from go to rust." (Feb. 4, 2020), [Online]. Available: https://discord.com/blog/why-discord-is-switching-from-go-to-rust (visited on May 22, 2023).

[5] "Rust for linux," [Online]. Available: https://rust-for-linux.com (visited on May 22, 2023).

[6] Dana Jansens. "Supporting the use of rust in the chromium project," Google Online Security Blog. (Jan. 12, 2023), [Online]. Available: https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html (visited on May 22, 2023).

[7] "Meet safe and unsafe - the rustonomicon," [Online]. Available: https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html (visited on Aug. 21, 2023).

[8] National Security Agency, "Software memory safety," National Security Agency, Nov. 2022, p. 7. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF (visited on May 23, 2023).

[9] "What unsafe can do - the rustonomicon," [Online]. Available: https://doc.rust-lang.org/nomicon/what-unsafe-does.html (visited on Aug. 21, 2023).

[10] P. Bjesse, "What is formal verification?" *ACM SIGDA Newsletter*, vol. 35, no. 24, p. 1, Dec. 15, 2005, ISSN: 0163-5743. DOI: 10.1145/1113792.1113794. [Online]. Available: https://dl.acm.org/doi/10.1145/1113792.1113794 (visited on Aug. 21, 2023).

[11] Peter Müller. "Rust verification workshop (RW2023)," [Online]. Available: https://sites.google.com/view/rustverify2023/ (visited on May 23, 2023).

[12] Rust Formal Methods Interest Group. "Welcome - rust formal methods interest group," [Online]. Available: https://rust-formal-methods.github.io/ (visited on May 23, 2023).

[13] G. Research. "Rust verification tools." (2021), [Online]. Available: https://project-oak.github.io/rust-verification-tools/ (visited on May 23, 2023).

[14] P. Bjesse, "What is formal verification?" *ACM SIGDA Newsletter*, vol. 35, no. 24, 1–es, Dec. 15, 2005, ISSN: 0163-5743. DOI: 10.1145/1113792.1113794. [Online]. Available: https://doi.org/10.1145/1113792.1113794 (visited on Aug. 25, 2023).

[15] I. Ivanov and M. S. Nikitchenko, "On the sequence rule for the floyd-hoare logic with partial pre-and post-conditions.," in *ICTERI workshops*, Taras Shevchenko National University of Kyiv, 2018, pp. 716–724. [Online]. Available: https://ceur-ws.org/Vol-2104/paper_259.pdf.

[16] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, Feb. 2009, ISSN: 0256-2499, 0973-7677. DOI: 10.1007/s12046-009-0001-5. [Online]. Available: http://link.springer.com/10.1007/s12046-009-0001-5 (visited on Aug. 26, 2023).

[17] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 305–343, ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10575-8_11 (visited on Sep. 6, 2023).

[18] "For and range - rust by example," [Online]. Available: https://doc.rust-lang.org/rust-by-example/flow_control/for.html (visited on Sep. 12, 2023).

[19] V. Astrauskas *et al.*, "The prusti project: Formal verification for rust," in *NASA Formal Methods*, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 88–108, ISBN: 978-3-031-06772-3 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_5. [Online]. Available: https://link.springer.com/10.1007/978-3-031-06773-0_5 (visited on Apr. 3, 2023).

[20] ETH Zurich. "Initial commit. · viperproject/prusti-dev@94b474f." (Dec. 11, 2017), [Online]. Available: https://github.com/viperproject/prusti-dev/commit/94b474f9672a1dfe416e50417b1df24bba55f878 (visited on Jun. 10, 2023).

[21] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–30, OOPSLA Oct. 10, 2019, Number: OOPSLA, ISSN: 2475-1421. DOI: 10.1145/3360573. [Online]. Available: https://dl.acm.org/doi/10.1145/3360573 (visited on May 29, 2023).

[22] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49121-8 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2. [Online]. Available: http://link.springer.com/10.1007/978-3-662-49122-5_2 (visited on Apr. 13, 2023).

[23] Chair of Programming Methodology - ETH Zurich. "Prusti assistant - visual studio marketplace," [Online]. Available: https://marketplace.visualstudio.com/items?itemName=viper-admin.prusti-assistant (visited on May 29, 2023).

[24] Rust Foundation. "Attributes - the rust reference," [Online]. Available: https://doc.rust-lang.org/reference/attributes.html (visited on May 29, 2023).

[25] "Pure functions | scala 3 — book | scala documentation," [Online]. Available: https://docs.scala-lang.org/scala3/book/fp-pure-functions.html (visited on Aug. 31, 2023).

[26] L. F. W. Anthony, "Supporting alternative SMT solvers in viper," Apr. 28, 2022. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Lasse%20F._Wolff_Anthony_PW_Report.pdf (visited on Aug. 12, 2023).

[27] "Boogie: An intermediate verification language - microsoft research," [Online]. Available: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/ (visited on Aug. 31, 2023).

[28] "Viper – programming methodology group | ETH zurich," [Online]. Available: https://www.pm.inf.ethz.ch/research/viper.html (visited on Sep. 4, 2023).

[29] Prusti user guide. "Absence of panics - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/panic.html?highlight=conserv#absence-of-panics (visited on May 31, 2023).

[30] "Counterexamples - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/tour/counterexamples.html (visited on Aug. 31, 2023).

[31] Prusti user guide. "Verification features - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/summary.html (visited on May 31, 2023).

[32] Stef Gijsberts, "Prusti in practice," Radboud University Nijmegen, Bachelor Thesis, Mar. 22, 2023. [Online]. Available: https://www.cs.ru.nl/bachelors-theses/2023/Stef_Gijsberts___1034031___Prusti_in_Practice_-_A_case_study_of_using_the_Prusti_auto-active_program_verifier_for_Rust.pdf (visited on Jun. 5, 2023).

[33] Prusti user guide. "Trusted functions - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/trusted.html (visited on May 31, 2023).

[34] Prusti user guide. "Predicates - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/predicate.html (visited on Jun. 4, 2023).

[35] Prusti user guide. "Pledges - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/pledge.html (visited on Jun. 4, 2023).

[36] Prusti user guide. "External specifications - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/external.html (visited on May 31, 2023).

[37] Prusti user guide. "Closures - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/closure.html (visited on Jun. 2, 2023).

[38] "Closures: Anonymous functions that capture their environment - the rust programming language," [Online]. Available: https://doc.rust-lang.org/book/ch13-01-closures.html (visited on Aug. 31, 2023).

[39] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers, "Modular specification and verification of closures in rust," *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 1–29, OOPSLA Oct. 20, 2021, Number: OOPSLA, ISSN: 2475-1421. DOI: 10.1145/3485522. [Online]. Available: https://dl.acm.org/doi/10.1145/3485522 (visited on Jun. 2, 2023).

[40] Prusti user guide. "Assertions, refutations and assumptions - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/assert_refute_assume.html (visited on Jun. 5, 2023).

[41] Prusti user guide. “Type-conditional spec refinements - prusti user guide,” [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/type_cond_spec.html (visited on Jun. 5, 2023).

[42] P. user guide. “Specification syntax - prusti user guide,” [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/syntax.html (visited on Aug. 19, 2023).

[43] ETH Zurich. “Prusti,” [Online]. Available: https://www.pm.inf.ethz.ch/research/prusti.html (visited on May 29, 2023).

[44] ETH Zurich. “Viperproject/prusti-dev: A static verifier for rust, based on the viper verification infrastructure.,” [Online]. Available: https://github.com/viperproject/prusti-dev (visited on Jun. 5, 2023).

[45] ETH Zurich. “Releases · viperproject/prusti-dev,” [Online]. Available: https://github.com/viperproject/prusti-dev/releases (visited on Jun. 5, 2023).

[46] ETH Zurich. “Issues · viperproject/prusti-dev,” [Online]. Available: https://github.com/viperproject/prusti-dev/issues (visited on Jun. 5, 2023).

[47] J. Schar, “Proving refinement in a rust verifier,” ETH Zurich, Master Thesis, May 10, 2023. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Jan_Schaer_MS_Thesis.pdf.

[48] Peter Müller. “Rust verification workshop (RW2021),” [Online]. Available: https://sites.google.com/view/rustverify2021/home (visited on Jun. 5, 2023).

[49] *Prusti – deductive verification for rust*, in collab. with Alex Summers, May 11, 2021. [Online]. Available: https://www.youtube.com/watch?v=C9TTioH5JUg (visited on Jun. 5, 2023).

[50] Rust Formal Methods Interest Group. “Tools - rust formal methods interest group.” (2021), [Online]. Available: https://rust-formal-methods.github.io/tools.html (visited on Jun. 19, 2023).

[51] Prusti user guide. “Introduction - prusti user guide,” [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/intro.html (visited on Jun. 5, 2023).

[52] ETH Zurich. “Prusti-dev/prusti-tests at master · viperproject/prusti-dev · GitHub,” [Online]. Available: https://github.com/viperproject/prusti-dev/tree/master/prusti-tests (visited on Jun. 6, 2023).

[53] “Ex_u32_checked_mul in vstd::std_specs::num - rust,” [Online]. Available: https://verus-lang.github.io/verus/verusdoc/vstd/std_specs/num/fn.ex_u32_checked_mul.html (visited on Sep. 6, 2023).

[54] Prusti user guide. "Final code - prusti user guide," [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/tour/final.html (visited on Sep. 17, 2023).

[55] Xavier Denis. "Initial commit · xldenis/creusot@d214a25." (Oct. 22, 2020), [Online]. Available: https://github.com/xldenis/creusot/commit/d214a2597e4937594a2be7c378c7618c11c55745 (visited on Jun. 10, 2023).

[56] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *Formal Methods and Software Engineering*, A. Riesco and M. Zhang, Eds., vol. 13478, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 90–105, ISBN: 978-3-031-17243-4 978-3-031-17244-1. DOI: 10.1007/978-3-031-17244-1_6. [Online]. Available: https://link.springer.com/10.1007/978-3-031-17244-1_6 (visited on Jun. 6, 2023).

[57] Y. Matsushita, T. Tsukada, and N. Kobayashi, "RustHorn: CHC-based verification for rust programs," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 4, 15:1–15:54, Oct. 31, 2021, Number: 4, ISSN: 0164-0925. DOI: 10.1145/3462205. [Online]. Available: https://dl.acm.org/doi/10.1145/3462205 (visited on Jun. 7, 2023).

[58] Xavier Denis. "Xldenis/creusot: Deductive verification of rust code. (semi) automatically prove your code satisfies your specifications!" [Online]. Available: https://github.com/xldenis/creusot (visited on Aug. 31, 2023).

[59] Xavier Denis. "Xldenis/whycode," [Online]. Available: https://github.com/xldenis/whycode (visited on Jun. 7, 2023).

[60] The Why3 Development Team. "7. other input formats — why3 1.6.0 documentation," [Online]. Available: https://why3.lri.fr/doc/input_formats.html#mlcfg (visited on Jun. 9, 2023).

[61] Why3. "Why3 - external provers," [Online]. Available: https://why3.lri.fr/#provers (visited on Jun. 9, 2023).

[62] X. Denis, J.-H. Jourdan, and C. Marché, "The CREUSOT environment for the deductive verification of rust programs," Inria Saclay - Île de France, Research Report RR-9448, Dec. 2021, Issue: RR-9448. [Online]. Available: https://inria.hal.science/hal-03526634.

[63] Johannes Hayeß, "Verifying the rust runtime of lingua franca," Technische Universität Dresden, Dresden, Master Thesis, Mar. 6, 2023. [Online]. Available: https://grk2767.tu-dresden.de/files/Images/people/chair-cc/theses/2303_Hayess_MA.pdf (visited on Jun. 6, 2023).

[64] Xavier Denis. "Issues · xldenis/creusot," [Online]. Available: https://github.com/xldenis/creusot/issues (visited on Jun. 10, 2023).

[65] S. H. Skotåm, "CreuSAT, using rust and creusot to create the world's fastest deductively verified SAT solver," University of Oslo, Master Thesis, 2022. [Online]. Available: https://www.duo.uio.no/handle/10852/96757.

[66] Peter Müller. "Rust verification workshop (RW2022)," [Online]. Available: https://sites.google.com/view/rustverify2022/home (visited on Jun. 10, 2023).

[67] Rust, *Creusot: A prototype tool for verification of rust software*, May 11, 2021. [Online]. Available: https://www.youtube.com/watch?v=b8sBtmzq0FM (visited on Jun. 10, 2023).

[68] Rust Formal Methods IG, *February session – creusot*, Mar. 1, 2022. [Online]. Available: https://www.youtube.com/watch?v=JP8Q8GYJzb8 (visited on Jun. 26, 2023).

[69] Xavier Denis. "Creusot/creusot/tests at master · xldenis/creusot · GitHub," [Online]. Available: https://github.com/xldenis/creusot/tree/master/creusot/tests (visited on Jun. 10, 2023).

[70] A. Lattuada *et al.*, *Verus: Verifying rust programs using linear ghost types (extended version)*, Issue: arXiv:2303.05491, Mar. 10, 2023. arXiv: 2303.05491[cs]. [Online]. Available: http://arxiv.org/abs/2303.05491 (visited on May 12, 2023).

[71] verus-lang. "Goals · verus-lang/verus wiki." (Mar. 2, 2023), [Online]. Available: https://github.com/verus-lang/verus/wiki/Goals (visited on Jun. 13, 2023).

[72] verus-lang. "Add verify directory and some initial verifier code for handling basic · verus-lang/verus@eaa340b." (May 21, 2021), [Online]. Available: https://github.com/verus-lang/verus/commit/eaa340bced222cb79ed55b5b7150ad6ded64bba0 (visited on Jun. 13, 2023).

[73] verus-lang. "Getting started - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/getting_started.html (visited on Jun. 14, 2023).

[74] Chris Hawblitzel. "Doc: Deprecated and recommended syntax, and upcoming changes · verus-lang/verus wiki." (Mar. 19, 2023), [Online]. Available: https://github.com/verus-lang/verus/wiki/Doc%3A-Deprecated-and-recommended-syntax%2C-and-upcoming-changes (visited on Jun. 13, 2023).

[75] Chris Hawblitzel. "Verus/source/CODE.md at main · verus-lang/verus · GitHub." (Nov. 11, 2021), [Online]. Available: https://github.com/verus-lang/verus/blob/main/source/CODE.md (visited on Jun. 14, 2023).

[76] Andrea Lattuada. "Port to rustc 1.68.0 and drop dependency on compiler fork by utaal · pull request #438 · verus-lang/verus." (Mar. 9, 2023), [Online]. Available: https://github.com/verus-lang/verus/pull/438 (visited on Jun. 14, 2023).

[77] Rust Analyzer. "User manual," [Online]. Available: https://rust-analyzer.github.io/manual.html (visited on Jun. 14, 2023).

[78] A. Lattuada *et al.*, *Verus: Verifying rust programs using linear ghost types – supplementary material*, Mar. 10, 2023. DOI: 10.5281/zenodo.7718498. [Online]. Available: https://zenodo.org/record/7718498 (visited on Jun. 17, 2023).

[79] Learning Rust With Entirely Too Many Linked Lists. "Final code - learning rust with entirely too many linked lists," [Online]. Available: https://rust-unofficial.github.io/too-many-lists/fourth-final.html (visited on Jun. 17, 2023).

[80] J. Yanovski, H.-H. Dang, R. Jung, and D. Dreyer, "GhostCell: Separating permissions from data in rust," *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 1–30, ICFP Aug. 22, 2021, ISSN: 2475-1421. DOI: 10.1145/3473597. [Online]. Available: https://dl.acm.org/doi/10.1145/3473597 (visited on Sep. 2, 2023).

[81] verus-lang. "Assert, requires, ensures, ghost code - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/requires_ensures.html?highlight=assume#assert-and-assume (visited on Jun. 17, 2023).

[82] verus-lang. "Quantifiers and spec closures - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/quants.html (visited on Jun. 17, 2023).

[83] verus-lang. "Loops and invariants - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/while.html (visited on Jun. 17, 2023).

[84] verus-lang. "Integers and arithmetic - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/integers.html (visited on Jun. 17, 2023).

[85] verus-lang. "Verus-lang/verus: Verified rust for low-level systems code," [Online]. Available: https://github.com/verus-lang/verus (visited on Jun. 11, 2023).

[86] verus-lang. "Issues · verus-lang/verus," [Online]. Available: https://github.com/verus-lang/verus/issues (visited on Jun. 18, 2023).

[87] M. Brun, "Verified paging for x86-64 in rust," ETH Zurich, Master Thesis, Oct. 2, 2022. [Online]. Available: https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/594366/1/Brun_Matthias.pdf (visited on Jun. 18, 2023).

[88] Rust Formal Methods IG, *Verus – SMT-based verification of rust systems code*, Mar. 2, 2023. [Online]. Available: https://www.youtube.com/watch?v=7WtWA0TTBqg (visited on Jun. 26, 2023).

[89] Rust, *Verus - verified rust for low-level systems code by andrea lattuada - rust zürisee june 2023*, Jun. 26, 2023. [Online]. Available: https://www.youtube.com/watch?v=ZZTk-zS4ZCY (visited on Sep. 3, 2023).

[90] verus-lang. "Verus overview - verus tutorial and reference," [Online]. Available: https://verus-lang.github.io/verus/guide/overview.html (visited on Jun. 19, 2023).

[91] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," *Proceedings of the ACM on Programming Languages*, vol. 6, pp. 711–741, ICFP Aug. 29, 2022, Number: ICFP, ISSN: 2475-1421. DOI: 10.1145/3547647. arXiv: 2206.07185[cs]. [Online]. Available: http://arxiv.org/abs/2206.07185 (visited on Jun. 11, 2023).

[92] AeneasVerif. "Initial commit · AeneasVerif/aeneas@09bfbde." (Nov. 2, 2021), [Online]. Available: https://github.com/AeneasVerif/aeneas/commit/09bfbde5ebdc60cef109b570b6c6f567a5b516b0 (visited on Jun. 22, 2023).

[93] AeneasVerif. "AeneasVerif/charon: Interface with the rustc compiler for the purpose of program verification," [Online]. Available: https://github.com/AeneasVerif/charon (visited on Jun. 22, 2023).

[94] AeneasVerif. "AeneasVerif/aeneas: A verification toolchain for rust programs," [Online]. Available: https://github.com/AeneasVerif/aeneas (visited on Jun. 22, 2023).

[95] Polonius. "What is polonius? - polonius," [Online]. Available: https://rust-lang.github.io/polonius/ (visited on Jun. 26, 2023).

[96] AeneasVerif. "Issues · AeneasVerif/aeneas," [Online]. Available: https://github.com/AeneasVerif/aeneas/issues (visited on Jun. 26, 2023).

[97] AeneasVerif. "Issues · AeneasVerif/charon," [Online]. Available: https://github.com/AeneasVerif/charon/issues (visited on Jun. 26, 2023).

[98] Rust Formal Methods IG, *Aeneas: Aeneas verification by functional translation*, Nov. 7, 2022. [Online]. Available: https://www.youtube.com/watch?v=9j9EE36lJJI (visited on Jun. 26, 2023).

[99] facebookexperimental. "MIRAI/documentation/overview.md at main · facebookexperimental/MIRAI · GitHub." (Dec. 23, 2021), [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/Overview.md (visited on Jun. 30, 2023).

[100] facebookexperimental. "Facebookexperimental/MIRAI: Rust mid-level IR abstract interpreter," [Online]. Available: https://github.com/facebookexperimental/MIRAI/tree/main (visited on Jun. 30, 2023).

[101] facebookexperimental. "First commit · facebookexperimental/MIRAI@8afab5d." (Nov. 6, 2018), [Online]. Available: https://github.com/facebookexperimental/MIRAI/commit/8afab5db34fb7330789e3765c3adf460ed72ed05 (visited on Jun. 30, 2023).

[102] facebookexperimental. "MIRAI/documentation/CallGraph.md at main · facebookexperimental/MIRAI · GitHub." (Jan. 24, 2022), [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/CallGraph.md (visited on Jul. 1, 2023).

[103] facebookexperimental. "MIRAI/documentation/WhyPlugIn.md at main · facebookexperimental/MIRAI · GitHub." (Dec. 24, 2021), [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/WhyPlugIn.md (visited on Jun. 30, 2023).

[104] "MIRAI/documentation/IncrementalAnalysis.md at main · facebookexperimental/MIRAI," [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/IncrementalAnalysis.md (visited on Sep. 15, 2023).

[105] karroffel. "Contracts - crates.io: Rust package registry," [Online]. Available: https://crates.io/crates/contracts (visited on Jun. 30, 2023).

[106] Herman Venter. "Mirai-annotations - crates.io: Rust package registry," [Online]. Available: https://crates.io/crates/mirai-annotations (visited on Jun. 30, 2023).

[107] facebookexperimental. "MIRAI/documentation/TagAnalysis.md at main · facebookexperimental/MIRAI · GitHub." (Sep. 18, 2021), [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/TagAnalysis.md (visited on Jul. 1, 2023).

[108] "MIRAI/checker/tests at main · facebookexperimental/MIRAI," [Online]. Available: https://github.com/facebookexperimental/MIRAI/tree/main/checker/tests (visited on Sep. 15, 2023).

[109] Graphviz. "Graphviz," [Online]. Available: https://graphviz.org/ (visited on Jul. 4, 2023).

[110] diem. "Diem/diem: Diem's mission is to build a trusted and innovative financial network that empowers people and businesses around the world.," [Online]. Available: https://github.com/diem/diem (visited on Jun. 30, 2023).

[111] "Initial commit · diem/diem@5e034dd," [Online]. Available: https://github.com/diem/diem/commit/5e034dde19a5320d7e2bdc9da25114e816b4454d (visited on Sep. 15, 2023).

[112] facebookexperimental. "Release version 1.1.8 · facebookexperimental/MIRAI." (May 9, 2023), [Online]. Available: https://github.com/facebookexperimental/MIRAI/releases/tag/v1.1.8 (visited on Jun. 30, 2023).

[113] facebookexperimental. "Issues · facebookexperimental/MIRAI," [Online]. Available: https://github.com/facebookexperimental/MIRAI/issues (visited on Jul. 5, 2023).

[114] Rust Formal Methods IG, *January session – MIRAI*, Jan. 31, 2022. [Online]. Available: https://www.youtube.com/watch?v=Slf1QWaRe2c (visited on Jul. 1, 2023).

[115] facebookexperimental. "MIRAI/documentation/decisions.md at main · facebookexperimental/MIRAI · GitHub." (Dec. 24, 2022), [Online]. Available: https://github.com/facebookexperimental/MIRAI/blob/main/documentation/Decisions.md (visited on Jul. 1, 2023).

[116] facebookexperimental. "MIRAI/examples at main · facebookexperimental/MIRAI · GitHub." (Jan. 8, 2023), [Online]. Available: https://github.com/facebookexperimental/MIRAI/tree/main/examples (visited on Jul. 1, 2023).

[117] "Rust-lang/project-stable-mir: Define compiler intermediate representation usable by external tools," [Online]. Available: https://github.com/rust-lang/project-stable-mir (visited on Sep. 3, 2023).

[118] "List.why," [Online]. Available: https://why3.lri.fr/stdlib-0.81/list.why.html (visited on Aug. 29, 2023).

[119] W. Min, "Analysis on bubble sort algorithm optimization," in *2010 International Forum on Information Technology and Applications*, vol. 1, Jul. 2010, pp. 208–211. DOI: 10.1109/IFITA.2010.9.

[120] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2022, ISBN: 0-262-36750-5.

[121] R. Wagner and M. Fredrikson, "Incremental proof development in dafny," [Online]. Available: https://www.cs.cmu.edu/~mfredrik/15414/lectures/17-notes.pdf (visited on Aug. 1, 2023).

[122] "Vec in std::vec - rust," [Online]. Available: https://doc.rust-lang.org/std/vec/struct.Vec.html (visited on Aug. 10, 2023).

[123] "Usize - rust," [Online]. Available: https://doc.rust-lang.org/std/primitive.usize.html (visited on Aug. 10, 2023).

[124] "Slice - rust," [Online]. Available: https://doc.rust-lang.org/std/primitive.slice.html (visited on Aug. 16, 2023).

[125] Andrea Höfler, "SMT solver comparison," Diploma, Graz University of Technology, Graz, Austria, Jul. 2014. [Online]. Available: https://spreadsheets.ist.tugraz.at/wp-content/uploads/sites/3/2015/06/DS_Hoefler.pdf (visited on Aug. 20, 2023).