Usability Lessons Towards Adopting Deductive Verification in Mainstream Rust Development

Abstract—**Can software engineers effectively leverage deductive verifiers for Rust without an established formal verification background? In this case study, we investigate the Prusti and Creusot deductive verifiers for Rust and use them to formally verify a practical union-find data structure. The main verification engineer in our study did not previously have a verification background and so serves as a proxy for the target users of deductive Rust verifiers. Over the course of this study, we observed user obstacles due to differences between the development process of verification contracts and the programming process that software engineers are typically used to. From this study, we created tool-agnostic recommendations to make the process more accessible for Rust programmers. During our work, we maintained direct communication with the developers of both tools. Thus, to design a scalable learning experience for a growing number of programmers, our recommendations focus on reducing the need for expert assistance during the verification process. We suggest that changes to the level of abstraction of the underlying verification mechanisms, which can be expressed in the user interface and learning resources for the respective tools, can reduce the logical complexity of verification and make these tools more accessible to a broader audience. Overall our work demonstrates that a sufficiently motivated developer can use current Rust automated verifiers on practical code, and develops recommendations to enable further adoption of deductive verifiers within the Rust programming community.**

Index Terms—**Formal Verification, Rust, Usability, Case Study, Deductive Verification.**

I. Introduction

Safety and correctness are essential to critical systems software. Usage of the Rust programming language serves as one approach towards achieving software safety. Rust ensures memory-safety and prevents out-of-bounds memory accesses such as the one responsible for the global CrowdStrike incident in 2024 [1]. Additionally, Rust is accessible to a wide range of developers that are not experts in programming languages, memory safety, or formal methods. As a result, Rust has seen increased use in critical systems such as the Linux kernel [2].

Although it is an important property, memory safety is only part of the safety and correctness requirements of critical systems software. One approach to ensuring these requirements is formal verification, which uses mathematical proofs to make guarantees about safety and correctness. Semi-automated verification languages such as Dafny allow programmers to write code specifically designed for verification [3]. There are a growing number of tools for formal verification of Rust programs, which can enable software engineers to ensure stronger correctness properties about their programs beyond just memory safety. One path towards widespread adoption of these tools is to ensure they are as accessible as Rust itself. A key question we ask here is this: **are current Rust verification tools ready to be used by programmers without prior experience in formal verification?**

To begin to answer this question, we performed an initial feasibility study to determine how a programmer with prior Rust knowledge but no prior exposure to formal verification tools would verify an existing Rust library. Focusing on a single programmer, rather than a group, made it feasible to capture a rich history of the programmer's progress and roadblocks that made the activity difficult. It also made it feasible to have discussions with the developers of these tools to provide detailed explanations of how to use the tools and to fix bugs in the tools that were encountered. The intent is to pave the way for future user studies in this area by documenting what information and learning was needed for a programmer without a formal methods background to succeed.

More specifically, this study focused on applying the deductive verifiers Prusti [4] and Creusot [5] to formally verify a union-find data structure from the Rust e-graphs library, *egg* [6]. This practical union-find implementation is simple and self-contained, yet it enables e-graphs to perform efficient and optimized code refactorings. In the process, we also contribute the first verification of union-find with path compression using an automated verifier.

For formal verification to integrate into the software development process, the user experience of developing contracts should be as close to developing code as possible. We identified several areas where usability improvements could significantly benefit Rust programmers in formal verification, including straightforward setup and onboarding, enhanced support for debugging verification failures, user guides aimed toward learners, and intuitive mechanisms for expressing logic to model data.

In summary, this paper contributes the following:

- The first verification with automated verifiers of a realistic union-find data structure (Section III).
- A report of the challenges encountered in verifying a union-find data structure with the deductive verifiers Creusot and Prusti (Section IV).
- A set of recommendations for designing usable deductive verifiers which would overcome the challenges encountered during the study (Section V).

Overall our work provides insights on the successful use (and pitfalls) of Rust verifiers by a typical Rust programmer, laying the foundation for mainstream use of deductive verifiers in software development with Rust.

II. Background

A. Verification tools

This study utilized two verification tools, Prusti and Creusot, which are both designed to prevent "panics" at runtime and to enable the development of contracts to allow users to specify correct behavior. The contracts of both of these tools aim to be syntactically similar to natural Rust code—utilizing Rust attributes in the form of `#[`*Attr*`]` as to be ignored during regular program compilation. Preconditions and postconditions are represented by `requires` and `ensures` statements respectively. Users may also write functional predicates, specify loop invariants, and reason over quantifiers in a similar manner with minor syntactical differences between the two tools. Neither of these tools expose users to separation logic—instead utilizing Rust's ownership properties to represent *prophecies* in Creusot and *pledges* in Prusti—and neither currently support verifying unsafe Rust code. Despite surface-level similarities, the usability, logic necessary to prove goals, and verification frameworks of these two tools differ.

1) Creusot: Creusot is a verification tool that translates Rust code into Coma, a custom intermediate verification language, to be analyzed by Why3 [7]. The Why3 platform leverages several different SMT solvers and proof strategies that can be executed automatically. Creusot users need to install Why3 IDE to understand where contracts fail in the original Rust program and to debug effectively. Creusot's specification language, Pearlite, best interprets data in terms of a *logical model* type which is compatible with Why3. For several standard Rust datatypes, appending `@` to the variable is enough to derive this model, otherwise a `ShallowModel` trait must be implemented for it [5].

2) Prusti: Prusti is an automated Rust verifier built on the Viper verification framework [8]. Its intended use is within the Prusti Assistant VS Code extension where it is integrated as if it were a "stricter compiler for Rust" [4]. Prusti Assistant displays errors and failed contract checks in the same style as Rust compiler errors, with failing contracts underlined in red. This familiar feedback simplifies understanding and debugging. Prusti can reason directly about several native Rust datatypes, but lacks support for others such as slices and arrays.

B. Union-find

In the union-find data structure, a collection of disjoint sets are represented as a forest of trees, where each tree is defined by a parent relation pointing from child to parent. This differs from a classical forest of trees, where the parent-to-child relationship is typically explicitly maintained. To achieve the desired asymptotic complexity for union-find operations, path compression techniques may be applied during queries, dynamically altering the tree structure to optimize future lookups. However, the child-to-parent representation and the dynamic nature of path compression complicate the expression of invariants, function termination criteria, and other properties necessary for verification.

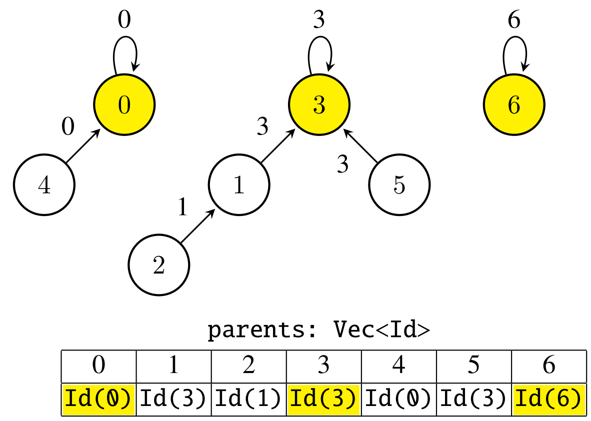

Fig. 1. A representation of a valid union-find data structure, demonstrating the relation of the underlying `parents` indices to nodes and `Id`s to directional edges. Roots are highlighted.

The representation of union-find in *egg* is implemented as a struct containing a `parents` vector where each index is a node. The vector holds `Id`s, which wrap unsigned 32-bit integers and represent directional edges corresponding to the index of each node's parent. A visual representation of `parents` is given in Fig. 1. This figure shows a forest where each node points to its parent along with the `parents` vector that corresponds to this forest. Root nodes, highlighted in yellow, point to themselves.

Each disjoint set in union-find contains a single root that represents the set. All paths along the child-to-parent edges must terminate at a root, a node whose parent is itself, preventing the existence of multi-node cycles. As long as the above properties hold, the union-find structure is valid and well-formed. The *egg* project presents several methods to read and mutate the data structure while retaining this definition of validity.

III. Verification in Creusot and Prusti

In this section, we compare and contrast two implementations of verification of this union-find—one in Creusot and one in Prusti. The goal of these verifications is to confirm the correctness of the union-find methods and ensure they do not invalidate the well-formedness of the union-find data structure. This was done by first typecasting the `Id` struct as `usize` to focus on verifying the algorithms. Because Creusot uses logical model derivations to reason about most values, re-implementing a verified `Id` did not require rewriting contracts. As for Prusti, the raw `usize` type was interpreted in contracts and would require significant rewrites. Thus, `Id` was left as `usize` in the final verified union-find in Prusti.

A. The Invariant

The basis of our verification relies on what we've defined as the union-find *invariant*. As a precondition and postcondition, it allows us to validate the instance of union-find before and after public function calls. If the invariant properties are not guaranteed, the internal functions may fail to execute properly

```
#[predicate]
fn invariant(&self) -> bool {
  pearlite! {
    self.len() <= u32::MAX@ &&
    self.len() == self.dist.len() &&
    forall<i: Int> 0 <= i && i < self.len() ==>
      self.parents[i]@ >= 0 &&
      self.parents[i]@ < self.len()
    forall <i: Int> 0 <= i && i < self.len() ==>
      (self.dist[i] == 0 && i == self.parents[i]@)
      || (self.dist[i] > 0 &&
        self.dist[i] > self.dist[self.parents[i]@])
  }
}
```

Creusot uses a logical `Int` type in specs to compare integer values. We also defined `self.len` to serve as shorthand for `self.parents.len`. `self.dist` is a sequence to correspond indices in `parents` to distances.

```
predicate! {
  fn invariant(&self) -> bool {
    self.size() <= u32::MAX as usize &&
    forall(|i: usize| (i < self.size()) ==>
      self.parent(i) < self.size())
    forall (|i: usize| (i < self.size()) ==>
      (self.dist(i) == 0 && i == self.parent(i))
      || (self.dist(i) > 0 && self.dist(i) >
        self.dist(self.parent(i)))
    triggers=[(self.dist(self.parent(i)))]
    )
  }
}
```

In Prusti, we defined `dist` as a function to compute the distance of a node to root. Prusti also has triggers to only instantiate the `forall` quantifier when computing a the parent of a node's `dist`. Because `Id`s are cast as `usize`, we only need to ensure an upper bound for `i`.

Fig. 2. Creusot vs Prusti `invariant`

or produce incorrect results. The following three invariant conditions must hold during function calls to maintain well-formedness:

1) The length of `parents` must be less than or equal to the maximum `u32` value, as to not overflow the `Id` representations.
2) Every value in `parents` must contain an `Id` representing a valid index in `parents`.
3) For all nodes, the distance of a node to its root is zero if and only if it is a root. Otherwise, the distance of the node must be strictly more than its parent's distance to root.

The existence of a "distance to root" for a given node proves that the path to the root is finite, ensuring path termination. There cannot be a root in a disjoint set containing a cycle, as the distance of each node in the set would be undefined. We define these distances as `dist` using two different representations to match the capabilities of each tool. Both invariant implementations can be found in Fig. 2.

Creusot allows us to represent `dist` as the sequence `Snapshot<Seq<Int>>` inside the union-find struct. `Snapshot` becomes a zero-sized type during regular execution. In this sequence, the indexes correspond to nodes, and the values represent distances to root. Thus, in our invariant, we include the condition that the length of `dist` is equivalent to the length of `parents`.

Because Prusti lacks a sequence type, we can represent `dist` as a recursive function to determine the exact distance from a node to its root. Beginning at zero, `dist` increments a counter for path length, on each step from child to parent, returning the value upon reaching the root. If any set in the union-find contains a cycle and therefore no root, then all nodes in that set will have non-terminating `dist`s that would evaluate to infinity. Thus, if the counter is equal to the length of `parents`, it is immediately returned. This not only ensures that the function terminates, but also that cyclic structures will violate the invariant condition that parents must be strictly less than their child `dist`s.

B. Methods

This union-find data structure was implemented with a focus on speed, favoring iterative methods over recursion. This results in a practical yet more challenging example for verification due to mutations over iterations.

The functions `size` and `parent` serve as simple accessor methods and are the easiest to verify since they are pure, i.e., they do not modify the data structure. Prusti does not support reasoning about `parents` and its values directly as Creusot does, so by labeling these accessor functions with `#[trusted]` and `#[pure]`, they can be re-used in contract logic for verifying other functions. The default `find` (without path compression) internally mutates an `Id`, `current`, during the execution of a `while` loop. Through defining loop invariants, reasoning about `current` is trivial—it must correspond to a valid index in `parents` and always have the same root as its initial value.

The functions `make_set`, `parent_mut`, `union`, and `find_mut` all take `&mut self` as the first parameter, a reference to the union-find itself which allows for mutations. Any mutation introduces the possibility of invalidating the union-find structure such that it no longer conforms to the invariant properties. The bulk of the verification effort focused on validating these four functions against the invariant, starting with a valid union-find as input.

1) make_set: At a high level, `make_set` creates a new disjoint set containing a single root node by deriving a new `Id` from the length of `parents` and pushing it to the vector, returning the new `Id`. If the length of `parents` is greater than $2^{32}-1$, the value of the new `Id` will overflow, causing Rust to panic. As a result, `make_set` must require that the incoming length of `parents` is strictly less than the maximum `u32` value so that the final union-find may still satisfy the invariant. We expect the length of `parents` to be increased by one with all of the original nodes left unmodified. Additionally, the resulting `Id` returned must be a valid root equal to the length of the array before it was pushed.

In Creusot, the final result of `self` is denoted as `^self`. By modeling `parents` as a `Seq`, we ensure that the final model

```
#[requires(self.len() < u32::MAX@)]
#[requires(self.invariant())]
#[ensures((^self).invariant())]
#[ensures(result@ == self.len())]
#[ensures(self.parents@[result@] == result)]
#[ensures(self.len()+1 == (^self).len())]
#[ensures(forall<i: Int> 0 <= i && i < self.len()
  ==> self.parents[i] == (^self).parents[i])]
#[ensures((^self).parents@
  == self.parents@.push(result))]
pub fn make_set(&mut self) -> Id {
  let id = Id::from(self.parents.len());
  self.parents.push(id);
  self.dist = snapshot! {
    self.dist.push(0) };
  id
}
```

In Creusot, the final result of `self` is denoted as `^self`. The original code was preserved, and `dist` was updated with ghost code before returning the new `Id`.

```
#[requires(self.size() < u32::MAX as usize)]
#[requires(self.invariant())]
#[ensures(result == old(self.size()))]
#[ensures(self.parent(result) == result)]
#[ensures(self.invariant())]
pub fn make_set(&mut self) -> Id {
  let id = self.parents.len();
  self.push(id);
  id
}
#[trusted]
#[requires(self.size() < u32::MAX as usize)]
#[requires(self.invariant())]
#[ensures(self.size() == old(self.size())+1)]
#[ensures(self.parent(old(self.size())) == value)]
#[ensures(forall(|i: usize| (i < old(self.size())
  ==> self.parent(i) == old(self.parent(i)))))]
#[ensures(self.invariant())]
fn push(&mut self, value: Id) {
  self.parents.push(value);
}
```

Prusti refers to the initial state of union-find as `old(self)` in postcondition contracts, and requires a custom `push` function on `self` to verify.

Fig. 3. Creusot vs Prusti `make_set`

```
#[requires(self.invariant())]
#[requires(query@ < self.len())]
#[ensures(self.parents[query@] == *result
  && (^self).parents[query@]== ^result)]
#[ensures(self.len() ==(^self).len())]
#[ensures(forall<i: Int> 0 <= i &&
    i != query@ && i < self.len() ==>
  self.parents[i] == (^self).parents[i])]
#[ensures(self.dist == (^self).dist)]
fn parent_mut(&mut self, query: Id) -> &mut Id {
  &mut self.parents[usize::from(query)]
}
```

Creusot is able to ensure that no new `Id`s were added or removed and that only the queried `Id` is mutated. The invariant cannot be ensured within the function's scope because of the returned mutable reference.

```
#[trusted]
#[requires(self.invariant())]
#[requires(query < self.size())]
#[after_expiry(
  old(self.size()) == self.size()
  && self.invariant()
  && snap(&self.parent(query))
    == before_expiry(snap(result))
  && forall(|i: usize|
      (i < self.size() && i != query) ==>
    self.parent(i) == old(self).parent(i)))]
fn parent_mut(&mut self, query: Id) -> &mut Id {
  &mut self.parents[query]
}
```

Despite use of `#[trusted]`, Prusti can verify pledges after the returned reference expires. `Id`s are casted as `usize` already, so `from` is not used.

Fig. 4. Creusot vs Prusti `parent_mut`

is the same as the initial after pushing it to the `Seq`. Using the `snapshot!` macro, we can provide ghost code to push 0 to `dist` so that the invariant condition holds.

The final value in `parents` must be a lone root, but Prusti does not allow reasoning directly about vector values to ensure this. Therefore, we implemented a trusted `push` function directly on union-find so that `parent` can be reused as a value accessor.

2) parent_mut: In `parent_mut`, a mutable reference to `self` is returned to reassign the parent at the index of the input `Id`'s value. As in `parent`, the queried `Id` must have a value less than the length of `parents`. The final state of `self` after this reassignment cannot be verified within the scope of the function, because the mutation would occur *after* the return. We can only verify that a single mutation occurs at the location of the queried `Id` while all other values and the vector's length remains unchanged. Whether the new `Id` is valid and does not introduce a cycle is dependent on the context in which it is called. Thus, to satisfy the invariant, we must verify conditions *after* the borrow expires.

In this implementation, `parent_mut` is not a public function and is only used internally by `union` and `find_mut`. Therefore, in Creusot, we must validate the invariant as it is used in those contexts. While in Prusti, `parent_mut` must be trusted as it involves access of the `parents` vector, but we may still utilize the `after_expiry` contract to check conditions after the returned reference's lifetime expires. Not only is it possible to ensure the invariant holds, but we can also use `before_expiry` to ensure the result before mutation is the same as an immutable call to `parent`. The contracts of `parent_mut` in both Creusot and Prusti are in Fig. 4

3) union: Given two valid root `Id`s, `root1` and `root2`, `union` reassigns `root1` as the parent of `root2` using `parent_mut`. Thus, all nodes with original root `root2` now are represented by `root1` and the total number of disjoint sets in the union-find is decremented by one. After execution, it must be ensured that `root1` is the parent of `root2` and no other indexes are changed. All other roots remain valid. With

```rust
#[logic]
#[requires(self.invariant())]
#[requires(i >= 0 && i < self.len())]
#[ensures(result >= 0 && result < self.len())]
#[ensures(self.parents[result]@ == result)]
#[variant(self.dist[i])]
fn find_pure(&self, i: Int) -> Int {
  pearlite!{
    if self.parents[i]@ == i {
      i
    } else {
      self.find_pure(self.parents[i]@)
    }
  }
}
```

Creusot uses #[logic] to denote purely functional code that may be reused in contracts. Creusot also requires defining an integer variant to converge on zero and ensure function termination.

```rust
#[pure]
#[requires(self.invariant())]
#[requires(i < self.size())]
#[ensures(result < self.size())]
#[ensures(result == self.parent(result))]
fn find_pure(&self, i: Id) -> Id {
  let parent = self.parent(i);
  if parent == i {
    i
  } else {
    self.find_pure(parent)
  }
}
```

Prusti uses #[pure] for code to be used in contracts. An Id is passed as a parameter to allow for the call to parent, rather directly accessing the parents vector.

Fig. 5. Creusot vs Prusti find_pure

additional ghost code in Creusot, the dist values among the original descendants of root2 are incremented by one with a recursive incr function.

As an additional step in verification, we ensured that all descendants of root2 now belong to the disjoint set represented by root1. Over all nodes, we must verify that this holds for nodes previously represented by root2 and that the representatives of all other nodes stay the same using find_pure. We defined find_pure as the recursive implementation of find. The implementation of this function using both tools can be seen in Fig. 5.

Because find_pure recurses on union-find, its results cannot be proven using a forall quantifier alone in neither union nor find_mut. Instead, it must be justified by find_union_lemma, an additional lemma which compares the result of find_pure with the initial and final version of self. find_union_lemma has the same function body as find_pure but ensures more specific postconditions: to verify that every node on every path either has the same root before the mutation, or if it was root2 initially, it belongs to root1 in the final union-find. In both Creusot and Prusti, it was sufficient to call find_union_lemma in a forall context on all Ids in parents to verify the above.

4) find_mut: find_mut has the same interface behavior as find, and along the way, it performs a *path halving* path compression algorithm [9]. It does this by iteratively reassigning the Id at the current node's parent with parent_mut to it's parent's parent, or grandparent. After, grandparent becomes the new current, and the loop iterates again until the root is found. Similar to union, the goals of find_mut include ensuring that *all* roots of each disjoint set remain unchanged and that mutations occur only along the path from current to root. The value returned by the operation must be the same as find without mutation, or find_pure.

In both verifications, an additional recursive lemma is required to ensure that all nodes remain in the same disjoint sets before and after the function. find_mut_lemma, in a similar nature to find_union_lemma, must ensure that descendants of current are now represented by the root of the new value, and all other roots returned by find_pure remain the same over every iteration. Because this new value is the grandparent of current and thus shares the same root, find_mut_lemma also must ensure the roots for all nodes remain constant throughout the function. find_mut_lemma is called after the mutation in a forall context over all ids with the current self, old, current as cur, and grandparent as gp.

Several contracts in find_mut_lemma utilize an additional predicate, is_descendant, to reason about directional relations between cur, gp, and a given Id, i. is_descendant(p, q) returns true if and only if there exists a path from p to q, providing more evidence than simply equating the results of find_pure on p and q. Both find_mut_lemma implementations can be seen in Fig. 6.

In Creusot, the snapshot! macro saves the intermediate state of the union-find as old in every iteration before the mutation with parent_mut. In find_mut_lemma, we generalize two conditions; that either a given Id is a descendant of cur, and its path to root has been modified, or it is not and its path is unchanged. Creusot can ensure this through modeling the mutation directly on a sequence model of old with set and requiring its equivalence to self. The fact that cur and gp have the same root needs to be proven in root_lemma. Invoked when cur is encountered, root_lemma recurses on the path from gp to its root to verify there has been no mutations on its path and that gp has the same root.

Validating that the dist at gp is strictly less than cur can be done in a single contract with index accesses in Creusot. In this implementation, dist also serves as an upper bound for distance and does not need to be mutated to satisfy the invariant. In order to ensure exactly how much the path is compressed, a decr function would need to be implemented, similar to incr for verifying union.

For Prusti to ensure the invariant and functionality of find_mut, we had to reason about intermediate states of self before and after the mutation as in Creusot. In place of the

```
#[predicate]
#[requires(self.invariant() && old.invariant())]
#[requires(old.len() == self.len())]
#[requires(cur@ < self.len() && gp@ < self.len())]
#[requires(i >= 0 && i < self.len())]
#[requires(old.parents@.set(cur@, gp)
  == self.parents@)]
#[requires(self.dist[gp@] < self.dist[cur@])]
#[ensures(!old.is_descendant(i, cur@) ==>
  self.find_pure(i) == old.find_pure(i))]
#[ensures(old.is_descendant(i, cur@) ==>
  self.find_pure(i) == old.find_pure(gp@))]
#[variant(self.dist[i])]
#[ensures(result)]
fn find_mut_lemma(
&self, old: Self, cur: Id, gp: Id, i: Int) -> bool {
  pearlite! {
    if old.is_root(i) {
      true
    } else if i == cur@ {
      self.root_lemma(old, gp@)
    } else {
      self.find_mut_lemma(
        old, cur, gp, self.parents[i]@)
    }
  }
}
```

While Creusot requires less preconditions, an additional lemma is called to ensure `gp`'s roots did not change. Creusot cannot prove that `self.find_pure(i) == old.find_pure(i)` until this lemma is called in a `forall` quantifier.

```
#[pure]
#[requires(self.invariant() && old.invariant())]
#[requires(old.size() == self.size())]
#[requires(cur < self.size() && gp < self.size())]
#[requires(i < self.size())]
#[requires(forall (|i: usize|
    (i < self.size() && i != cur)
  ==> self.parent(i) == old.parent(i)))]
#[requires(old.grandparent(cur) == gp)]
#[requires(self.parent(cur) == gp)]
#[requires(!self.is_descendant(gp, cur))]
#[ensures(old.is_descendant(i, cur) ==>
  old.is_descendant(i, gp))]
#[ensures(self.is_descendant(i, cur) ==>
  self.is_descendant(i, gp))]
#[ensures(old.is_descendant(i, cur) ==>
  self.is_descendant(i, gp))]
#[ensures(self.find_pure(i) == old.find_pure(i))]
#[ensures(result)]
fn find_mut_lemma(
&self, old: &Self, cur: Id, gp: Id, i: usize) -> bool {
  let parent = self.parent(i);
  if self.is_root(i) {
    true
  } else {
    self.find_mut_lemma(old, cur, gp, parent)
  }
}
```

Even with greater limitations to reasoning in Prusti, defining relationships between `Id`s in `old` and `self` is enough to prove that roots remain the same during `find_mut`.

Fig. 6. Creusot vs Prusti `find_mut_lemma`

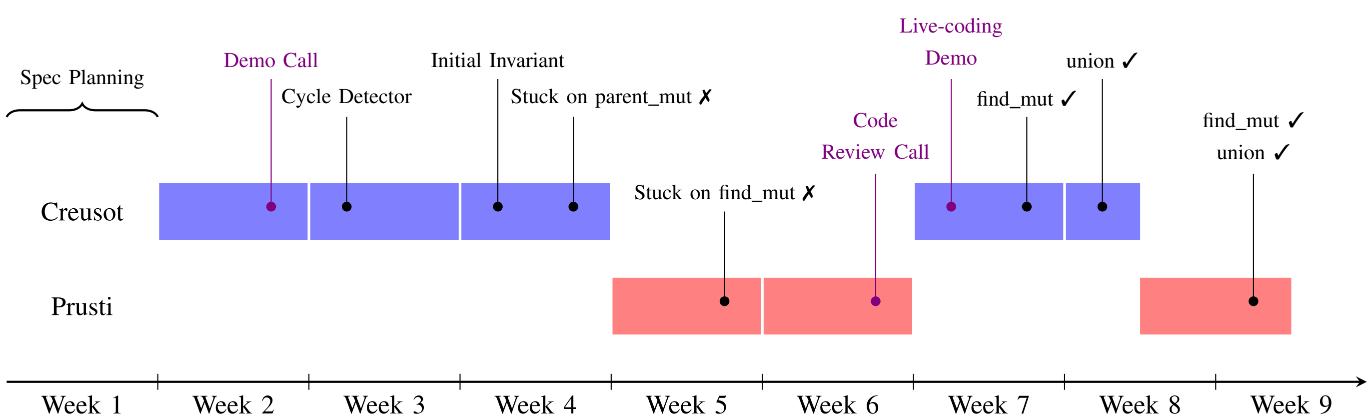

Fig. 7. The nine week verification timeline. This includes planning, learning to use the tools, interactions with tool developers, and milestones.

ability to snapshot values outside of contracts, the loop body had to be rewritten as its own function, allowing us to write preconditions and postconditions for `self` and `old`. Because we could not directly model the mutation as a setting of `cur` to `gp`, more preconditions than in Creusot are necessary to define the relation between `cur` and `gp` from `old` to `self`.

To prove equivalence of `find_pure` for a given `i`, it must be ensured that if `i` is a descendant of `cur`, then it must also be a descendant of `gp` in `old` and `self` respectively. Then, it can be ensured that if `i` was originally a descendant of `cur` in `old`, then it must still be a descendant of `gp` in `self`. Prusti can then guarantee that `i` has the same root in `old` as in `self`.

IV. Verification Timeline

In this experience, formal verification was not a linear or straightforward process. This verification was performed over nine weeks by the first author. Initially only meant to be done in Creusot, the decision to include Prusti was made in response to encountering obstacles in Creusot's logic in Week 4. Here we detail the development timeline, which is depicted in Fig. 7.

A. Strategizing (Week 1)

Creusot was chosen as it is one of the most popular verification tools for Rust. To contribute to other Rust correctness efforts, we examined the e-graphs project, *egg*, and chose to verify its union-find implementation. We initially defined our verification goals along two requirements that define a union-find:

1) All nodes must point to some other node or itself.
2) All paths along every node must not contain a cycle, terminating with a self-loop at a root.

B. Setup and Learning Specs (Week 2)

Installing dependencies for Creusot, Why3, and WhyCode—a VS Code extension made by the Creusot developers to replace Why3 IDE—took significant time. I began looking at existing projects verified with Creusot such as the SAT solver CreuSAT [10]. Creusot's use of Rust syntax in predicates and logic made it easy to understand and define my own specifications. However, there are very few works available which use Creusot, and simply examining well-defined contracts offers little insight into the process behind their design.

As WhyCode development is still in the early stages, I had trouble executing it and tried building it from source. This led me to begin talking with the main developer for Creusot, and after a video call, I understood how to use Why3 IDE and refined the predicate for goal 1.

C. Understanding Limitations (Weeks 3-4)

Towards verifying goal 2, I referenced loop-detection algorithms I was familiar with in programming. This involved tracking and ensuring the uniqueness of visited nodes, but became difficult to ensure within the SAT solver time limits in Why3.

I also tried expressing these goals as type invariants. However as a type invariant, goal 2 could not be verified in the case of `parent_mut`, which returns a mutable reference to an `Id` in the union-find. From a paper on implementing type invariants in Creusot, I learned it can only make guarantees about values within the scope of functions with *prophecies*, while Prusti's *pledges* and `after_expiry` can reason about the mutable reference outside of a function's scope [11]. Therefore, verification in Prusti was considered to address these roadblocks.

D. Switching Scenery to Prusti (Weeks 5-6)

As Prusti and Creusot's syntax share many similarities, learning the tool's keywords and porting existing contracts into Prusti was fairly straightforward. The major limitation was that an equivalent *model* type in Creusot for reasoning about vectors is not available in Prusti. All accesses to vector indices must be trusted, and `find_root` was adapted from `find_pure` as a predicate to ensure that a root could be found after traversing a path less than or equal to the length of `parents`. Setting `find_root` as an invariant condition over all nodes for goal 2 was not enough to verify the `current-grandparent` relationship in `find_mut` on its own.

At the end of week six, I was able to video call the principal investigator for Prusti. I learned that the invariant definition could not be unraveled without induction on `find_mut`. I also gained a better understanding of features not well covered in the user guide, such as quantifier triggers, which improve verification efficiency.

E. Utilizing Induction with Recursion (Weeks 7-8)

At the beginning of week seven, a live debugging session from the main Creusot developer on a similar path compression implementation helped to reform the predicate and develop a strategy for `find_mut`. I learned that incorporating a `dist` sequence and recursive helper lemmas for `find_mut` was the missing piece. This approach allowed me to verify that the mutable implementation preserved the results of `find_pure` for all nodes. In the following dats, this strategy was successfully applied to `union`, ensuring that only the root of the unioned components was modified. The union-find was fully verified with Creusot in week eight, alongside `Id`, which was easily re-implemented due to most of the logic using model types.

F. Applying Strategies Across Tools (Weeks 8-9)

Redefining the `dist` sequence as a function to align with Prusti's requirements was necessary in progressing the verification of the union-find. Similar to Creusot, both `parent_mut` and `union` in Prusti required additional recursive lemmas to complete the verification. The differences between implementations are shown in Section III.

Ultimately, understanding the limitations and capabilities of the underlying frameworks, alongside how to build up to stringer contracts—whether through additional contracts or lemma functions—was crucial to verifying this data structure. To help software engineers reach this stage more efficiently, changes to the design of these tools are necessary.

V. Discussion

While intended to be used by Rust programmers, we argue that these tools require additional design work to be used efficiently by programmers today without beforehand knowledge of formal methods. We develop four recommendations to bridge this gap.

A. Straightforward Setup and Onboarding

The first obstacle a developer meets in the formal verification process is in installation and setup. It is critical to simplify this process in order to increase adoption, which requires significant engineering effort on behalf of the tool designers [12].

Prusti streamlines the setup process with Prusti Assistant, a VS Code extension that automatically installs Prusti and related dependencies. Prusti Assistant can be configured to verify-on-save, emulating the immediate compilation feedback from popular extensions such as Rust-Analyzer [13]. Its interface is intuitive to users of modern IDEs with underlined failing code and hover-to-view details.

While the WhyCode extension is under development, it currently only displays failing contracts as they appear in Coma, Creusot's intermediate verification language (IVL). To install Why3 and other dependencies, Creusot requires `opam`, OCaml's package manager. Unlike Prusti and Viper, Creusot and Why3 have different development teams. Other tools such as Why3find [14] are compatible with Creusot, but users must manage Why3 versioning compatibility between them as Coma is in the process of integration as an official Why3 language.

The current recommended process is to compile with Creusot and launch the Why3 IDE, which opens an external executable window. From there, the user may manually select provers and strategies or have them run automatically. While the UI is antiquated to modern standards, it is able to transpose Coma logic back onto the original Rust code, re-contextualizing failing contracts for debugging.

Prusti offers a more familiar first introduction compared to Creusot. The easy installation and interface is similar to tools Rust programmers are already using. Creusot's multi-step build process can be time-consuming and use of the Why3 IDE interface provides too much detail for most cases of verification. Despite wielding more advanced capabilities, Creusot may make it harder for programmers to get started verifying code. If formal verification can be integrated smoothly into the development pipeline, more users may adopt it and gradually explore its advanced features.

B. Greater Support for Debugging Verification Failures

Rust has commendable error messages which often provide hints and suggestions to resolve the error [15]. It is significantly different from the experience of debugging formal verification logic, as contract failures often provide little guidance toward how to resolve them. It is tempting to fall into "guess and check" tactics of debugging to deduce whether logic is incorrect due to a trivial mistake or that not enough context has been presented to be verified.

Prusti's feedback is displayed minimally via red underlines in the IDE. This is helpfuls point out issues in simple cases, such as forgetting to ensure a precondition for a function when it is called inside another context. Prusti has a number of flags that can be set in `Prusti.toml` as well. The `counterexamples` flag is useful for identifying simple integer logic errors in contracts. However, when this flag is enabled, Prusti will attempt to generate counterexamples for every failing contract. This often results in nonsensical counterexamples and may even panic if the operation involves unwrapping an `Option` type, which could possibly be `None` in a counterexample.

Code with Creusot contracts must first be compiled with `cargo creusot`, which is able to catch syntax and type errors. There are several instances where error messages could be designed to provide more context. For example, attempting to use a `for` or `while` loop in `pearlite` blocks throws a parse error, but a new user might not know that loop logic can't be used in pure contexts. The error message could serve as a learning opportunity about purely functional programming. If the user accidentally appends `@` to the end of a `Int` or `Seq` type to produce a model, they get the error that `ShallowModel` is not implemented for that type. This is because `Int` and `Seq` are already model types, so a more informative error message might explain this fact and suggest to remove the `@` in Rust-like fashion.

Because Creusot exposes the Coma IVL, it provides more information on what specifically fails to verify. Such is the case for defining recursive variants to ensure function termination. These hidden conditions involve checking if a variant is strictly decreasing or is always positive, which are implied by the variant contract but only explicitly implemented in the Coma translation. For more advanced debugging in Why3, users can also manually apply tactics such as `split_vc` to further split the Coma verification goals into subgoals.

Trivial contract failures benefit from simple feedback, but sometimes a deeper look is required to understand the full scope of why a verification fails. Ideally, programmers should be able to receive immediate feedback and examine the execution of their contracts within the same tool. Programmers typically don't need to use a debugger for every error, but having access to debugging tools helps identify issues in regular code development. The same should apply to formal verification to match this expectation.

C. Pedagogical User Guides and Documentation

The user guide is meant to inform users about a tool's functionalities and keywords. While this level of depth may be adequate for completing simple proofs of safety and correctness, there is little support to help users develop strategies to tackle verifying complex properties. Without a formal methods background, Rust programmers may need additional instruction on how to construct efficient and effective contracts.

Creusot's user guide is minimal, focused on explaining syntax and features. There are a few simple examples to demonstrate contract usage, but it lacks strategies in how to use them in bigger contexts. The guide contains outdated features like the `ghost!` macro and has yet to cover features such as `DeepModel` for comparing model types. `Seq`, or logical Sequence, is mentioned briefly, but is missing some key documentation in both the guide and crate documentation among other logical types. As a result, to know which functions can be performed on `Seq`, the user has to reference its implementation in the source code. Logical data structures are powerful tools in Creusot for modeling Rust types but are obscured from new users by lack of documentation.

Prusti's guide assumes that the user has a basic understanding of Rust and contains a tutorial project verifying a singly-linked list. Though the logic of the example is simple, covering only functional-style Rust, it shows how to fully verify a data structure from start to finish using the capabilities of Prusti. The feature sections contain tips for effective use and include features still in development. There remain some features with underdeveloped explanations, such as `triggers`, where the user must turn to the Viper guide to gain a better understanding of how they work. Prusti has a well explained user guide, but lacks depth in tactics for verifying realistic Rust code.

Another tool, Verus, [16] also has a user guide, and assumes the user has established knowledge of Rust but not of formal verification. In addition to a quick-reference, the guide explains why proofs might fail, how to utilize recursion to perform induction, and how to ensure the tool is being used optimally for efficient verification. This textbook-level detail is closer to what Rust users need by explaining just enough of the backend functionality and formal methods theory to use the tool successfully.

Development teams ultimately have the deepest knowledge of their tools. Consulting with the Creusot and Prusti developers throughout this study gave us invaluable guidance that wasn't otherwise available online. If formal verification is to become more common within software engineering in Rust, this approach does not scale. Creating tutorials and video demos that showcase realistic and complex examples of the iterative process of developing and debugging contracts over time would address this gap in publicly available learning resources. It's crucial to keep learning resources updated while considering the prerequisite knowledge of the growing userbase.

D. *Easily Expressed Logic for Modeling Data*

Minimizing the complexity required to verify code is a major contributor toward tool usability. The logic that is exposed to the user can complicate verification if its abilities and limitations are not well expressed. One example of the logic programmers currently must manage independently is modeling data into types that can be interpreted in the verification layer.

Prusti can deal with Rust datatypes without need for a model type in logical contexts, allowing for contracts that are more syntactically similar to Rust code. While Prusti doesn't have a complete sequence model, currently only supporting `Int` or `Bool` type models, it wasn't necessary to verify non-trivial functions in our union-find implementation. The use of `#[extern_spec]` can be used to supplement verification for some type methods, but not in all cases such as the `push` function on vectors. Operations that are unsupported require the `#[trusted]` label, which, if misused, can compromise soundness. The more the tool must be allowed to trust functions, the more difficult it is to fully verify programs.

Creusot, on the other hand, allows programmers to cleanly derive model types for use in contracts. Once users understand how to derive a model, the data becomes more powerful in verification as it can be directly interpreted by Why3. However, complications arise as types can convert into their `ShallowModel` type but not vice versa. As a result, almost all logic must be verified through model types. If a custom type can be accurately modeled by the user, then it can be verified. Despite requiring more effort on behalf of the user to conceptualize and define models, it allows for greater expression in verification and ultimately a stronger proof of correctness.

VI. Related Work

1) Rust verification: Rust, while being a new language, has received significant interest from the verification community due to its intended use in safety-critical systems, influence from type theory in design of ownership types, and due to its relatively broad adoption as a memory-safe systems language of the future. Projects such as Oxide [17] and RustBelt [18] have built on a rich literature of separation logic [19], [20] and linear type theory [21] to formalize the behavior of Rust programs. RustBelt even mechanizes these programs using the Iris [22] separation logic in the Coq theorem prover [23]. We focus on the usability of Rust *automated deductive verifiers*.

a) Deductive Verifiers for Rust: There are relatively few deductive verifiers for Rust. Two polished tools are Prusti and Creusot, which extend the Viper [8] and Why3 [7] verification frameworks to handle Rust respectively and are directly examined in this paper. In addition there is Flux [24], which implements Liquid Types for Rust. There is also GillianRust [25], which extends the Gillian language [26] with support for Rust by using the RustBelt semantic typing paradigm. Verus [16] is a verification framework for converting (a subset of) Rust code with logical annotations (such as contracts and assertions) into corresponding SMT queries. RefinedRust [27], Aeneas [28], and Heapster [29], [30] are semi-automated tools that use automation in interactive theorem provers to discharge simple goals, while leaving more complex goals as obligations for the verification engineer.

Of these tools, Prusti, Creusot, and Verus are best suited for verifying complex properties and integrating with a traditional Rust development workflow. Flux's refinement type logic, while predictable and automated, lacks the expressiveness required to encode many realistic program properties (such as the quantified invariants necessary in our case study). The implementation of GillianRust is not publicly available and Heapster is incomplete, lacking crucial support for lifetimes. Finally, RefinedRust and Aeneas are similar to Creusot in flavor but with a bias towards enabling complex interactive proofs.

The remaining candidate tools are Prusti, Creusot, and Verus – of these, we targeted Prusti and Creusot. Our study (and recommendations) would probably translate straightforwardly to Verus, as it has a similar logic and user experience as Prusti. One extra layer of complexity is how Verus exposes recursive function fuel and SMT triggers to the verification engineer. These subtleties might pose a challenge for encoding our inductive proofs, and also might lead to more user experience design recommendations.

2) Verification case studies: Verification case studies are a popular way to demonstrate the applicability of a verification tool or technique. Notable examples include Frama-C in the aerospace [31] and the automotive industries [32], microprocessor verification using PVS [33] and Forte [34]–[36], and symbolic model checking at AWS [37]. These studies demonstrate the feasibility of formal methods, when used by teams of verification experts, to scale out to challenges in industrial environments. By contrast, our study focuses on a verifying a single complex algorithm (union-find) with an explicit goal of distilling general usability takeaways for Rust verifiers.

a) Contrastive Verification case studies: Several case studies contrast proofs using different theorem provers on the same verification problem. Vazou et al contrast LiquidHaskell and Coq on monoidal string matching [38]. Chen et al contrast Why3, Coq, and Isabelle/HOL on Tarjan's Strongly-Connected Component graph algorithm [39]. The key difference in our work is that we focus on automated deductive verifiers for Rust.

b) Union-find case studies: Union-find has seen a lot of attention as a verification benchmark for novel logics in interactive theorem provers. The Archive of Formal Proofs (AFP) in Isabelle/HOL has an implementation using a separation logic for Imperative HOL [40]. Chargueraud and Pottier use CFML and a novel ghost logic to verify functional correctness and amortized complexity [41], [42]. Guttmann replicates the verification from the AFP Isabelle/HOL effort with a novel Kleene Relation Algebra proof technique [9], [43]. In contrast to these works, we focus on automated verifiers, developing the first correctness proof for union-find in an automated deductive verifier.

3) Programming Language Usability Studies: Human-factors concerns and usability studies are an increasingly popular evaluation for programming language techniques. This can be applied to fundamental language paradigms, such as how programmers write statically-typed functional programs [44] or the design of web automation languages [45], and also to proposed extensions of existing languages. LiquidJava, which extends Java with Liquid Types, evaluated the accessibility of the refinement type annotations [46]. Glacier extends Java with immutability and the extension was also evaluated through a user study [47]. Obsidian is a smart contract language with support for *typestate*, and ran a user study on the usability of the typestate system [48]. Our work is similar in spirit, with an additional emphasis on contrasting two Rust deductive verification techniques from the perspective of a typical Rust programmer.

Most closely related to our work is Juhosova's survey of usability barriers for 35 novice Agda programmers [49], which incorporated Agda into two weeks of an undergraduate Functional Programming (FP) course. The main differences between this work and Juhosova's survey is in scope, target audience, and tooling: Juhosova examined the interactive theorem proving experience for multiple FP novices on textbook verification challenges using a proof assistant; while our study examines one experienced Rust developer on realistic and intricate verification challenges using two automated verifiers. Due to these differences, most of the design takeaways from the studies are orthogonal. However, one common takeaway from both studies is the importance of pedagogical tutorials to help verification novices build a mental model for the underlying verifier.

References

[1] I. CrowdStrike, "External technical root cause analysis — channel file 291," 2024, [Online; uploaded Aug 6, 2024]. [Online]. Available: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf

[2] S. Vaughan-Nichols, "Rust in linux: Where we are and where we're going next," 2023, [Online; uploaded Nov 14, 2023]. [Online]. Available: https://www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/

[3] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[4] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[5] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: a foundry for the deductive verification of rust programs," in *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, ser. Lecture Notes in Computer Science. Madrid, Spain: Springer Verlag, Oct. 2022. [Online]. Available: https://inria.hal.science/hal-03737878

[6] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434304

[7] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128.

[8] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.

[9] W. Guttmann, "Relation-algebraic verification of disjoint-set forests," 2024. [Online]. Available: https://arxiv.org/abs/2301.10311

[10] S. H. Skotåm, "CreuSAT, using Rust and Creusot to create the world's fastest deductively verified SAT solver." [Online]. Available: https://www.duo.uio.no/handle/10852/96757

[11] D. Stolz, "Type invariants and ghost code in rust verification with creusot," Master's thesis, Technische Universität München, 2023.

[12] A. Reid, L. Church, S. Flur, S. de Haas, M. Johnson, and B. Laurie, "Towards making formal methods normal: meeting developers where they are," *CoRR*, vol. abs/2010.16345, 2020. [Online]. Available: https://arxiv.org/abs/2010.16345

[13] Ferrous Systems and contributors, "rust-analyzer," https://rust-analyzer.github.io, 2023.

[14] L. Correnson, "Packaging proofs with Why3find," in *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, Saint-Jacut-de-la-Mer, France, Jan. 2024. [Online]. Available: https://hal.science/hal-04407129

[15] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, Aug. 2021, pp. 597–616. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton

[16] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[17] A. Weiss, O. Gierczak, D. Patterson, and A. Ahmed, "Oxide: The essence of rust," *arXiv preprint arXiv:1903.00982*, 2019.

[18] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158154

[19] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[20] P. O'Hearn, "Separation logic," *Commun. ACM*, vol. 62, no. 2, p. 86–95, jan 2019. [Online]. Available: https://doi.org/10.1145/3211968

[21] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397587900454

[22] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 637–650. [Online]. Available: https://doi.org/10.1145/2676726.2676980

[23] G. Huet, G. Kahn, and C. Paulin-Mohring, "The coq proof assistant a tutorial," *Rapport Technique*, vol. 178, 1997.

[24] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala, "Flux: Liquid types for rust," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591283

[25] S. Élie Ayoun, X. Denis, P. Maksimović, and P. Gardner, "A hybrid approach to semi-automated rust verification," 2024. [Online]. Available: https://arxiv.org/abs/2403.15122

[26] P. Maksimović, S.-É. Ayoun, J. F. Santos, and P. Gardner, "Gillian, part ii: real-world verification for javascript and c," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 2021, pp. 827–850.

[27] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer, "Refinedrust: A type system for high-assurance verification of rust programs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1115–1139, 2024.

[28] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, aug 2022. [Online]. Available: https://doi.org/10.1145/3547647

[29] P. He, E. Westbrook, B. Carmer, C. Phifer, V. Robert, K. Smeltzer, A. Ştefănescu, A. Tomb, A. Wick, M. Yacavone, and S. Zdancewic, "A type system for extracting functional specifications from memory-safe imperative programs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485512

[30] P. He, "Rely-guarantee semantics for separation-logic-based specification extraction," Ph.D. dissertation, University of Pennsylvania, 2024.

[31] R. A. B. e Silva, N. N. Arai, L. A. Burgareli, J. M. P. de Oliveira, and J. S. Pinto, "Formal verification with frama-c: A case study in the space software domain," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1163–1179, 2016.

[32] J. Hecking-Harbusch, J. Quante, and M. Schlund, "Formal runtime error detection during development in the automotive industry," in *Verification, Model Checking, and Abstract Interpretation: 25th International Conference, VMCAI 2024, London, United Kingdom, January 15–16, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 3–26. [Online]. Available: https://doi.org/10.1007/978-3-031-50524-9_1

[33] S. Miller and M. Srivas, "Formal verification of the aamp5 microprocessor: a case study in the industrial use of formal methods," in *Proceedings of 1995 IEEE Workshop on Industrial-Strength Formal Specification Techniques*, 1995, pp. 2–16.

[34] M. D. Aagaard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C.-J. H. Seger, "Formal verification of iterative algorithms in microprocessors," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 201–206. [Online]. Available: https://doi.org/10.1145/337292.337388

[35] R. Kaivola and K. Kohatsu, "Proof engineering in the large: Formal verification of pentium®4 floating-point divider," in *Correct Hardware Design and Verification Methods*, T. Margaria and T. Melham, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 196–211.

[36] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel® coretm i7 processor execution engine validation," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 414–429.

[37] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 11–20. [Online]. Available: https://doi.org/10.1145/3377813.3381347

[38] N. Vazou, L. Lampropoulos, and J. Polakow, "A tale of two provers: verifying monoidal string matching in liquid haskell and coq," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 63–74. [Online]. Available: https://doi.org/10.1145/3122955.3122963

[39] R. Chen, C. Cohen, J.-J. Levy, S. Merz, and L. Théry, "Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle," in *ITP 2019 - 10th International Conference on Interactive Theorem Proving*, J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141. Portland, United States: Schloss Dagstuhl–Leibniz-Zentrum f{'u}r Informatik, Sep. 2019, pp. 13:1 – 13:19. [Online]. Available: https://inria.hal.science/hal-02303987

[40] P. Lammich and R. Meis, "A separation logic framework for imperative hol," *Archive of Formal Proofs*, November 2012, https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

[41] A. Charguéraud and F. Pottier, "Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation," in *Interactive Theorem Proving*, C. Urban and X. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 137–153.

[42] ——, "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits," *Journal of Automated Reasoning*, vol. 62, no. 3, pp. 331–365, 2019. [Online]. Available: https://doi.org/10.1007/s10817-017-9431-7

[43] W. Guttmann, "Verifying the correctness of disjoint-set forests with kleene relation algebras," in *Relational and Algebraic Methods in Computer Science*, U. Fahrenberg, P. Jipsen, and M. Winter, Eds. Cham: Springer International Publishing, 2020, pp. 134–151.

[44] J. Lubin and S. E. Chasins, "How statically-typed functional programmers write code," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485532

[45] M. H. Fischer, G. Campagna, E. Choi, and M. S. Lam, "Diy assistant: a multi-modal end-user programmable virtual assistant," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 312–327. [Online]. Available: https://doi.org/10.1145/3453483.3454046

[46] C. Gamboa, P. Canelas, C. Timperley, and A. Fonseca, "Usability-oriented design of liquid types for java," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1520–1532.

[47] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for java," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 496–506.

[48] M. Coblenz, J. Aldrich, B. A. Myers, and J. Sunshine, "Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428200

[49] S. Juhosova, "How novices perceive interactive theorem provers," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*, ser. TyDe 2024. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://sarajuhosova.github.io/assets/files/novices/extended-abstract.pdf