



The Prusti Project: Formal Verification for Rust

Vytautas Astrauskas¹, Aurea Bílá¹, Jonáš Fiala¹, Zachary Grannan²,
Christoph Matheja³, Peter Müller¹, Federico Poli¹,
and Alexander J. Summers²(✉)

¹ Department of Computer Science, ETH Zurich, Zurich, Switzerland
`aurea.bily@inf.ethz.ch`

² University of British Columbia, Vancouver, Canada
`alex.summers@ubc.ca`

³ Technical University of Denmark, Kongens Lyngby, Denmark

Abstract. Rust is a modern systems programming language designed to offer both performance and static safety. A key distinguishing feature is a strong type system, which enforces by default that memory is either shared or mutable, but never both. This guarantee is used to prevent common pitfalls such as memory errors and data races. It can also be used to greatly simplify formal verification, as we demonstrated by developing the Prusti verifier, which can verify rich correctness properties of Rust programs with a very modest annotation overhead. In this paper, we provide an overview of the Prusti project. We outline its main design goals, illustrate examples of its use, and discuss important outcomes from the perspectives of a user, a verification expert, and a tool developer.

Keywords: Rust · Deductive verification · Separation logic

1 Introduction

Systems programming languages have traditionally had one dominating design goal: performance. To achieve this goal they give programmers maximum freedom in organising their code and data. They allow unrestricted aliasing and freely bypassing the safety checks of the language, for instance through unchecked type casts. This freedom enables the development of highly efficient programs, but also makes it all too easy to introduce errors and vulnerabilities, such as buffer overflows, memory errors, data races, and subtle functionality bugs.

Rust is a modern systems programming language that is built on a different premise: it is designed to maximise *both* performance and static safety. Rust employs a strong type system that prevents many common errors at compile time. In particular, it eradicates memory errors (e.g. accessing uninitialised or freed memory), various sources of program crashes (e.g. null-dereferencing), and data races. In cases where the type system is too restrictive, programmers can escape into unsafe Rust, which permits direct pointer manipulation like in traditional systems programming languages. However, according to Rust’s design philosophy [25, 32], unsafe operations are typically confined to libraries and encapsulated behind safe abstractions, while client code is written in safe Rust [5, 30].

This design makes Rust a promising target for program verification. Not only does Rust’s type system prevent certain errors, such that verification need not deal with them, but it also provides strong compiler-enforced restrictions on aliasing and mutable state, which can be leveraged to simplify verification. There is also an important social motivation: Rust is often chosen for projects with high safety and security requirements, whose members are likely open to program verification as an additional means of achieving these requirements.

To explore this opportunity, we started the Prusti project in 2017. Prusti [6] is a general-purpose deductive verifier for Rust. We had three key design goals:

1. *Enable the verification of expressive program properties.* These go beyond the absence of exceptions (called *panics* in Rust, e.g. due to overflows or out-of-bounds accesses) to include invariants of data types, and more-general functional correctness properties. We initially focused on safe Rust code, but a designated goal of the Prusti project has been to generate self-contained proofs that are valid independently of the guarantees of safe Rust. For the properties guaranteed by safe Rust, this so-called *core proof* is redundant (assuming Rust’s type system is sound), but it forms a reusable basis for layering correctness arguments for more complex properties on top, and (eventually) extending verification to common usages of unsafe code.
2. *Reduce the annotation burden for programmers by leveraging Rust’s design.* Prusti addresses this goal along two dimensions. First, it reduces the *complexity* of annotations. Safe Rust’s restrictions on aliasing and mutations allow Prusti to use annotations based on Rust expressions, without the need to expose programmers to non-trivial logics such as separation logic [39,43]. The resulting annotations are similar to classical contracts [35], but enable sound, modular verification of heap-manipulating programs. Second, Prusti reduces the *amount* of necessary annotations. Mainstream verification techniques such as separation logic or dynamic frames [29] require a large upfront investment to declare and manipulate predicates and ghost state that describe the shape of data structures, and to prove memory safety as the basis for more advanced properties. In contrast, Prusti extracts this information automatically from Rust’s type system, allowing programmers to focus immediately on the functional properties they care about.
3. *Integrate smoothly into the workflow of Rust programmers.* Integrating verification tools into development workflows is widely regarded as a major obstacle for their adoption [18]. Prusti simplifies integration in two ways: First, since Prusti requires no upfront investment, it enables a workflow where programmers can incrementally write more annotations to obtain stronger guarantees. It offers a mode that does not check panic freedom, such that it can be run on unannotated Rust programs. Panic freedom can generally be proved by adding a small (often zero) number of simple annotations (e.g. function preconditions); richer properties can be expressed and proved by adding postconditions and invariants. Second, Prusti integrates smoothly into the compiler infrastructure. It operates on the same representations of programs that the Rust compiler uses. This avoids discrepancies with the compiler (which, in the absence of a

formal language specification serves as a working definition) and makes sure the verifier does not drift out of sync as the Rust language and compiler evolve. It also gives a unified view on potential errors: verification issues are reported in the same way as compilation errors.

In this paper, we give an overview of the Prusti verifier and discuss the central design decisions and relevant outcomes so far from the perspective of a user (Sect. 2), a verification expert (Sect. 3), and a tool developer (Sect. 4). We discuss related work (Sect. 5) and conclude with some directions for future work (Sect. 6).

2 Prusti from a User’s Perspective

We first consider the Prusti verifier from a Rust programmer’s perspective. Prusti builds upon the standard Rust compiler `rustc`. The command `prusti-rustc` can be used as a drop-in replacement for `rustc` to verify individual files; the command `cargo prusti` uses Rust’s package manager `cargo` to run Prusti on Rust projects. Alternatively, Prusti can be used through an extension for Visual Studio Code (VSCode), which is a popular editor for Rust programming [49].

A key feature of Prusti is that it supports incremental verification with an initial annotation effort of (almost) *zero*: developers get guarantees beyond those of safe Rust and useful feedback by just running Prusti on their code; they can then choose to invest more effort to obtain more powerful guarantees. We will illustrate Prusti’s capabilities by proving increasingly complex properties for safe Rust programs. Further details and examples are available online [47].

2.1 (Almost) Zero-Cost Verification

By default, Prusti checks that a Rust program will not *panic* (terminate with an unrecoverable error) at runtime, whether due to an explicit `panic!(...)` call¹ or e.g. due to bounds-checks and integer overflows. Prusti can perform these checks directly on the input program, with *no* modification and *no* user-supplied annotations; in particular, it does not require the specifications of data structures and side-effects required as upfront investment by verification techniques for other imperative languages. For many examples, the checks for panic freedom succeed immediately; others require a small amount of simple annotations. In the following, we present examples for both cases.

As a first example, consider the Rust function in Fig. 1, which performs a binary search for a value `key` on a slice of integers `a`, i.e. a contiguous subsequence of the elements in a collection. Compiling this function with `rustc` produces no errors. However, running `prusti-rustc` reveals a potential bug: the statement `let mid = (low+high)/2` on line 7 might overflow for a very large slice `a`. This automatically detected bug is non-trivial: it remained undetected for years in a similar implementation provided by the Java standard library [9].

¹ or its siblings `unreachable!()`, `unimplemented!()`, `assert!(false)`, etc.

```

1 fn search(a: &[i32], key: i32)
2     -> Option<usize> {
3     let mut low = 0;
4     let mut high = a.len();
5     while low < high {
6         // Addition may overflow
7         let mid = (low+high) / 2;
8         // Bound check at runtime
9         let mid_val = a[mid];
10        if mid_val < key {
11            low = mid + 1;
12        } else if mid_val > key {
13            high = mid;
14        } else {
15            return Some(mid);
16        }
17    }
18    return None;
19 }

```

Fig. 1. Buggy binary search.

```

> prusti-rustc search.rs
error: [Prusti: verification error]
assertion might fail with attempt to
add with overflow
--> search.rs:7:5
|
7 |         let mid = (low+high) / 2;
|         ~~~~~
5     while low < high {
6         body_invariant!(high <= a.len());
7         let mid = low + ((high-low) / 2);
8         assert!(mid < high);
9         let mid_val = a[mid];
// ...
17    }

```

Fig. 2. Reported error and fixed loop.

Whenever Prusti fails to verify the absence of panics, it reports potential issues like compiler errors, as in Fig. 2 (upper half); these naturally benefit from any IDE highlighting of errors. Programmers can understand and handle such warnings as if Prusti were a stricter compiler for Rust.

We can fix the bug by rewriting line 7 to `let mid = low + ((high-low)/2)`. Now Prusti is able to infer both that `high-low` cannot underflow (from the loop guard: `low < high`) and that `low + ((high-low)/2)` cannot overflow.

While this property can be proved without any help from the programmer, others require annotations. Prusti verifies loops according to the guarantees of the Rust type system and any user-provided loop invariants. After fixing the overflow error in our example, Prusti cannot show that, in every loop iteration, the slice access `a[mid]` (line 9) is within bounds. To establish this it suffices to add a simple loop invariant² stating that, during every iteration, `high <= a.len()` holds just inside the loop body. The annotated code accepted by Prusti is shown in Fig. 2 (lower half). Prusti proves that the loop invariant holds (inductively); the invariant, along with the loop guard `mid < high` and the (implicit) unsigned types of these index variables, allows Prusti to prove that `a[mid]` is safe).

This simplest way of using Prusti requires almost no user annotation: Prusti’s underlying reasoning accounts for path conditions, value ranges and (not shown here) non-aliasing guarantees implied by `rustc`’s type-checking. Additional local properties of interest can be added with standard Rust `assert` macros (e.g. line 8 in Fig. 2), and checked statically with Prusti rather than (only) at runtime; the initial friction in using Prusti this way is as low as for using a code linter.

² In slight contrast to classical loop invariants, a `body_invariant!(...)` need only hold for every loop iteration *reaching* this location inside the loop body.

2.2 Modular Verification of User-Specified Contracts

After using Prusti for proving panic freedom, developers may decide to invest annotation effort step-by-step to obtain stronger correctness guarantees about their Rust code. To this end, every function can be annotated with a *contract*: a specification consisting of pre- and postconditions. Functions are verified modularly against these contracts: when verifying calls to the function, only its contract and type signature are used, not its concrete implementation. Besides facilitating scalability and supporting recursion, a modular approach enables decoupling verification of client code from e.g. specific library implementations.

Continuing our example from Fig. 1, consider the following contract:

```

1 #[requires(a.len() < usize::MAX / 2)]
2 #[ensures(if let Some(idx) = result { idx < a.len() && a[idx] == key }
3         else { true })]
4 fn search(a: &[i32], key: i32) -> Option<usize> { /* ... */ }
```

Specifications in Prusti consist of (a large subset of) side-effect free Rust expressions with a few carefully chosen extensions, as we discuss below. The above postcondition `ensures(...)` uses the special Prusti variable `result` to refer to the function’s return value³. It specifies that whenever `search` returns some position `idx`, then the value `a[idx]` equals the search `key`. Prusti checks this property and also that the slice access `a[idx]` in the postcondition is in bounds.

The precondition `requires(...)` states that `search` can be called only on slices whose length is at most half of the largest number of type `usize`—Prusti will report an error if a caller attempts to pass a longer slice. Under this precondition, the original overflow bug could never be triggered, and Prusti can also verify the *unmodified* code from Fig. 1 (for calls allowed by the precondition).

2.3 The Prusti Specification Language

We will now explain and illustrate numerous features of Prusti’s specification language via its usage on a binary search tree (BST), given by:

```

1 // A binary search tree data structure (elements should be sorted)
2 pub enum Tree<T: Ord> {
3     Node(T, Box<Tree<T>>, Box<Tree<T>>),
4     Empty,
5 }
```

Every element of a `Tree` is either an `Empty` leaf or a `Node` storing pointers to its left and right subtree, and a value of (generic) type `T`; the bound on `T` requires that this type must implement the `Ord` trait so that values can be compared. We assume that this BST represents a set, i.e. duplicate entries will never be stored.

Prusti’s specification syntax (e.g. for pre- and postconditions) reuses Rust expressions as far as possible. Not *all* Rust expressions are accepted: the evaluation of expressions used in specifications must not have side-effects (specifications

³ The `if let` construct is standard Rust, branching on whether the value can be pattern-matched against `Some(idx)` (taking the second branch if not, i.e. for `None`).

should not affect program execution), and must be deterministic and terminate, to ensure that specifications have an intuitive meaning for programmers (a clear mathematical interpretation for the verifier). Prusti identifies a *pure subset* of Rust with the above properties allowed in specifications, including dereferencing, branching, pattern-matching etc., as used in our `search` postconditions above.

Importantly, Prusti allows calls to *functions* within specifications, *if* they have the Prusti-specific attribute `#[pure]`. The body of a function labelled as pure must fall into Prusti’s pure Rust fragment described above. As of now, Prusti checks that pure functions have no side-effects and are deterministic (termination checking is not yet performed, but will be added in the near future).

A common case of pure functions are queries (or *getters*) of a data structure, such as the `contains` function below, which often appear in specifications.

```
1 impl<T: Ord> Tree<T> {
2   #[pure]
3   pub fn contains(&self, find_value: &T) -> bool {
4     // ... with the natural (recursive) definition in Rust ...
```

This function is implemented as a straightforward recursive traversal over the BST [48], naturally satisfying the requirements for a pure function⁴. As `contains` is declared pure, Prusti will treat it analogously to a mathematical function and unroll its definition (in a bounded way, to avoid non-termination) instead of relying solely on the function’s contract (as for ordinary methods). Annotating the function as pure suffices for proving simple code such as the following:

```
1 let v = 0;
2 let t = Tree::Node(v, Box::new(Tree::Empty), Box::new(Tree::Empty));
3 assert!(t.contains(&v));
```

While it is reassuring that such unit-test-like programs can be statically verified automatically, the real power of pure functions is that they provide API-specific building blocks for defining richer functional specifications, as we show next.

Type Invariants. Our next goal is to specify that `Tree` objects maintain a fundamental invariant, namely that they model binary search trees. Assume, for the moment, that we already have a specification of the search tree property given by a pure method `bst_invariant(&self) -> bool`. Prusti’s `#[invariant(...)]` annotation then allows us to directly attach the invariant to the `Tree` type:

```
1 #[invariant(self.bst_invariant())]
2 pub enum Tree<T: Ord> {
```

Now, Prusti will ensure that whenever a `Tree` instance is passed as function argument or return value, the invariant is guaranteed; it is correspondingly assumed for function parameters (by the callee) and return values (by the caller).

⁴ Values of generic type `T` are compared with the library function `cmp` from trait `Ord`, which is specified to satisfy the standard properties of total orders using an *external specification*; this Prusti feature is explained in Sect. 2.4.

```

1 predicate! {
2   pub fn bst_invariant(&self) -> bool {
3     if let Tree::Node(value, left, right) = self {
4       forall(|i: &T| left.contains(i) ==
5         (matches!(i.cmp(value), Less) && self.contains(i)))
6       && forall(|i: &T| right.contains(i) ==
7         (matches!(i.cmp(value), Greater) && self.contains(i)))
8     } else { true }
9   }
10 }

```

Fig. 3. Predicate expressing the invariant of a binary search tree.

Quantifiers and Predicates. Our invariant `bst_invariant` needs to capture the following informal search tree property: any value v of type T in the left (resp. right) subtree of a BST instance t with root value v' is smaller (resp. greater) than v' according to T 's ordering. Rather than implementing this property as a pure function in Rust, the above description suggests *quantifying* over all values. Prusti specifications may contain both universal (syntax: `forall(|vars| expr)`) and existential (syntax: `exists(|vars| expr)`) quantifiers, where the declaration of quantified variables `vars` is analogous to declaring Rust closure parameters.

We can now precisely define our intended invariant with this powerful mix of logical quantifiers and pure functions denoting data-structure-specific abstractions. However, since quantifiers are not Rust expressions, the invariant itself cannot be defined in a Rust function. Instead, Prusti provides the feature of *predicates*, which are similar to (pure) Rust functions whose bodies can be any expression allowed in Prusti's specification language. Our formal Prusti specification of the invariant is shown in Fig. 3. Prusti checks that predicates are only ever invoked in specifications; they cannot be called from executable code (general quantifiers need not have an executable semantics).

Old Expressions. Now that we have established the search tree property as an invariant of `Tree`, we may decide to add further contracts to functions working with trees. For instance, Fig. 4 shows a method `insert` that inserts a new value into a binary search tree; it is equipped with a simple postcondition (line 1) stating that, once the function terminates, the tree contains the new value. Since `insert` mutates the given tree, we may also want to make sure that, apart from adding the new value, no other values have been added or removed. Prusti specifications can include `old(...)` expressions in postconditions to refer to the memory before execution of the function's body. As shown in lines 2–3, we can then specify that, for all values except the new one, the function `contains` returns the same result when executed on the tree before and after running `insert`.

Pledges. One of the most advanced specification features Prusti adds to its base language of Rust expressions tackles specification of *reborrowing*: func-

```

1  #[ensures(self.contains(&new_value))]
2  #[ensures(forall(|i: &T| !matches!(new_value.cmp(i), Equal)
3      ==> self.contains(i) == old(self).contains(i)))]
4  pub fn insert(&mut self, new_value: T) {
5      if let Tree::Node(value, left, right) = self {
6          match new_value.cmp(value) {
7              Equal => (),
8              Less => left.insert(new_value),
9              Greater => right.insert(new_value),
10         }
11     } else {
12         *self = Tree::Node(new_value,
13             Box::new(Tree::Empty), Box::new(Tree::Empty))
14     }
15 }

```

Fig. 4. Insertion into a binary search tree.

tions that both take and return mutable references. An example is the function `get_root_value` below, which hands out a reference to the root value of the tree.

```

1  pub fn get_root_value(&mut self) -> &mut T {
2      if let Tree::Node(value, _, _) = self { value } else { panic!() }
3  }

```

Rust’s type system (generally forbidding the combination of usable aliases and mutability) makes the reference `self` *blocked* after calling this function, until the returned reference’s lifetime expires (it is no longer used). This creates an interesting challenge if (as we did for Prusti) one wants a specification language which is in-keeping with both Rust expression syntax and its typing rules, to aid programmer understanding. The key challenges [6] are: (1) one wants to specify guarantees that will be true for `self` *once it becomes accessible again*, but in the post-state of the call one cannot (according to the type system) talk about the blocked reference to `self`, and (2) *some* facts that one cares about cannot even be determined in the post-state of this call, since the value that the root will have when the reborrow expires is *not yet known*: it depends on what the caller does with the returned reference to this root value.

Prusti solves both problems with *pledges* [6], a novel specification feature which allows one to express specifications about points in the *future* of this call, when the returned reborrow expires. Pledges use two specification constructs: `after_expiry(e)` (which describes what `e`’s value *will be* once the returned reference expires), and `before_expiry(e)` (which describes `e`’s value *just before* the returned reference expires). Using these constructs, one can write e.g. a postcondition `after_expiry(self.contains(before_expiry(result)))`, to express that once the returned reference `result` expires, the BST `self` is guaranteed to contain whatever value `result` stores by the time it expires. More examples are discussed in our earlier paper [6].

```

1  #[requires(matches!(self, Tree::Node(..)))]
2  #[assert_on_expiry(
3      // Must hold before result can expire
4      if let Tree::Node(_, left, right) = old(self) {
5          forall(|i: &T| left.contains(i)
6              ==> matches!(i.cmp(result), Less)) &&
7          forall(|i: &T| right.contains(i)
8              ==> matches!(i.cmp(result), Greater))
9      } else { false },
10     // A postcondition of 'get_root_value' after result expires
11     if let Tree::Node(ref value, _, _) = self {
12         matches!(value.cmp(before_expiry(result)), Equal)
13     } else { false }
14 )]
15 pub fn get_root_value(&mut self) -> &mut T {
16     if let Tree::Node(value, _, _) = self { value } else { panic!() }
17 }

```

Fig. 5. A rich specification combining many Prusti and Rust features.

Given our desired BST invariant, client code should modify the root’s value only in a way that guarantees to preserve the BST invariant. This can be enforced with the more-advanced pledge construct `assert_on_expiry(e’,e)`. This construct expresses `after_expiry(e)` and, in addition, asserts `e’` at the point where the reborrowed reference expires. To demonstrate the expressiveness of these features combined, we show a very general specification for `get_root_value` in Fig. 5, which exploits the power of Prusti’s specifications to combine pledges, old expressions, pure functions, quantifiers along with standard Rust features. Notably, the constraint on the reborrowed reference `result` relates only its (single) value to the old version of the tree: the rest of the tree structure is guaranteed immutable while the reborrow is live, and Prusti’s underlying separation logic proof (discussed in the next section) captures this directly.

2.4 Incremental Verification in Practice

As illustrated above, Prusti’s design enables developers to verify a codebase by incrementally trading annotation effort for stronger guarantees. In this subsection, we report on preliminary experiences from an ongoing project in which Prusti is used this way to analyse the `ibc` [21] crate, an implementation of the Interblockchain Communication Protocol [19] containing >20,000 lines of code.

At the time of our first experiments, Prusti could run on roughly 70% of the functions in the two crates (495/716 and 545/738) analysed; the remainder used features unsupported by the verifier. Without specifications the vast majority of these functions were proved panic-free automatically. Prusti identified a small number of potential panics, due to manual `assert!` calls (conceptually expressing preconditions) or potential overflows due to expressions such as

`self.revision_height + delta` where `delta` was a `u64` function parameter. Making manual `assert!`s into preconditions (which are then checked at call sites!) is easy since Prusti’s specifications can be Rust expressions; adding preconditions to rule out overflows was also simple, e.g. this precondition for the case above:

```
1 #[requires(u64::MAX - self.revision_height >= delta)]
2 pub fn add(&self, delta: u64) -> Height { /* ... */ }
```

This ruled out language-level panics for all supported functions, but (as is common) the code also uses standard library functions such as `Option.unwrap()`, which panic at runtime if called incorrectly. To extend Prusti’s reach to uncovering such panics, we need to add a precondition for `Option.unwrap()`, but since this is standard library code, we also can’t (and don’t want to) edit it.

For this purpose, Prusti offers the *external specifications* (`extern_spec`) feature, which allows attaching contracts to functions (including library functions) separately from their implementation⁵. Such specifications look like a regular implementation block for a Rust type except that functions have no bodies (mimicking Rust’s trait declaration syntax).

For instance, the following external specification makes sure that calls to `Option.unwrap()` won’t cause panics, which is naturally expressed as a Prusti specification by identifying `is_some` as a pure method:

```
1 #[extern_spec]
2 impl<T> std::option::Option<T> {
3     #[pure]
4     fn is_some(&self) -> bool;
5
6     #[requires(self.is_some())]
7     fn unwrap(self) -> T;
8 }
```

As a user, one can take an incremental approach to adding such specifications to called functions, adding those which are most worthwhile for the user’s goals. For our panic-freedom pass, we pragmatically focused on the most widely used functions known to panic (from `Option<T>` and `Result<T>`), which already gave us stronger guarantees than our initial run with no such specifications.

After ruling out (most) panics in this way, we added specifications to check important domain-specific requirements, for example, that the height and time of each block in the blockchain increases monotonically. We used Prusti to verify that various functions in the `ibc` crate maintain these monotonicity invariants.

Inevitably for such a large codebase, we found functions that use currently unsupported language features. We *can* still attach contracts to such functions, which will subsequently be used by Prusti to deal with calls. We can tell Prusti not to check these specifications with a `#[trusted]` annotation. For example, in `ibc`, some time-related functions, such as `from_nanos` below, relied on unsupported types exposed by the `chrono` [28] crate and were marked as `#[trusted]`. The specification below expresses that `from_nanos` returns a valid result (rather than the error case of `Result`) if the `u64` parameter `nanos` fits within an `i64`, but Prusti does not check the function’s body to verify that the specification holds.

⁵ External specifications can also be used for functions inside the same crate, allowing developers to apply Prusti without modifying source files, if desired.

```

1  #[trusted]
2  #[ensures(nanos <= i64::MAX as u64 ==> result.is_ok())]
3  pub fn from_nanos(nanos: u64) -> Result<Timestamp, TryFromIntError> {
4      let nanos = nanos.try_into()?;
5      Ok(Timestamp {time: Some(Utc.timestamp_nanos(nanos))})
6  }

```

While trusted specifications must be written carefully, they enable developers to pragmatically focus on specifying and proving those properties they consider most relevant without imposing an excessive verification burden.

These features provide a further degree of freedom in the verification workflow: developers may initially use many `#[trusted]` annotations in a first iteration, and later attempt to reduce the number of trusted functions in subsequent iterations. As such, both trusted functions and external specifications further facilitate the incremental verification of realistic Rust code using Prusti.

3 Prusti from a Verification Expert’s Perspective

At the heart of Prusti lies the *core proof*, i.e. a memory safety proof written in separation logic [23, 39, 43], the de-facto standard for verifying resource-manipulating programs. Conceptually, the Prusti project explores three main questions, upon which we will reflect in this section:

1. To what extent can intuitive reasoning about most Rust programs be captured by an off-the-shelf separation logic?
2. To what extent can the generation of core proofs be automated?
3. To what extent can core proofs be leveraged for verifying interesting functional correctness properties?

3.1 Core Proofs in an Off-the-Shelf Separation Logic

Separation logic nowadays comes in numerous flavours, ranging from simple logics for verifying sequential heap-manipulating code to highly specialised variants targeting intricate concurrency or weak-memory models (cf. [39]). It is thus not surprising (but still very challenging!) that one can construct *some* separation logic which allows precise reasoning about all aspects of Rust’s memory model; RustBelt [27] is the most impressive attempt in that direction so far.

By contrast, the Prusti project aims to enable *intuitive* formal reasoning about *most* Rust code. We believe that this approach matches Rust’s design philosophy of enabling “fearless programming”: *safe Rust* code, i.e. code without any *direct* usages of unsafe language features should be understandable, without low-level concerns. Recent studies [5, 16] confirm that Rust code in the wild largely adheres to this philosophy: the vast majority of function implementations are written in safe Rust; they *may* call functions that are implemented using unsafe features, but shield clients from these details through encapsulation.

More concretely, Prusti embeds an annotated Rust program (cf. Sect. 2) in the Viper intermediate verification language [38], which is based on Implicit

Dynamic Frames (IDF)—a variant of traditional separation logics with a clear formal connection to standard separation logic [40]. Building upon an off-the-shelf logic has the advantage that the overall soundness of the embedding is analogous to soundness arguments that are well-understood for separation logic reasoning; it also allows us to draw on substantial prior work and expertise, particularly when it comes to proof automation.

The original Prusti paper [6] describes the embedding in detail. Overall, we found that the read and write capabilities governed by Rust’s flow sensitive type system have almost identical properties to the assertions governing heap accesses in IDF. In particular, Rust structs can be modelled as (possibly nested and recursive) *predicates* representing unique access to a type instance. Moreover, moves and simple usages of Rust’s shared and mutable borrows resemble ownership transfers in the permission reading of separation logic assertions [10]; *reborrowing* is modelled directly by *magic wands*: when a reborrowed reference is passed back to a caller, it comes with a magic wand representing the ownership of all borrowed-from locations *not* currently in the proof.

Prusti’s underlying logic champions simplicity and fits well into Rust’s overall design philosophy: at every point in Prusti’s core proof, there is direct representation of ownership in separation logic terms. This is different from RustBelt [27], where ownership and the connection between reborrowed and borrowed-from locations is handled via an indirection through a custom *lifetime logic* designed to express general semantic requirements on how lifetimes are manipulated, including via ad hoc manual policies implemented by unsafe code.

However, the simplicity of Prusti’s underlying logic has also made some (safe) Rust features harder to incorporate. One key example is struct types with explicit *lifetime parameters* (used to accommodate reference-typed fields), for which it is sometimes convenient to treat the struct as a single resource, and sometimes convenient to consider it as multiple individual resources borrowed for a certain lifetime. RustBelt achieves this via the more fine-grained resources of its lifetime logic; it is unclear whether this complexity is inevitable.

3.2 Full Automation of Core Proofs for Type-Checked Rust

As explained above, Prusti’s underlying model introduces nested and potentially recursive predicates to model instances of Rust types. However, general reasoning about such separation logic predicates is known to be undecidable [3, 22]. Verifiers such as Viper require additional annotations to guide reasoning about predicates, e.g. by inserting explicit statements to *unfold* and *fold* predicate definitions into a Viper program. For example, when a field of a struct is accessed in the Rust program, this requires unfolding the predicate modelling the capabilities for accessing the struct; the obtained capabilities cannot always be re-folded into a predicate since the field might be borrowed or moved-out.

While fold and unfold statements cannot be inferred automatically for *arbitrary* code with recursive predicates, Prusti infers them automatically for type-correct Rust code. The essential point is that the Rust compiler, when enforcing the flow-sensitive typing rules for the language, requires book-keeping similar to

that of unfolding and folding our predicates. For example, enforcing the check that fields moved out from a struct are (all) moved back in before the struct can be returned is conceptually analogous to refolding its corresponding predicate definition in Prusti’s model.

Prusti performs a pass over the encoded Rust program to add the necessary fold and unfold operations: essentially it performs a symbolic execution, tracking the accessible places at each program point and their current depth of unfolding (differentiating, say, between a struct being accessible and its fields being accessible). In addition to fold/unfold annotations, Prusti also infers all of the necessary Viper annotations for reasoning about magic wands [45] modelling reborrows. In all, the annotations required make up a large chunk of the generated Viper code, but they are generated *fully automatically* for all Rust programs supported by Prusti. This degree of automation is challenging to achieve but (we believe) an important objective for a tool that tries to raise the conceptual level at which a user interacts with a verifier. It ensures that Prusti users do not have to understand the sometimes intricate logical encoding of their programs. To our knowledge, Prusti was the first tool to be able to automatically produce formal proofs about a substantial fragment of Rust that could be automatically checked by a program verifier.

3.3 Incorporating Rich Functional Specifications

Prusti’s underlying logic is Viper’s dialect of Implicit Dynamic Frames. Although closely related to separation logic, a key feature of this logic is that one can conjoin functional specifications concerning heap *values* directly onto the resources such as permissions and predicate instances. In this sense, once the core proof is in place, layering functional specifications on top comes essentially for free.

Our first versions of Prusti exploited this technical feature to embed *all* aspects of user-written specifications (i.e. Rust annotations) into corresponding expressions in the generated Viper code, i.e. the core proof. A more-recent extension of Prusti’s core model equips each predicate instance with a *snapshot*: a value used as a mathematical identity for the current state of the (possibly composite) portion of the program memory accessible via this predicate. This technique originates (we believe) from the implementation of the VeriFast program verifier [24], and is also used extensively in Viper’s symbolic execution engine [46]. RustHornBelt [33] uses a similar technique to layer functional specification on top of RustBelt [27] predicates. Snapshots simplify encoding properties guaranteed by reasoning methodologies *other than* the basic separation logic framing built into Prusti’s core proofs. For example, (in work with Fabian Wolff) we use the flexibility provided by snapshots to layer guarantees about the heap on top of the core proof to extend Prusti’s support for a rich class of specifications about Rust closures [52].

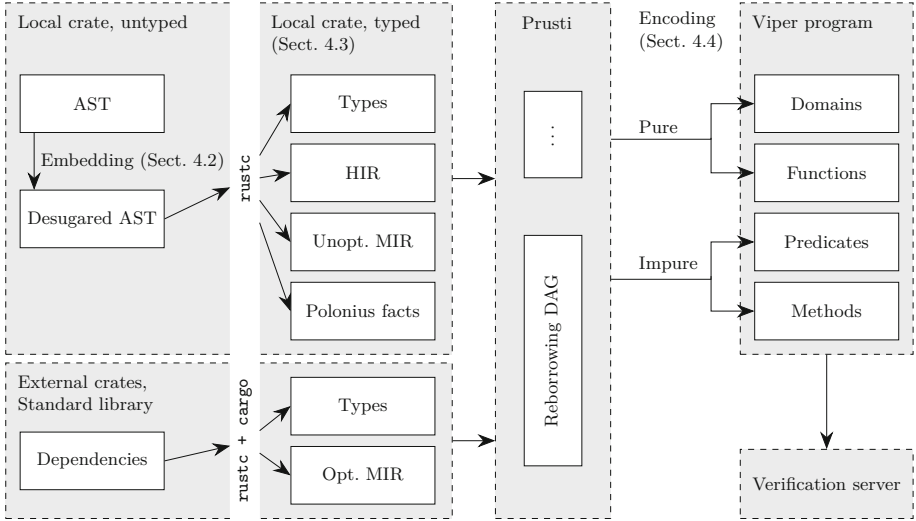


Fig. 6. Overview of Prusti’s encoding process.

4 Prusti from a Tool Engineer’s Perspective

Prusti targets real-world code in Rust, itself a mature and complex language. Accordingly, Prusti is designed to reuse existing functionality from the Rust compiler whenever possible, in order to reduce the implementation burden and faithfully maintain compatibility with the constantly-evolving Rust ecosystem.

4.1 Architecture and Design Overview

Prusti is implemented as a *compiler driver*, reusing the standard `rustc` compiler extensively; its overall workflow is presented in Fig. 6. Prusti launches and interacts with a full instance of `rustc`, used both for its program representations and analysis results (second column; cf. Sect. 4.3). To have Prusti-specific specification features (Sect. 2.3) type-checked analogously to regular Rust expressions (including error-reporting), Prusti performs a *specification embedding*, reusing existing Rust features whose type-checking rules are analogous (top-left; cf. Sect. 4.2). Prusti has `rustc` map the sources for both the program and (embedded) specifications down to `rustc`’s mid-level representations as for standard compilation. Prusti performs its own analyses (third column), and assimilates all necessary information to generate a Viper program (last column) that it sends to a further Prusti component which performs verification through a Viper wrapper. If verification fails, Prusti maps the Viper errors to user-readable Rust errors reported via the compiler API.

The compiler driver architecture is used by popular tools such as Clippy [11] and Miri [36]; it has two main advantages. First, it raises confidence that the

semantics used by Prusti is faithful. Prusti directly obtains a control-flow graph (CFG) representation of any parsed Rust function from the compiler, instead of inventing its own representation, which could lead to errors or semantic differences over time. The CFG-based representation used by Prusti, called *unoptimised MIR*, has a simple order-of-execution semantics and a limited number of statements; at this stage, many of the more-subtle aspects of Rust’s evaluation semantics have been *already handled* by the compiler. For example, Prusti does not need to be aware that Rust uses short-circuiting semantics for Boolean operators, because Boolean expressions are already transformed by the compiler into multiple statements evaluating individual operators. Unoptimised MIR maintains all type-checker information, along with back-links that allow the compiler (and thus also Prusti) to translate error messages back to the source code.

Second, the above architecture enables Prusti to reuse compiler components. Besides building upon unoptimised MIR, Prusti reuses the compiler’s type and borrow checker to ensure that user-written Prusti annotations follow typing rules analogous to regular Rust expressions, as explained in Sect. 4.2. Similarly, Prusti reuses the Rust compiler’s error reporting component to display verification errors. This way, the default syntax of the reports is familiar to Rust programmers and the compiler can be configured to report machine-readable errors. The latter simplifies integrating Prusti with other tools. For example, IDE extensions like the official Prusti Assistant extension for Visual Studio Code, but even Prusti-unaware tools such as Rust-analyzer [44], can be configured to report Prusti verification errors generated by running `cargo-prusti` instead of `cargo check`.

4.2 Specification Embedding

Prusti-specific annotations (e.g. method contracts) are implemented with *procedural macros* [1]. These macros are defined to generate nothing when compiled using the regular Rust compiler. However, when compiled with Prusti, a *specification embedding* is performed: to make the compiler both type-check and translate (to MIR) these specifications, corresponding methods are added to the program. For Prusti-specific constructs the specification embedding is more involved, replacing them with usages of Rust features which have the right type-checking requirements. For example, quantifiers (Sect. 2.3) are embedded as *closures*.

Prusti uses a Pratt parser [41] to perform the embedding of Prusti-specific constructs, before invoking the `syn` [13] Rust parser on the result, yielding an AST representation. The resulting specification expressions are embedded into the bodies of methods with unique names. Prusti constructs a mapping between these generated methods (called *specification items*) and the relevant construct in the original source code (e.g. for a precondition, the method it is a precondition of). By feeding the program augmented with specification items through the compiler, we both check that the specifications type-check and can obtain corresponding MIR representations of the specifications. The type-checking and evaluation semantics reflected by this translation to MIR are those of standard

`rustc`; this approach reuses the standard semantics of the Rust language for specification checking and compilation.

4.3 Compiler Interface

Prusti obtains various information from `rustc`'s data structures, as illustrated in the second column of Fig. 6. Given how Rust compilation works, different information is available (and used by Prusti) for the *local* crate (i.e. the crate being compiled/verified) and *external* crates (the dependencies of the local crate).

Local Crate. For the local crate, Prusti obtains a high-level AST representation (HIR), the type definitions, the unoptimised CFGs of the functions (MIR), and borrow-checker information (Polonius facts), defining the compiler-determined lifetimes of references. Prusti uses HIR, in which function names have already been associated to their definition, to retrieve specifications embedded in specification items, as described in Sect. 4.2. Prusti uses type definitions to generate Viper predicate definitions for the core proof (cf. Sect. 3), while unoptimised MIR is used to generate the corresponding Viper code itself (cf. Sect. 4.4).

The compiler offers various versions of MIR at different stages during the compilation process. Prusti uses the *unoptimised* version because it is the only one on which the borrow-checker runs. This also has a semantic advantage, since we do not need to worry whether compiler optimisations preserve the strong type properties that Prusti exploits⁶. Prusti uses the results from the Polonius borrow-checker, also called *facts*, to automate the generation of annotations such as folding and unfolding of Viper predicates (cf. Sect. 3.2).

Previously, the compiler API did not expose Polonius facts, but the compiler developers were very supportive in accepting our proposed additions to the API [4]. Our changes have since been used by at least one other static analysis tool, Flowistry [12], to access precise aliasing information.

External Crates. For external crates, the compiler offers strictly less information than for the local one, primarily for performance reasons. Type definitions and *optimised* MIR are available (Prusti uses the former to encode calls), but the HIR, the unoptimised MIR, and the Polonius facts are not present. Since Prusti's overall methodology is modular, the only real limitation this imposes is that any Prusti specifications written *in* an external crate will not be seen. As explained in Sect. 2.4, Prusti supports external specifications to be applied to these functions from the local crate. Nonetheless, following the example of the MIRAI static analyser [17], we believe that, in the future, previously-compiled Prusti specifications could be recovered for external crates from a combination of the optimised MIR and persisting some information to disk between compilations.

⁶ See for example <https://github.com/rust-lang/rust/issues/46420> for an optimisation that used to copy non-duplicable mutable references.

4.4 Encoding to Viper

Finally, Prusti uses the information assembled from the Rust compiler to encode an annotated Rust program to a Viper program for verification. As shown in the right half of Fig. 6, there are *two* different encodings: a *pure encoding* to Viper expressions and an *impure encoding* to Viper statements.

Pure Encoding. Prusti’s pure encoding is used for specifications and pure functions (which may be invoked from within specifications), and is necessary as Viper specifications must be Viper expressions (which are side-effect-free, unlike statements, which are a distinct notion in Viper).

Pure Rust expressions (cf. Sect. 2.3) are encoded to Viper expressions using a backwards symbolic execution through their CFG, starting from the variable which stores the final result (easily determined in MIR); the steps are reminiscent of a standard weakest-precondition calculation.

To represent Rust values in pure code, Prusti uses the snapshot technique presented in Sect. 3.3. Snapshots are encoded to Viper domains; that is, abstract type definitions with uninterpreted functions and axioms that describe the relation between the snapshot of a type and the snapshot of its inner instances (e.g. variants of an enumeration or fields of a structure). These are computed from the compiler’s type definitions.

Impure Encoding. Like the pure encoding, the impure encoding processes the unoptimised MIR and analyses the CFG of a method. However, in the impure case, the output is a Viper *method* containing heap-mutating statements. Viper methods can also contain `goto` statements, which allows us to encode the MIR CFG without having to reconstruct loops or standard control flow structures.

To encode mutable references, Prusti needs to know the program point at which references expire and which places receive the no-longer-borrowed ownership, such that magic wands that encode the ownership flow can be applied in the right order to form the core proof. To do so, Prusti elaborates the borrow-checker facts to automatically compute a directed acyclic graph (DAG) of the borrowing relations for each program point: each node with exit edges represents a reference and each edge points to the places that it blocks. When a set of references expire, a topological sort of the DAG determines the order in which the magic wands associated to the edges should be applied. This *Reborrowing DAG* is generalised to appropriately account for conditional paths through the CFG.

5 Related Work

RustBelt [27] is a long-standing verification project for Rust. RustBelt focuses on proving that abstractions provided by internally unsafe libraries are safe; verification is performed in Coq [8] over a simplified language based on Rust. By contrast, Prusti is designed for general-purpose verification (with an emphasis on safe Rust), and directly uses the representations in the Rust compiler.

Several verification approaches have been developed which avoid explicitly modelling Rust’s memory (and aliasing) for safe Rust (only). Electrolysis [50]

applied *purification* of such programs to convert them to functional programs to be verified in Lean [37]. More recently, RustHorn [34] and Creusot [14] leverage Rust’s ownership semantics to model mutable references using a technique similar to prophecy variables [2] rather than explicitly modelling the heap. The soundness of the approach was shown in RustHornBelt [33], a unification of RustBelt and RustHorn. To our knowledge, automatic generation of core proofs in these underlying models remains an open problem. Although not for Rust, the Move Prover [15] employs a reborrowing DAG similar to Prusti’s, although it then employs techniques similar to purification to eliminate heap reasoning.

Several automated static analysers have been developed for Rust, including the abstract interpreter MIRAI [17]. The Kani Rust Verifier [51] applies bounded model-checking. Other tools analyse the generated LLVM: e.g. Klee Rust performs symbolic testing [31], Smack applies bounded verification [7], Project Oak [42] provides an evolving portfolio of complementary tools. None of these tools use the ownership guarantees of the type system, to our knowledge.

Stacked Borrows [26] is another formal model for Rust aiming to precisely define notions of *undefined behaviour* for the Rust language; it is accompanied by the interpreter Miri [36], which can be used to dynamically check for rule violations. We are not aware of corresponding static tools based on this model.

6 Conclusions and Future Work

We have presented the Prusti project, and reflected on its key features and most-notable design decisions from three different perspectives: for users, verification experts, and authors of other Rust analysis tools. From a user’s perspective, notable features include the close relationship between specifications and Rust expressions, and the flexible trade-offs between annotation effort and richness of guarantees, which supports incremental usage of the tool on large-scale projects. For verification experts, a notable goal is the reuse of long-standing program reasoning techniques for reasoning about (primarily) safe Rust code. For tool builders, the extensive reuse of compiler data structures, analyses and error reporting mechanisms has proven powerful; these techniques are largely reusable.

A key goal for future work to benefit users is to enable richer specifications (when desired), via built-in types (such as mathematical sets) and add dedicated features for *ghost code*, as well as improving verification performance. Of more interest to verification experts, we are exploring the adaptation of Prusti’s core model and proofs to both structs with lifetime parameters and some usages of unsafe code. On the tooling front, we aim to support persistence of compiled Prusti specifications, and offering built-in specifications for common Rust libraries.

Acknowledgements. We warmly thank Nicholas D. Matsakis, Nick Cameron, Derek Dreyer and Ralf Jung for extensive discussions and feedback in the early stages of this project, and are very grateful to Florian Hahn for his work on a precursor to Prusti [20], as well as numerous Master’s and undergraduate students who have since contributed via projects.

This work was partially funded by the Swiss National Science Foundation (SNSF) (Grant No. 200021_169503), the Natural Sciences and Engineering Research Council of Canada (NSERC) (ref. RGPIN-2020-06072), Amazon Research Awards, Meta (then Facebook) Research and the Interchain Foundation.

References

1. Procedural macros documentation (2022). <https://doc.rust-lang.org/reference/procedural-macros.html>
2. Abadi, M., Lamport, L.: The existence of refinement mappings. In: Proceedings of the 3rd Annual Symposium on Logic in Computer Science, pp. 165–175, July 1988. <https://www.microsoft.com/en-us/research/publication/the-existence-of-refinement-mappings/>, IICS 1988 Test of Time Award
3. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 411–425. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_27
4. Astrauskas, V.: Enable compiler consumers to obtain MIR: Body with Polonius facts. <https://github.com/rust-lang/rust/pull/86977>
5. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe Rust? Proc. ACM Program. Lang. **4**(OOPSLA), 1–27 (2020)
6. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019). <https://doi.org/10.1145/3360573>
7. Baranowski, M., He, S., Rakamarić, Z.: Verifying rust programs with SMACK. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 528–535. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_32
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions. In: Texts in Theoretical Computer Science. An EATCS Series, pp. XXV–472. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-662-07964-5>
9. Bloch, J.: Extra, extra - read all about it: Nearly all binary searches and merge-sorts are broken, June 2006. <https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>
10. Bornat, R., Calcagno, C., O’Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 259–270 (2005)
11. Clippy developers: Clippy: A collection of lints to catch common mistakes and improve your Rust code. <https://github.com/rust-lang/rust-clippy>
12. Crichton, W.: Flowistry: Information flow for Rust. <https://github.com/willcrichton/flowistry>
13. Tolnay, D.: Parser for Rust source code (2021). <https://crates.io/crates/syn>
14. Denis, X., Jourdan, J.H., Marché, C.: The Creusot environment for the deductive verification of Rust programs (2021)
15. Dill, D., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, E.: Fast and reliable formal verification of smart contracts with the Move prover. arXiv preprint [arXiv:2110.08362](https://arxiv.org/abs/2110.08362) (2021)
16. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 246–257. IEEE (2020)

17. Facebook: MIRAI: an abstract interpreter for the Rust compiler’s mid-level intermediate representation. <https://github.com/facebookexperimental/MIRAI>
18. Garavel, H., Beek, M.H., Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) FMICS 2020. LNCS, vol. 12327, pp. 3–69. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_1
19. Goes, C.: The interblockchain communication protocol: an overview. arXiv preprint [arXiv:2006.15918](https://arxiv.org/abs/2006.15918) (2020)
20. Hahn, F.: Rust2Viper: building a static verifier for Rust. Master’s thesis, ETH Zurich (2015)
21. Informal Systems Inc. and ibc-rs authors: Rust implementation of the Inter-Blockchain Communication (IBC) protocol (2021). <https://docs.rs/ibc>
22. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 201–218. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_15
23. Ishtiaq, S.S., O’Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL, pp. 14–26. ACM (2001)
24. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
25. Jung, R.: The scope of unsafe, January 2016. <https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html>
26. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: an aliasing model for Rust. Proc. ACM Program. Lang. **4**(POPL), 1–32 (2019)
27. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. **2**(POPL), 1–34 (2017)
28. Seonghoon, K., et al.: Chrono: Date and Time for Rust (2021). <https://docs.rs/chrono>
29. Kassios, I.T.: The dynamic frames theory. Formal Aspects Comput. **23**(3), 267–289 (2011)
30. Klabnik, S., Nichols, C.: Unsafe Rust (2022). <https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html>
31. Lindner, M., Aparicius, J., Lindgren, P.: No panic! Verification of Rust programs by symbolic execution. In: 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), pp. 108–114. IEEE (2018)
32. Matsakis, N.D.: Unsafe abstractions (2016). <http://smallcultfollowing.com/babysteps/blog/2016/05/23/unsafe-abstractions>
33. Matsushita, Y.: Extensible functional-correctness verification of rust programs by the technique of prophecy. Master’s thesis, University of Tokyo (2021)
34. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. In: ESOP, pp. 484–514 (2020)
35. Meyer, B.: Design by contract. In: Mandrioli, D., Meyer, B. (eds.) Advances in Object-Oriented Software Engineering, pp. 1–50. Prentice Hall (1991)
36. Miri developers: Miri: An interpreter for Rust’s mid-level intermediate representation. <https://github.com/rust-lang/miri>
37. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE

2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
38. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
39. O’Hearn, P.: Separation logic. *Commun. ACM* **62**(2), 86–95 (2019)
40. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. *Log. Methods Comput. Sci.* **8**(3:01), 1–54 (2012)
41. Pratt, V.R.: Top down operator precedence. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 41–51 (1973)
42. Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., Laurie, B.: Towards making formal methods normal: meeting developers where they are. *arXiv preprint arXiv:2010.16345* (2020)
43. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74. IEEE (2002)
44. Rust-analyzer developers: Rust-analyzer: A Rust compiler front-end for *ides*. <https://github.com/rust-analyzer/rust-analyzer>
45. Schwerhoff, M., Summers, A.J.: Lightweight support for magic wands in an automatic verifier. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, vol. 37, pp. 614–638. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2015)
46. Schwerhoff, M.H.: Advancing automated, permission-based program verification using symbolic execution. Ph.D. thesis, ETH Zurich (2016)
47. The Prusti Team: Prusti User Guide (2020). <https://viperproject.github.io/prusti-dev/user-guide/>
48. The Prusti Team: Prusti NFM 2022 Online Appendix (2022). https://github.com/viperproject/prusti-dev/tree/master/prusti-tests/tests/verify_overflow/pass/nfm22
49. The Rust Survey Team: Rust survey 2019 results: Rust blog, April 2020. <https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html>
50. Ullrich, S.: Simple verification of Rust programs via functional purification. Master’s thesis, Karlsruher Institut für Technologie (KIT) (2016)
51. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in Rust (2022)
52. Wolff, F., Bílá, A., Matheja, C., Müller, P., Summers, A.J.: Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.* **5**(OOPSLA), 1–29 (2021)