Developing IDE Support for a Rust Verifier

Bachelor's Thesis

Julian Dunskus
Supervised by Federico Poli & Vytautas Astrauskas
under Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich
Zurich, Switzerland

August 13, 2020

Abstract

Verifiers enable programmers to check their code for full functional correctness, ensuring software robustness and reliability. Prusti is a verifier for the Rust programming language aiming to make verification more accessible. However, there were some issues with Prusti inhibiting widespread adoption, several of which we address in this project. The IDE extension, aiming to abstract the details of working with Prusti, required manual setup and otherwise offered a mediocre first-time experience. Further, Prusti would ideally just be a part of the user's build workflow, automatically running during compilation, but its execution times worked against that, taking up to ten seconds for even the most trivial of programs. Through the addition of a server mode, this project makes a big step in terms of reducing runtime, reusing components across verification operations in order to eliminate startup delays and enable more optimization through Java's just-in-time compilation. IDE integration is also improved, lowering the barrier to entry by easing the initial installation and helping retention by ensuring a visual presence, in addition to benefitting from the aforementioned server-based improvements.

Contents

1 Introduction

It is important to confirm functional correctness of programs in order to create robust and reliable systems. Over time, programming languages are coming ever closer to this goal, with powerful, modern languages like Rust [1] allowing programmers to express their intents very explicitly, enabling the compiler to verify static correctness and perform optimizations. However, as much as the landscape has changed, compilers are as yet insufficient to prove full functional correctness of one's code. This is where verifiers come in: they can analyze code in detail, e.g. performing symbolic execution, simulating program execution to identify problems with inputs the programmer may not have thought about. Verifiers empower you to statically verify otherwise inexpressible[1] properties like array population or range bounds on numbers, ensuring software reliability in a tight feedback loop.

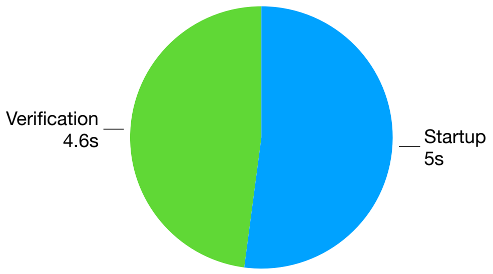
One such verifier is Prusti [2], a Rust verifier developed by the Programming Methodology Group at ETH Zurich. Prusti aims to make verification available to regular programmers, by being readily applicable to their existing code as well as offering user-friendly integration with their IDE (integrated development environment). By offering powerful yet easy-to-use verification tools, we hope to increase adoption of verification, making the software landscape more robust. Ideally, the experience of verifying your code would be a natural and unnoticeable addition to regular compilation, but Prusti currently falls short of that goal. The core goal of this thesis is to bring Prusti closer to that ideal experience of seamless integration by addressing various shortcomings, as laid out in the next few paragraphs.

To verify programs, Prusti uses the Viper [3] verification infrastructure, which encompasses an intermediate language as well as a back end to verify code written in that language. Prusti translates its compiled Rust code input to a Viper program and verifies that using the Viper back end, then associates the results with the original Rust source. It can verify `assert!(...)` assertions in code as well as preconditions, postconditions, and invariants on functions, specified via Rust's attribute system. Prusti does all this by running as a Rust compiler (`rustc`) plugin, which makes it easy for users to run it during compilation, although this poses some restrictions on input and output.

Prusti's execution times can get rather long (taking several minutes for complex algorithms), by virtue of the computationally-intensive verification techniques employed. However, even for simple programs, verification often takes upwards of 10

[1]barring avant-garde type systems like Idris' (`https://www.idris-lang.org`) dependent types

seconds, which can feel unresponsive and might put people off from using Prusti. If we take a look at what is happening in this duration, we can see that only around half the time is spent on actual verification, while the other half is spent starting up the Java Virtual Machine (JVM) and initializing the Viper verifier (Figure 1). This presents a great opportunity for optimization: if we can keep an initialized JVM and verifier running, we can shave off roughly five seconds from Prusti runtimes across the board, which is especially significant for simple programs. In order to achieve this, we split off the part of Prusti that manages the JVM and Viper verifier into a server, which can keep running continuously while the compiler plugin runs multiple times, simply acting as a client and using the server for its actual verification.



```
fn main() {
    assert!(true);
}
```

Figure 1: How much time is spent on the two main parts of a verification operation, on a trivially correct program (on the right).

Further improvements to Prusti would benefit from insight into its performance, allowing us to identify its performance bottlenecks, which calls for profiling. Before this project, profiling Prusti was very difficult because the interaction with Viper used the Java Native Interface (JNI). This interaction confused the employed profiling tools and made the results hard to evaluate. Fortunately, our approach to the above problem of reusing Viper verifiers results in the JNI-interfacing part being split off (as the server), allowing us to easily profile our front end running as a client.

Prusti Assistant is a plugin for Microsoft Visual Studio Code [4] (VS Code) that allows users to verify their code using Prusti without using the command-line interface. The new client-server architecture for Prusti meshes naturally with Prusti Assistant: Prusti Assistant can launch and manage a running Prusti server, simply running client instances for the verification requests. Thus, the Prusti execution time improvements incurred above will translate very well to the IDE experience with Prusti Assistant.

In addition to this, the implementation of Prusti Assistant was previously rather minimal, leaving a number of user experience improvements to be desired.

Some cursory usage of Prusti Assistant (in its state before this paper) brings up several usability concerns, e.g.: a lack of progress indication while it is installing its dependencies, which involves a significant download[2]; a lack of discoverability, requiring users to either browse through the settings to enable "Verify on Save" or manually run the verify command from VS Code's command palette. This project addresses several common concerns like that, aiming to enhance the user experience and first-time setup of Prusti Assistant in various ways. The impact of these enhancements is hard to measure numerically (large-scale quantitative usability testing notwithstanding), and currently there are not many people using Prusti Assistant regularly and providing feedback. However, we trust that the enhancements' value is evident simply from our descriptions thereof, as we see them as natural additions to the plugin, and they have already proven useful to us in testing other components.

The main contributions of this project are as follows:

- We added a server mode to Prusti, allowing users to forgo startup times and benefit from JVM optimizations. This also improves the Prusti development experience by reducing latency for its test suite.
- We opened an avenue toward future performance improvements by addressing issues causing common profiling tools to fail on Prusti.
- We improved Prusti Assistant's integration with VS Code, reducing barrier to entry and working to improve long-term retention.
- We extracted reusable components and systems to a Node module useful for any VS Code extension that performs verification.

The next section provides a high-level description of our approach to these various opportunities for improvement. The following implementation section explains in detail the decisions we made and our reasoning behind them, as well as covering some difficulties we encountered. In the evaluation section, we critically examine our changes, evaluating the actual value gained from them. Finally, the conclusion tersely summarizes our findings, while the future work section lays out areas with potential for future improvements.

[2]e.g. 70.1 MB for the macOS version, as of 2020-08-03

2 Approach

2.1 Execution Times

2.1.1 Prusti Server Mode

As outlined in the introduction, our main approach to reducing Prusti execution times is to avoid repeated setup logic by reusing the environment across multiple verification operations. In order to achieve this, Prusti has gained a "server mode", i.e. a way to run only the backend, keeping around a running Java Virtual Machine (JVM) and managing initialized instances of the Viper verifiers Prusti relies on for the actual verification. When given a server address, Prusti now functions as a client, only encoding the Rust program to a Viper program and sending it to the server The server in turn verifies that using the Viper backend, and sends back the results, which the client outputs as before. In essence, we simply replaced the part of the client that performs the actual verification logic with a component that either does just that or sends its input (the Viper program) to a server and returns the response as output.

The communication between client and server is implemented using the REST (Representational State Transfer) architecture, which is widespread for use cases like these. The server provides an endpoint for verification, to which the client can send (using a REST `POST` request) a verification request. This request contains the Viper Program encoded from the Rust program, as well as some configuration for the verification back end, like what arguments to instantiate the Viper verifier with. The server looks at the configuration and checks if it already has an initialized Viper verifier available which matches that configuration, and otherwise instantiates a new one. The Viper verifier is used to perform the verification, after which it is returned to storage, which consists of a cache with a fixed size, automatically evicting the least-recently-used verifier instances to make space for more recent ones. Finally, the server takes the produced verification result—which expresses whether the verification succeeded or, if it failed, what caused it to fail—and encodes it into a response, which it then sends back to the client.

2.1.2 Profiling

There were attempts in the past to run profiling tools on Prusti, but those had issues due to incompatibility between those tools and the way we are using the Java Native Interface. However, now that the Java-interacting logic can be relegated to a server, we can simply start up a server and profile Prusti running as a client, not requiring any direct interaction with Java. This decoupling newly allows us to run common profiling tools, at least on the client-side logic, i.e. mostly the encoding process from Rust to Viper. This gives us valuable insights into Prusti's performance and allows future work to efficiently identify possible optimizations, further improving execution times.

2.2 IDE Integration

Prusti Assistant benefits particularly from the server mode, while also automating the manual steps involved in managing a running Prusti server and passing its address to clients. On load, once it is ensured that Prusti is installed, the extension launches a server, noting down its port, which it then uses for verification tasks. Alternatively, users can provide an external server address, in which case the extension does not launch its own server, instead simply communicating with the one at the provided address. The latter approach also has the benefit of offloading some work off the user's machine, as well as letting multiple users verify using the same central server. Prusti Assistant also responds to crashes of its managed server or connection failures to the external server by notifying the user and, if applicable, providing actions to fix it, like offering to restart the managed server. All in all, Prusti Assistant automatically uses a Prusti server, letting everyone benefit from the improved execution times without requiring any setup, while still offering customization points.

Previously, in order to use Prusti Assistant, you had to manually install a specific nightly build of Rust (which Prusti relies on for its interaction with the compiler). Automating this process is relatively straightforward, as the `rustup` utility—the de-facto way of installing Rust—produces easily-parsed output and lets us check for and install the correct Rust version with just two commands. Since this installation does take a bit of time and disk space, as well as affecting (if only additively) the user's environment, Prusti Assistant asks users who are missing the required Rust version if they would like to install it automatically, needing just one button press. This lowers the barrier to entry for people using

Prusti Assistant for the first time, automating the last part of the installation process that still required manual intervention.

Prusti Assistant manages its own installation of Prusti, and so another enhancement is the addition of a dependency management system. Previously, the process of the extension installing Prusti was very opaque—there was no user-facing indication of progress until it had completed, and there was no way to customize it. In order to cleanly implement progress reporting, and extract some useful components in the process, we designed a modular dependency installation system with built-in progress reporting, encapsulated in the `vs-verification-toolbox` Node [5] module [6]. This system allows you to define your dependencies and their installation steps in a declarative way, using a variety of simple predefined components, e.g. downloading a file or unarchiving a zip archive, adding completely custom steps if necessary. Each component provides a way to perform its task, reporting back its progress as it works on that. These components compose easily thanks to the sequence component, which can run any number of steps in sequence, collecting the progress across them, which it reports in turn. All this comes together with VS Code's built-in way to display a progress bar in a floating dialog, allowing the user to watch the installation progress through its steps, also seeing a description of what is happening at any point.

Additionally, this dependency management system provides an API for specifying different build channels, e.g. stable and nightly. Prusti does not yet offer stable builds (the only download is nightly), but once it does, that will be a trivial addition to the extension. What the extension does already allow, though, is the option of using a local build of Prusti—this has proved invaluable for Prusti development and integration.

Moreover, we improved the status bar integration in several ways. It now displays a button to quickly verify the currently-open file, aiding discoverability of Prusti Assistant's functionality and providing a mouse-only way to invoke Prusti. Various pieces of text shown in the status bar are now accompanied by glyphs, providing e.g. a spinner animation during verification or an intuitive visual indication of the verification result (like a checkmark or a cross) once complete.

Lastly, we made some relatively minor changes:

- We discovered an already-implemented but not user-visible setting to verify the entire workspace rather than just one file, which we now surface to the user.

- We restructured several parts of the codebase, encapsulating pieces of logic in their own modules and conforming to best practices like TypeScript's strict mode.

3 Implementation

3.1 Prusti Server Mode

There is a large design space of possible approaches to implementing Prusti's server mode. Most basically, we had to decide where to split client and server, for which Prusti's layered architecture presented multiple options. Following this, we had to choose a communication protocol and encoding to facilitate this split.

3.1.1 Server Architecture

In designing the server mode, we had to decide what parts of Prusti should be part of the server and what should still happen on the client. Considering the overall structure of the Prusti project (Figure 2), there already several options there:

- **Having Prusti Assistant communicate directly with the Prusti server:** This would require moving compilation to the server, which not only increases latency when using a remote server but also necessitates reproducing everything necessary to build the code, which is not realistic. In order to benefit from compiling on the server, it would also have to somehow run the Rust compiler as a server, which `rustc` does not support yet.
- **Splitting off a part within Prusti:** This approach encapsulates all the server interaction within Prusti, making client-server communication an implementation detail. As communication occurs entirely within Rust, this gives us the biggest variety of options for data transfer. Further, it lets us split off exactly the logic we want in the server, and no more.

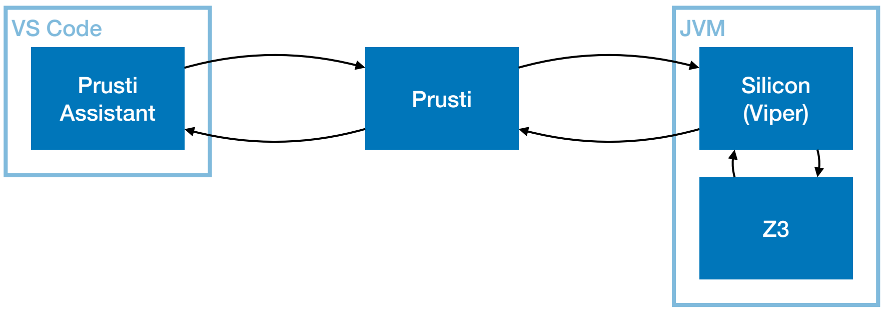


Figure 2: An overview of the Prusti project's layered structure.

- **Using the Viper server from Prusti:** The Viper project itself already offers a server, which Prusti could use for its verification. However, this server would have to be extended for our use case, because in its current form, we could not track the relation between the encoded Viper program and the original Rust code, which is essential for reporting back to users exactly where their program failed verification. This, and other helpful extensions like it, would create a lot of work and move the focus of this project from Prusti to Viper. Perhaps most importantly, this approach would require a well-defined API which is always kept in sync between Prusti and Viper, creating a large maintenance burden. Finally, we would not be able to use any of the functions Viper provides for working with its ASTs on the client, since initializing a separate JVM would defeat the whole purpose of this split.

We ended up choosing the second option, specifically opting to leave compilation and encoding to the client. We decided to specify the server such that it contains the parts of Prusti that most benefit from reuse, i.e. the initialized JVM and Viper verifiers, leaving the compilation to the client. In its overall process, Prusti receives the compiled Rust program in form of Rust's mid-level intermediate representation (MIR), which it in turn encodes to a Viper program, represented by way of an AST (Abstract Syntax Tree), hereafter referred to as the Viper Intermediate Representation (VIR). While it might be interesting to perform the MIR-to-VIR encoding on the server, the MIR is not stable, increasing effort required to update our `rustc` version. Additionally, because we do not own MIR, it would be difficult to serialize and deserialize—for VIR, we can just derive almost all serialization logic using a popular library. Using an AST rather than raw source code, in addition to being easier to create and transform programmatically, also allows us to attach information to its nodes, which lets us encode the corresponding source locations in the original Rust code where applicable. Taking all this into account lead us to choosing VIR for this task, resulting in a meaningful separation of concerns: Only the client interacts with `rustc`, and only the server interacts with Java.

A restriction that heavily influenced our options in client-server communication frameworks is that we were limited to an older version of Rust in our implementation, specifically a nightly build from June 2018. This is because the API `rustc` provides for plugins (like Prusti) is still experimental and thus only available in nightly builds, which locks us into using that same version for all the code we write and verify. There have been efforts to update this, which is an elaborate undertaking, and the branch for this version update was recently (July 27, 2020) merged into master. Unfortunately, that was too late for the purposes of this thesis, so we had to work under the aforementioned constraints. The main detrimental effect of using this outdated version is that it was much harder to add and update dependencies, since they would often use language features not available yet, so we often had to limit ourselves to outdated versions of our dependencies in order to use them at all.

For our communication, we initially looked at RPC (Remote Procedure Call) frameworks, since that is all we really need. After considering our options, we decided to try Google's Tarpc library [7], which looked nice at first. However, for the reasons listed above, we were forced to use an older version, which resulted in several issues. This version of tarpc was insufficiently documented, while simultaneously making it hard to find undocumented information through its extensive use of macros. Additionally, it would be harder to update our code using Tarpc to the new compiler version because it had changed so much in those 2 years. Further, it did not at all lend itself to customization, making it all but impossible to adjust what we wanted to. In the end, we decided to instead use HTTP libraries, since despite technically operating at a lower level, that ecosystem looked more refined and provided us with better options, especially for our compiler version.

In its final form, the Prusti server runs as an HTTP server, providing a `POST` (REST) endpoint for verification. The basic HTTP interaction of the server and client rely on the Warp [8] and Reqwest [9] libraries, respectively. Warp provides a builder-pattern API for defining provided endpoints and their behavior. At the same time, Warp also provides the many opportunities for customization Tarpc lacked, e.g. allowing us to configure how many threads can concurrently work on processing requests. Reqwest, in turn, makes it easy to access those endpoints from the client by similarly providing a builder API. If we ever need to make our client asynchronous (rather than blocking until the request is complete), Reqwest has that covered too. In summary, these libraries are expressive (pretty and readable), while also allowing the customization we need and ensuring robustness across compiler version upgrades.

When it comes to the encoding used, the server actually provides its verification endpoints in two variants, using JSON (JavaScript Object Notation) or Bincode [10] as encoding. JSON has the advantage of being human-readable and thus easier to debug, while Bincode minimizes encoding overhead (by simply not labeling values, among other things), resulting in an efficient encoding both in speed and size. The encoding process is handled by the Serde [11] library, which lets us avoid writing boilerplate by automatically deriving the implementations for de-/serializing our data structures, only customizing the encoding where needed. Serde also has the benefit of being encoding-agnostic, allowing us to easily support both JSON and Bincode without needing to cover both encodings in all our data structures. This enabled us to simply offer endpoints with both encodings, leaving it up to the client how it wants to encode its request, which will help with tracing when something does go wrong.

The Prusti server does not need to interact with `rustc`, so we decided to create a new executable for it, which does not require the `rustc` libraries to be available. However, isolating dependencies like that proved to be a bit difficult by virtue of our limited module setup: the server needed to use parts of a module that linked against `rustc`. Specifically, that module contained our representation of the encoded Viper program, by way of an AST (Abstract Syntax Tree), hereafter referred to as the Viper Intermediate Representation (VIR). Evidently, both the server and the client needed access to the VIR in order to communicate the programs, so the module it was in should be part of both. As such, we decided to create a new module for things common to all parts of Prusti, which could also contain some utilities, but expressly did not depend on `rustc` nor Java: the `prusti-common` module. Moving the VIR definitions to this new module was a delicate task, since it affected so much of the codebase—we even decided to pause all development for the duration of this move, so as to avoid excessive merge conflicts. Fortunately, all went well in that process and the new `prusti-common` module now correctly provides the VIR as well as some minor utilities to all parts of Prusti, also giving us a canonical place for such things in future.

Finally, we identified many places across the codebase using a near-identical, copy-pasted timing construct in order to print runtimes for sections of Prusti's execution. This was exactly the kind of use case `prusti-common` could cover, so we experimented with different approaches to a reusable timing API. A simple method call is nice because it is self-contained, ensuring that starting a timer always results in it later stopping and logging its duration. However, this proved to be rather restrictive and would have required significant code changes in several places, mostly because all the output generated in the timed code section would have to be explicitly returned and then stored. Instead, we investigated a macro-based approach: Macros would allow us to superficially look and work like method calls, while allowing us to work around some limitations by running the provided statements on the top level. Unfortunately, this had some issues too: it was harder to understand, especially when something went wrong, and had to take individual statements in order to place them at the top level, which did not feel right at all. Finally, we settled on an object-based design, where one creates a `Stopwatch` object with a name for the section to time, after which one finalizes the section, logging its name and duration to the console, in one of three ways: dropping it or calling `finish` simply ends the current section and consumes the instance; calling `start_next` with a new section name ends the current section and immediately starts timing a new one, allowing for a detailed breakdown with little effort.

3.1.2 Server Integration

Integrating the server was of course the point of the above efforts, and this exercise also made us aware of some important issues. There are two places where we integrated the new server mode: Prusti's own unit test suite and Prusti Assistant, the former of which was much easier than the latter. One obvious difficulty of integrating with Prusti Assistant is that we had to interact with our Rust programs from within another language, specifically TypeScript. An unforeseen challenge, however, was with killing the server process, as the Prusti architecture involves spawning subprocesses. Finally, we needed some way to automatically choose a free port, so as to not collide with other running applications with open ports. This section describes our efforts to integrate Prusti and overcome those challenges in more detail.

Prusti sports an extensive unit testing suite, executing its verification process with a wide variety of examples, which typically takes over an hour to complete. This presents a great opportunity for the new server mode: if we simply keep one server instance around during all these tests, we can shave away a big portion of our setup times for each test run. Indeed, using the server for our test suite proved to be as simple as adding one method call to run a server off-thread and providing its address to the client instances via an environment variable. The benefits are undeniable, almost cutting the runtime of our complete test suite in half, which also reduces response times from continuous integration.

Another place where the server mode really shines is the IDE integration, which provided the main motivation behind all this in the first place. Prusti Assistant already handles all interaction with Prusti for the user, so it was a natural extension to have it manage the server too, rather than requiring one to be launched through other means. Upon load, Prusti Assistant launches the server (once it has ensured that everything is installed and ready to go), providing a port of zero. This zero port has a special meaning, delegating the choice of port to the operating system, which then assigns it a free port. The server takes note of the port it gets assigned and prints it to standard output (`stdout`), which Prusti Assistant parses and proceeds to use for client instances. Because logging behavior is often seen as an implementation detail, we made sure to provide a unit test for this flow, ensuring that the first line printed to `stdout` does indeed express the assigned port in the decided format. Of course, this is also advantageous to users launching the server themselves, as they can just ask the OS for a free port and see what the server receives. Thus, we have constructed a robust way to start the server on a free port (as provided by the OS) while being able to access this information in Prusti Assistant.

Just like the regular Prusti executable, the server is actually two executables: a driver that performs the actual functionality, and a launcher that simply locates the libraries the driver needs and launches it while providing those, so the user does not have to explicitly link those libraries in order to run Prusti. However, this design creates some difficulties for integration tests and IDE integration, in that it is unreasonably hard to shut down the server if you did not directly launch the driver, often resulting in an orphaned driver process with no easy way to kill it. This is especially unfortunate for Prusti Assistant, since it needs to shut down the server in various circumstances, e.g. when the configuration changes, requiring a server restart, or when the user explicitly requests a restart. Effectively, there is currently no way to cleanly shut down the Prusti server; at least exiting VS Code recursively kills any child processes Prusti Assistant spawned, as long as they were not orphaned. Luckily, even an orphaned server instance, while undeniably displeasing, mostly just takes up some RAM, which can even be swapped to disk; the impact on other system resources is negligible. Furthermore, a major problem would arise if one of the subprocesses Viper spawns, e.g. a Z3 instance, gets stuck in an infinite loop. This would clearly have a significant performance impact and hence pose a major problem, although it thankfully appears to be rare, as it has never come up in our usage. We did not get around to addressing this issue yet, but it is surely an important future addition and should be treated with priority.

In summary, integrating the server into both Prusti's test suite and Prusti Assistant proved very beneficial, and we were able to address most of the issues. We found a clean solution to finding a free port, where we just delegate this task to the operating system. However, the issue of managing the driver subprocess unfortunately remains unresolved.

3.2 Dependency Management

In order to ensure a smooth user experience, Prusti Assistant should automatically manage dependencies for the user, resulting in a simple first time setup and update process. The main dependency is Prusti itself, which comes packaged as a downloadable ZIP, regularly exported from the current master branch. This download also packages Viper and Z3, Prusti's own dependencies, but there are several other dependencies it can or should include, like the Java Runtime Environment or the necessary specific Rust version, the latter of which we addressed. Further, we revised the dependency management process, making it more testable and structured. We extracted this system, as well as several smaller generally-useful utilities, to a separate package which is bound to be useful for other verifier IDE extensions like Prusti.

3.2.1 VS Verification Toolbox

It was important to us to build up reusable systems and components wherever possible, rather than hardcoding specific use cases. To that end, we created the `vs-verification-toolbox` Node module, which provides a place for utilities common to not only Prusti Assistant but also other IDE integration efforts for verifiers, such as:

- Abstraction over platforms: where previously one had to use error-prone string comparisons, the toolbox provides a simple enumeration listing all three major operating systems, naturally lending itself to exhaustive `switch` statements.
- Filesystem interaction, another place where we could shave off boilerplate and improve ergonomics: We added a `Location` class, which wraps a filesystem path, providing easy-to-chain, discoverable ways to navigate through file hierarchies, create or delete files, or even append the platform-dependent executable extension. This has the further upside of clarifying the type used for paths, separating them from unspecific strings.
- A wrapper for VS Code's progress-reporting API, working around some design flaws, like the fact that it only shows a progress bar when shown as a notification, which is likely to confuse users. Further, it has a rather unusual design of expecting relative progress numbers since the last report rather than an absolute number, which proved difficult to work with reliably.

Thus, the toolbox hopes to smooth out the bumps we encountered in our work by making more intuitive design choices and providing ergonomic ways to achieve things that would otherwise require more boilerplate code.

Applying this same philosophy of reusability to dependency management, we examined what steps installing dependencies can consist of and how we might modularize that process. We ended up with a surprisingly simple system, where the installation process for a dependency is defined by composing so-called installers, expressed by the `DependencyInstaller` interface. This interface only requires a single method named `install`, which takes a filesystem location as input and asynchronously returns a new location. Another input to this method is a boolean deciding whether the installer should perform its work even if it already has, effectively updating that part. Finally, the method takes a progress listener, which it is expected to call repeatedly to provide progress updates, passing in a number from zero to one representing its progress and a description of what it is doing. The beauty of this system is in its composability, which allows for basic components to be combined to express any installation process.

In order to further illustrate this installation system, these are some of the basic components it provides:

- `FileDownloader`, which takes a file URL and downloads it to the folder passed as its input location, returning that file's location.
- `ZipExtractor`, which expects a ZIP file at its input location and proceeds to extract it to a new folder within the same enclosing folder.
- `LocalReference`, which simply returns a predefined filesystem location, e.g. one set by the user in the settings.
- `InstallerSequence`, which chains multiple installers together, piping each installer's output into the next as input. It wraps its own progress listener in calling its children, rescaling their progress to the corresponding range within itself.

A big advantage to splitting everything up into basic units like these is that they are easy to understand and test, improving maintainability and robustness, while still providing insightful feedback to users as the task progresses. By combining these provided installers, even a complex installation processes can be defined in a simple, declarative way, all the while providing ample opportunity for customization where necessary.

3.2.2 Automatic Rust Version Installation

The canonical way to manage Rust versions is the `rustup` command-line utility, so this is what we rely on to ensure the correct version is installed. In detail, we call `rustup list` to get a list of all installed versions, then parse its output to check for the version we need. If the check succeeds, nothing else needs to happen, but if it fails, we display a dialog to the user alerting them to the issue, also offering to install the version in question with just a single button press. To install the new version, we simply run `rustup toolchain install`, passing in the desired version, and asynchronously wait for it to complete. In order to avoid hardcoding specific versions of rust, we kept things flexible by simply including the `rust-toolchain` file from the root of the Prusti source tree in the distributable ZIPs. This file states which build of Rust we are using for Prusti and thus need to install in Prusti Assistant. All in all, this turned out to be easy to automate and an easy way to streamline the first-time setup experience for new users.

3.2.3 Settings Change Detection

Prusti Assistant offers several customization points to the user in its section within the VS Code settings. However, simply changing a setting does not do anything by itself—users would have to realize nothing happened and think to restart VS Code in order to apply the changes. To improve this process, we now subscribe to the config change event which VS Code offers. This allows us to notice when users have changed our configuration, at which point we identify exactly what settings that change affected and automatically update affected components. For example, when the user changes the build channel, we automatically ensure the corresponding version is installed, downloading it if necessary. By automatically handling changes to our configuration, Prusti Assistant becomes more responsive and seems to anticipate users' actions.

4 Evaluation

4.1 Execution Times

In order to evaluate the impact the server mode had on execution times, we looked at a variety of programs of different complexities and verified each once without and once with the server. Additionally, the JVM, which Viper runs in, gets faster at executing time as it warms up, because it sees which parts of the code are executed the most and can spend more time optimizing their execution with its just-in-time compilation techniques. Consequently, we also test a program which takes a lot of time to verify, showing how that time changes over multiple verification operations on the same server (and thus JVM). This section aims to examine the effect of the server on execution times, both by mitigating startup times and by warming up the JVM.

Prusti contains an extensive unit test suite, running itself on a wide variety of examples. The GitHub repository has been set up to use continuous integration (CI) via GitHub Actions to automatically run all these tests on push, which also displays a duration. Before the server, these tests would consistently take around 65-75 minutes, which the switch to using a server for the tests instantly reduced that to 36-42 minutes. Simply using the server to run Prusti on our test suite almost halved its runtime, making it much more viable to run locally and reducing the delay before CI would inform you of a failed run.

In order to ensure a meaningful comparison, we chose a variety of examples from Prusti's test suite, specifically all the same ones as the original paper introducing Prusti [2] listed, several of them updated to work better with our newer version of Prusti. We extended these with one further example: The trivial correct program from Figure 1, for completeness. On all these examples, we simply ran Prusti with no special arguments and looked at the timing information it printed. Specifically, for each example individually, we: ran Prusti once as a standalone executable; booted up a throwaway Prusti server and ran Prusti again, connecting to that address, then killing the server; ran Prusti a third time, connecting to an existing server that had been warmed up by verifying every listed example once. Thus, each server instance was used exactly twice: once in an initial "cold" run, then again as it had warmed up. Finally, we repeated this whole process three times, averaging the timings measured between them and calculating the standard deviation. Running tests on a cold server is functionally almost the same as in standalone mode, so we decided to relegate those numbers to the appendix, and simply state that those timings are indeed all but identical. Table 1 lists the results from the other two setups across its two column groups: Standalone and Warm Server. Each run is further separated into its component timings:

- "Total" lists the total duration of Prusti's verification.
- "Client" is the total duration of all client-specific logic, i.e. everything it takes to generate the VIR program, most notably the encoding process from MIR to VIR.
- "Server" comprises all server-specific logic, which involves recreating the VIR AST on the JVM and actually running the Viper verifier. The JVM startup time, which amounted to no more than 116 ms in any of these runs, is not counted anywhere, as it is not a useful statistic and would only introduce noise.
- "Verifier" refers to the Viper verifier startup time, which of course does not apply to the warmed-up server as there is already an initialized verifier ready for reuse at that point.
- "Unacc." lists the remaining time unaccounted for among the above sections, compared to the overall verification time for the encoded program.

	Standalone									Warm Server						
Example	Total	σ	Client	σ	Server	σ	Verifier	σ	Unacc.	Total	σ	Client	σ	Server	σ	Unacc.
100 Doors	**10.84**	2.93	0.36	0.09	7.85	2.20	2.63	0.60	0.003	**2.88**	0.55	0.30	0.05	0.13	0.02	2.443
Binary Search (generic)	**10.65**	3.00	0.29	0.09	7.73	1.99	2.63	0.87	0.003	**2.93**	0.60	0.27	0.03	0.08	0.01	2.575
Heapsort	**19.31**	3.00	0.74	0.10	15.62	2.38	2.95	0.47	0.004	**6.46**	1.58	0.59	0.09	0.25	0.05	5.624
Knight's Tour	**120.03**	8.92	3.71	0.09	112.99	8.49	3.32	0.26	0.005	**78.53**	13.84	2.45	0.43	0.60	0.12	75.480
Knuth Shuffle	**6.27**	0.17	0.17	0.01	4.32	0.19	1.77	0.03	0.003	**1.39**	0.11	0.22	0.01	0.08	0.00	1.101
Langton's Ant	**16.09**	0.81	0.86	0.06	12.65	0.57	2.58	0.17	0.003	**4.95**	0.76	0.63	0.05	0.43	0.04	3.894
Selection Sort (generic)	**22.76**	2.99	0.81	0.12	18.87	2.51	3.07	0.38	0.004	**8.03**	1.62	0.56	0.07	0.19	0.02	7.276
Ackermann Function	**9.66**	1.51	0.13	0.02	6.59	0.94	2.94	0.48	0.003	**1.97**	0.31	0.13	0.01	0.10	0.01	1.745
Binary Search (monomorphic)	**16.72**	1.74	0.72	0.07	13.18	1.34	2.82	0.32	0.004	**6.23**	0.90	0.59	0.06	0.20	0.02	5.440
Fibonacci Sequence	**11.70**	0.56	0.40	0.02	8.52	0.35	2.78	0.18	0.004	**3.45**	0.63	0.38	0.04	0.19	0.02	2.873
Knapsack Problem	**125.04**	4.54	2.62	0.12	119.50	4.27	2.91	0.17	0.005	**108.78**	3.40	2.32	0.34	0.65	0.10	105.816
Linked List Stack	**15.55**	1.58	0.64	0.08	12.93	1.25	1.98	0.25	0.004	**5.74**	0.25	0.60	0.02	0.23	0.02	4.919
Selection Sort (monomorphic)	**31.06**	1.77	1.06	0.05	27.28	1.56	2.71	0.13	0.005	**13.01**	1.45	0.72	0.04	0.30	0.02	11.979
Towers of Hanoi	**6.94**	0.23	0.03	0.00	4.34	0.17	2.57	0.08	0.004	**1.21**	0.14	0.07	0.01	0.03	0.00	1.117
Borrow First	**8.13**	0.06	0.15	0.00	5.38	0.01	2.58	0.05	0.004	**1.60**	0.15	0.16	0.01	0.06	0.00	1.377
Message	**9.52**	0.06	0.21	0.01	6.77	0.11	2.54	0.11	0.004	**2.13**	0.24	0.21	0.01	0.07	0.01	1.843
Trivial	**6.59**	0.24	0.01	0.00	3.82	0.15	2.76	0.09	0.003	**0.95**	0.06	0.05	0.00	0.01	0.00	0.888

Table 1: Results from timing Prusti's execution on various example programs, along with their standard deviation across the three runs. All these tests were run on macOS 11 on an 8-core Intel i9-9880H with 2.3 GHz (turbo boost 4.8 GHz)[3], using Java 14.0.1. Full results including the cold server times available in Table 2 in the appendix.

[3]Full details at `https://everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i9-2.3-eight-core-16-2019-scissor-specs.html`

While very short, the unaccounted-for time contains important information about our communication overhead, where applicable (i.e. not in the standalone setup): among other smaller bits, everything from serializing the VIR on the client to deserializing it on the server is included in this duration. Comparing it between the different runs shows that communication overhead understandably increases for more complex programs, as there is simply more to encode, but remains negligible in relation to the remaining execution time.

Examining the execution times in more detail, we can see that there is some variance: it would take more than three repetitions to truly stabilize the numbers. Overall, however, this variance does not favor standalone over cold server or vice versa, so we can largely ignore it. Without exception, these times reflect exactly the effect one would assume the server to have. Client logic remains largely unchanged, and so do its runtimes across the different setups. Notably, even at its worst, client logic makes up a rather small part of Prusti's overall runtime, demonstrating its efficiency. Verifier startup times vary a lot, but don't show a clear trend between the two setups they are a part of. Server logic takes about the same time in the standalone setup as on a cold server, but shows significant improvement on a warmed-up server, as the JVM has warmed up and can execute code faster.

The JVM extensively uses just-in-time compilation, and it speeds up as the same code is run multiple times, because it realizes that this code is important

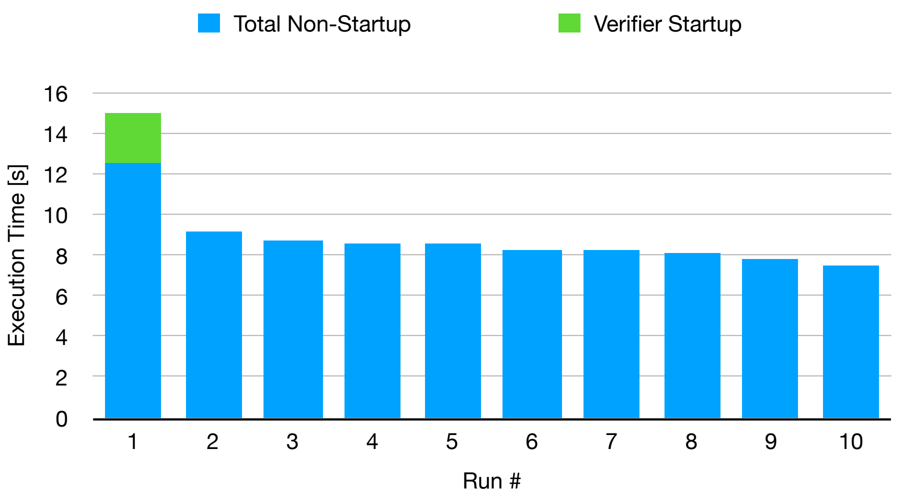


Figure 3: The time taken to verify the Langton's Ant program, with repeated runs using the same server. Note that only the first run has to initialize the Viper verifier, which is hence colored separately.

and sacrifices some immediate performance to compile it to native machine code. As the same code runs more and more, the JVM optimizes it more aggressively and compiles more of it. Thanks to the server setup reusing its JVM, we can now gain a lot of performance from this approach for free. The ideal situation of running near-identical code many times corresponds closely to the typical use case of working on a program, where the user compiles often, running Prusti in the process. In order to examine the effects of this system, we started up an instance of the Prusti server and repeatedly ran Prusti, using the server, on the Langton's Ant program from Table 1. Figure 3 shows the improvements we gain from these repeated runs, and performance indeed continues improving well after the first run, which is entirely down to the JVM's optimizations. There is a limit to this, of course, since processors can only execute code so fast, and the optimized code can only get so efficient. Furthermore, Viper itself uses Z3 as a low-level back end, which runs as a separate process outside the JVM, and is thus not affected by these optimizations. In summary, the benefits of reusing components through the Prusti server is clear, providing users with faster and faster verification as they continue to use Prusti on the same server.

4.2 IDE Integration

The first issue users would hit when installing Prusti Assistant before this project was that it requires a specific nightly build of Rust, which it would be exceedingly improbable for them to already have installed. To make matters worse, there was no check that the required Rust version was actually installed, resulting in cryptic errors for which users should not be expected to find the root cause. As one of Prusti's core goals is to make verification more available and easier to use, this stands in stark contrast to the rest of its functionality. With the changes we have made, this is now a smooth experience: Once the extension activates for the first time, it checks for the required Rust toolchain version, providing a simple dialog if it recognizes that this version is missing. This dialog offers a button to install the missing version, forwarding feedback from `rustup` to the status bar as the installation proceeds. Effectively, the first-time setup has been simplified to remove the last big barrier to entry, requiring nothing but a single button press now, making Prusti more accessible to new users.

Prusti Assistant of course needs to download and install Prusti to function, which can take some time. Previously, this process was completely opaque to the user—nothing would happen for a while, then the verification functionality would silently become available at some indeterminate time, once Prusti had been installed. Now, Prusti Assistant displays a dialog notifying the user about this installation process, also sporting a progress bar and a description of the current step, coming across more responsive and helping users to understand what is happening. The latter is especially helpful when something goes wrong, e.g. if the connection to the internet is slow.

The central functionality Prusti Assistant provides is verification, but this functionality can be surprisingly hard to access. Formerly, in order to verify a program, the user needed to either open the command palette, search for Prusti Assistant, and run the verify command, or alternatively enter the settings, navigate to Prusti Assistant's section, and enable the "verify on save" option—if this is even desired. Further, if users chose the former approach, it was easy for them to forget about the extension later, as it did not provide any affordances in VS Code's UI reminding users of its existence. To remedy this, we added a "verify" button to the status bar, which not only makes it easier to run Prusti but also keeps it visually and thus mentally present to the user, improving the likelihood of continued use.

5 Conclusion

The goals of this thesis centered around Prusti's core goal of bringing state-of-the-art verification to a wider audience of developers by making it easier to use and verifying programs even without any specific annotations or refactoring from the programmer's side. To that end, we focused on improving Prusti's execution times as well as its IDE integration and the resulting user experience. Our approach to improving execution times was to provide a server mode, which can not only reuse our underlying verification infrastructure but also gain a lot from the JVM's optimizations for often-used code paths. This approach has proved successful, shaving off almost half of the runtime for simple programs, where startup is a comparatively big part, and also showing steady improvement through continued use. The effect in practice is clearly demonstrated by Prusti's test suite of example programs, whose runtime was nearly halved from over an hour to just over half an hour, reducing latency in showing Prusti developers if they broke something, either locally or via continuous integration. This significant reduction in runtimes makes it less jarring to add Prusti verification to one's workflow by ensuring quicker feedback, which contributes to improving adoption of verification.

Additionally, our changes to Prusti Assistant offer major improvements to accessibility, vastly improving the first-time setup experience and thus lowering the barrier to entry. Installing the extension now no longer requires manually installing a dependency, which can put people off from using it. The part of the installation process that is automated was improved as well, giving a clear visual indication of its progress. Further, Prusti Assistant now offers a choice of build channels, easing coordination with a local build of Prusti during development and letting users choose between stable and nightly builds once Prusti reaches stability. Also in continued use, Prusti Assistant is now more present, providing the affordance of a status bar button, which reminds users of its existence and thus helps ensure continued use past the initial installation. Finally, Future development of not only Prusti Assistant but also other VS Code extensions providing verification capabilities will be eased by our extraction of reusable components and generally-useful systems to a Node module.

6 Future Work

Throughout this project, we stumbled upon several avenues worth exploring or other potential additions that we did not have time for or were out of scope. This section lists the most important ideas, in hopes of spurring on future work on Prusti and Prusti Assistant.

6.1 Prusti

- Provide a way to cleanly shut down the Prusti server, e.g. via a process signal or an HTTP request.
- Ensure killing the process used to launch the server also kills all the child processes it spawned, rather than leaving them around as orphans. This would also translate well to any other executables Prusti provides using this same launcher-driver pattern.
- Investigate having the server notice when a client disconnects and discard any verification requests it would otherwise waste computing time on.
- When a verification request arrives for which there is no matching verifier instance yet, initialize one in parallel with the usual request handling up until the point where it delegates to Java. This would further reduce execution times for verification operations on the server.
- The Viper verifiers Prusti uses to perform its verification are currently not thread-safe, so the server has to limit itself to only handling one request at a time. Gaining concurrency on the server would further improve performance, facilitating a pattern of having a central server that provides verification to many users.
- Use the profiling capabilities we have unlocked to identify performance bottlenecks and further improve execution times.

6.2 Prusti Assistant

- Provide a progress bar for verification operations, perhaps using previous operations' duration as an estimate to give a sense of progress.

- Extend the unit test suite to cover more of Prusti Assistant's codebase/behavior.
- Extract more reusable logic from Prusti Assistant to VS Verification Toolbox. Some useful candidates might be the subprocesses spawning method or the logging methods, including user-facing dialog/status bar interaction.
- Add a timeout when running Prusti, so that users are offered an easy remedy when it or Viper gets stuck.
- Automatically check for updates and notify the user if there is a new version of Prusti available, offering to download it.

References

[1] The Rust Programming Language. `https://www.rust-lang.org`. Accessed 2020-08-04.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), volume 3, pages 147:1–147:30. ACM, 2019.

[3] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In International Conference on Verification, Model Checking, and Abstract Interpretation, pages 41–62. Springer, 2016.

[4] Visual Studio Code. A code editor. `https://code.visualstudio.com`. Accessed 2020-08-04.

[5] Node.js. A package manager for JavaScript. `https://nodejs.org`. Accessed 2020-08-04.

[6] VS Verification Toolbox. Useful component to build VS Code extensions for verifiers. `https://github.com/viperproject/vs-verification-toolbox`. Accessed 2020-08-04.

[7] Tarpc. An RPC framework for Rust with a focus on ease of use. `https://github.com/google/tarpc`. Accessed 2020-08-08.

[8] Warp. A super-easy, composable, web server framework for warp speeds. `https://github.com/seanmonstar/warp`. Accessed 2020-08-08.

[9] Reqwest. An easy and powerful Rust HTTP Client. `https://github.com/seanmonstar/reqwest`. Accessed 2020-08-08.

[10] Bincode. A binary encoder/decoder implementation in Rust. `https://github.com/servo/bincode`. Accessed 2020-08-08.

[11] Serde. A serialization library for Rust. `https://serde.rs`. Accessed 2020-08-08.

A Full Timing Data

	Standalone									Cold Server									Warm Server						
Example	Total	σ	Client	σ	Server	σ	Verifier	σ	Unacc.	Total	σ	Client	σ	Server	σ	Verifier	σ	Unacc.	Total	σ	Client	σ	Server	σ	Unacc.
100 Doors	**10.84**	2.93	0.36	0.09	7.85	2.20	2.63	0.60	0.003	**10.97**	3.39	0.37	0.11	7.87	2.48	2.71	0.77	0.015	**2.88**	0.55	0.30	0.05	0.13	0.02	2.443
Binary Search (generic)	**10.65**	3.00	0.29	0.09	7.73	1.99	2.63	0.87	0.003	**11.32**	2.32	0.30	0.07	8.32	1.61	2.69	0.61	0.012	**2.93**	0.60	0.27	0.03	0.08	0.01	2.575
Heapsort	**19.31**	3.00	0.74	0.10	15.62	2.38	2.95	0.47	0.004	**21.09**	1.15	0.78	0.06	17.12	0.59	3.18	0.52	0.017	**6.46**	1.58	0.59	0.09	0.25	0.05	5.624
Knight's Tour	**120.03**	8.92	3.71	0.09	112.99	8.49	3.32	0.26	0.005	**103.50**	7.11	2.24	0.36	99.13	6.64	2.12	0.31	0.019	**78.53**	13.84	2.45	0.43	0.60	0.12	75.480
Knuth Shuffle	**6.27**	0.17	0.17	0.01	4.32	0.19	1.77	0.03	0.003	**7.88**	0.59	0.22	0.01	5.37	0.57	2.29	0.03	0.008	**1.39**	0.11	0.22	0.01	0.08	0.00	1.101
Langton's Ant	**16.09**	0.81	0.86	0.06	12.65	0.57	2.58	0.17	0.003	**18.64**	3.12	0.97	0.08	14.68	2.61	2.96	0.33	0.021	**4.95**	0.76	0.63	0.05	0.43	0.04	3.894
Selection Sort (generic)	**22.76**	2.99	0.81	0.12	18.87	2.51	3.07	0.38	0.004	**21.88**	2.84	0.85	0.13	17.69	2.14	3.33	0.60	0.017	**8.03**	1.62	0.56	0.07	0.19	0.02	7.276
Ackermann Function	**9.66**	1.51	0.13	0.02	6.59	0.94	2.94	0.48	0.003	**9.31**	1.19	0.12	0.02	6.24	0.63	2.94	0.50	0.010	**1.97**	0.31	0.13	0.01	0.10	0.01	1.745
Binary Search (monomorphic)	**16.72**	1.74	0.72	0.07	13.18	1.34	2.82	0.32	0.004	**16.22**	1.32	0.70	0.04	12.73	1.07	2.77	0.19	0.013	**6.23**	0.90	0.59	0.06	0.20	0.02	5.440
Fibonacci Sequence	**11.70**	0.56	0.40	0.02	8.52	0.35	2.78	0.18	0.004	**11.43**	0.64	0.39	0.01	8.28	0.49	2.75	0.15	0.012	**3.45**	0.63	0.38	0.04	0.19	0.02	2.873
Knapsack Problem	**125.04**	4.54	2.62	0.12	119.50	4.27	2.91	0.17	0.005	**126.86**	8.04	1.85	0.34	122.97	7.49	2.02	0.32	0.017	**108.78**	3.40	2.32	0.34	0.65	0.10	105.816
Linked List Stack	**15.55**	1.58	0.64	0.08	12.93	1.25	1.98	0.25	0.004	**17.33**	0.59	0.84	0.05	14.01	0.42	2.47	0.15	0.016	**5.74**	0.25	0.60	0.02	0.23	0.02	4.919
Selection Sort (monomorphic)	**31.06**	1.77	1.06	0.05	27.28	1.56	2.71	0.13	0.005	**30.33**	2.62	1.12	0.09	26.43	2.29	2.76	0.23	0.019	**13.01**	1.45	0.72	0.04	0.30	0.02	11.979
Towers of Hanoi	**6.94**	0.23	0.03	0.00	4.34	0.17	2.57	0.08	0.004	**6.74**	0.27	0.03	0.00	4.19	0.18	2.52	0.07	0.009	**1.21**	0.14	0.07	0.01	0.03	0.00	1.117
Borrow First	**8.13**	0.06	0.15	0.00	5.38	0.01	2.58	0.05	0.004	**7.95**	0.25	0.15	0.00	5.19	0.20	2.60	0.06	0.009	**1.60**	0.15	0.16	0.01	0.06	0.00	1.377
Message	**9.52**	0.06	0.21	0.01	6.77	0.11	2.54	0.11	0.004	**9.82**	0.66	0.23	0.01	6.82	0.45	2.76	0.18	0.011	**2.13**	0.24	0.21	0.01	0.07	0.01	1.843
Trivial	**6.59**	0.24	0.01	0.00	3.82	0.15	2.76	0.09	0.003	**6.83**	0.45	0.01	0.00	3.94	0.25	2.88	0.18	0.007	**0.95**	0.06	0.05	0.00	0.01	0.00	0.888

Table 2: Full results from timing Prusti's execution on various example programs, along with their standard deviation across the three runs. All these tests were run on macOS 11 on an 8-core Intel i9-9880H with 2.3 GHz (turbo boost 4.8 GHz)[4], using Java 14.0.1.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Developing IDE Support for a Rust Verifier

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Dunskus	Julian

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Uerikon, 13.08.2020	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.