Summary of the field(s) of research within which the topic of the proposed seminar lies

Over the past decade, the Rust programming language has rapidly gained traction as a safer alternative to languages such as C or C++ for low-level, security-critical programming. This success stems in large part from several key features of the language: Rust supports the low-level idioms and performance commonly associated with C or C++ while providing memory safety by construction thanks to its rich, ownership-based type system. As a result, Rust is now at the forefront of the Safe Coding methodology, with both governments and corporations advocating for a transition to Rust in order to guarantee the safety and reliability of critical software infrastructure; as an example, high-profile projects such as the Windows and Linux kernels are currently developing new features using Rust.

Despite its promises, however, using Rust is not a silver bullet. First, to ensure memory safety, Rust relies on runtime checks to determine whether accesses into arrays are in bounds; if not, programs cleanly abort (panic in Rust parlance). While better than memory vulnerabilities in C or C++, this behavior obliges programmers to manually establish the *panic-freedom* of their code. Second, while highly effective, the Rust *borrow-checker* (a key component of Rust's ownership type system) can be too restrictive in its quest to ensure that Rust code abides by the ownership-based discipline underlying memory safety. To work around these restrictions, Rust provides an *unsafe* escape hatch, allowing for more complex aliasing and memory patterns, at the cost of compile-time safety guarantees, such that programmers are trusted to ensure the safety of their code. Errors in unsafe code may compromise the guarantees Rust is aiming to provide. Last, beyond safety, Rust does not guarantee *correctness*, *reliability*, or *security*, leaving Rust code potentially susceptible to a range of errors and vulnerabilities.

To address these issues, the formal verification community recently started investigating the development of static analysis and verification tools, aiming to further raise the confidence and reliability of newly developed, security-critical Rust code. Many teams either repurposed and extended existing, well-established tools for other languages (e.g., C) to target Rust, or developed new approaches specifically targeting Rust specificities, in both cases leveraging invariants provided by the type system to simplify analysis and verification.

Initial results are promising: cryptographic libraries, microkernels, security monitors, and storage systems have been verified using Rust-based tools. In particular, code written in safe Rust *does* relieve the verification engineer of memory-based proof obligations, and productivity gains have indeed been observed. However, these initial results either target specific application domains (e.g. a cryptographic function that is a pure function of its inputs), or systems that do not necessarily represent the reality of industrial Rust code.

In effect, if we want to scale up these verification results to larger, real-world Rust systems, notably those using a mixture of unsafe code, interior mutability (a.k.a. dynamic borrow-checking), concurrency and asynchronous tasks, several scientific challenges loom large, for which the existing techniques and tools propose no definitive answer. Hence, we propose to bring developers of these techniques and tools together to address these common challenges, both from a scientific and an engineering standpoint.

Description of the seminar topics

This seminar will focus on the theory and practice of Rust formal analysis and verification. Specifically, we will address three intertwined problems: first, the aspects of the theoretical foundation of the Rust language crucial to analysis and verification; second, techniques for the analysis and verification of Rust programs; and third, the engineering practices and tool ecosystem that implement those techniques.

Rust foundations are currently shaky, in that many corners of the language are ill-defined. While foundational work (notably, RustBelt) has succeeded in shoring up the theoretical basis of the language, many corners of the language remain unexplored (from a foundational perspective), such as its trait system. Furthermore, several attempts at improving the Rust language have stalled, either because of the lack of confidence in how they interact with the rest of the language in terms of soundness, or simply because enunciating their theory cleanly and neatly remains difficult, for lack of a good reference semantics for the language. Examples include the view types proposal (never implemented), ongoing rewrites of the borrow checker (still not ready to be merged), or a proposal for a new trait system for Rust (whose behavior differs from the previous implementation, but which, lacking a formal model, cannot be conclusively determined to be the correct behavior).

It therefore remains difficult to confidently develop analysis and verification techniques, when the semantics of Rust are still undergoing such debate. We will not aim to produce a comprehensive specification or dwell on the theoretical aspects of the language's design. Instead, we will focus on (a) identifying the parts of the language where clear specifications would most benefit analysis and verification, and (b) establishing a consensus on what we as a community believe the current semantics for those parts are or what they should be.

Next, many analysis and verification techniques leverage Rust's unique ownership discipline to simplify reasoning, or to make reasoning tools more efficient. As a second axis, an important focus of our seminar is the extension of these techniques to the full Rust language. Moreover, we propose to focus on a broader re-thinking of existing techniques in the context of Rust; how Rust's ownership impacts, e.g., resource analysis (as exemplified by RaRust), property-based testing, and many other techniques. While existing program verification frameworks do leverage

ownership as a means to automate or simplify reasoning, we believe there is more potential to be tapped in order to make the analysis of Rust programs generally more efficient thanks to the ownership guarantees offered by the language.

As a third axis, we propose to focus specifically on tooling. There is a considerable amount of design and implementation work that needs to be addressed in this area, notably: - What are the theoretical and practical challenges of integrating the infrastructure necessary for analysis and verification into the Rust compiler? Could the community standardize on one framework (such as the Charon project or Place Capability Graphs), and if not, what are the roadblocks? - How can we design a unified Rust program specification language that enables interoperability between different tools and allows the community to develop common verified libraries? How could such a language simultaneously serve a full spectrum of testing, analysis, and verification tools?

Finally, as a transversal axis, we propose to put all of the ideas above into practice by identifying language features, implementation idioms, and program properties that are not sufficiently supported by state-of-the-art techniques and tools, resulting in a list of action items for the community to steer further research and assess progress.

We remark that browsing the Dagstuhl archives, we could not find a single seminar focusing on Rust specifically; given the excitement about Rust in the research community, we think it is timely and important to gather researchers on this particular topic as soon as possible!

Composition of the organizing team

Our organizing team represents a broad cross-section of the Rust analysis community, with four different tools represented (Verus, Prusti, Charon, and Aeneas), a variety of seniority levels, geographical diversity (two US and two European organizers), and one industry member who has been involved in driving verified Rust code into production.

The variety of tools also reflects a variety of approaches: - the Verus Rust verifier emphasizes an SMT-first approach while leveraging Rust's ownership to efficiently encode Rust programs to SMT - the Prusti Rust verifier relies on the Viper infrastructure, where Rust's ownership discipline guides the application of rules in separation logic - the Aeneas Rust verifier relies on backwards functions to translate Rust programs into a pure equivalent that is sent to an interactive prover, with a particular emphasis on Lean.

We believe this variety will prevent one particular style from being over-represented, and will encourage a broad discussion of possible approaches.

Finally, two of the organizers have significant experience with the Charon Rust analysis toolkit, which is currently used by several tools, notably Aeneas (Rust

verifier), Eurydice (Rust to C compiler), RaRust (Rust resource analysis), and several more.

Expected results (outcome) of the seminar

We expect several outcomes, in no particular order: - new lines of work towards clarifying the semantics of Rust and ultimately improving the language specification; - a call to arms for formalization experts, pointing them at critical, understudied places in the Rust language; - a survey of the state of Rust verification and analysis tools, which will help establish a shared understanding of the advantages and disadvantages of existing techniques; - new directions for combining the best ideas from existing analysis techniques; - a taxonomy of Rust language features, coding idioms, and program properties to guide the further development of analysis and verification techniques; - a benchmarking suite based on this taxonomy, including a series of verification challenges of increasing difficulty for analysis and verification tools, fostering comparison and development of tooling; - new ideas for increasing adoption of Rust-related tools and techniques by the Rust community; - a systematic approach for interfacing with the Rust compiler in order to facilitate the development of reasoning tools, along with concrete tasks to implement it; and - a shared common infrastructure to develop new tools for Rust.

Ideas about the structure of the seminar

Day 1, morning: Quick presentations from all participants (10 minutes each) about their tools, scope, expressiveness, and common difficulties interacting with Rust

All working groups come with a summary to the broader audience at the end of each session.

Day 1, afternoon: Working Groups

- Analysis and verification challenges (language features, idioms, program properies)
- Rust semantics with a focus on unclear concepts
- A common specification language for Rust

Day 2, morning: Working groups

- Benchmark suite
- Focus groups for new analysis areas
- Discussion of well-established C verification tools, and connections/applicability to Rust
- Interacting with the Rust compiler, and common engineering challenges

Day 2, afternooon (relying on topics prepared ahead of time by participants):

- "VerifyThis"-like challenge for deductive verification tools
- "AnalyzeThat"-like challenge for static analysis tools, using benchmarks established in the morning

Day 3, morning:

- Debrief of differences, strengths and weaknesses of existing tools, based on day 2 challenge

Day 3, afternoon (parallel sessions / focus groups):

- Hacking on common engineering infrastructure
- Prototypes of novel analyses
- Larger benchmarking suite and proof ladder of increasiningly challenging examples for different tools

Difference to other Dagstuhl Seminars within related topics, in particular those within the same topic

This seminar proposes to focus exclusively on Rust; looking through the Dagstuhl seminar archives, it appears that no other seminar has focused on Rust before.

The following seminars are loosely related:

- 25412 Sound Static Program Analysis in Modern Software Engineering. This seminar focused on static analyses at large, and how they interact with software engineering processes. A particular emphasis was put on Web Applications. We instead focus on Rust specifically and put more emphasis on verification (deductive and model checking), besides program analysis.
- 26111 Formal Analysis and Verification in Quantum Programming Languages. Verification is also something we aim for, but not in the context of quantum languages.
- 26071 Behavioural Types for Resilience. Types are a language-based technique, but in our context, it is unlikely that a new type system will be devised for Rust.

Conferences and research projects within related topics, and why it is justified to hold such a seminar at Dagstuhl

There are many developer-facing Rust conferences, such as RustConf, RustNation, and RustParis. However, these conferences focus on language discussions from the perspective of language designers, and not from the perspective of formal analysis specialists. Notably, not a single talk at RustConf mentioned the word "formal".

On the research side, there is RustVerify, a workshop with no proceedings, traditionally associated with ETAPS. However, RustVerify follows a traditional conference format, meaning that the presentations are very static, and leave few opportunities for interactions in small groups with measurable outcomes.

We envision that Dagstuhl will provide the necessary environment to get engagement from both sides of this community (researchers and industrial practitioners) and create new collaborations.