Engineering of Reliable and Secure Software via Customizable Integrated Compilation Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Institut für Technologie (KIT)

genehmigte

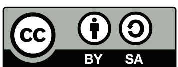
Dissertation

von

Dipl.-Inf. Oliver Scherer

Tag der mündlichen Prüfung: 06.05.2021

1. Referent: Prof. Dr. Veit Hagenmeyer
2. Referent: Prof. Dr. Ralf Reussner



This document is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0):
https://creativecommons.org/licenses/by-sa/4.0/deed.en

Abstract

Lack of software quality can cause enormous unpredictable costs. Many strategies exist to prevent or detect defects as early in the development process as possible and can generally be separated into proactive and reactive measures. Proactive measures in this context are schemes where defects are avoided by planning a project in a way that reduces the probability of mistakes. They are expensive upfront without providing a directly visible benefit, have low acceptance by developers or don't scale with the project. On the other hand, purely reactive measures only fix bugs as they are found and thus do not yield any guarantees about the correctness of the project. In this thesis, a new method is introduced, which allows focusing on the project specific issues and decreases the discrepancies between the abstract system model and the final software product. The first component of this method is a system that allows any developer in a project to implement new static analyses and integrate them into the project. The integration is done in a manner that automatically prevents any other project developer from accidentally violating the rule that the new static analysis checks. The second component is a way to directly integrate system models (e.g. from UML) into the project by treating the model as a direct input to the compiler, just like any other source code. These two components together allow developers to handle complex situations that are only relevant to the given project. The entire project gets analyzed for the correct usage of nontrivial APIs or other hazards which either are bugs or are likely to turn into bugs in future refactorings. Thus, the new method permits the incremental introduction of formal analysis without forcing a project's developers to change to unfamiliar habits or styles. At the same time, it allows preventing classes of defects automatically, yielding immediate gains from the first usage of the new method.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Overview over the state of the art of reliable and secure software . . . 3

1.3 Open aspects of existing solutions . . . 4

1.4 Goals of this thesis . . . 5

2 State of the art of reliable and secure software **7**

2.1 Reliability . . . 7

2.1.1 Availability . . . 8

2.1.2 Redundancy . . . 9

2.1.3 Security . . . 10

2.1.4 Comprehensibility . . . 10

2.2 Software engineering of reliable systems . . . 10

2.2.1 Software development process . . . 11

2.2.2 Continuous integration . . . 11

2.2.3 Model checking . . . 13

2.2.4 Typestate . . . 13

2.2.5 Restricting Programming Languages to a safe subset . . . 15

2.2.6 Static Analyses . . . 20

2.2.7 Project Specific Static Analyses . . . 21

2.2.8 Safety through Programming Languages . . . 22

2.2.9 Compiler extensions . . . 23

3 New concept of an extendable compilation system with integrated static analysis 36

3.1 Components of compilation systems 36

3.2 Automation in software engineering 38

3.3 Analysis of requirements for extensible compilers 38

3.3.1 Coverage of language constructs 39

3.3.2 Direction of dataflow 39

3.4 Features enabled by the new concept 41

3.4.1 Employing static analyses 41

3.4.2 Model driven development 47

3.4.3 Checking code against models 48

3.4.4 Code generation from models 49

3.4.5 Code generation vs code checking 49

3.4.6 Abstract interpretation 50

3.5 Project specific compiler extensions instead of generally applicable tools 51

3.6 Lint categories 53

3.7 (Semi-)automatically resolving static analysis results 55

3.8 Incremental steps towards full formal analysis 55

3.9 Summary 56

4 Comparison of the new concept with related works 58

4.1 Components of compilation systems 59

4.2 Automation in software engineering 60

4.3 Compiler extensions 61

4.4 Formal analysis 61

4.5 Model driven development 62

4.5.1 Code generation 62

4.5.2 Manual transcription 62

4.5.3 New concept 62

4.6 Static analyses 63

5 Implementation of the components of the new concept **64**
5.1 Integrating static analyses into the development process 65
5.1.1 Developing a new static analysis 65
5.1.2 Choosing a responsiveness level 66
5.1.3 Forward compatibility of compiler extensions 67
5.1.4 Diagnostics 67
5.1.5 Problem resolution instructions 68
5.2 Model driven development 69
5.2.1 Code generation 69
5.2.2 Code checking 74
5.3 Abstract interpretation 76
5.3.1 Running abstract interpretation on test suites 77
5.3.2 Restraining the virtual machine 78
5.3.3 Employing abstract interpretation for debugging purposes . . 80
5.3.4 Summary 81

6 Aspects of the practical application of the new concept **82**
6.1 Role of false positives on the usability of static analyses 82
6.1.1 Stability across compiler changes 85
6.2 Compiler extension schemes 89
6.3 Generating static analyses from examples 89
6.3.1 The problem 89
6.3.2 The new tool and its usage 90
6.4 Community driven database 93

7 Evaluation **96**
7.1 Analysis of the goals of this thesis 96
7.2 Usability study 97
7.2.1 Collected data 98
7.2.2 Analysis 98
7.2.3 Limits of this study 98

8 Conclusion **99**

8.1 Contributions . . . 101

8.2 Limits . . . 102

8.3 Outlook . . . 102

Abbreviations **104**

Bibliography **106**

A Semantic versioning **118**

B Forward compatibility guarantees **119**

C The Rust programming language **121**

C.1 Basics . . . 121

C.2 Constructors . . . 121

C.3 Destructuring . . . 122

C.4 Refutable destructuring . . . 123

C.5 Ownership . . . 124

D Rustc internals **126**

D.1 AST Items . . . 126

D.2 AST Types . . . 127

D.3 AST Expressions . . . 127

E Pointer equality operation during abstract interpretation **131**

F Code generation from model **132**

G Custom driver for project specific static analyses **135**

H `init` function call before `foo` function call demo **137**

I Survey about the ease of tool integration **145**

J Full list of related publications **146**

J.1 Peer reviewed conferences and journals 146

J.2 Language design and standardization (peer reviewed white papers) . 146

J.3 Peer reviewed open source 147

Chapter 1

Introduction

1.1 Motivation

Adding new ways to access any system, be it physical, informational or digital, comes with the inherent risk of giving access to unauthorized parties. Physical security has been a part of humanity since its dawn and part of the animal kingdom before that. Protecting offspring or property via preparation has clearly been a beneficial trait. Even before the existence of writing knowledge has been kept secret or hidden in code words. Communicating information without fear of interception by either the messenger or third parties was an issue well before the beginning of our calendar [63]. But only with the invention of automated information processing has the protection against malicious information become an end in itself.

When it became possible to connect to computers via phone lines, systems did not have any IT-security (from here on just referred to as "security") concept except for the secrecy of the phone number used to connect to it. Thus, the cat and mouse game of inventing new attacks and defending against them began with automated systems calling every possible phone number to find computers. After these systems were considered *reasonably secure*, the internet opened a new avenue of attack, because a system was not limited to a single incoming connection anymore. Instead, multiple inbound and outbound connections are emulated via packets and identifiers, but the underlying principle of probing arbitrary physical addresses still applies. Multiple orthogonal and complementary technologies have evolved to ensure the security of the virtual connections. Unfortunately these technologies only add to the growing complexity of the systems themselves. Since any reasonably complex system is guaranteed to exhibit unintended or forgotten behavior [3], these interconnected systems have frequently been exploited for various goals. More recently the security technologies themselves have been attacked, while the base system would have been invulnerable without it.

The convenience of a unified communication system greatly outweighs the potential for issues that have arisen with it. This convenience has led to further industries

adopting internet technologies. Medical systems [84], factories [40], vehicles [71, 83], home automation [36, 126] and even the future energy supply and distribution systems [48] are opening up their information systems in order to take advantage of the thus possible features.

The issues revealed themselves beginning with the web industries and personal computers. Viruses, worms, ransomware, bot networks, etc. spread uncontrolled across these vulnerable systems, ignoring all classical security addons like firewalls and anti-virus-software. The companies supplying the affected software components countered by hardening their software by either using the attackers' tools themselves in safe environments or by moving to entirely new systems eliminating entire avenues of attack.

While it would have been expected from the other industries to follow suit, the lack of attacks and a higher quality of software gave a false sense of security. Train schedule systems [95], entire energy grids [79], hospital networks [24], vehicular control software [46] and even critical aircraft systems [110] have been successfully attacked. These attacks have been made possible by the experience of attackers gained in the web, enabling them to attack more protected systems. If left unchecked, a repetition of the (partially) automated and widespread attacks in this millennium's internet is not only possible, but the logical consequence.

It is possible to design a theoretical system with perfect security. On the other hand, implementing it will unavoidably introduce defects [14]. Irrelevant of the care taken during implementation, any physical manifestation of a theoretical system will not represent the system exactly. Any reasonably complex system will have even more possibilities for mistakes to be made during the implementation. Various software engineering methods exist to minimize the risk by a clean upfront design and a transparent process that allows tracing all implementation components back to components of the theoretical system and vice versa.

Model driven development, formal analyses and similar holistic approaches are too expensive upfront for the application in small and medium sized companies, as well as hard to sell to large companies [77]. These methods are mainly used in safety critical applications like military, aerospace, space and transportation, where a software defect is legally considered to be the author's negligence. These industries additionally introduce hardware redundancy and software diversity to decrease the chance of accidental failures even further. This attribution of fault comes from the direct danger to human life and health that a software defect incurs. Security defects on the other hand require an attacker in order to be exposed and exploited. It is natural to place the blame solely on the attacker instead of applying a proportionate amount of effort into securing the software.

The advent of the German Energiewende (energy transition) multiplies the mentioned dangers from mediocre software engineering by introducing software development and interconnected systems to companies previously developing stand-alone components containing no or little software. Awareness of the existence of issues

with unintentional or careless exposition of systems has motivated these companies to search for security solutions. While "obvious" technologies like cryptography and signatures are a necessity, they are not sufficient. A perfectly secured steel door with iris scans and voice identification is useless if the walls around the door are made of thin wood. In software terms this means that one needs to take security into account throughout the entire software and its development process, instead of thinking of it as an add-on that can be added via security technology.

In summary, safety and security can not be done as a half-measure, but forcing companies to adopt the latest research is not practical due to the high upfront costs and the lack of experience or confidence in software engineering practices.

1.2 Overview over the state of the art of reliable and secure software

Every recurring mechanical activity that a human has to perform, is being considered a bug in software development. Continuous integration is being employed heavily in innovative industries and slowly gaining usage in existing large scale industries [85], significantly speeding up the time between the implementation of a feature or bugfix and the time when that implementation is ready to be deployed to customers. Continuous deployment automates even the deployment to customers, automatically taking care of notifying customers about the new versions and providing convenient access to the updated pieces of software. Similarly the software infrastructure needed to support the development is being automated in the same manner (IaC: Infrastructure as Code), enabling the maintainers of that infrastructure to quickly and reproducibly react to requests and events.

Similar ideas exist in model driven development. Instead of building a model and then manually transcribing it to the source and manually ensuring that the source and the model match, code generation is commonly applied, automatically generating source code that matches the model exactly. Each modelling language supports different features that are targeted towards their specific niche use case. The UML attempts to unify different modelling concepts into a single model and exposing subsets of the single model to the user via diagrams. This approach aims to maximize interoperability and total capabilities of the modelling language while still giving humans approachable subsets that allow them to concentrate on the components that are of interest in specific situations.

Another avenue at reducing unnecessary mechanical work are high level language features. The choice of programming language has a significant impact on project success [130] and defect rates [75]. For safety critical systems where maintenance is hard or outright impossible Ada[1] is a common choice due to the safety guarantees that are built in and don't need to be validated after the fact [39]. Even at such a

[1] a programming language for safety critical systems

high level of safety, there is desire to restrict the language further to preemptively eliminate avenues for mistakes. In the case of Ada the Ravenscar profile prevents all deadlocks, while the SPARK subset completely eliminates runtime errors. In less safety critical systems like web or open source tool development there was a recent (2010-2020) influx of new programming languages (Go, Swift, Rust), that aim at eliminating common sources of bugs. At the same time they increase developer contentment, e.g. resulting in Rust holding the title of most loved language for five years in a row in developer surveys [127].

When the choice of programming language is given by an existing large project, static analyses offer a mid-way by restricting the programming language to a safer subset, eliminating undesirable features or coding patterns. For many programming languages multiple static analysis tools exist, allowing developers to not just choose from the tools, but also configure the tools to choose which analyses to use. Some compilers even allow developers to write their own static analyses via compiler extensions [5, 121]. These compiler extension APIs are often meant as an experimental, unstable API only meant for research and experimentation, making them unsuitable for commercial projects.

1.3 Open aspects of existing solutions

Chapter 2 highlights stagnation in the actual application of the best practices in the engineering of reliable software. Basic approaches like modelling, formal analyses, different stages of testing, continuous integration and traceability of requirements are applied in modern projects [142]. While there still are minute improvements happening and the tooling around the software development cycle has many avenues for advancement, the process itself still requires not insignificant amounts of human interactions. Increasing the level of automation in software engineering can reduce costs and improve the software quality at the same time, because developers can focus on the risky parts of the development instead of being busy with the many simple, but repetitive tasks.

Furthermore, the long cycle from a change in the models or the code to obtaining feedback from the analysis or testing slows down the advancement of software development processes over the course of a project's lifetime. Heterogenous toolchains and outdated compilation systems are made up of many sequential steps leading to long waiting times even for trivial changes. Each step often has to repeat work done by previous steps, because of incompatibilities of the heterogenous components. The developers themselves cannot solve the incompatibilities due to their closed-source nature, and the small benefit from paying the component author is rarely enough to convince decision makers. Additionally, developers frequently work around the issues, which is inefficient and will be more expensive than fixing the underlying problems if the component is used long enough.

Modelling and then manually checking the project against the model does not scale

and quickly leads to obsolete models. Code generators on the other hand require a lot of work whenever the model changes, and thus penalize fixing issues in the model. Model checkers enforce specific structures on the code, even if a slightly different project structure would improve the implementation significantly. Putting aside the flaws in the tooling, modelling also suffers from the dangers of over- and under-modelling. If the models are too abstract there is no real benefit to the project, if the models are too concrete, they are most likely the result of model authors attempting to program with models, which should be performed in the implementation language. There are guidelines and best practices to determine the "sweet spot" where the model is abstract enough to allow implementation freedom and actually help comprehending the software, while being concrete enough to be useful. Finding and staying at this optimum is nontrivial and frequently not succeeded at.

These issues are known and there is work being applied at fixing them. Unfortunately the turnaround times are commonly between 1-10 years, which means that by the time the problem has been solved, the workarounds in place are well established or the problem is not relevant anymore due to other changes. Together with a continuously increasing rate of bugs with increasing project size amongst other factors in long lived projects, the slow turnaround time causes the issues to stack up [104].

The turnaround time needs to be reduced significantly to the magnitude of hours in order to prevent this above-linear growth of issues both in the process and the project itself.

1.4 Goals of this thesis

The gap between software development research and practice has been growing over the last years. There exist many methodologies for formal correctness analyses and model driven development, but in practice, most projects do not employ such schemes due to the high upfront cost. This thesis aims to permit developers to incrementally work towards the more advanced software development schemes by incrementally introducing concepts as they become beneficial to the project.

The goals of this thesis are

- to increase the quality of software projects by automating flaw detection,
- to encode developer experience in tooling to ensure it is not lost if the developer leaves the project,
- to take review and teaching load off experienced developers allowing new developers to quickly and automatically have their code reviewed without having to consult another developer,
- to create a work flow and software engineering methodology that enables and encourages the introduction of static analysis and models at any point in the project's lifetime,

- to implement prototypes and tools to employ the new methodology in real life projects, and
- to contribute to open source projects related to compiler extensions and static analysis to ensure that the work done for this thesis directly benefits existing projects and indirectly validates the introduced concepts.

In Chapter 2 the existing terminology and methodology for reliable and secure software is established. In Chapter 3 the new concept of an extendable compilation system with integrated static analysis is introduced, followed by a comparison with other preexisting solutions in Chapter 4. The implementation aspects of the components of the new concepts are discussed in Chapter 5. Various aspects of the application of the new concept in practice are addressed in Chapter 6. Survey, experimental and anecdotal evidence for the effectiveness of the new concept is discussed in Chapter 7. In the Chapter 8, the thesis is concluded and results of using the prototypes in real world project's development cycles are briefly presented. Figure 1.1 depicts this structure graphically.

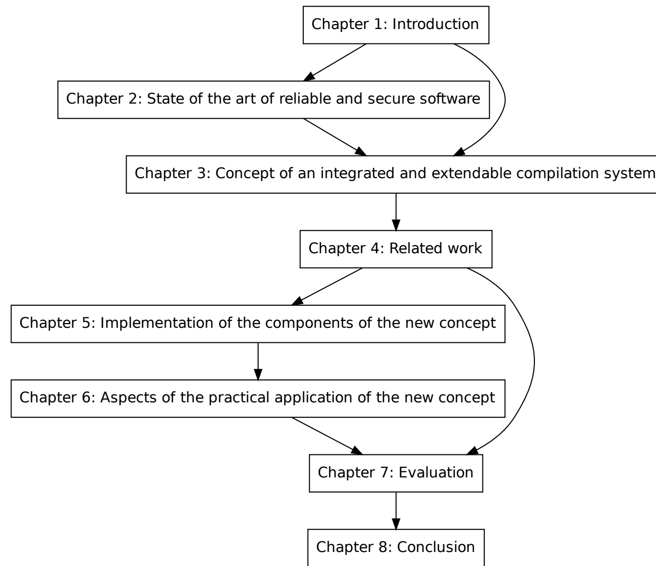


Figure 1.1: Structure and possible reading orders of this thesis. Bypasses show which chapters can be skipped without missing information about the theoretical aspects of the new concept.

Chapter 2

State of the art of reliable and secure software

This chapter presents and evaluates the preexisting work in ensuring a specific set of constraints or increasing the quality of software projects. The first part addresses the terminology around reliable software. The second part covers software development processes, software modeling and programming languages. The third part evaluates the current standards in static analysis tool and compiler interfaces and practical API[1] usability.

2.1 Reliability

A reliably crashing system does not make a reliable system. Thus, before addressing *why* many software systems have a low reliability, one needs to clarify *what* reliability means in this context. Frequently reliability is confused with availability. A small example demonstrates why this inference is insufficient: A factory that produces one item per 100 seconds but crashes every 10 seconds and reboots in one millisecond, has a high availability (99.99%), but is entirely unreliable, as it will never produce even a single item due to crashing before the item is finished and starting over.

There is no generally agreed upon measure or statistic for reliability, which lies in the very nature of reliability: its definition depends on the use case. A general rule in engineering is summarized in [31] as

> Reliability is the probability that a product will operate or a service will be provided for a specified period of time (design life) under the design operating conditions (such as temperature, load, volt ...) without failure.

E.g. a bridge is guaranteed to not collapse under the stress of weather conditions for a specific time, a specific number of vehicles that cross it and a maximum weight of

[1] application programming interface

vehicle that is allowed to cross it. Furthermore, a bridge is always over-engineered to handle stresses in excess of the specified maxima, and, instead of collapsing, will degrade gracefully (e.g. by allowing cracks to form and by deforming in general).

Software reliability cannot be specified in the same way [76]. While a software system can have graceful degradation in the presence of bugs or overloading, a piece of software by itself will always "collapse". Even if the software is designed to detect bugs, it cannot bend or degrade, it can only perform emergency shutdowns and report the failure. Additionally, one cannot protect software systematically against all bugs, only against specific bugs or increase the cost of exploiting a bug by obfuscating the protection method. Instead of trying to mitigate the existence of bugs, safety critical software development strives to reduce the kinds and number of bugs below acceptable levels by an error avoiding upfront design, extensive use of strong typing and the application of formal analyses.

2.1.1 Availability

Reliability is often used interchangeably with availability. Intuitively this makes sense in many everyday systems. A pen that emits ink 99.99% of the time it is in contact with paper can be considered reliable because of its high availability. Being stuck for less than a second after an hour of continuous usage does not devalue the pen. An alternative way to reach 99.99% availability is for the pen to emit ink for 999.9 milliseconds, not emit ink for 0.1 milliseconds and then immediately emit ink again. Such a pen would still be considered reliable, since on paper, no visible interruption has occurred. At worst a slight brightening of the pen color might be noticeable, but most likely it would be within the standard deviation of uninterrupted writing. Figure 2.1 compares two possible interpretations of 80% availability.

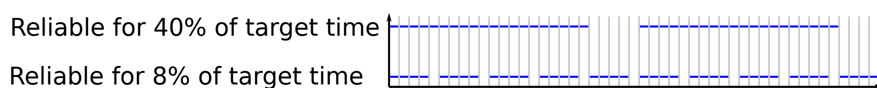


Figure 2.1: The first line is uninterrupted for 40% of the range, while the second line is interrupted after 8%.

When moving from a continuous system to a discrete system, a system with 99.99% availability can be entirely unreliable. Reiterating on the example from the introduction, consider an operation requiring `n` seconds to finish, but if interrupted needs to start from the beginning. A system with the illustrative 99.99% availability could finish 9999 operations successfully, then go into a downtime for `n` seconds, and repeat the process. In that case, on can speak of the system having a high reliability, as in 99.99% of the cases, it was reliable. On the other hand, the system could crash after `(m * 0.9999) < n` seconds, restart in `m * 0.0001` seconds, and repeat this process forever without ever finishing even a single operation uninterrupted. Since

the operation starts over from the beginning after an interruption, it will never finish before being interrupted again. This means while the availability is 99.99%, the system is entirely unreliable as it was reliable in 0% of the cases.

The discrepancies between availability and reliability are further amplified in software engineering. It is impossible to test the reliability of any reasonably large software. The number of tests required to prove the absence of undesired behavior commonly grows exponentially with the number of branches a software has. Every conditional loop, if condition or virtual function call must be considered a branch, and, considering the extensive use of conditionals in any software project, this would require more test code than actual project code. In the case of multi-threading or interrupt based systems, the program can be interrupted after every single machine instruction. Since this means that each machine instruction essentially becomes a possible branch site, it becomes infeasible to even attempt to test all cases. Out of economic reasons many projects decide on a "best effort" testing scheme, assuming that a software that has run under real circumstances for a reasonable amount of time has encountered all the problems it will experience in practice. [65]

2.1.2 Redundancy

Mechanical safety systems apply a trivial method for increasing the reliability of a system: redundancy. If a component has a 1% chance of failing during a year of continuous operation, if you have a backup component, you have a 0.01% chance of both failing simultaneously. A prominent example is the breaking system in cars, which consists of two separate and diagonal connections from the break paddle to the breaks in the wheels.

Deploying software redundantly does not increase the reliability of the system. If one copy of the software fails, the other copy will fail, too because it is identical. In contrast to physical components, there are no "production variances", because the production is just a digital and thus flawless copy.

Instead, software can be developed diversely

- by using multiple programming languages,
- by using multiple developer teams,
- or by using optimization leeway

to generate different binaries or even sources from the same requirements. These different binaries should exhibit the same behavior under all tested situations. Since unknown failures are by definition not tested, it is assumed that the different binaries will exhibit different behavior in the case of a failure. If either binary detects that another is diverging from its own behavior, an alarm is raised. In the case that a majority of binaries produces the same behavior, the operation does not need to be interrupted, although the event must still be recorded and analyzed in order to increase the reliability in the future.

2.1.3 Security

While testing software can increase the confidence that there will be few failures due to random or environmental causes, the presence of an attacker invalidates all assumptions that can be made about the probabilities of defects influencing the normal operation of the software. Depending on the defect, the attacker can either directly trigger the defect, or must manipulate the system environment in order to increase the likelihood of the defect becoming available. In either case, the defect is guaranteed to be exploited once discovered. Due to the systematic search for defects by the attacker, the chance for a defect to be discovered by the attacker is significantly higher than the randomized detection happening during testing and production. Thus the security of a software system cannot be quantified by judging the trend of found defects over time. An insecure system can thus by its very definition not be classified as reliable in case it is accessible by attackers in any manner. [65]

2.1.4 Comprehensibility

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. – Brian Kernighan*

Program code is rarely a finished product. Most program code evolves in iterations - new features are added, deficiencies are fixed, or the software is adjusted to changed requirements. Since modifying software first necessitates understanding the software, it is imperative that the software is designed in a readable way - both visually and logically. Small projects still require fixing bugs late in the development, they do are not as strongly affected as large projects, since less code means large refactorings are less work [92]. The required effort for future changes will unproportionally increase with the project's size and lifetime [87]. Furthermore, long lived software is not just guaranteed to require developers to modify it some time in the future, but the likelihood increases that the developers modifying it will not be the developers that wrote it, but their successors. A repeated experiment [105, 111] shows that even trivial standard readability features like syntax highlighting have a significant positive impact on time needed to comprehend a piece of code. In summary, low comprehensibility of software negatively impacts the reliability of software in the long term due to the friction introduced in resolving problems properly.

2.2 Software engineering of reliable systems

Existing modeling tools and their corresponding code generators produce textual input for the compiler to process. This means that the compiler not only loses all semantic information relating to the source the code is generated from, but also the

ability to any means for producing comprehensible diagnostics. Static analysis tools suffer from the exact opposite problem. They need to process source code and emit diagnostics, requiring them to replicate source code parsing and diagnostic styles to match those of the compiler. The replication also replicates the sources of bugs that occur in a regular compiler and carries the risk of diverging from the compiler's behavior. The effort of replicating compiler components is the major speed bump for advanced practical static analyses.

2.2.1 Software development process

The applied software development processes vary significantly between companies, in its performance, its cost and the level of manual work required. Safety critical software components at BASF are developed with the classical V-Model [74] with a focus on traceability of requirements to code and vice versa. The firmware for the flight control of SpaceX rockets on the other hand is designed with agile software development methods [15]. Initial criticism by NASA [30] about the lack of a maturity process has been addressed by integrating these processes in an automatable way instead of falling back to classical software development. While these systems are not targeted by private attackers, it is certainly conceivable that governments could attempt to influence space hardware. Even ignoring attackers, the high variance and uncertainty about the environment in which the hardware is in use, coupled with an inherent lack of maintenance capabilities, make a robust software base a significant factor in the success of projects. Industries outside or new to the safety critical or software development realms (e.g. companies developing electrical network components expanding to developing smart meters [25]) only address functional safety and software security, without considering the software safety aspects that become relevant when opening up software components to wider networks like the internet. There is a strong desire to make energy system components communicate and interact more. The inherent distributed nature of the energy system components makes it hard to protect the components by physical means. Protecting the systems from attackers by addressing software security concerns grows in importance, while the companies creating those systems are not yet as sensitized to those concerns when compared to companies offering services on the internet.

2.2.2 Continuous integration

The goal of continuous integration is that the product is ready for deployment at every moment in development. Continuous integration is an approach where running static analysis tools, unit tests, coverage tests, integration tests and deployment is automated. Instead of a human going through all these steps by hand before a release, a release is something that automatically happens after every change to the code base. Many projects already automatically run their tests on the entire code base on a "nightly" basis, meaning whenever the fewest developers are active. The

step from testing regularly to continuous integration is to test the entire code base after every change to the code base. If the tests pass, the change is accepted, otherwise the developer is notified and has to adjust their code change accordingly. After the change is accepted, the continuous integration system additionally generates the artifacts necessary for deployment to the customers or the project's platform. The deployment to the customers is usually automated by introducing an update distribution platform that the customers' software polls regularly. Deployment to hardware platforms is already offered as a service or system [64], but rarely applied outside of well-funded projects like SpaceX [15] or the automotive industries.

Architecture Analysis & Design Language

The AADL[2] was specifically designed to model hardware and software components of real-time systems. AADL is significantly less abstract than UML. The software model separates processes, threads, data and subprograms. Subprograms can be written in any language supported by the concrete tool and are not part of the model. Deadline, period, WCET[3] and priority can be chosen for every thread. Various annexes extend the AADL to allow modelling of error handling, data structures and the behavior of these data structures [35].

UML

The UML strives to allow humans to comprehend complex systems by using abstraction, separation of concerns and visual diagrams. It is intended to be used in a top-down (software) development approach. The diagrams can be classified into structure and behavior diagrams [100]. Structure diagrams show the parts of the system that do not change during runtime. Behavior diagrams visualize the interactions of the structural components. Although UML specifically allows merged diagrams, few tools support this, and none in any meaningful manner. An attempt to combine state machine diagrams with sequence diagrams [16] yielded a form of petri net that treats state machines and sequence diagrams as views into the same model instead of separate models. Stereotypes allow annotating UML diagrams in order to connect model components to desired project specific semantics [66]. Stereotypes improve upon the general nature of model components, but are further disconnected from tool support since they may be entirely user defined without a connection to either the target platform or the concrete modeling tool.

UmlSec [62] is an extension to the UML that uses tagged associations to directly model the communication method (LAN, Internet, wire, etc.) used for exchanging information between objects. Additional metadata for the security levels of information and objects enforces that information only flows from low security levels to high security levels.

SecureUml [82] focusses on modelling role-based access control within UML. Stereotypes for permissions, roles and users are attached to associations and classes, allowing fine grained control over security requirements without burdening the model

[2] Architecture Analysis & Design Language
[3] worst case execution time

author with unnecessary detail levels.

2.2.3 Model checking

Model checking is the process of verifying that a model upholds certain properties. A model can be anything from a hardware description, over program code to high level models like UML diagrams. When focussing on checking program code, often, instead of specifying properties to be checked, code is matched against a high level model. The code needs not only to correctly match the model criteria, but also to exhaustively cover all aspects of the high level model. Not everything is or can be specified in the high level model, because that would invalidate the need for program code, making the high level model essentially become a programming language. Thus the code is allowed to implement more behavior or behavior in more detail than the model specifies. Unfortunately this leeway can result in unwanted behavior if the model cannot or does not exclude said behavior [53, 69, 114].

2.2.4 Typestate

The programming language concept typestate [129] formalizes inherent state that the programmer keeps in mind or documentation. Using this concept would make several concepts introduced in this thesis obsolete. Therefore both typestate's lack of success in practice as well as its known downsides are discussed in this section. Every typestate has its own set of methods and fields. An example structure for a file handle containing the type states `uninitialized`, `error` and `open` is depicted in Table 2.1

Table 2.1: A list of typestates, their transition methods and their data components.

typestate	method	public fields	private fields
uninitialized	open, create		
error		error_kind	
open	write, close		os_handle

In the beginning every object of the file handle typestate is *uninitialized*. The method `open` can be used to open an existing file on the filesystem and changes the typestate to either *error* in case the file doesn't exist or the filesystem forbids opening the file due to permission levels. The kind of error is stored in the *error* typestate's `error_kind` field and can be inspected by the user. The `create` method creates a new file on the filesystem in case the file does not yet exist (in which case the new typestate would again be *error* with the appropriate value in the `error_kind` field). As shown Table 2.1, the `error` typestate has no methods, as it is not possible to recover from errors in this simple file handle example. In case

the `open` or `create` method succeed in their respective tasks, the *open* typestate is reached (see Figure 2.2 for all possible transitions).

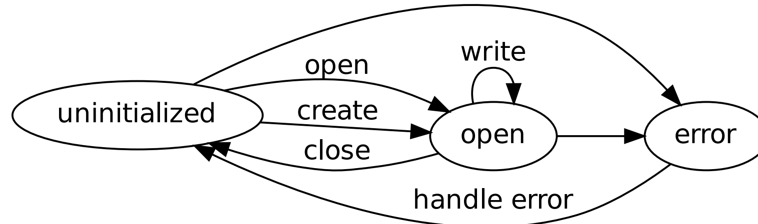


Figure 2.2: A state machine of the file example

This state guarantees that a functioning file handle to the opened or created file exists. Thus it is now possible to write data to the file through the `write` method. Since the file systems background storage might run out of free space or suffer sudden unrelated errors, calling the `write` method may fail. In case of such an error condition, the typestate is either *error* or again *open* after processing a `write` method call. Finally there is a way to guarantee that the file has properly been written, by calling the `close` method which either changes the typestate to *error* or *uninitialized*.

The research programming language plaid [2, 144] replaces Java's classes with typestates, allowing the inheritance of states instead of classes. Constructors exist in all starting states of an object (a file object might be initialized as a closed file and then opened or directly initialized as an open file). Development on plaid appears to have come to a halt in 2012.

The programming language Rust [149] supported typestates until version 0.4 when user-defined typestates were removed due to the fact that common patterns like loops are either heavily restricted in a typestate environment, or require complex annotations that actually hinder code maintainability instead of improving it. Language-defined typestates are *uninitialized*, *initialized* and *moved-out-of* (equivalent to *uninitialized*, but the distinction significantly improves compiler error messages). This concept is the base for the ownership semantics that are at the core of Rust. A detailed example is shown in Appendix C.5.

The domain specific language hanoi [88] allows defining typestates for existing java code or binaries to apply model checking without interfering with the existing code base.

The JML[4] consists of annotations inside Java comments. It has pre- and postconditions for functions as well as type invariants. JML has been extended with

[4] Java Modelling Language

typestates [68] which are translated to JML pre- and post-conditions. The Plural Specification Language and corresponding implementation as Pulse-Tool [124] has implicit typestate support along with access permissions for object references and “dimensions” for dynamic typestates.

2.2.5 Restricting Programming Languages to a safe subset

Programming language design rarely considers safety and error reduction. The only commercially viable programming language before 2009, which was explicitly designed for safety and which is successfully used outside of academia is the language Ada. In quick succession Go (2009 [150]), Rust (2010 [151]) and Swift (2010 [152]) were designed in order to eradicate the memory safety issues and common bug sources of the C, C++ and Objective-C languages.

The authors in [51] analyze various language subsets and their effectiveness. They note that such standards are either informal or formal, where the latter sees little use in practice due to the user being required to understand formal notation. The informal standards on the other hand suffer from ambiguities. It is also noted, that creating language subset standards is an expensive process, ranging usually from $50000 to $100000. These figures do not include the development of automated rule checking software.

MISRA-C

The automobile industry uses the MISRA-C subset of the C programming language to eliminate common programming mistakes and simplify the code to increase the readability.

For example the usage of variable assignments (`x = y`) inside other expressions is a construct forbidden under MISRA-C. The syntactical similarity of `x == y` and `x = y` has caused many errors even in heavily reviewed code.

An example of an error prevented by the MISRA-C standard’s rule set can be found in the following (either erroneous or needlessly complex) piece of code

```c
int x = some_value;
if (x = 5) {
    // do some action
}
```

This piece of code contains three related symptoms of an issue.

1. the assignment operation in c yields the value of the assigned variable after the assignment
2. the curious oddity that every nonzero value of a non-aggregate c type causes a branch to be taken

3. assigning to a variable that was never read since the previous assignment is useless and the compiler will very likely optimize out the first assignment, which opens up the possibility to many more issues due to code that should be executed not getting executed.

The `if`-statement's condition will *always* assign `5` (a nonzero value) to the `x` variable. Then the assignment operation will *always* yield `x`'s value `5`. Thus the branch will *always* be taken. A branch that is always taken will confuse future programmers that read the code, because a branch suggests that it will only be taken when some condition is true. In case the condition is always true, the entire `if`-statement can be removed and replaced by the branch's code.

Considering all of the guesswork done in this analysis, it is most likely that the programmer meant to write

```
if (x == 5) {
    // do some action
}
```

instead. This variant of the code (note the triviality of confusing `==` with `=`) does not change the value of the `x` variable and only enters the conditional branch in case `x` is equal to `5`.

In 2018, the 4 most forked C++ projects on github.com (`bitcoin`, `cocos2d-x`, `opencv`, `tensorflow`) contained 20k equality operations (`==`) in `if` conditions, and further 30k `if` conditions without an equality operation. In constrast, there are only 150 assignments in `if` conditions. Since equality operations are much more common than assignments, it is prudent to ensure that such a trivial error (forgetting to type the second `=` character) will not occur.

As with many code constructs that are possible but often erroneous, there are also valid use cases. A function call might yield a nonzero error code in case of an error, and a zero value in case no error occurred.

```
if (error = some_function(some_argument)) {
    // print error and terminate program
}
```

The above code example uses this kind of assignment condition to only enter the branch in case of an error, and also keep the value of the error available for introspection inside the branch. The alternative

```
if (some_function(some_arguments)) {
    // print error and terminate program
}
```

has no access to the error code returned by `some_function`. The alternative suggested by the MISRA-C standard

```
error = some_function(some_argument);
if (error) {
    // print error and terminate program
}
```

is more verbose (it requires an additional line of code). While it is argued that it is more explicit, the counterargument is that it is such a common pattern in code that the additional verbosity actually makes the code less readable. Developers will expect something more complex to be going on when an additional line of code is used instead of the shorter expression. On the other hand, of the in total 50k `if` conditions in the projects mentioned earlier, fewer than 150 have legitimate uses of assignments in their conditions. Even if the additional line of code suggested by the MISRA-C standard were objectively "worse" than doing the assignment in the condition, it would affect fewer than 1% of all `if` statements in the example C++ projects.

Ada Ravenscar

The Ada Ravenscar profile [10, 18] is a subset of the Ada language that decreases the chance for accidentally introducing non-real-time constructs and makes the programs easier to analyze. In addition to removing various Ada features, a best effort analysis for detecting blocking actions within protected operations (critical sections) is performed. A blocking action like accessing files or the network has no guarantees on the amount of time it will take or whether it will terminate at all. If such an action were allowed within protected operations, other tasks waiting to access the same operation would also be blocked.

A subgroup of the ravenscar restrictions is focused on forbidding dynamically changing the settings or number of interrupts, tasks, priorities, since such changes make the program and especially its timings entirely unpredictable.

Further restrictions target obvious non-real-time components like implicit synchronization between tasks or calendar access which might suddenly change timestamps due to daylight saving time or similar changes.

Ada SPARK

The SPARK[5] language is not a pure subset of the Ada language - it is a subset augmented by semantic comments. Any SPARK program can be compiled with an Ada compiler, but the additional SPARK checks encoded in Ada comments will only be processed by a SPARK checker. With the Ada 2012 standard, the contracts API (pre and post conditions) has been added to the language itself and is not part of SPARK anymore. SPARK uses the contracts differently though: instead of inserting

[5] a subset of Ada which allows formal proofs

runtime checks like Ada does, SPARK checks whether the contracts can be proven to be always upheld.

This methodology is used throughout SPARK by eliminating all exceptions from the language and instead requiring a SPARK prover to prove that the situation raising the exception can never occur at runtime. An example of such a proof is the elimination of division by zero. Any division by a variable can possibly result in a division by zero. If the division is in a branch of a condition checking whether the divisor is not zero, then the division is safe.

Pre/Post-Conditions

A precondition is a boolean expression that is required to be true *before* a function is called. The expression has access to global state (or at least the global state that the function will access) and all arguments of the function.

A postcondition is a boolean expression that is required to be true *after* a function is called. The expression has access to the same values that the precondition has and additionally the expression can observe the return value. Some languages with postconditions create an additional copy of all (or some) arguments in their pristine state before the function call, so that the postcondition can additionally access the unchanged arguments.

Pre- and postconditions can either be verified dynamically, by aborting the program through the appropriate means when the condition does not hold, or statically. In the static verification case, the compilation is aborted with an error if the compiler cannot prove that a condition is held for a specific function call. Manually adding runtime checks before the function call may thus allow the compiler statically allow a function call that it didn't allow before the runtime checks were added. An example is a `sqrt` (square root) function that has a precondition requiring its argument to be zero or positive.

```
#[pre="x >= 0"]
fn sqrt(x: f64) -> f64 { /* implementation irrelevant for example */ }
```

The above function declares that the argument must be zero or positive via a precondition. The correct usage of the function is verified in the use sites below.

```
fn test(x: f64) {
    let y = sqrt(x);
    //~^ ERROR: precondition x >= 0 not statically held
    if x < 0 {
        println!("x is negative");
    } else {
        // this is fine, since in this branch of the
        // `if` condition, the precondition of `sqrt`
        // is always held
```

```
        println!("sqrt({}) = {}", x, sqrt(x));
    }
}
```

Dataflow Annotations

The SPARK language allows annotating functions with the dataflow between input arguments and output arguments, return value or globals. A missing annotation assumes that all input arguments contribute to the value of all output arguments. The SPARK prover then verifies that the code inside the function follows the specified dataflow. An example is the swap procedure, which has a dataflow from `X` to `Y` and vice versa, but the output of `X` is not allowed to also depend on the value of `X` before the function call.

```
procedure swap(X: in out SomeType; Y: in out SomeType)
    with Depends => (X => Y,
                     Y => X);
```

More advanced `Depends` annotations can also ensure that no dataflow exists from `Y` to the return value, by leaving out that dataflow edge.

```
function test_and_set(X: in out SomeType; Y: in SomeType)
    with Depends => (X => Return,
                     Y => X,
                     X => X);
```

Provably Safe Parallel Computation

Similar to multiple humans operating a machine simultaneously, parallel computation comes with the inherent risk of agents undoing the others' work or even destroying it irrevocably. Consider the case of turning a clock hour handle to `12`. A simple algorithm might check the current time, subtract the time from `12` and then move the handle by that many hours. If the algorithm is simultaneously run twice, both instances might read the current time (e.g. `9`). Both instances thus realize they need to move the hour handle by 3 hours. Moving the handle twice by 3 hours does obviously not yield the appropriate result.

These situations are called `data races` or `race conditions`. Most languages and libraries with parallelism capabilities enable some form of race conditions and thus also provide mechanisms to prevent race conditions from occurring. While these mechanisms allow writing code that is free from race conditions, only few languages guarantee that *any* code written in them is free from race conditions. The languages Rust and SPARK are addressed here, because they both guarantee that code is free from race conditions before ever executing the code and are used extensively outside of academia.

The SPARK language defines thread safety in terms of parallel tasks communicating solely via synchronized objects [1]. A synchronized object is either an atomic object and thus each operation is guaranteed by hardware to be uninterruptible, or the object is of a synchronized type. A synchronized type is either a type protected by the language via underlying hardware synchronization primitives or an array or aggregate type consisting solely of synchronized types. In order to enforce that only synchronized operations are used to communicate between tasks, the language enforces that all global variables are either synchronized, or not visible outside a task. Thus, since all unsynchronized global variables are only ever visible from a single task, no race conditions can occur.

The Rust language implements the synchronized type feature via a `trait` called `Sync`. In this situation a Rust `trait` can be considered to be equivalent to a Java/Ada `Interface` or a haskell[6] `typeclass` allowing types to be grouped together by an attribute they have in common. Every type which is made up of types implementing the `Sync` `trait` implements `Sync` automatically. All threading related functions require that objects passed to them are of a type that implements `Sync`. Since global variables are accessible by all code, they are required to be `Sync`. Rust does not have global variables only visible in a specific task/thread and its ownership semantics penalize global variables in general. These ownership semantics enable the same code as SPARK's single task globals.

2.2.6 Static Analyses

The search for software bugs can take various forms. In the software industry unit testing and code reviews (automated and manual) are the prominent tools to achieve that goal. Both unit testing and code reviews suffer from the fact that they will only uncover bugs that are known and thus looked for. On the opposite end of the spectrum one finds formal analyses as often applied by academia, military, railway and the aviation and space industry. Formal analysis, while offering *correctness*, requires experts in the field of formal proofs [70] for writing the proofs and interpreting the results when faults are detected. At the same time there is a significant cost due to the replication of parts of the source code in the proof.

Static Analyses offer a midway between those two worlds [97]. The analyses themselves are designed and written on the basis of insights gained in previous projects. This process requires deep knowledge of compiler internals, language design and experience with issues occurring in large software projects. These analyses are then integrated as part of the automated code reviews and their results can easily be interpreted by the developers of the software project.

One of the earliest examples of Static Analyses tools is Lint [57], a collection of analyses for the C language, developed in 1977 in Bell Labs. Later examples for the C language include the MISRA-C conformance tools [153] and the various compilers

[6] a popular functional language

enforcing the MISRA-C rules. With the advent of interpreted dynamically typed languages like `ruby`, `python` and `javascript` static analyses have again gained traction, due to the many conventions that have to be upheld to make developing large projects in these languages feasible.

2.2.7 Project Specific Static Analyses

Libraries often have certain constraints on their API that users need to uphold. Some libraries might be less efficient (e.g. python's data analysis framework [90]) when the constraints are not upheld, others might throw exceptions (e.g. modifying data structures while using Ada's iterators [55]) or trigger assertions, and on the extreme end there are libraries that outright cause undefined behavior if their API is not used correctly [103].

Most (if not all) such invariants can be encoded in the type system of languages with strong type systems. Done correctly one can guarantee that a successfully compiling program upholds all encoded runtime invariants. Like with all silver bullets, there's a flip side too: excessive use of the type system to enforce invariants often leads to hard-to-read code and complex, long diagnostic messages if invariants are not upheld. Additionally it can be hard for developers to figure out all the puzzle pieces needed to achieve their goal [52].

Some projects have thus opted for the use of simple-to-use-but-unsound APIs and guarantee soundness via additional custom static analyses. One prominent example is the custom garbage collector of the servo[7] project [4]. A garbage collector needs a root node in the managed heap in order to be able to figure out which allocations are still alive. This root node is not allowed to change its position in memory, because there are outstanding pointers to it. The static analysis guarantees that the root node is not moved or discarded before the managed heap is completely deallocated. Further analyses aid in preventing accidentally leaking managed memory to code knowing nothing about managed memory.

Since it is inherently impossible to write a complex C/C++ API that is guaranteed memory safe, there has been some previous work towards writing compiler extensions that analyze the source code [7, 32, 123]. These works focus on compiler experts writing static analyses and integrating them with the compiler at hand [5].

Teaching project concepts to developers new to a project can be a very time consuming, repetitive and generally stressful component of code review and project management [8]. Some of this work can be offloaded to project specific static analyses that catch these issues and thus not only save time for the experienced developers, but also for the new developers who get feedback on their code the moment it is written instead of only during review.

[7] a browser engine written in Rust

2.2.8 Safety through Programming Languages

Make illegal states unrepresentable –Haskell mantra

The choice of programming language used in a project can have an enormous impact on the error rate. Historically various steps were taken to reduce the error rate in writing software. Initially one had to manually punch holes into punch cards used as software-input for computers. Since the input was not always decided on a single hole, but from combinations of holes encoding some binary value, humans easily made mistakes by punching the wrong holes.

The next step were punching machines that allow the programmer to press a key representing the intended binary value, which caused the correct combination of holes to be punched. While punching machines enabled programmers to enter entire values at the press of one key, the values had no semantic value enforced on them. Thus commands and letters were intermingled. If a command required that the next value is a letter, the user was still able to input another command, which was either interpreted as an error by the computer or simply caused different behavior depending on arbitrary internal states.

This lead to the first assembler languages which separated the machine code from the logical representation. The assembler code written by developers was translated by a compiler to machine code. Due to concerns with modularity, programming languages were designed which abstracted commonly found patterns like loops and branches into reusable templates which produced assembler code. These procedural languages improved readability of the code due to a more natural language and reduced errors introduced by repeatedly writing the same constructs with only minor changes.

Object oriented languages replaced many of these early programming languages with another abstraction layer. These new objects were often able to statically inherit behavior from more abstract types, thus increasing modularity and scalability. More comprehensible code was the result, which lead to a further decrease in bugs, since the developers needed to spend less time on *how* to write code and could spend more time on *what* code to write.

In parallel to this evolution the functional languages were developed from the purity of mathematical computation independent of concerns about running on hardware. While functional languages are known for their lack of unplanned side effects and low error rate, they are impractical for many safety and security use cases due to their high overhead and lack of real time capabilities. Thus this chapter will only address procedural programming languages.

After the advent of object oriented languages, the widespread availability of personal computers and their connection to the internet "easy" languages like Java, Javascript, Ruby, python and PHP increased in popularity. This is mainly due to the low entrance barrier, quick prototyping capabilities and detailed runtime errors of these languages.

The internet also lowered the barrier for viruses, worms and other malware to spread itself. While initially anti virus software, firewalls and intrusion detection systems were able to counteract these assailants, recent findings show that fixing the faulty software is more effective than introducing new software with its own defectss into the system [11].

Many low hanging fruit of system hardening were eliminated quickly by ASLR[8], static analyses, fuzzers and other techniques. While this made attacking code harder, the capability to write code containing security defects stayed within the languages.

Another round of programming language development lead to memory safe *and* compiled languages. The memory safety was guaranteed by a garbage collector (Go, Swift) or by language inherent static analyses that proved memory safety at compile time (Swift, Rust).

Choosing the correct language for a given project is out of scope for and will not be handled in this thesis. Nevertheless there has been some work in comparing languages in anecdotal and scientific ways. Anecdotal evidence shows that the safety introduced by languages often conflicts with performance and/or real time constraints. Discussions for comparing languages often lead to so called programming language wars, where neither side can yield to the arguments of the other side. While scientific comparisons of code written in different languages are rare, the few studies that exist do not compare bug rates or only come to the trivial conclusion that new programmers' code contains more bugs than experienced programmers' code [128]. Controlled experiments have shown that statically typed languages significantly reduce development and debugging times. The usage of design patterns has also been shown to improve development time and the correctness of the code [138].

Statically typed languages and the usage of complex structures not supported natively by the language (like design patterns) lead to a significant decrease in comprehensible compiler diagnostics [133]. In [12] a study on compiler errors was carried out, comparing regular Java error messages displaying the language specification violation with enhanced error messages highlighting the error from a programmer's perspective. Many compilers are undergoing extensive changes to their diagnostics output in order to improve the situation [125, 147, 148].

2.2.9 Compiler extensions

This section shows the various methods employed by compilers to support nonstandard use cases. These generally include everything that is not simply invoking the compiler on a set of input files to produce a binary output file. Examples are

- Static Analyses,
- Abstract Interpretation,

[8] address space layout randomization

- Code generation,
- Model checking, and
- Language extensions.

While in some compilers support for such use cases is standard due to being part of the language, generally these use cases are either second class or entirely unsupported.

Static analyses

Programming a static analysis tool for compiled languages requires one to duplicate or reuse large parts of the compiler. In the simplest version, a tokenizer, a parser and an AST are required. This allows the static analysis to traverse the AST and notify the developer of stylistic mistakes and of issues that do not require type information to verify. The next step is to process the AST to do name resolution. After name resolution, new kinds of analyses can be written, which require the ability to look up definitions (function and types) from use sites (function calls, variables and operations). While the AST and name resolution allows many kinds of analyses, analyses for control and data flow are very complex, because the AST represents the code in a structural way. To alleviate developing more powerful analyses, the tool needs to partially compile the code into an IR[9] that is suited for control and data flow analyses. At this point, the tool has mostly replicated the compiler (modulo various checks like type checking and aliasing analyses that the compiler already does). This is obviously not a convenient way to develop static analyses, and it requires keeping the entire processing chain in sync with the compiler. Instead, many compilers offer various ways to interact with them to be able to be consistent with the compiler, reduce code repetition and increase the convenience of writing and using the static analyses [32].

Furthermore, some languages' generics system either accidentally or by design ended up being Turing complete. Since expanding generic items is a necessary prerequisite to analyzing much seemingly trivial code, the tools necessarily include a Turing machine capable of interpreting the generics. A prominent example is a C++ template which compiles indefinitely, computes prime numbers in the process and emits them as compiler errors [135].

For any given Turing machine it is possible to construct an input that requires exponential amounts of time or memory relative to the input's length, compilers usually include shortcuts and optimizations to reduce that exponential growth. Without these shortcuts, the tool would most likely be useless for applications with a large enough code base, because it would require hours, days or potentially years to complete running just a single static analysis.

[9] intermediate representation

The Rust compiler

The exemplary compiler chosen for this work is the Rust Language Compiler which itself is written in the Rust language. It was chosen over compilers for other languages and Rust compilers written in other languages [131] due to the active development, easy integration of compiler plugins and the error avoiding features of the Rust language.

A compiler is a piece of software transforming user input into machine code. Usually the user input is some form of text, but there are also development environments which feed structured data to the compiler [137]. Starting with the user input, various intermediate steps process the previous step's output into a new form, which is again passed as input to the next step. A simple example is an assembler, since it consists of 3 steps (shown as edges in Figure 2.3).

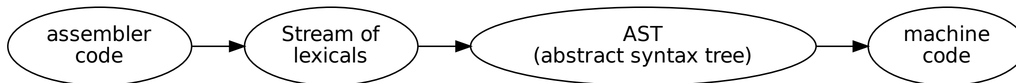


Figure 2.3: An example assembler pipeline converting human readable assembler to machine code

The first step in a compiler is the **lexer** which processes character data (e.g. from a file, a pipe, the network, etc.). Since user input can contain arbitrary (and thus invalid) characters, the user input is converted into a list of lexicals. Lexicals are any valid symbols that the source code can contain. This is not limited to deciding whether a raw character (e.g. `0`, `1`, `2`, …) is allowed, but can merge multiple characters into a single lexical. Keywords of the given language (e.g. `if`, `for`, …) will be converted into a single lexical representing that keyword. Contiguous sequences of digits will also be represented as a single lexical, containing the entire number. The following table depicts the lexer transformations for a simple math-language:

characters	lexical(s)
'+'	Plus
'-'	Minus
'0'	Number(0)
'42'	Number(42)
'-42'	Minus, Number(42)
'5 + 6'	Number(5), Plus, Number(6)
'a'	Error
'5 ++ 6'	Number(5), Plus, Plus, Number(6)
'5 + + 6'	Number(5), Plus, Plus, Number(6)

It can be seen, that such a simple lexer does not know about negative numbers and instead treats them as a sequence of the "Minus" lexical and the digits themselves grouped into a "Number" lexical. The combination into a negative number is part of the parsing step. Any characters that are not legal will abort the lexing process and report an error. More advanced lexers may also know about paired symbols like `{` and `}` or `[` and `]`, representing such pairs as a single lexical, containing a list of all lexicals between the paired symbols.

The Rust Compiler's lexer is a handwritten lexer, which is tested against a lexer automatically generated from a symbol table. The handwritten lexer can recover from errors like mismatched paired symbol (for example `{]`), by applying heuristics on the encountered symbols. This and several other similar error recovery features significantly improve the diagnostics of the handwritten lexer over the diagnostics by the generated lexer.

Not every sequence of lexicals is a valid program. In the example math language it would not be legal to have two numbers separated only by spaces, without a mathematical operator in between, or to have the program end with an operator, since the given operators require a number after them. Thus, the lexicals are parsed into an **Abstract Syntax Tree** (AST). This AST is then an (often recursive) structure, which can represent any syntactically valid program. Figure 2.4 shows the AST data structure and the relationships between its components. While the data structures aren't a tree, instances of it are.

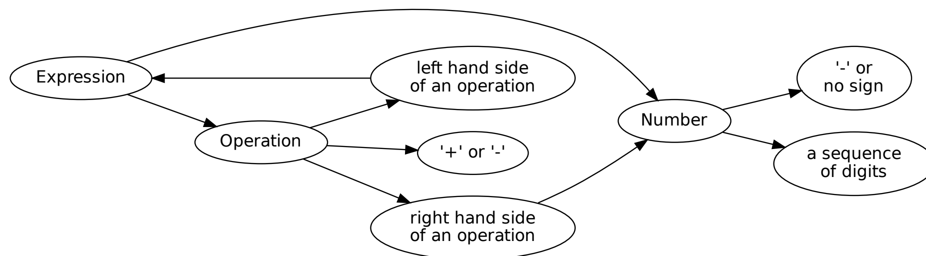


Figure 2.4: Subset of the AST data structures. Arrows point from components to their subcomponents.

The right hand side of an operation is not an expression, since '+' and '-' are left associative, meaning `5 + 6 + 7` is represented as `((5 + 6) + 7)`, where the parentheses represent an expression of the AST.

The Rust Compiler's AST contains several (mutually) recursive elements. The root element of any compiled program is the `crate`. It contains the root `module`, which is a variant of the recursive element `item`. The possible `item` variants are listed in Appendix D.1. Further recursive elements are type references and expressions. Type references can either be a type name or an aggregate type like arrays (see Appendix D.2).

Rust's expressions have eliminated all operator precedence in the AST, since every expression can only consist of a single operator with sub expressions. Figure 2.5 shows the ASTs of parenthesized expressions without explicitly storing the parentheses in the AST itself. The AST of the expression `a + b * c` is compared against the AST for `(a + b) * c` in Figure 2.5

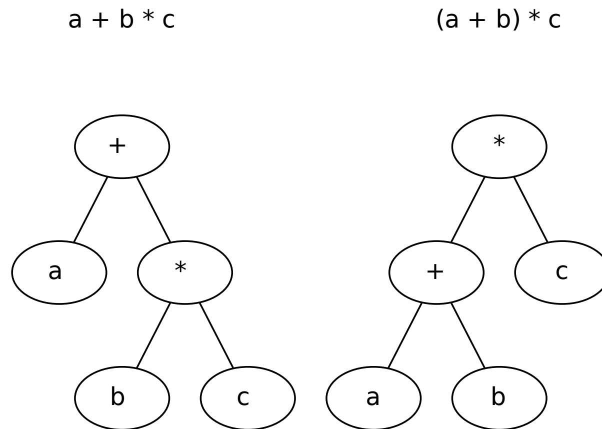


Figure 2.5: Expressions and their AST. Root expressions are on top, with their components as branches below.

The operator precedence has been eliminated in the parser, to simplify further processing of the AST. The full list of expression variants is available in Appendix D.3

Expressions, types and items each are a recursive structure and are mutually recursive with all others. Block expressions can contain items and functions, while constant and static items can again contain expressions. Cast expressions contain type references, and array types contain an expression for the array length.

Expressions also often contain patterns, which can be recursive, but contain no mutual recursion with other structures. The Rust Compiler contains the following pattern variants:

- a wildcard pattern (_) matching any value,
- an identifier pattern for extracting a struct field, optionally with a condition pattern,
- a struct pattern for matching struct fields,
- a tuple struct pattern for matching tuple struct fields,
- a path to a constant value to match against,
- a tuple pattern for matching against tuple fields,
- a box pattern for skipping one allocation level (or extracting the value from an allocation),
- a reference pattern for skipping one reference indirection level,

- a literal value to match against,
- a range pattern for matching against a range of float or integer values,
- a slice pattern for matching against arrays and slices of different sizes and contents, and
- a macro expanding to a single pattern.

The **parser** converts a sequence of lexicals into their corresponding AST or aborts with an error in case the lexicals cannot be represented by the AST and are thus syntactically invalid. Any sensible language's syntax can be represented by a context free grammar (Chomsky type 2). The parser of such a language can be automatically generated from the rules of the grammar. Many parser generators take an extended Backus-Naur-Form as their input, which is a standardized (ISO/IEC 14977) self-describing language for describing the syntax of languages [89, 136].

Unfortunately, the error reports obtainable by such an automatically generated parser are hard for humans to interpret (citation needed). Therefore it is preferable to write a parser by hand and tailor the error reports to the given language.

The Rust compiler contains a handwritten parser which is frequently tested against a parser generated from an EBNF[10]. The handwritten parser has the ability to recover from parsing errors and keep parsing the further correct parts of the code. This recovery has several schemes:

1. look for the next `;`, and continue afterwards with the next item or statement
2. match brackets and ignore their content
3. look for the next expression separator (mathematical operators or `.`) and treat everything up to it as a single erroneous expression

Once the AST has been fully parsed, the Rust compiler begins the `resolve` step which is in charge of name resolution across modules and crates. Name resolution is a lazy algorithm, because Rust's modules can have recursive references. At this stage it is possible to construct a type `A` which has a field of type `B`, while type `B` has a field of type `A`. Similar recursion errors are caught in a later step, but the `resolve` step ensures that every name and path in the code points to an existing element and caches the result of this resolution so later steps can quickly look up the actual element when given a name or a path.

Rust has two independent namespaces: values and types. Within a single group, no name may occur multiple times, but a name may occur in the value namespace and in the type namespace simultaneously. While this theoretically permits both a variable and a type named `foo`, there are soft capitalization rules (`CamelCase` for types and `snake_case` for variables) that the compiler informs the user about. Every scope (module, impl block or function body) has its own namespace pair to allow functions of the same name to exist e.g. in order to allow multiple types to implement a `new` method without causing a name conflict. This means that there

[10] extended Backus-Naur form

can be both a `TypeFoo::new` method and a `TypeBar::new` method without these methods standing in conflict with each other.

If a referenced path or name does not exist in the current scope or a name exists multiple times within the same scope, an error is emitted. In order to maximize the helpfulness of the error message, a second round of name resolution tries to find the name within other modules or crates and even searches for typing mistakes with a levenshtein distance of 2. If the second round succeeds with one or more resolutions, these resolutions are added to the error message as suggestions so the user can quickly find a matching name.

Even in the simple math language the compiler would be unnecessarily complex if it had to compute the result of the program by looking at the AST directly. The reason for this complexity is that the conversion to an AST only guarantees syntactic correctness of the given program, but there might be semantic dependencies between elements.

Therefore the AST is further converted into an **intermediate representation** that knows about the relationship between negative numbers and the `+`/`-` operators. Since the right hand side of an operation can only be a number (optionally preceded by a negative sign), all `-` operations can be eliminated by toggling the sign of the right hand side number. This next level representation of the simple math language is visualized in Figure 2.6.

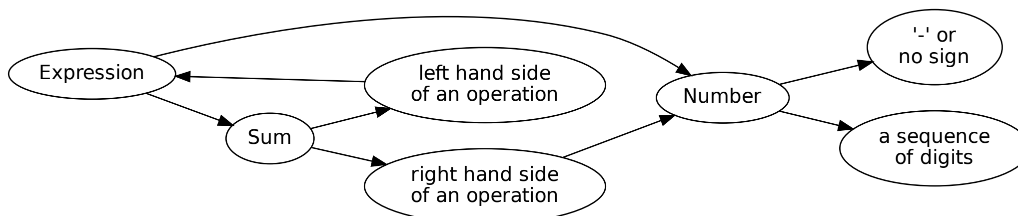


Figure 2.6: The data structure representation of the next level representation. Arrows point from components to their subcomponents.

The Rust compiler has multiple intermediate representations with decreasing similarity to the original source and increasing generalization towards the final machine code. The HIR[11] does not contain syntactic sugar like `for` loops, `while` loops, `if let` statements, `while let` statements and the `?` operator.

The step converting from the AST to HIR is called "lowering". During this step the lowered parts are marked so later error reporting can produce specialized messages for lowered code instead of showing errors in the lowered code to users, because users have no control over how the code is expanded and thus would not be able to

[11] high-level intermediate representation

fix the issues. The parts that are kept from the original code are not marked, since the user can edit them as required.

For example, to lower a `for` loop, it is split into its parts: the creation of the iterator, a bare `loop` and a match to either extract the loop pattern from the next iterator element or abort the loop if the iterator has reached its end. The transformation is shown in Figure 2.7.

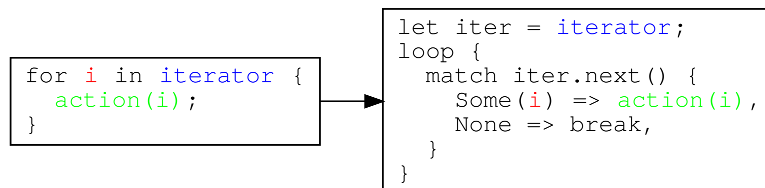


Figure 2.7: Lowering of a `for` loop into a `loop` and a `match`

After the lowering step, the AST is not required anymore and is deallocated in order to reduce maximum memory consumption within the compiler.

At this stage, since all paths and names are resolved and the representation is simplified significantly by reducing redundant elements, the internal types of all expressions and declarations can be computed. This step is called `typeck` (type-check) and simultaneously computes types and checks whether their uses are valid. While intuitively one would assume that the type computation and type checking steps would be separate (with the type checking coming after the type computation), the Rust compiler, similar to many other compilers cannot separate them, because the type of some expressions depends on the resolution of other expressions. An example is the invocation of a method `foo` on a variable `bar`. Depending on the type of `bar`, there might be zero, one ore many possible methods `foo` that can be applied. So in order to choose the method `foo` and thus compute the type of `foo`, one needs to know the type of the result that is expected. A very prominent example is the statement

```
let x = some_iterator.collect();
```

Without knowing more about the type of `x`, the correct method `collect` cannot be chosen. `x` would most commonly be of type `Vec<T>`, but it might just as well be of type `HashMap<K, V>` if `T` is `(K, V)`. This information cannot be inferred without further help from the user. Due to situations like this, some type checking will have to be done by the type computation step. It is possible to transform any type check failure into a generic structure that will have to be checked during type computation. Therefore the type checking stage would duplicate all checks from the

type computation stage. To prevent this duplication it was decided to move the type checking entirely into the type computation step.

The internal type representation is a simplification of the AST's types. There are no path or name types anymore; instead each type directly represents either

- a primitive type (`bool`, `char`, `f32`, `f64` or one of the integer types),
- a (mutable) reference or raw pointer to a type,
- a slice of elements of a type,
- an array of a specific number of elements of a type,
- a tuple of types,
- variants with a structure consisting of named or numbered fields with their own type,

or a special type that does not fit into the above categories like

- a trait object,
- a string slice,
- the never type (no value can exist of this type),
- or a function pointer type (containing just the function signature).

A further complication of the `typeck` step is the computation of compile time constants. In order for arrays (which have a statically known size) to be useful, it must be ensured that the types `[T; 4]` and `[T; 2 * 2]` are identical, because otherwise the assignment to `x` would not succeed in the following snippet.

```
let y: [T; 4] = ...;
let x: [T; 2 * 2] = y;
```

Historically the Rust compiler contained an interpreter, which evaluated HIR expressions directly. This interpreter only supported evaluating expressions. It was not possible to create new variables or to use loops or other control flow statements. In order to allow evaluating more complex code during constant evaluation, a new interpreter for Rust code `miri` [99] was co-developed by the author. Once it was production ready, the author integrated it into the Rust compiler, replacing the entire existing constant evaluation and processing infrastructure. `miri` facilitates the MIR[12] which encodes function bodies as a form of high level and platform independent assembler.

The MIR is generated from the HIR and the type information collected by the `typeck` pass. Reiterating on the for loop example lowering from AST to HIR, the `loop` and `match` are converted into a graph (see Figure 2.8).

[12] medium-level intermediate representation

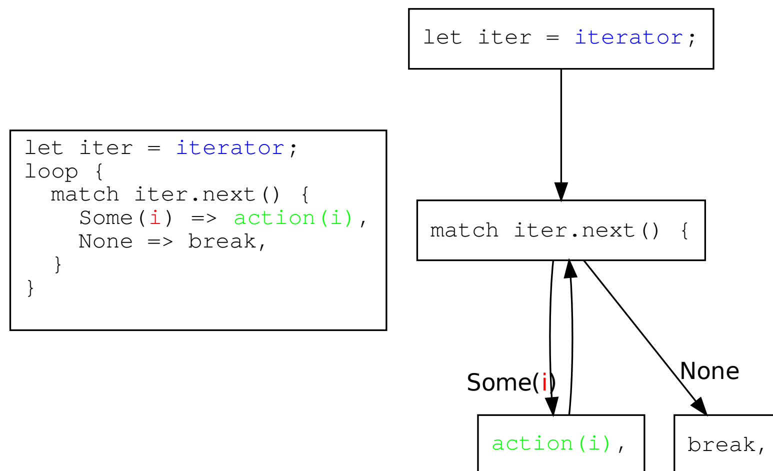


Figure 2.8: Lowering of `loop` + `match` to `MIR`

Compiler interfaces

Plugin interfaces

Modern compilers like gcc[13] [154], ghc[14] [155], clang[15] [156] and rustc[16] [157] offer a plugin interface, where the developer can hook into various steps in the compilation process and access all information that the compiler has generated for this step. The available information includes type information, control flow, AST, HIR, memory layout, etc. These interfaces also offer an API that allows plugin developers to reuse the compiler's error reporting infrastructure to report custom tailored errors and warnings. The Rust compiler also allows suggesting code snippets that users can directly paste over their own code or IDEs can automatically use to replace the wrong code.

Compiler as a library

Another variant offered by the ghc [158] and Rust compilers is to use the compiler as a library. An application using the compiler's library API can be distributed as a stand alone tool without exposing the usually complex compiler CLI[17] or API. Further, the developer gains even more control over the compilation process, as it is now possible to modify the process entirely without being constrained by the "regular" compilation process. Compiler steps can be left out, which prevents various optimizations from obfuscating the produced binary and speeds up the compilation

[13] the GNU Compiler Collection
[14] Glasgow Haskell Compiler
[15] a C language family frontend for LLVM
[16] the Rust Compiler
[17] command line interface

by leaving out safety checks (useful if the checks are run somewhere else anyway and would therefore be redundant). The compilation process can be aborted early since a static analysis tool has no need for producing an executable binary and syntax conformance checkers can abort even before types are computed. Additional steps can be inserted to hardcode new analyses into the compiler, hardcode compiler arguments to reduce CLI noise, run custom tailored (for the project at hand) optimization passes, and inject profiling and fuzzing code. Compilation steps can be replaced, e.g. allowing IDE[18]s to directly feed their AST to the compiler instead of going through a textual representation. IDEs can also replace the error reporting step to forward all errors directly to their own error display API instead of having the compiler produce output which has to be parsed again.

Stability Guarantees

Offering access to the compiler's API through the library interface or the plugin interface restricts the changes that can be done to the compiler by the compiler developers or requires plugin developers to adjust their plugin to any API changes done by the compiler developers. An example is a name change of a function from the API, which will cause any plugin that uses that specific function to stop compiling because the function can't be found anymore under the original name.

Few mainstream programming languages are known that guarantee API stability. The haskell compiler ghc 6.10.1 [158]

> "is still in flux and may change quite significantly between major releases while we provide new features or simplify certain aspects."

There exists a guaranteed stable limited code generation API to the Rust compiler [159]. This API offers exactly one stable function at the time of this writing. The function takes valid Rust code in textual form as input and outputs text to append after that Rust code.

The Ada language specifies the Ada Semantic Interface Specification (ASIS[19]) which

> "[..] is an interface between an Ada environment as defined by ISO/IEC 8652:1995 (the Ada Reference Manual) and any tool requiring information from this environment. An Ada environment includes valuable semantic and syntactic information. ASIS is an open and published callable interface which gives CASE tool and application developers access to this information. ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of software engineering tools while relieving tool developers from having to understand the complexities of an Ada environment's proprietary internal representation." [108]

[18] integrated development environment
[19] Ada Semantic Interface Specification

Unfortunately this API has not been updated to the language changes that came after the 1995 release. This means that while code written in the newer Ada versions can be checked with ASIS, any code using new language features is invisible to ASIS and will appear to not exist. This is infectious, so even code which itself is written in Ada 95 won't be detected if newer features are used in the same function, file or even package.

Ease of Integration

Compiler extensions generally require not only extensive setup effort by the developers of the extension, but also by the users of the extension in production. Commonly encountered issues are

- that the compiler version does not match the one required by the extension,
- that the extension requires building the compiler instead of being able to use pre-existing binaries,
- that building the extension requires complex or many setup steps,
- that using the extension requires changes to the users' build system,
- or that using the extension requires changes to the users' code.

Google's static analysis projects experienced that the acceptance of the tools hinged on their usability [109].

To address these issues while still allowing all use cases of the extensions, an experiment with the clippy[20] static analysis tool was performed. Instead of choosing one path, three ways to use the tool were exposed

1. Integration into the rls[21] and thus support any IDE with the LSP[22] feature.
2. Distribution with the compiler to require only a single setup with automated updating.
3. Manual building of the tool was tied to a checker for the minimally required compiler version. This allows users to quickly determine the version of the tool that matches their compiler version. This method has been deprecated due to the lack of users.

When using any IDE with LSP support, the clippy tool can be used by opting into the lints with the `#![warn(clippy::all)]` attribute. The opt-in mechanism exists to prevent the lints from overwhelming projects which so far are not using any static analysis tool. The exact rules around this have been co-authored by the author in [44] and were reviewed by the Rust community and the Rust compiler team via the RFC[23] process before becoming finalized.

[20]a static analysis tool for the Rust language
[21]Rust Language Server
[22]Language Server Protocol
[23]Request For Comment, a peer reviewed process for changing the Rust language

The process for installing clippy required every user to build the tool on their machine. While this does not incur any work on the user side, since it's automatically downloaded, compiled, and installed by invoking `cargo install clippy`, it does require some non-insignificant compilation time. Thus, the author additionally bundled the clippy tool with the compiler. The tool is now distributed with every compiler release (starting with version `1.29`) as an experimental feature and became officially supported by compiler version `1.31`. clippy includes various static analyses and static analysis design tools discussed in Section 6.3.

Chapter 3

New concept of an extendable compilation system with integrated static analysis

This chapter introduces new concept that allows achieving a high level of automation and self reflection which is high enough to ensure that the software development process stays efficient. The concept builds on compiler-centric development, which heavily relies on project specific compiler extensions. Significant focus is put on enabling developers of all experience levels to participate in this scheme, while allowing for more powerful systems once the developers gained more experience.

The core features enabled by the concept are

- static analyses authored by regular developers,
- checking code against models where code generation is infeasible,
- eliminating common downsides to code generation, and
- employing abstract interpretation when all other bug avoiding avenues become impractical.

The next sections introduce the general aspects of the new concept. After that, each of the above mentioned features is discussed in its own section.

3.1 Components of compilation systems

This thesis proposes to use a centralized system, where the compiler is the core entity. All other parts of the compilation system obtain and deliver information via the compiler. Figure 4.1 shows the information flow in the new system. For example: a code generator gets its input delivered from the compiler and delivers its generated code (or other data structures) directly to the compiler. There are no discernible steps, instead all components request their input from the compiler, which in turn

may trigger requests to other components. A static analysis tool would request the list of elements to analyze, analyze them and emit a list if diagnostics to the compiler. Any system consuming diagnostics would then request the diagnostics from the compiler. While some annotations may exist that allow e.g. diagnostic consumers to detect that the diagnostics are not from the compiler but from another component, these consumers are still strictly separated from the diagnostic emitting components.

No specific exchange formats, APIs or even communication protocols are suggested by this thesis. The reason for that is manyfold. Different languages, systems and communities prefer or are capable of using different exchange schemes, and it is neither the goal of this thesis to help deciding on a scheme, nor to suggest that there is an ideal scheme. Rather it must be considered that the scheme with the least implementation friction will ease the introduction the new concept to existing systems.

What is being proposed is the rewrite of all tools to obtain information about the source code, object files or metadata solely via the compiler. This may require new APIs in the compiler, but reduces heterogeneity of state about the build process that is kept. If all state is known by the compiler, other tools can reuse that state, reducing duplication in both code computing said state and in the state itself.

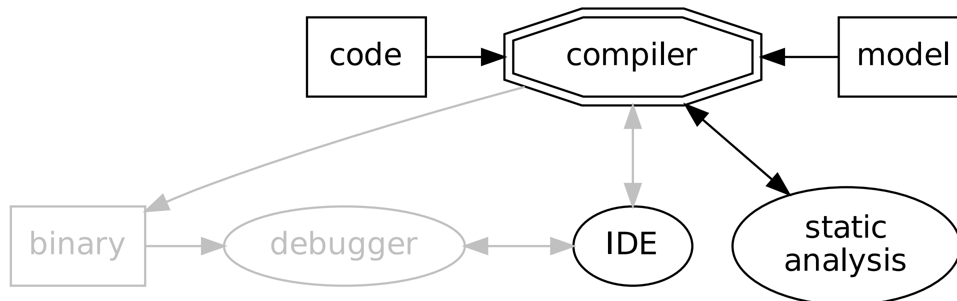


Figure 3.1: Depiction of the compiler-centric system. Parts that are the same in existing systems are grayed out. Arrows depict the dataflow direction

The compiler-centric system does not have any direct connections between the separate components of the system, instead requiring all communication between components to go through the compiler. This approach requires structured APIs in the compiler that separate the components. While this appears to be restrictive on a first glance, in practice it promotes the development of components due to the modular design. Extending complex designs with many interactions is a difficult task even for developers with experience in the system. With an extensible compiler in the center of the system, one only needs to ensure the correct usage of the API of

the compiler, instead of having to also know about all other components.

3.2 Automation in software engineering

Repetitive work causes developers to make mistakes due to inattention. Additionally developer time is wasted with mundane tasks and could be spent much more productively with solving actual issues. Commonly such tasks are automated fully or at least to some extend.

With the new concept, the developers can focus on solving their problem directly instead of being held up for unpredictable amounts of time when their problem has to be fixed by the static analysis developer. This new scheme is depicted visually in Figure 4.2.

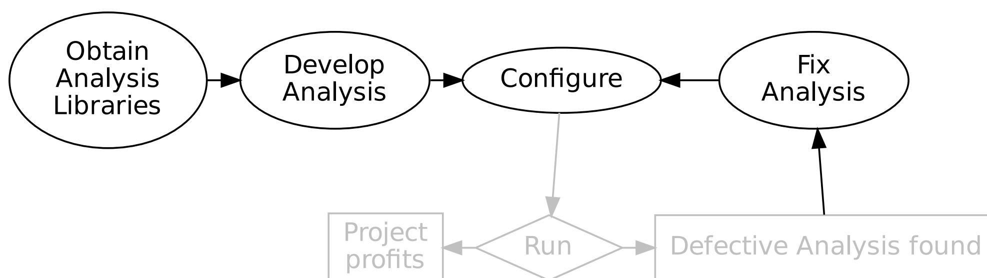


Figure 3.2: The process for developing. The parts that are the same as in existing schemes are grayed out. Arrows depict the possible state transitions.

3.3 Analysis of requirements for extensible compilers

Designing code analysis tools requires replicating many components found in compilers. The initial step of processing source files to produce an analyzable in memory representation should be identical to the compiler's source processing. The analysis tool can therefore skip checking for errors and does not need to report errors in a comprehensible manner, because the compiler should already have done so. Even so, the tool will need to replicate the parser and large parts of the type analysis passes in order to be able to produce high quality diagnostics. This duplication often leads to incompatibilities similar to those found between multiple compilers for the same programming language.

To prevent this duplication, many compilers offer APIs for programmatic access to the internal compiler processes. The APIs coverage and quality differ in many aspects like

- the stability across compiler changes,
- the coverage of language constructs,
- the direction of data flow,
  - allowing to process data and emit diagnostics,
  - or to even inject new data into the compilation process, and
- the ease of use from an analysis developer or analysis user perspective.

These aspects will each be addressed individually in the remainder of this section as they strongly influence the popularity and lifetime of analysis projects.

3.3.1 Coverage of language constructs

The compiler API might only expose a small subset of the language constructs that developers may want to interact with. APIs are kept minimalistic in many compilers in order to decrease the maintenance burden on the compiler developers. Access to internal structures and functionality is only exposed on an on-demand basis, and even then it is a design goal to keep them as minimal as possible. The exact opposite strategy exists, too, where everything is exposed, but no guarantees on stability are given (this topic is discussed more in Section 2.2.9).

3.3.2 Direction of dataflow

The popularity of compiler APIs is positively influenced by the possibilities offered by the API and negatively influenced by the frequency of breaking changes that the developers have to cope with.

Common features that API users desire are,

- analyzing code at various compilation steps,
- reporting diagnostics via the compilers diagnostic mechanisms,
- injecting new (commonly generated) code
  - from a completely different source (e.g. models),
  - or after analyzing the existing code and extending it (essentially compile-time reflection [86] with a subsequent code generation [22]), and
- conditionally enabling and disabling parts of the code base.

The ability to reuse the compiler's error, warning and information reporting scheme is a prerequisite to all other extensions. Without the ability to report that something is amiss, using further extensions would lead to hard-to-debug situations. The lack of diagnostic reporting capabilities is commonly circumvented by writing to the command line or storing all messages in a custom file. While both variants work for

some use cases like manually invoking the compiler and its extensions, it complicates the use in IDEs and CI[1]s due to the ad-hoc interface.

It commonly suffices to expose a system to report diagnostics at a file, line and column position. The diagnostics themselves can be simple text (Figure 3.3), GUI elements that can be interacted with (Figure 3.4) or even be shown directly at the error site in the code (Figure 3.5). Modern IDEs and editors are gaining capabilities for more individualized diagnostics, requiring compiler developers to follow suit.

```
gcc -c main.adb
main.adb:5:04: "foomp" is undefined
gnatmake: "main.adb" compilation error
```

Figure 3.3: A classical error message display in the terminal

Figure 3.4: A modern textual error in an interactive graphical tree

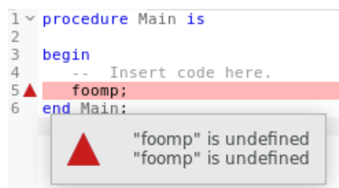


Figure 3.5: A visual inline error displayed over the error in the editor

Instead of having to write adapters for each of the possible output schemes, the compiler can expose a structured diagnostic reporting API. This way new diagnostics only have to target the compiler's API, while gaining every output scheme implemented by the compiler. Since other tools will be consuming the compiler's diagnostics either by via terminal output or via structured means like JSON[2] or by using the other side of the structured diagnostics API. Such schemes exist in the clang and rustc compilers, but are missing in other compilers. Lack of such an API makes the adoption of the proposed concept harder or potentially infeasible.

Let `N` be the number of diagnostic producers and `M` be the number of diagnostic consumers. Instead of requiring `N*M` implementations on only requires `N + M` implementations, since the diagnostic API decouples reporting from emitting. Figure 3.6 shows this with concrete producers and consumers.

[1] continuous integration
[2] a self-describing data storage and transfer format

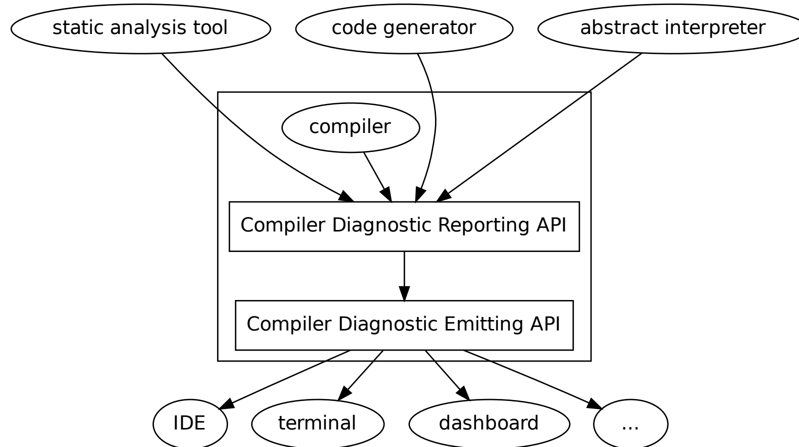


Figure 3.6: The arrows depict the information flow between components. Since the compiler's diagnostic API is part of the compiler, they are bundled together in this representation.

3.4 Features enabled by the new concept

This section gives a summary of the necessary building stones for the new concept. These are discussed in more depth in Chapter 5 and Chapter 6. After this introduction, each major feature enabled by the new concept is discussed in its own subsection.

Not all building stones are necessary prerequisites, but can rather be classified as desirable features or as adoption enablers. The following list begins with the prerequisites and is sorted from most necessary to least necessary:

- An extensible compiler allowing the extraction of information about code and the injection of new code in various levels of abstraction.
- Extension authors should have their focus on problems specific to their projects at hand instead of attempting to fix general problems in universal ways.
- A well documented and convenient API for creating such extensions.
- A reasonably restricted target language that allows many high level assumptions about any code at hand.
- A semantically versioned extension API reducing the overhead in maintaining compiler extensions.
- A feature rich diagnostics system allowing extension authors to emit high quality diagnostics.

3.4.1 Employing static analyses

Writing a static analysis tool from scratch is nontrivial. While its general features are those of a compiler that is lacking a machine code generator, the focus is much more on the algorithms that detect problematic code and reporting that to the user. While it is the author's opinion that diagnostic reporting is something that compilers should also strive to do well, this is neither true for many compilers nor the focus of this thesis. The algorithms for the analyses usually make up the least amount of code in a static analysis tool, with the majority of the code being the usual lexer, parser, type-checker combination of a compiler. It seems prudent to deduplicate this "largely uninteresting" (from a static analysis perspective) code to allow static analysis tool authors to focus on their main interest.

All open source compilers analyzed by the author [121]

- clang
- gcc
- ghc
- rustc
- scala

support some form of "extending the compiler" that either allows creating standalone tools that use the compiler as a library or allows attaching additional code to the compiler to be run during compilation.

These compilers allow a static analysis author to obtain ASTs, types and often even different stages of intermediate representations of the code. These data structures can be inspected to detect undesirable code patterns.

Furthermore, compilers already do various analyses themselves. If these analyses or their results are exposed, the static analyses can reuse these preexisting compiler algorithms to ease analyzing code. An example for such preexisting algorithms is looking up type names and obtaining a data structure that describes the actual type's information.

Once a pattern has been detected, the user should be informed of its existence or the detection emitted to a file for later analysis. Since this is something that compilers are doing anyway and often also expose an API for, static analysis authors do not need to invent a new system, but can instead reuse the existing infrastructure. This further permits seamless integration into any system already using the compiler's output. IDEs, CI or even just humans reading the compiler diagnostics do not have to be adjusted at all to support these new analyses as they are indistinguishable from regular compiler diagnostics.

Configurable lints

Some lints aren't purely algorithmic, but have certain constants that change the way the lint behaves. As an example, take the simple lint for counting lines in a source

file and complaining about too large files, recommending to split them. Irrelevant of the usefulness (or uselessness) of that lint, finding a good value for the number of lines, starting at which the lint triggers, is not easy. It seems thus prudent to make the actual number configurable, so that projects can use different settings depending on their needs.

Lints can be configured at three levels of ease and code granularity:

- by changing the constants in the source code of the lint (this applies to all triggers of the lint)
- by loading the values from a configuration file when running the lint (this applies to all triggers of the lint within the current project)
- by loading the values from source code annotations (this applies to all triggers of the lint within the annotations' scope)

These levels are not mutually exclusive. The default value can be a constant in the source code of the lint. Each project may have a configuration file overwriting the default value. And finally, the source code may overwrite the values again with arbitrary granularity. It may even do so multiple times (e.g. mark a function with one value, but inside the function use different lint configurations for inner functions).

As an example, consider the following pseudo-code of a function `foo`, which may not have any branching or loops. This is achieved by setting the permitted maximal cyclomatic complexity to 1, which means that the function body must be completely branchless.

```
@max_cyclomatic_complexity = 1
begin function foo
    x = 42
    y = x + 3
    print y
end function foo
```

If a function `bar` were nested inside the function `foo`, the cyclomatic complexity can be overwritten for the inner function in order to allow branching.

```
@max_cyclomatic_complexity = 1
begin function foo
    @max_cyclomatic_complexity = 5
    begin function bar
        // check if the user entered a `y`
        if user_input() == 'y' then
            print "yes"
        end if
```

```
    end function bar
end function foo
```

Since each method of configuration value granularity changing requires some form of code, no matter how little, it might be undesirable to make each lint fully configurable. It is left to the lint author to choose the features they need. In hand with design rules for project specific tools, one should not implement features that are unused in the project at hand.

Enforcing the correct usage of APIs

API design requires juggling usability and maintainability to obtain a practical API. Usability in itself also requires trade-offs. Improving the usability by exposing many functions that users may need also increases the chance of such functions getting used wrongly. While the correct usage may be documented, it is additionally desirable to encode API requirements in the type system.

Consider the following common pattern in the C language:

Before being allowed to call any functions of a library, one must call an initialization function that sets up global variables, connects with special hardware or does other sorts of initializations that other functions depend on already having happened. This invariant is documented, but not statically enforced. It would be perfectly possible to call the other functions without having performed the initialization, causing undefined behavior.

The same issue can be prevented in e.g. C++ or Rust by using the type system to encode this invariant. A value of the type `InitToken`

```
struct InitToken {
    InitToken();
private:
    int private_field = 0;
};
```

can only be created by using the constructor. The constructor can then do any special initialization necessary:

```
InitToken::InitToken() {
    // do the necessary initialization here
}
```

Any code which requires the initialization to have happened can then take a reference to such a token:

```
// Precondition: initialization has been done already
void some_action(InitToken& token) {
    // do some action expecting initialization to have occurred
}
```

There is no need for the `some_action` function to access the token argument at all. Just the fact that it is required for any caller to also provide a reference to the token, necessitates that the `InitToken` constructor has been called.

Unfortunately using type systems to enforce API requirements results in either useless error messages (e.g. "cannot create value of type `InitToken` due to private fields" when trying to create one by using a wrong Constructor) or requires additional effort by the developer to work with these type system "tricks". If every API applied such type system proofs rigorously, most functions would end up with very complex types and more token arguments than real arguments. Since this scheme neither scales well nor boosts productivity, a methodology based on project specific static analyses is proposed.

API lints

lints have the capability to track arbitrary information orthogonally to what is specified in the code. Going back to the initialization function example, this means that an analysis has to lint about calls to `some_action` that occur without `init` being called before. A library can define both of these functions

```
void init() {
    // do the initialization here
}

void some_action() {
    // do some action expecting initialization to have occurred
}
```

without there being any connection between them in the type system. Users of the library can then invoke them in arbitrary order and get informed by the static analysis if they are invoking them in a wrong order.

```
void foo();
void test(fn_ptr_type);
void bar();

void main() {
    foo();
    // not ok, all type information is lost when using pointers
    fn_ptr_type fn_ptr = some_action;
    // `test` might call `some_action` via the function pointer
    test(fn_ptr);
    init();
    bar();
}
```

Since the analysis doesn't know whether `some_action()` is called in `foo`, `test` or `bar` just by looking at the function call, it needs to observe the function bodies together with all the call sites. The `foo` function calls `some_action` without `init` having been called before the first invocation of `foo`.

```
void foo() {
    // not ok, there's a call to `foo` before `init` has been called
    some_action();
}
```

The `test` function can't know anything about the function pointer it is given, so the check is left to any code that creates function pointers.

```
void test(fn_ptr_type f) {
    // cannot reason about `f`, so the caller of `test` needs to ensure
    // `test` is not given a pointer to `some_action`
    f();
}
```

The `bar` function is only called after `init` has been called. It may call `some_action` at its leisure.

```
void bar() {
    // ok
    some_action();
}
```

This simple analysis is guaranteed to prevent `some_action` being called before `init`, but it also triggers on various legal systems. It is not necessary for such static analyses to have no false positives, as long as they have no false negatives (see Section 6.1 for a more in depth discussion). While more powerful analyses may allow function pointers by tracing where they will be called, this simple example already demonstrates that the guarantees for execution order of `init` and `some_action` can be kept without burdening users with token types or other type system complexities. The implementation of this static analysis can be found Appendix H.

Instead of requiring the user to do the easy but time-consuming work of tracing the initializedness via token types, the static analysis performs the tracing and either reports user errors or analysis limitations via hand-crafted diagnostic messages. These diagnostics can be fine tuned to the analysis at hand instead of being a general purpose diagnostic about type system violations, improving the comprehension of the issue and thus the speed at which the problem is resolved.

3.4.2 Model driven development

MDD aims to address the loss of abstraction that occurs when implementing a system. The program code itself intermingles implementation details with design decisions resulting in difficulties in verifying the correctness or at least consistency of the design.

There are two paths in using models to drive the implementation:

1. Generating code from the model
   - requiring the code to only fill in the implementation details
2. Checking whether the code conforms to the model

Both methods require handwritten code and a designed model. The difference is that the checker only permits the code to become the final code instead of adding new code to the handwritten code (see Figure 3.7).

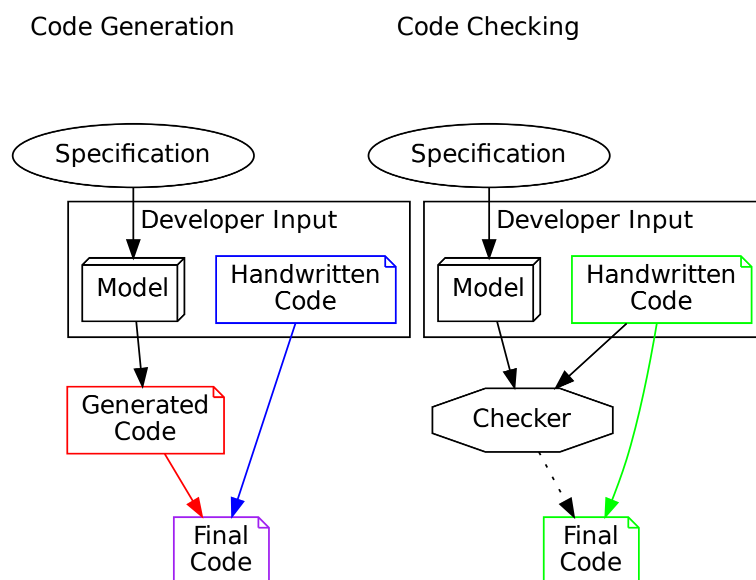


Figure 3.7: Comparison of the workflows for code generation and code checking. The dataflow is depicted with arrows. The dotted arrow is not actual dataflow, but solely there to visualize the blocking nature the checker has on the final code. The final code in the code generation case is a merged version of handwritten and generated code, as hinted at by the color merging.

The following two sections will discuss both variants in more detail, followed by a comparison of their respective advantages and disadvantages.

3.4.3 Checking code against models

Checking whether written code upholds all constraints required by a system model is essentially the same as a configurable lint whose configuration is said model. Instead of creating a fully generalized model and the corresponding analysis from the start, one starts with the most basic version necessary for the use case and only adds features when immediately needed.

This concept is best described by an example. Take for instance the class diagrams of UML, which in the 2017 edition make up more than 50 pages of detailed specifications[101]. Fully implementing the entire specification without using it is not just an enormous task, it completely misses the goal of using UML [160]. Having the ability to use every specified feature of class diagram models will overwhelm developers with choices [56] reducing their capability to make good choices.

The complexity of the model can be incrementally increased from a trivial structural model enforcing the existence of modules and types in the correct hierarchy to more complex models containing runtime constraints, type relationships and similar.

Such static analyses need to link model components to items in the code, requiring an identification scheme to be shared between the two. Compilers already require such a scheme internally to identify items (e.g. to resolve which function should be called in a function call expression) and frequently export a way for static analyses to reuse it. Checking whether the code adheres to the model has several advantages over classic MDD[3] via code generation. These advantages will each be discussed in the next paragraphs.

Instead of generating program code, writing it to disk, the compiler reading it back from disk and compiling that, the model is verified in memory against the handwritten source code's semantic intermediate representation in the compiler. Additionally, due to using the compiler APIs while the compiler is processing the source, no work is repeated between the compiler and the checker.

If the model's generated program code causes a type error then the compiler will emit a generic diagnostic message pointing into the middle of the generated code as being at fault. Even worse, it might be pointing into the handwritten code even though the mistake lies in the generated code. These kind of errors occur frequently when the generated code contains templates that are instantiated by the user code. Debugging such errors is very time consuming and frustrating.
Code checking on the other hand allows the checker author to write handcrafted error messages specific to the model. These messages can additionally refer back to the model instead of generated code, allowing the developer to track down issues much faster.

A project without a formal model can be transitioned to using a formal model by incrementally introducing parts of the model to the code. References between the model and the code are made either

[3] model driven development

- by annotating key source code points like type or function declarations with the respective model component's name, or
- by annotating the model component with the source code point's module path and name.

This scheme allows the model itself to evolve with the project instead of enforcing an existing model and thus often requiring developers to work around the ways that the model cannot fit the project.

3.4.4 Code generation from models

Common modeling systems like UML and AADL are very powerful and generic tools. They allow modeling almost every aspect of any system. Unfortunately, once code has been generated from the model, several issues come up:

1. Integrating the generated code into the build system
2. The generated code has to interoperate with the user-written code
3. The generated code has lost all semantic links to the model

The author proposes to thus move away from classical code generation where the intermediate code is written to disk and integrated into the build system just like user-written code is. Instead the code generation happens during compilation of the user code, by extending the compiler to invoke the code generator. The code generator is adjusted to not produce just textual code, but instead link every code snippet to its model component in a way that permits the compiler to report error messages that reference the model instead of the generated code.

Both existing methods have advantages and disadvantages making neither the unequivocal better solution. This chapter will weigh the methods against each other and then show how each can be implemented by reusing compiler APIs to allow a full integration of the models in the development cycle. The differences to classical model driven development are that

- only *used* model components are implemented instead of using the full UML and picking components, but still needing to support the entire suite.
- the intermediate generated code is not generated on a textual level, but semantically integrated into the larger code base.

3.4.5 Code generation vs code checking

Advantages and disadvantages of code generation

Generating code from models and linking the user-written code against the model requires little to no repetition of information between the model and the user-written

code. The generated code represents all of the model's semantics and is solely extended by the user-written code. This strict separation allows large changes in the model to be implemented more easily, since these changes only have a limited effect on the user-written code. Vice versa, changes in user-written code do not affect the model directly.

Additionally to the standalone features of code generation, there is also a concrete advantage over code checking: There is no need to implement a scheme for finding a model component's corresponding code elements. The model interacts with the code, but does not mirror it.

On the other hand, the generated code is often very bloated and not easily comprehended, even for small models. Many components of the generated code are uninteresting to the developer [145] while still requiring manual work by the developer to integrate the generated code into their code base.

Advantages and Disadvantages of Code Checking

Code checking allows developers more freedom in designing the code counterparts of the model. This can often result in better performing designs as the intent of the model is transcribed to the code, instead of just its structure. The redundancy from having both a model and user-written code for the same information does not just check that the code adheres to the model, but additionally ensures that the model does not contain any designs that are inherently impractical. Finally, changing the model does not cause conflicts due to code generation modifying user code or code referred to by user code. While the code needs to be adjusted to match the model changes, this happens on a much higher level, informing the user about conflicts instead of just emitting compiler errors that have no knowledge about the model.

The manual authoring of code and subsequent linking to the model is a mechanical and thus error prone task. Developers are required to manually implement state machine logic, inheritance graphs and similar constructs that are concise in the model.

3.4.6 Abstract interpretation

> *Testing shows the presence, not the absence of bugs - Dijkstra* [19]

Addressing software defectss via static analyses preemptively ensures the freedom from specific issues, but some issues are hard or impossible to detect statically. Static code analysis is inherently limited in its ability to detect abnormal situations by the abstraction of runtime behavior as compile-time constraints. Essentially the problem boils down to a variant of the halting problem: It is impossible to tell whether a program will have a problem at runtime without actually running the program. In order to cover the runtime cases not reasonably detectable by static

analyses, the software can be interpreted for testing purposes instead of executed on real hardware.

Interpretation of a compiled language has many of the advantages that dynamic languages have implicitly:

- Reproducibility (All input is under full control of the interpreter)
- Analyzability of program state
  - Additional metadata can be encoded (e.g. memory can log all accesses)
  - Inspection of program state like in dynamic languages
- Collection of statistics (e.g. code coverage, heat maps, …)
- Easy data injection (e.g. for fuzzing or testing fault detection)

Compilers usually do not go straight from syntactical representations of the source code to machine code, but instead have further levels of intermediate representations that progressively get more similar to machine code. These intermediate representations can be used as the "assembly" of a virtual machine. While this may not be able to interpret every plausible scenario, especially if the code attempts to interact with hardware, large amounts of unit tests are self contained enough to allow interpretation.

For a single compiler, only one interpretation engine must exist for the scheme to work. It is not necessary for each project to create its own like with static analyses. Nonetheless, projects may have very specific needs that would actually be wrong in other projects. They can thus extend the interpretation engine with their custom needs.

Such a scheme requires a reasonable coverage of the source code via the test suite, and while the author is strongly in favour of extensive test suites, this discussion is out of scope of this thesis.

3.5 Project specific compiler extensions instead of generally applicable tools

Software tools for static analyses outside the open source communities generally grow into single large, sometimes even monolithic, tools which support all real and imagined use cases that are of interest to developers [23]. This trend has multiple motivating factors. The most prominent ones are

- the (lack of) support for modularity in the language, model language or build system [17],
- companies selling everything to their users instead of just the bare minimum (higher per-sale profits, since even components useless to the user are sold) [67, 122],

- companies shipping a single product, where features that the user can additionally buy are enabled by changing product keys or configuration values,
- user-lock-in by only allowing data exchange between components of the single product [146], and
- preventing the need to support "creative" uses of the software by prohibiting any unintended uses.

This methodology forces projects to adjust their problem to the tool instead of the other way around. The tools chosen can severely limit projects and cause them to either fail to become successful in the short term or from being able to survive in the long term.

Unix systems follow the exact opposite strategy: They expose many simple tools which are useful completely independently of all the other tools, but allow users to choose a combination of the tools that fit their exact use case. Instead of having to learn how to perform well-known tool `A`'s action in tool `B`, the user can just attach tool `A` to tool `B` to process `B`'s output via `A`.

Adjusting a general purpose software to a new use is commonly very expensive due to the software architecture being at odds with the new use case. Small modifications in a seemingly unrelated part of the software can lead to unintended changes in existing behavior in a completely different part of the software. Even though adding new features is usually nontrivial, this kind of software tends to collect an arbitrary set of features without an overall design. The features are often only tested in exactly the use case that caused the feature to be implemented, even though the feature enables various other use cases. These untested use cases commonly are the overwhelming majority of features as shown by the enormous effectivity of running fuzzers on projects with preexisting large test suites [91].

While this argument disregards the complexity of interactions between tools, it is the root of the Unix philosophy [38] that

> "1. Small is beautiful." and "2. Make each program do one thing well."

These concepts have allowed complex systems to evolve from the smaller components, resulting in inherently modular design where each component is easily replaced or improved.

Furthermore, the scheme of developing small tools for specific use cases is a much more targeted solution to the actual problem at hand. Not only does this reduce the time and work needed to develop the tool, it also leads to a higher test coverage since the developer will test their use case either explicitly via tests or by actually using the tool. The hypothetical use cases covered by a more general tool are often not tested at all. Even if there are tests for these use cases, they frequently are unrealistic use cases. The moment the tool is used in a real environment, edge cases likely expose the limits of the tool.

Usually writing a small tool that takes care of one specific problem is a lot less effort than finding a tool that supports the use case, obtaining said tool, integrating it into the development process and finally configuring it to actually perform (just) the desired action. The configuration of general purpose tools commonly borders on actual programming while being significantly more restrictive than a general purpose programming language.

The maintenance burden of a small targeted tool is also much lower, since in the worst case the tool can be quickly rewritten with newer technologies. Regular maintenance is also simpler, since new users or even the original developer can obviously comprehend a small code base faster than a large code base.

An analogy to this methodology is the problem of digging a round hole with a diameter of one meter and a depth of one meter. The expensive but fast way is to rent an excavator to dig an unshaped hole of about the right dimensions and then fix up the hole to have the desired shape. Building an overly complex general purpose tool would be to design e.g. a drill with a one meter diameter and the machinery around it. Finally the solution proposed in this work is to use a shovel and a jackhammer to quickly get the problem solved before either of the other solutions even get their budget approved. If one wants do dig 1000 of these holes, then one can still evaluate the other options.

3.6 Lint categories

A lint [57] is a specific static analysis implementation, the interpretation of its result and a diagnostic message. While there exist linter[4]s, which are tools grouping many lints together, these are often too generic to solve many of the problems that developers encounter or can't detect semantically complex problems [94]. Similarly to the previous section's discussion about the advantages of developing project specific targeted tools, it is also possible to design a project specific linter. When designing a lint for a specific issue in one's project, there's no need to ensure that it is generally applicable to other software. If an analysis applies to unrelated code or just needs additional information that cannot be obtained from the source alone, one can enrich the code with attributes that the analysis can process [143].

lints can generally be categorized into

- stylistic preferences (often subjective and highly debated),
- performance issues (possibly specific to the users hardware setup),
- semantic complexity (that could be reduced by changing the code without changing the behavior) [20],
- correctness (either generic bugs or misuses of specific APIs).

[4] a tool containing one or many lints that can be run on program code

The author co-authored an RFC [44] discussing the above categorization and its separation.

As an example for where a lint for semantic complexity could be applied, consider the case of computing the sum of all elements of an array in C++:

```
auto sum = 0.0;
for(size_t i = 0; i < array.size(); i++) {
    sum += array[i];
}
```

The immediately obvious way of improving this code snippet is to use iterators instead of indexing:

```
auto sum = 0.0;
for(auto x: array) {
    sum += x;
}
```

Finally, common operations are directly available as functions in the standard library, so one can compute the sum over the elements of any iterator with the `accumulate` function.

```
#include <numeric>
auto sum = std::accumulate(array.begin(), array.end(), 0.0);
```

The code has neither changed in performance nor logic, but the readability for humans has increased significantly. This is due the reduction in "state" that one has to keep in mind during reading. If such transformations are applied across the entire code base, developers can assume that if they see a `for` loop without iterator syntax, that something nontrivial with the range of elements is going on (e.g. skipping the first element). If a `for` loop that looks like it's summing up elements is encountered, the developer knows that something more complex must be going on, because if it were a simple summing up, the loop would have been replaced by `accumulate`.

Usually the more verbose code is not written to begin with, but results from refactorings. Having complexity reducing lints available frees up the developer to think about the larger scope of the refactoring instead of having to get hung up on also considering the future readability of the code.

The `accumulate` example can be extended to similar patterns in user-written libraries and project-private data structures. E.g. a library offering a `Matrix` type can detect manually iterating over the diagonal elements of a matrix and suggest using a dedicated method instead. Examples in the wild are discussed in Section 2.2.

3.7 (Semi-)automatically resolving static analysis results

While automating the detection of code defects is already a boon by itself, it still leaves the (potentially inexperienced) developer with the (potentially hard) problem of addressing the code defect. The lint author, while writing the lint, is knowledgeable about the defect they are detecting. In most cases they know how to fix the defect. The burden of fixing the defect can thus be offloaded to the lint, too, by generating fixed code and including it in the diagnostic message. Experience with the clippy tool and the clang compiler [42] shows that this usually requires little additional code on the lint side.

Even in cases where some effort is required to produce fixed code automatically, the advantage is that it will allow users to fix the code with a single click in their IDE or at least via copying the fixed code from the diagnostic output. Instead of having to write the fixed code themselves, which might end up taking a few minutes, they can fix the issue within seconds. Writing fixes every single time a lint is triggered also has the danger of introducing new bugs. When designing the fix as part of the lint, the lint author only has to consider the potential pitfalls once in a controlled environment instead of at every use site.

3.8 Incremental steps towards full formal analysis

The larger a step from ad-hoc development towards full formal analysis is, the more effort and time has to be applied to adjusting the project to match the new constraints. In order to reduce the burden on developers and mitigate the risks that come with large refactorings, the step size needs to be reduced. Ideally, no step requires a single large refactoring, but can be applied incrementally.

E.g. instead of forbidding C-functions that process arrays, but do not take the length of the array as an argument, one can require that any change to the system must either reduce the number of arrays without explicit lengths or at least keep it the same. Simply outlawing undesired code or structures would require developers to adjust the entire code base, even the parts that they have little knowledge about. By allowing the incremental transition, the developers can learn to work with the new rules while only touching code that they understand. Another example is an aliasing analysis as it is commonly applied to Java code to enable more complex analyses. Such an aliasing analysis will fail to process code taking advantage of obfuscating features like reflection [45, 49]. A preliminary step could be to eliminate reflection from the code base, instead of having to match all requirements for the aliasing analysis in one step.

The new concept introduced in this thesis makes it feasible to design project specific analyses that pave the way for future refactorings or ensure that project invariants are upheld. The following non-exhaustive list introduces the most prevalent open aspects of existing systems that are mitigated or resolved by the new concept.

- Few static analysis tools are extensible and those that are require not only a special analysis language, but are usually very limited in the capabilities of the custom analyses.
- Static analysis tools replicate large parts of a compiler and are thus susceptible to the same problems.
- Designing static analyses is currently "best left to the experts". Deep knowledge of compiler processes is required in order to be effective and successful in implementing a new analysis.
- Integration of static analysis tools into the development process is not always trivial and difficulties with the integration are a major concern for users new to static analyses.
- Running a static analysis tool on an existing project will produce overwhelming amounts of diagnostics without prioritization or a gradual introduction.
- Requiring user input for repetitive work does not scale to large projects.

To address these issues, existing workarounds and best practices have been analyzed for their effectiveness and avenues for generalization were evaluated. The various methods of extending compilers were compared against each other by evaluating their use and the issues in existing medium and large open source projects. The APIs and processes necessary to address the issues are established. Social and technological aspects hindering acceptance or practicality are discussed and a path forward for incremental introduction is presented. Static analyses ranging from trivial syntax tree pattern matchers, over configurable type and dataflow analyses, to full-blown model checkers, are merged into a single concept erasing the borders between them and instead allowing a continuous spectrum of analyses. This allows "upgrading" analyses in small steps by increasing their scope, capabilities or configurability. Finally, to speed up the implementation of new static analyses, automated generation of the analysis from examples is discussed.

3.9 Summary

The central component of the thesis is the removal of the classical toolchain around the compiler and instead incrementally replacing them by directly interacting with the compiler. It has been shown how the centralization can reduce repetition of work and make new advanced features feasible. The four most prevalent features are

- static analysis,
- code checking,
- code generation, and
- abstract interpretation.

While static analysis and code generation are standard practice in software development and abstract interpretation has seen some adoption, code checking is inherently

impractical without the new concept proposed in this thesis. All these concepts are empowered by the new concept to allowing tailoring to project specific needs instead of just covering generic problematic situations. This level of fine-tuning allows makes incrementally introducing good software development practices easier and paves the way to advanced concepts like formal analyses, which unfortunately not yet see significant use outside safety critical industries.

The new concept has various implementation aspects that users need to be aware of. These aspects are discussed in Chapter 6.

Chapter 4

Comparison of the new concept with related works

While it is generally known that projects benefit from clean upfront design in the long term, many projects suffer from lingering defects introduced in the ad-hoc design phase that occurred in the early development of the software. The pressure for fast release cycles in not-mission-critical software development motivates developers and project managers to skip over necessary but time consuming tasks. Even though such decisions lead to short term time savings, the testing and bug fixing phase during deployment will be unproportionally lengthened [87]. Trying to incur a large scale change in software engineering has so far been unsuccessful. Instead of fighting the trend of short design phases and development cycles, agile software engineering practices embrace them and take generic software development towards the quality required by mission critical software. In order to enable the agile processes to achieve similar level of quality, the software development process needs to be heavily automated in order to reduce the amount of time developers spend on the overhead of the short development cycles. Simultaneously, additional tooling that requires no developer interaction, can be employed to automatically detect defects and risky code structures.

Unfortunately the current state of tooling is a heterogeneous, incompatible chain of tools with varying levels of integration. This means the developer sees a different subset of tools than the one used by the testing and integration system, while the deployment systems has yet another toolchain [28, 59].

Developing and improving these automation and quality assurance tools must become part of the software development itself. The entire software development process needs to be self-reflective and self-modifying [54] in order to allow the timely elimination of upcoming bottlenecks and inefficiencies while ensuring that deficiencies are eliminated and prevented by class instead of on a case-by-case basis [80].

This can be compared to manufacturing factories' policies to clean up the workspace when a product is finished. The cleanup process is adjusted by the workers them-

selves e.g. by choosing more convenient places for frequently needed tools.

It is the author's experience that while software tools like automated formatters and bug search tools are common practice in software development, adjusting the tools' code to increase the effectiveness is not. Instead most issues with tooling are worked around by creating checklists that developers have to go through when using the tool. These lists have a tendency to grow in an unconstrained manner, pushing more and more "boring" automatable work to human developers instead of allowing them to automate these processes.

This chapter will compare the various preexisting methodologies (already introduced in Chapter 2) for improving the quality of software in a scalable and exhaustive way with the new solutions proposed in this thesis.

The chapter begins with a side-by-side comparison of classical development environment structures with a component structure built by extending the root of all development: the compiler. An analysis of bottlenecks during software engineering is given and solutions to address them via automation are presented.

4.1 Components of compilation systems

Classic compilation systems generally consist of separate components that are bundled together. It is usually preferable to use more decentralized and independent/redundant components, but if the separate components only work if used in one specific configuration together with other components, all the advantages of decentralization are lost.

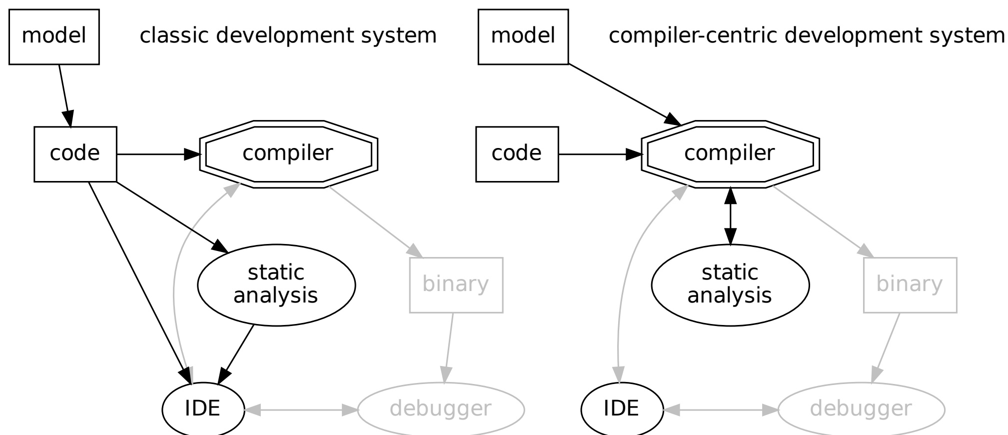


Figure 4.1: Comparison of the compiler-centric and the classic development system. Parts that are the same in both systems are grayed out. Arrows depict the dataflow direction

It is clearly visible that the classical development system has shorter paths due to

a higher connectivity, allowing freely obtaining information from other components as desired.

4.2 Automation in software engineering

The project specific nature of the automatable tasks requires targeted tools and cannot be meaningfully solved by a configurable tool. Instead, commercial tools like continuous integration services or test suites only offer a framework which is then programmatically adjusted to the needs at hand.

Unfortunately this form of customizable automation does not exist for model driven development beyond the use of e.g. macros in C and C++, introspection in dynamic languages like Java and python or code generation independent of the language.

While some static analysis tools have limited support for customization [161, 162], they still force the developer into a specific work-flow. A general pattern matcher for code [164] was developed as a research project [139, 140] but the project has been discontinued.

Instead of using a finished tool with customization capabilities, developers should be given the building blocks needed to quickly produce exactly the tool they need for their specific problem.

Thus the overhead and inadequacy of configurations is averted and the developers can focus on solving their problem instead of being held up for unpredictable amounts of time when their problem has to be fixed by the static analysis developer. This new scheme is compared to the status quo in Figure 4.2.

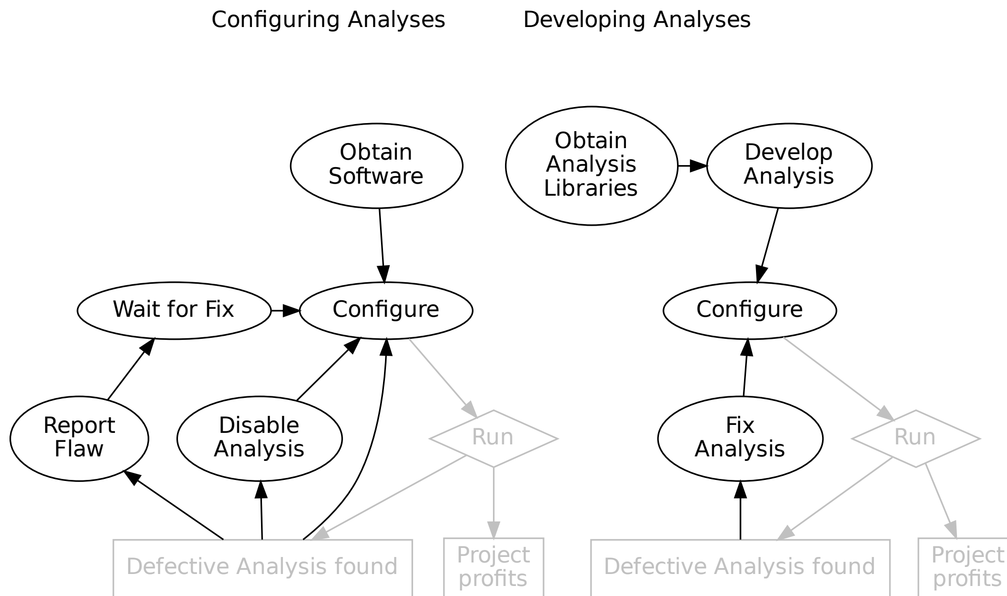


Figure 4.2: Comparison of the processes for developing and for configuring analyses. The parts that are the same in both schemes are grayed out. Arrows depict the possible state transitions.

4.3 Compiler extensions

Compiler APIs are frequently difficult to use and require expertise in the respective compiler's internal data structures [6]. Even the application of an already implemented compiler extension to the source code of a project can be a nontrivial undertaking. Most compiler projects treat the use of their API as something to be done for research or experimentation, not for creating a new product or tool. While this discourages the use of these APIs as proposed in this thesis, there are notable exceptions to this rule as discussed in Section 2.2.9.

4.4 Formal analysis

Formal analyses, while being able to prove certain invariants across entire projects, are hard to integrate into existing projects. Even if the project is designed from the ground on to work with formal analyses, developing in such a project is akin to writing in an entirely different language. The concepts from this thesis do not make a difference between existing and new projects, and can be introduced incrementally, both in the coverage and in the "extremity" of the analyses introduced.

4.5 Model driven development

Model driven development is classically performed by

- generating code that is compiled together with the handwritten code,
- generating code with placeholders where handwritten code is inserted, or
- manually transcribing the models into the code.

4.5.1 Code generation

Code generation generally suffers from disconnecting the generator input from the generated code. This is usually visible via compiler diagnostics which report errors in the generated code. The developers will not just have to understand their handwritten code, but also work with the generated code. Code generators usually do not generate readable code, making this process hard and frustrating. Furthermore, any change to the model will change the generated code in (sometimes unpredictable) ways, requiring large scale changes to the handwritten code.

Even if the code generator works by keeping handwritten code in mind, either

- by having sections of code that do not get overwritten during code generation, or
- by inserting handwritten code into the generated code,

developers still need to handle changes in the model that affect the handwritten code.

4.5.2 Manual transcription

Without code generation, developers are forced to manually convert models into program code. The required work is very systematic and mundane, while the chance for mistakes due to the repetitiveness is very high. This alone makes manual transcription not very feasible. Since models are supposed to also change and evolve, the effort for keeping the code in synchronization with the model is enormously high.

The only positive aspect of manual transcription is the fact that the developers have the most control over how to represent model components.

4.5.3 New concept

With the concepts from this thesis, it is possible to treat the model as an integral part of the code, without a separate code generation step. The compilation will instead take the model as input and merge it with the code within the compilation

process, allowing much richer diagnostics, which refer back to the model instead of into the generated code.

Since the model integration now happens via the compiler, changing the model is now different from changing any part of the source code. It also permits arbitrary levels mixing model and handwritten code. In fact, the developers do not have to decide on a specific mixing scheme, but can use different mixing schemes. In each mixing situation, the most fitting scheme can be used. If requirements change, the scheme can be changed later.

4.6 Static analyses

Writing static analyses is an exercise reserved for compiler developers. In order to learn how to write a static analysis, a developer needs to have intricate knowledge about the specific compiler they are using in addition to knowledge about compiler theory. This severely limits the scope of developers from which new static analysis authors can be recruited.

All concepts in this thesis have been designed with developers of any experience level in mind. Where complex situations are required, appropriate processes for obtaining skeletons or full analyses from examples have been established. Such processes flatten the learning curve, as developers start out with a static analysis, that at least catches their example, instead of having to start from scratch. Once the developers are confident with the current level of static analyses that they are writing, they can expand their knowledge in any direction in small steps.

Chapter 5

Implementation of the components of the new concept

One of the first steps when designing a prototype is to choose a programming language and compiler. Due to the projects through which the work on this thesis was financed, there are certain constraints on the language in order for it to be considered:

- It must not require garbage collection or other real-time incompatible features.
- It must be able to compile to native code for embedded devices.

Furthermore there are desirable features in languages that make it easier to implement the various concepts introduced in this thesis:

- The language must not require a textual preprocessor as this would break the mapping from source code to its semantical representation, making it hard or sometimes even impossible to emit high quality error messages to the user.
- The language should have error avoiding features in order for
  - the majority of static analyses to be of a high level nature instead of cleaning up defects of the programming language itself and
  - being able to make assumptions about invariants that are guaranteed to hold, which significantly helps in keeping static analyses simple and thus less prone to exhibit false positives.
- The compiler should have a reliable runtime system and be written in an error avoiding language.

The analyzed language and compilers are summarized in the following table:

	rustc	clang	javac	ghc	gnat	go
No Garbage Collector	x	x			x	
Native code	x	x			x	x

	rustc	clang	javac	ghc	gnat	go
Memory Safe	x		x	x	~	x
No Preprocessor	x		x	x	x	x

For this thesis the language Rust and a compiler for it that is itself written in Rust was chosen. The choice was made both on the various features mentioned above as well as the fact that it was in rapid development during the time of this thesis. This permitted a stronger interaction of the author with the development of the compiler than would have been possible with the established compilers and languages.

Initially intended as a proof of concept implementation, the author's contributions to the clippy static analysis project have been added as an extension to the official Rust compiler. While line-of-code-wise the major contributions are innovative static analyses not found in any commercial or open source static analysis tools, the work towards convenient application of the analyses and the automated generation of analysis code from examples has had the largest impact on the Rust community. Furthermore, while authoring or mentoring new static analyses, common mistakes were noticed and eliminated by writing new analyses with the sole purpose of detecting issues in the clippy project [115] or even the Rust compiler itself [112].

This chapter covers the implementation of the four main applications of the new concept. These applications are

- static analyses as part of the project's development process,
- integrating models directly as input to the compiler without an additional code generation and integration step,
- static analyses for checking whether code adheres to a model, and
- abstractly interpreting code to detect complex situations that is infeasible as a static analysis.

Overarching topics and topics that are not inherently necessary, but ease the application of the new concept are addressed in Chapter 6

5.1 Integrating static analyses into the development process

This section starts out with a general description of how a static analysis is developed and then discusses the various criteria that make up a static analysis tool with a good user experience.

5.1.1 Developing a new static analysis

Writing a new lint for Rust requires multiple mechanical steps that do not change between different static analyses:

1. Choose a unique name for the lint following the Rust naming guidelines for lints.
2. Create the boilerplate[1]
   - Write Documentation:
     - A short description of the lint.
     - An example of code that it triggers on.
     - Mention possible false positives.
   - Create the lint pass object type:
     - Implement `LintPass` trait, and
     - implement `LateLintPass` trait.
3. Implement the `LateLintPass` trait methods relevant to the lint (e.g. `check_expr` if the SA[2] should be run on expressions).
4. Destructure (see Appendix C.4) the Rust syntax component of interest (e.g. checking whether the current expression is a `+` operation and extracting the operands).
5. Report the lint through the compiler diagnostic system and optionally add automatically applicable replacements.

5.1.2 Choosing a responsiveness level

Historically static analyses have been implemented as independent tools that either need their own setup to find the source files or can be used as drop-in replacements for the compiler in the surrounding build system. Both methods come in three levels of response times:

1. Run during regular code cleanups (once a month or less)
2. Integrate into continuous integration (sanity test, always do this)
3. Integrate into compilation process (slow response time)
4. Integrate into IDE to react while typing (fastest response time)

While the IDE integration has the fastest response time and is less frustrating for developers, it can be difficult to provide a real-time experience depending on the complexity of the analyses. An algorithmically hard to analyze problem will require nontrivial amounts of time to run the analysis. If the slow analysis blocks the user from receiving fast analyses' diagnostics, this will degrade the experience. It seems prudent to not arbitrarily choose one of the response time options, but instead place analyses into categories of importance and speed [34, 41].

Some less important but slow analyses would frustrate users due to their low priority but slow response time and possible nontrivial resolution. These analyses are a better fit for code cleanup runs.

[1] supporting code that does not contribute to the logic, but is still required
[2] Static Analysis

Very fast but low importance analyses are accepted by users if they are trivial to address, e.g. by IDE support to automatically resolve the issue with a key press or click.

5.1.3 Forward compatibility of compiler extensions

Updating the compiler used in a project often requires some adjustments to the project. Even though e.g. the Rust compiler promises that updating it will not cause compilation to break or change the behavior of the final program. New compiler warnings may be emitted, which need to be silenced before the new compiler is rolled out throughout the project. Security bug fixes might break compilation, even if they currently did not cause the project to be endangered. These are rolled out as a warning first, so a previous compiler update should have already raised a warning.

Compiler extensions like static analyses and some code generation use cases do not yet have first class support and are thus allowed to be broken between compiler updates. In most cases this comes down to trivial type or function renamings. This adds an additional cost to compiler updates, which needs to be calculated into the creation cost of writing the analyses in the first place.

Since the compiler version being used is tracked together with projects, the compiler update can be rolled out together with fixes for the mentioned breakages. This prevents other developers from being inconvenienced by the compiler update. In the worst case, all compiler extensions must be updated when the compiler is updated.

5.1.4 Diagnostics

It is no surprise that the easier error messages can be comprehended, the faster the underlying issue is resolved by the user [13]. The "ignore all warnings" philosophy commonly associated with warnings from many compilers due to their compiler-developer-centric view is completely inverted in many modern compilers. Due to the high quality of diagnostics, rare bad diagnostics in edge cases are even accepted as indicators of the edge case and thus an potential error source. These warnings are either addressed by locally silencing the warning or by changing the code to something more comprehensible.

In order to replicate not just the quality of the diagnostics but also the user experience in general, diagnostics for compiler extensions should be using the same diagnostic API that the compiler itself is using. This allows any tooling which consumes the diagnostic output to work with the compiler extension without requiring any adjustments (see Figure 5.1).

The author refactored the Rust compiler's JSON diagnostic output to support multiple machine-applicable suggestions per diagnostic [116]. This feature saw immediate use in clippy and has recently been adapted in rustc itself. Another related feature implemented by the author is allowing JSON-consuming tools to additionally show

the diagnostic as it would show up on the command line [117]. This allowed IDEs to render the human readable diagnostic directly as a tooltip and simplified various tools that were processing compiler output.

Reusing the compiler diagnostic API has advantages beyond the consistent tooling consumption. Developers learn to skim diagnostic messages to get at the parts that interest them right now. Replicating the same layout for messages allows the developers to immediately be effective even with unknown diagnostic messages. Thus, a new tool that emits diagnostics will not require any diagnostic-consuming tooling or humans reading the diagnostics to adjust anything.

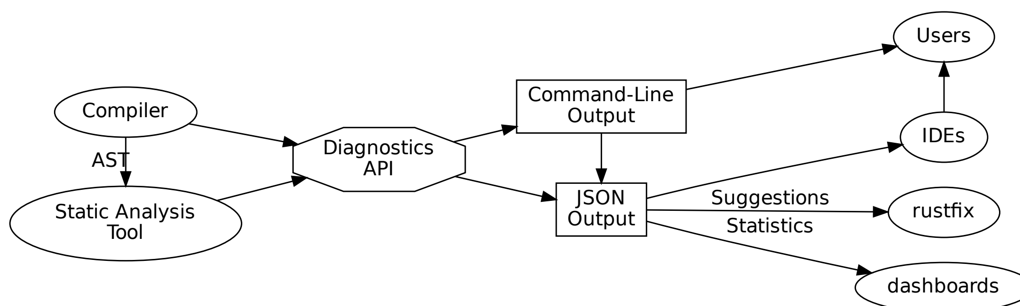


Figure 5.1: Dataflow diagram from compiler to user across the various diagnostic API participants.

5.1.5 Problem resolution instructions

Some diagnostics occur frequently but are easily resolved. While this keeps the code tidy, mechanical tasks are tiring to developers. The advantage of mechanical tasks is that they can often be automated or at least semi-automated. Some compilers expose a system for attaching suggestions to diagnostics. These suggestions range from general freeform text to fully structured automatically applicable code replacements.

As a very common example consider the situation where the programmer is using a function which has not been imported into scope:

```
x = sin(42);
```

Many languages place the `sin` function in some sort of `math` module, file, class, or other grouping concept of the language. It is often not available by default, but requires the user to import, include, or load the `sin` function. Instead of just telling the user some generic diagnostic message like

```
error: No function names `sin` in scope
```

an additional suggestion like

```
suggestion: The `math` module exports a function called `sin`
```

While on a first glance nothing is automatically applicable here, this is due to the command line output nature of the diagnostic emitter. Compilers may additionally support exporting a machine processable version (e.g. in JSON or XML format) of the diagnostics, which can then be consumed by IDEs and other tooling. The machine processable format contains a note for consumers mentioning the location where to insert code that will fix the problem. This can often be observed in command line output like

```
suggestion:
Import the `sin` function from the `math` module as shown:
`import math.sin`
```

The suggestion shows line the `import math.sin` snippet. The structured format can additionally suggest to insert that snippet at a specific line in the code. If other `import` statements existed already, the compiler could suggest to place the snippet next to the existing use statements. Lacking any existing `import` statements, the compiler suggests placing the new `import` statement at the top of the module that requires the item. In this case the module is the entire file, so the new `import` statement is placed at the top of the file.

The author converted the previously unstructured free-text suggestions in the Rust compiler into structured suggestions [116, 118] as described above.

5.2 Model driven development

For demonstration purposes, a simple modeling language is invented. It will be a modeling language supporting solely state machines (specifically Mealy machines). All other model concepts of UML are ignored in order to keep the demonstration simple. In practice, instead of using a custom syntax and restricted model, one would be taking the XMI representation [98] of the UML model as input instead of manually transcribing to a simple model language.

5.2.1 Code generation

This section explains the various ways that code generation extensions can be created in Rust, and why the last way is the one that will be used in this thesis.

1. a build script that generates code [132]
2. a `macro_rules` macro [72]

3. a syntax extension plugin to the compiler
4. a procedural macro [73]

Build scripts permit the classical code generation scheme employed by modelling tools, but automate the process by regenerating the generated code whenever the model changes on disk. The compiler can then integrate the generated code just like it would integrate a user-written file.

`macro_rules` only allow a small subset of code generation, but are trivial to use and reuse. They are a DSL[3] for converting input (potentially containing Rust expressions) to Rust code.

Syntax extension plugins are unstable and are being phased out in favor of procedural macros and are therefore not further discussed here.

Procedural macros are equal to `macro_rules` in ease of use and reuse, but lack the restrictions on what kind of generation can be done. A procedural macro can be seen as a function that takes the macro invocation input as an argument and returns valid Rust code as the output. The function body is written in Rust (in contrast to the DSL used for `macro_rules`), and have only a single restriction: they may not store their argument in a global static variable in one invocation and access it again in a future invocation. Doing so will not cause undefined behavior, but it will simply emit an error and abort compilation. This restriction applies because otherwise rustc would have to either never deallocate previous invocations' arguments (thus causing not insignificant increases in memory usage) or require a more complex API for little gain. Other than this single restriction, procedural macros may do anything regular Rust code may do like

- accessing files,
- accessing the network,
- using arbitrary crates (Rust libraries) to compute the output,
- using random numbers, and
- many others.

While many of these concepts are usually nonsensical in macros, as reproducibility is very desirable in any software development process, they are not inherently problematic from a compiler-perspective.

An example model is shown in Figure 5.2 which can be represented in the exemplary model language as

```
Phone,
begin_incoming_call: HangedUp -> Ringing,
accept_call: Ringing -> Speaking,
end_call: Speaking -> HangedUp,
```

[3]domain specific language

```
begin_outgoing_call: HangedUp -> Dialing,
finished_dialing: Dialing -> Waiting,
call_accepted: Waiting -> Speaking,
```

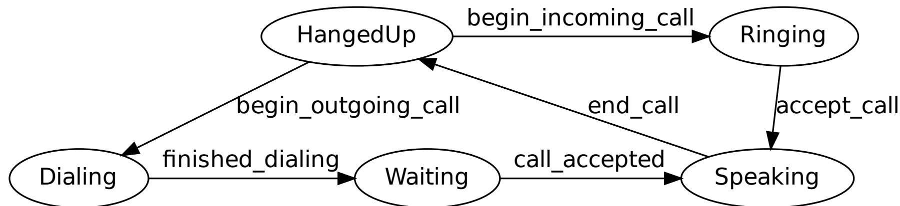


Figure 5.2: A small UML state machine for handling a phone

The first line contains the model name. All further lines are transitions. The name before the colon is the transition trigger, and the names on the right are the initial state required for the transition and the new state after the transition. Guards and actions from UML state machines have been left out in order to keep the model simple.

The goal of the code generator is to convert the above state machine to the following Rust code, although the code will never actually appear in this format, and instead will be generated in-memory in a non-textual format.

```rust
enum Phone {
    HangedUp,
    Ringing,
    Speaking,
    Dialing,
    Waiting,
}
impl Phone {
    fn begin_incoming_call(&mut self) {
        match *self {
            Phone::HangedUp => *self = Phone::Ringing,
            _ => panic!("`begin_incoming_call` can only transition from HangedUp"),
        }
    }
    //... all other transitions implemented similarly
}
```

In order to define the procedural macro, one needs to define a function and add a `#[proc_macro]` attribute to the function:

```
#[proc_macro]
pub fn state_machine(input: TokenStream) -> TokenStream {
    // actual code goes here
}
```

For simplicity, this example uses no helper libraries for the parsing of the arguments. Instead the input is converted to a `String`

```
let input = input.to_string();
```

The `String` is then converted into an iterator over the comma separated entries

```
let mut lines = input.split(',');
```

The first line is the model's name, so the first element of the iterator is consumed via the `next` method. The `lines` variable still exists afterwards, but further calls to `next` will only yield the rest of the lines and not the first one again. In this simple example error reporting is not addressed and thus the first line can be `unwrap`ped. `unwrap` will report a default diagnostic message in case the input did not contain even a first line. As a further cleanup the first line is `trim`med, meaning whitespace on the left and right of the line is removed, yielding just the content without any padding.

```
let type_name = Ident::new(lines.next().unwrap().trim());
```

After the model name is extracted, only transitions are left to parse. Each transition starts with its name (like `begin_incoming_call`) which will be stored in

```
let mut names = Vec::new();
```

The source and target states of each transition are stored in

```
let mut sources = Vec::new();
let mut targets = Vec::new();
```

respectively, while a set of all states is built from the states mentioned in the transitions:

```
let mut states = HashSet::new();
```

In order to consume the rest of the lines of the `lines` iterator, it can be consumed entirely via a `for` loop:

```
for line in lines {
```

While the actual processing could be done by using a real parser, that would incur a lot of code overhead, obfuscating the interesting parts of the example at hand. Instead, the a handwritten string processor is written. The string processor of each line can be found in Appendix F

Once the model has been parsed, the `quote` library is used to generate Rust code via a template mechanism that allows using variables from the scope in the template by using the `#` symbol. Sequences of elements are flattened by enclosing their uses in `#(` and `)*` which cause the enclosed code to be repeated for every element of the sequence.

The `quote` code generator is invoked via

```
quote! {
```

As a first step, the `enum` of all the states is generated by using the model name as the `enum` name. The list of states uses the repetition feature of the `quote` library and generates a comma separated list of the states. Note the `,*` repeat declaration which just generates commas between states, but not after the last one.

```
enum #type_name {
    #(#states),*
}
```

Now, for each state transition, a function is generated that modifies a value of the `enum`. In Rust, functions can be attached to types (so they can be called via `value.function()` syntax). This is done by wrapping the functions in an `impl TypeName` block, where `TypeName` is the name of the type to which the functions are attached.

```
impl #type_name {
```

Each function can then take either a `&mut self` or a `&self` argument as the first argument in order to have access to the value either via mutable reference or constant reference. Since the transition modifies the state, `&mut self` is required.

The following function header will be repeated for each entry in the `names` vector.

```
#(
    fn #names(&mut self) {
```

Inside the function, a single pattern match is generated, which checks whether the current state is in fact the source state for the given transition.

```
match self {
    #sources =>
```

If the arm matches, the current state can be reassigned to the target state.

```
*self = #targets,
```

For any other source state, an error is reported, because the transition is only legal in the given state

```
    _ => panic!("invalid transition"),
```

While it is also possible to create the macro output without using any libraries, the above template is both shorter and more readable than a dependency-free version. The full example can be found in Appendix F and archived online github.com/oli-obk/model-demo.

5.2.2 Code checking

The same model can also be checked against user-written code. To save the parsing phase in this demonstration, the model is represented directly in the code. In a real environment, this model would be loaded from XMI.

```
let transitions = vec![
    ("begin_incoming_call", ("HangedUp", "Ringing")),
    ("accept_call", ("Ringing", "Speaking")),
    ("end_call", ("Speaking", "HangedUp")),
    ("begin_outgoing_call", ("HangedUp", "Dialing")),
    ("finished_dialing", ("Dialing", "Waiting")),
    ("call_accepted", ("Waiting", "Speaking")),
];
let transitions = transitions.into_iter().collect();
// register the code checker
ls.register_late_pass(
    None,
    false,
    box StateMachine { start: "HangedUp", transitions },
);
```

The above is the only code that changes between different models. Everything from here on is the generic matching code that would also have to be written when matching a UML model from XMI against the code.

The analysis looks at every function in the user written source code by obtaining its MIR

```
let mir = cx.tcx.optimized_mir(def_id);
let mut states: IndexVec<mir::BasicBlock, HashSet<&'static str>> =
    IndexVec::from_elem(HashSet::new(), mir.basic_blocks());
// seed the function with an initial state
states[mir::START_BLOCK].insert(self.start);
```

Now, the states that every node in the MIR's graph can observe are filled in.

```
for (bb, bbdata) in mir.basic_blocks().iter_enumerated() {
    match &bbdata.terminator().kind {
        mir::TerminatorKind::Call {
            func,
            destination: Some((_, succ)),
            ..
        } => { /* function calls transition the state */ },
        _ => { /* everything else just forwards the states */ },
    }
}
```

When encountering a function call, the called function's name is matched against the state machine transitions' names.

```
if states[bb].contains(start) {
```

If it's a match, the current block's list of states doesn't contain a source state for the transition, the transition is not valid at this site and an error must be reported about it. If the state transition can be performed, the new state is inserted into the set of states that the next block can contain.

```
// update the next block's states
states[*succ].insert(end);
```

In case the source state is not a legal source state for the transition, the user is informed about this via the diagnostic infrastructure. Custom diagnostics, also called "lints" have a generic name for the analysis (in this case `STATE_MACHINE`), a location in the code that is reported to the user as the location of the mistake, and a message that explains to the user what went wrong.

```
cx.span_lint(
    STATE_MACHINE,
    bbdata.terminator().source_info.span,
    &format!(
        "state transition `{}` not applicable for states {:?}",
        transition,
```

```
        states[bb],
    ),
);
```

Instead of just causing a runtime panic like in the code generator version, this version can detect simple cases at compile-time already and emit custom diagnostic messages that point the developer to the cause of the problem. Consider the following small user-written program which violates the rule that `end_call` can only be called while in the `Speaking` state.

```rust
// Create a variable of `Phone` type in `HangedUp` state
let mut x = Phone::new();
// Transition from `HangedUp` to `Ringing`
x.begin_incoming_call();
// Transition from `Speaking` to `HangedUp`
x.end_call();
```

`begin_incoming_call` transitions the state machine into the `Ringing` state and the correct next transition would be `accept_call`. Instead, the analysis emits

```
error: state transition `end_call` not applicable for states {"Ringing"}
  --> $DIR/state_machine_fail.rs:29:5
   |
29 |     x.end_call();
   |     ^^^^^^^^^^^^
   |
   = note: #[forbid(state_machine)] on by default
```

While the diagnostic message illustrates the custom reporting scheme, there certainly is room for improving the message both grammatically and in terms of being helpful for finding the solution to the problem.

5.3 Abstract interpretation

Testing shows the presence, not the absence of bugs - Dijkstra [19]

This section analyzes a novel minimal virtual machine implemented as an extension to the Rust compiler. Its various uses, effects, limitations and enabled technologies are discussed in the light of previous attempts and hypothetical and in-development future uses.

An interpreter for Rust code (MIR *i*nterpreter `miri` [99]) was extended to support most deterministic programs. The interpreter was then split into a generic subset (called `miri-engine` henceforth) and the full `miri` interpreter. Additionally a second

interpreter (the `const` evaluator) was developed that only supports code that is legal in constants in the Rust language. The author merged this new `const` evaluator and the `miri-engine` into rustc [119], allowing `miri` to be just a compiler extension hooking into the `miri-engine`. Gradually more of the advanced `miri` features have been merged into the `miri-engine`, allowing the const evaluator to also take advantage of these features.

Figure 5.3 shows that `miri` reuses the compiler's internal API to build a standalone tool. The compiler internally uses the same API to realize its `const` evaluator. Further extensions like the graphical debugger `priroda` [120] build on top of `miri`, using both `miri` as a library as well as communicating with rustc's API.

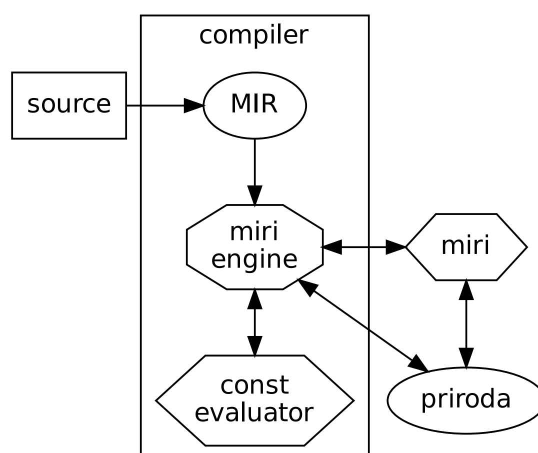


Figure 5.3: Information flow between compiler extensions and components. The arrows show information flows. In some cases there is an exchange of information, e.g. the const evaluator has to provide the `miri-engine` with information about what code to interpret.

5.3.1 Running abstract interpretation on test suites

Abstract interpretation can run essentially all unit tests and significant parts of integration tests. Unit tests are generally deterministic - even in the presence of random number generators, since a random number generator seeded with a constant value is deterministic - and thus sufficiently advanced abstract interpretation can interpret the tests, producing the identical results that actual execution would yield.

The difference between running the tests on hardware and via abstract interpretation is that it is possible to add arbitrary checks to the virtual instructions instead of just executing the instructions. Various operations that regularly have to be checked by the developer can now be checked automatically, even if they are uncheckable on hardware. As an example: it is not generally possible to detect at runtime whether

a pointer dereference is actually legal by the rules of the programming language in which it was authored. During abstract interpretation, the virtual machine can keep track of legal addresses. The term "legal" is used here to define what the language considers legal, not just stating that the memory behind the address actually exists. Interpretation can then be aborted if a dereference operation happens on an illegal address.

The ability of the abstract interpreter to collect addition data during the interpretation allows it to report more than just the fact that an illegal operation has happened. The interpreter can additionally point to sites that previously interacted with the same value or memory, which often helps to quickly narrow down the root cause instead of just exposing a symptom of the problem.

5.3.2 Restraining the virtual machine

In contrast to executing tests on regular hardware, the virtual machine can be restricted on purpose in order to simplify the detection of undesirable actions. As an example of such a restriction, this section will cover pointer handling in virtual machines. Comparing two pointers for equality is, by itself, a seemingly innocent operation. Consider the following program:

```
// Declare two immutable variables
let x = 42;
let y = 42;
// Obtain the addresses of the memory of the variables
let x_ptr: *const u32 = &x;
let y_ptr: *const u32 = &y;
// Compare the addresses
assert!(x_ptr != y_ptr);
```

A programmer might have written such a piece of code in order to obtain two guaranteed unique pointers to be passed to some unknown code and being able to identify the pointers later again. Since the values of `x` and `y` are both equal and known to never change, the compiler can unify them into one variable. This would lead to `x_ptr` and `y_ptr` being equal, thus aborting the above code when the assertion fails.

This optimization is neither guaranteed nor deterministic, but depends entirely on the rules of the optimizer. Thus it might be entirely reasonable for a software to work during testing, but not when deployed to the customer. While the example is very contrived for demonstration purposes, pointer comparisons are problematic in general due to memory safety concerns. Nonetheless, even this simple example can be defended against by forbidding comparing pointers into anything but heap allocations, static items and mutable local variables. The interpreter virtual machine can raise a warning when such potentially deduplicated pointers are compared.

Never deduplicated memories include heap allocations, mutable variables and global statics. Heap allocations are guaranteed unique, because their initial value and future modifications are unknown and thus cannot be optimized away. Static items are defined as being unique, thus guaranteeing differing addresses. Mutable local variables can also not be deduplicated, because that would cause modifications to a variable to also change a distinct variable.

An abstract machine can treat pointers as more than just addresses (which they are [61]), and handle the above situations in special ways. When the interpreter encounters an operation that attempts to compare addresses, it does not solely compare the identity of the address, but also checks the kind of memory being pointed at.

In contrast to immutable variables, constants do not actually have memory addresses. Instead, they are instantiated at each use site. If instantiated during the evaluation of another constant or in the type system (e.g. for array lengths), they are solely used for constant evaluation. If instantiated in a runtime operation (e.g. regular code inside functions), it is possible that an unnamed temporary variable is created. This variable will then have its own address. Multiple uses of a constant can arbitrarily reuse that variable or create new variables, depending on language rules and optimizations. Thus comparing the addresses of two constants does not yield any meaningful value and needs to be assumed unequal, even for trivial statements like `&FOO == &FOO`. While that is a legal interpretation of comparison, it can be very surprising for users [165].

Dangling pointers can be equal to other dangling pointers or even pointers to live allocations, because memory is reused after being deallocated. The following example demonstrates such a case:

```
let x_ptr: *const i32;
{
    let x = 42;
    x_ptr = &x;
}
let y_ptr: *const i32;
{
    let y = 42;
    y_ptr = &y;
}
// x_ptr == y_ptr depending on optimizations
```

The `x` variable is dead after its surrounding block is closed. The `y` variable is thus allowed to occupy the same memory that `x` used to occupy. Comparing the addresses of `x` and `y` can yield `true` or `false` depending on optimization parameters, but must be assumed to be `false`.

Immutable local variables can be deduplicated by the compiler if their values are equal. There is no reason to keep around two variables, if they are indistinguishable except by their address. Consider

```
let x = foo();
let y = bar();
if x == y {
    // from now on, the compiler can reuse `y`'s memory e.g. for `z`
    let z = 99;
    // `y` thus just becomes an alias for `x`.
} else {
    // x is distinct from `y`
}
```

Even if x and y are often different, since they are immutable after initialization, the moment it is known that they are identical, but their identity serves no purpose, they may get deduplicated. Primitive integers fulfill this role: they can be arbitrarily duplicated by users simply by assigning them, thus they do not have identity. Comparing integers for equality just compares the bits of the integers, instead of requiring more complex logic like in the case of floating point numbers.

Since functions may be duplicated across compilation units (files, modules, libraries), pointers to functions cannot be compared, as they might point to different binary code copies of the same function. At the same time, completely different functions might get optimized to the same machine code and subsequently deduplicated.

Essentially one needs to consider every address to be only comparable to addresses to entirely different types of memory or to types of memory with identity (e.g. static items). The full table of comparisons can be found in Appendix E.

5.3.3 Employing abstract interpretation for debugging purposes

When comparing the debugging experience of JIT[4] or VM[5] based languages (like Java, python, matlab, etc.) with the debugging experience of compiled languages (like C++, Rust, Ada, etc.) one quickly realizes that the stability of debugging and the extend of introspection capabilities is significantly lower in the compiled languages.

While there certainly are many avenues to improving the debugging information associated with binary files, the controlled and dynamic nature VM based languages is impossible to reach on real hardware. Hardware is the real culprit in this situation: Most compiled languages specify an idealized abstract machine against which all behaviors and features are explained. There have been previous attempts on implementing such machines in the form of an interpreter [27, 47, 50, 78, 134].

[4] Just In Time compilation
[5] Virtual Machine

These projects are all meant to be run solely headless and emit a textual report on any found issues. While some even permit adding custom constraints beyond those of the C/C++ languages, they are inherently limited to detecting undefined behavior. A new tool was implemented in the process of this thesis. The `priroda` [120] graphical debugger is based on `miri`, which is an officially supported and maintained component of the Rust compiler. Even though the abstract machine model of the Rust language is still under active development, `miri` already implements the most likely outcome of the model. Further changes to the model can be easily integrated. `priroda` allows the user to not only have all the undefined behavior analysis of `miri`, it additionally permits stopping evaluation at arbitrary points (e.g. when an assertion fails). When the interpretation is stopped, the stack and heap can be inspected graphically, without being restricted to type based memory analysis.

5.3.4 Summary

A virtual machine for MIR was developed. In the process of this thesis, development was continued to the point where most Rust code can be interpreted in any target's memory model without ever producing machine code. At that point, the author split out the core of the virtual machine and merged it into the Rust compiler, where it now powers the constant evaluator [119].

Chapter 6

Aspects of the practical application of the new concept

In this chapter the usability of existing static analysis tools is evaluated. Then the new compiler extension methodology (see Section 2.2.9) is applied to static analyses and the effects are analyzed.

Several factors play into the usability. The effort required to run a tool on an existing code base for the first time can quickly demotivate potential users. In order to be effective in the long term, continuous integration and IDEs need to be able to apply the tool without user action or intervention. A large number of false positives can lead to users ignoring all diagnostics due to the effort required for distinguishing false positives from useful diagnostics [9, 81, 133]. Finally the diagnostics themselves need to be comprehensible from a user perspective instead of a compiler or language author perspective.

6.1 Role of false positives on the usability of static analyses

False positives are situations where a static analysis believes a piece of code to contain a defect, when this is not true in fact. The following table depicts the full spectrum of false/true positives/negatives:

<table>
<tr><th></th><th>false</th><th>true</th></tr>
<tr><td>positive</td><td>trigger on ok code</td><td>trigger on problematic code</td></tr>
<tr><td>negative</td><td>do not trigger on problematic c</td><td>ode | do not trigger on ok code</td></tr>
</table>

Static analyses are usually broader than they are required to be. Additionally to catching problematic code, they may trigger on perfectly valid code that simply has a similar structure to the problematic code. Implementing the static analysis in a way that ensures it only triggers on problematic code often leads to some problematic

code not being detected. If the rate of highlighting valid code is low enough, it is often considered safer to mark some code as "triggering the analysis but ok" than to have some problematic code fall through the cracks [21].

> In short: ideal are only true positives and true negatives without any false positives or false negatives, but false positives are acceptable if they prevent false negatives.

An artificial example is visualized in Figure 6.1. The read and blue curves denote a different static analysis. Each dot on the curve is a different configuration of the static analysis. Take the example of a static analysis that informs the user about functions that are too large and must be split up into multiple functions. While more reasonable measures than the lines of code of a function exist, we'll use a threshold of the lines of code to detect functions that are too large. The choice of threshold affects the amount of functions that the static analysis triggers on. A low threshold will likely cause acceptable functions to be marked (false positive), while a large threshold may miss functions that should be split (false negative). A threshold above a certain value will not have any false positives, as e.g. any function with more than 1000 lines of code can be considered "too large" by any measure. Decreasing the threshold will at some point mark functions as too large, even though a human reading it would have still considered it acceptable. This is where the false positive rate goes above zero in the red curve. Ideally a static analysis' ROC curve is in the top left corner with zero false positives and 100% true positives. In reality false positives can often be prevented entirely, but unfortunately only up to a certain point. Increasing the number of true positives can at some point cause false positives to occur.

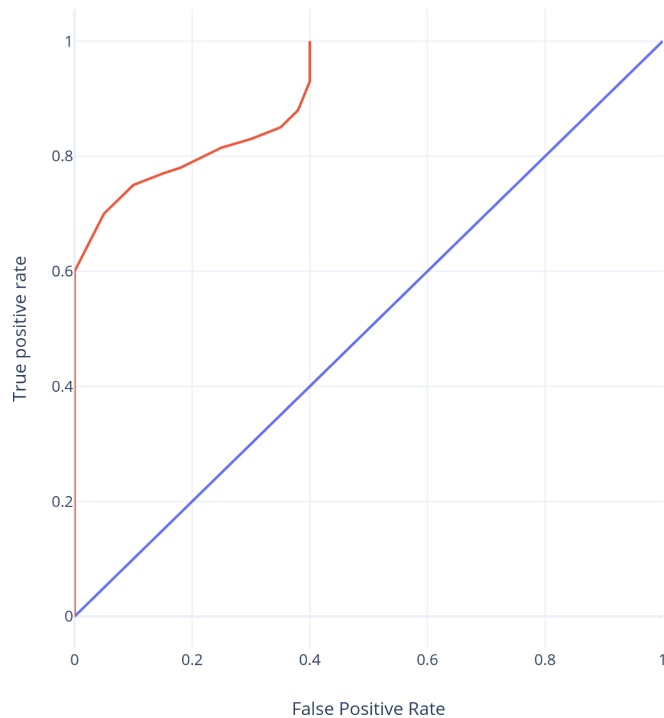


Figure 6.1: A graphical representation of the ratio of false positives to true positives. The blue curve is a hypothetical situation where for every true positive a false positive is emitted.

Additionally, while it may be possible to design the static analysis in a false positive and false negative free way, the effort in writing the analysis or in running the analysis may not be justifiable depending on how useful the freedom of false positives is. It may be significantly less expensive to address the false positives manually than to design a false positive free static analysis.

This line of reasoning is taken from physical safety rules, where one would rather replace or triple inspect a perfectly fine component that can't be proven safe by a simple inspection than take it along unchecked and end up with a crashing airplane or similar catastrophic failure.

Other static analyses might make perfect sense in one environment, but are of limited use in others. This situation is often encountered when trying to balance often-at-odds design choices like generalizability vs low complexity or performance vs readability.

Static analyses, being subject to bugs like any other software, can also produce false positives by simply triggering on unrelated code. These bugs are different from overly broad false positives, because they can be fixed by adjusting the analysis' implementation, without introducing false negatives. One example is a static analysis detecting code that adds `0`. While it is indeed useless to add `0`, the code may not directly contain `0`, but e.g. be `x + SOME_CONSTANT`. If `SOME_CONSTANT` happens to be zero in the current compilation, that does not mean that the `x + SOME_CONSTANT` can be reduced to just `x`, as `SOME_CONSTANT` may depend on platform information

like the number of bits that a pointer has or even on build-time configuration values [26].

6.1.1 Stability across compiler changes

As discussed in Section 2.2.9, compiler APIs come in wide ranges of stability guarantees. From

- inherently unstable and breaking analyses weekly on average (e.g. Clang plugins [156]),
- to fully stable across compiler versions (e.g. Rust procedural macros [73]), and
- even across compilers from different vendors (e.g. ASIS for Ada [108])

every kind of guarantee exists. Of course this is a trade-off between enabling the redesign of the API for improving usability, performance or correctness and on the other hand committing to a specific design indefinitely.

As a middle ground, semantically versioned APIs can be introduced. These do commit to a specific major version being accessible indefinitely, but do not guarantee that old major versions will obtain any new features. The current major version can then be extended with features, while the old major version APIs remain unchanged.

The compiler backend needs to support all APIs at the same time. Often a backend change that requires adjusting one API will require changing one or many other APIs, too. This effort will only grow with an increasing number of APIs that need to be supported. The additional effort decreases the motivation to improve the API in ways that requires a new major version, even if the change would impact the usability of the API positively. Figure 6.2 shows that each API directly depends on the compiler backend and thus needs to be adjusted to any changes within it.

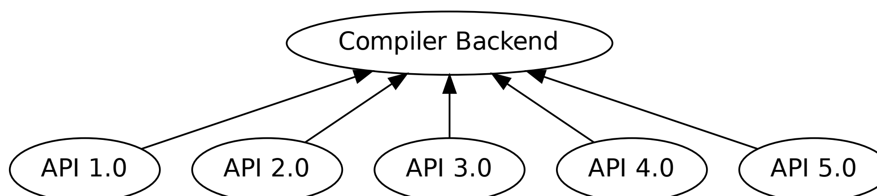


Figure 6.2: Dependency graph of the non-scalable API evolution. The arrows go from dependents their dependency.

A major downside of semantically versioned APIs is that they accumulate old APIs that need to be maintained. Since any API with a higher version number should support all use cases of a lower version API, the old API can be rewritten in terms

of the newer API. This is a one-time effort during the introduction of a new API instead of a long term maintenance commitment. In other words, if each new API implements the old API, the backend only needs to support the newest API, and all backend changes only affect the most recent API (see Figure 6.3). Inefficiencies introduced due to the additional layering can be used as motivators for consumers to move to newer APIs.

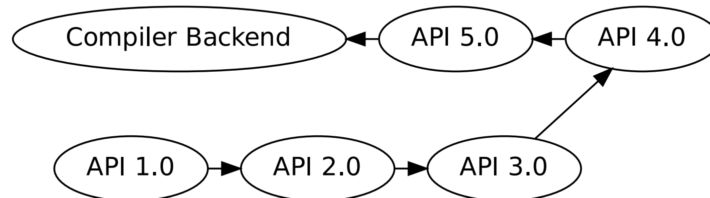


Figure 6.3: Dependency graph for an API evolution where each old API depends only on its successor. There is only a single dependency on the compiler backend. The arrows go from dependents their dependency.

This system even supports the incremental introduction of a new API by simply duplicating the current API and forwarding all calls of the old API to the new one. Then the new API can be evolved with breaking changes that only need to be addressed in the old API's backend. As an example consider a C++ API that allows adding

```
void add_name(
    /// Mutable access by reference to the API
    Api_1& api,
    string name,
) {
    api.name_set.insert(name);
}
```

or removing "names":

```
void remove_name(
    Api_1& api,
    string name,
) {
    api.name_set.remove(name);
}
```

Further features of such an API may be the ability to query if a name exists.

```
bool has_name(
    // Const reference access to the API.
    // Guaranteed that `has_name does  not change
    // the background state of the API
    const Api_1& api,
    const string& name,
) {
    return api.name_set.contains(name)
}
```

The developers might decide to expose the internal storage directly in order to allow users more convenience with respect to the manipulation of the name list. The same convenience could be reached with more functions exposing specific features of the internal storage, but would essentially require duplicating the entire API of the internal storage. Thus exposing the internal storage decreases the maintenance burden for the API while simultaneously improving the API usability due to users being able to access standard container features they already have experience with.

```
HashSet<string>& names_mut(Api_1& api) {
    return api.name_set;
}
const HashSet<string>& names(const Api_1& api) {
    return api.name_set;
}
```

In a future API it might be desirable to eliminate the `add_name`, `remove_name` and `has_name` functions in order to simplify the visible API surface. Removing those functions cannot be done in the current API because users might still be using the functions, even if a better alternative is available. As a first step of creating the API version 2, the old API is replicated exactly.

```
void add_name(const Api_2& api, string name) {
    api.name_set.insert(name);
}
void remove_name(const Api_2& api, string name) {
    api.name_set.remove(name);
}
bool has_name(const Api_2& api, const string& name) {
    api.name_set.contains(name)
}
HashSet<string>& names_mut(Api_2& api) {
    api.name_set
}
const HashSet<string>& names(const Api_2 api) {
```

```
    api.name_set
}
```

The second step is to change all functions in the old API to refer to the new API.

```
void add_name(const Api_1& api, string name) {
    add_name(api.api2, name)
}
void remove_name(const Api_1& api, string name) {
    remove_name(api.api_2, name)
}
bool has_name(const Api_1& api, const string& name) {
    return has_name(api.api_2, name);
}
HashSet<string>& names_mut(Api_1& api) {
    return names_mut(api.api_2);
}
const HashSet<string>& names(const Api_1& api) {
    return names(api.api_2);
}
```

In the third step the old API is adjusted to not use the undesirable functions.

```
void add_name(const Api_1& api, string name) {
    names_mut(api.api2).insert(name)
}
void remove_name(const Api_1& api, string name) {
    names_mut(api.api_2).insert(name)
}
bool has_name(const Api_1& api, const string& name) -> bool {
    return names(api.api_2).contains(name);
}
HashSet<string>& names_mut(Api_1& api) {
    return names_mut(api.api_2);
}
const HashSet<string>& names(const Api_1& api) {
    return names(api.api_2);
}
```

In the fourth and final step, the undesirable and now unused functions are removed from the new API.

For changes that do not solely extend the API surface, but modify it extensively, the process is applied analogous. The fourth step is then the actual modification of the new API, instead of just a cleanup step. For traceability it is advisable to split the fourth step into as many small substeps as possible.

6.2 Compiler extension schemes

In order to include custom static analyses in the development process of a project, compilers offer various schemes for integration. A commonly chosen scheme (due to its simplicity) is to replace all invocations of the compiler during the build process with an invocation of the static analysis tool. The static analysis tool will then run its analysis in addition to invoking the rest of the compilation process or even the compiler itself.

In Rust, a custom compiler can be written via the driver API. A driver contains the static analysis itself and reuses the compiler by invoking it via a library API. A step by step introduction on creating a custom driver with a custom static analysis can be found Appendix G.

The resulting binary is equivalent to rustc in its command line interface, but additionally runs the new static analysis when invoked.

6.3 Generating static analyses from examples

A major concern when delegating the design and implementation of static analyses (SA) to project developers instead of specialists is the effort and quality of the produced SA. Implementing a new SA from scratch is very repetitious and thus error prone. In this thesis a tool was designed which allows the developers to annotate broken code and automatically generate a template SA that will detect the pattern. Some handwritten code is required, especially for complex SA, but the tool reduces the amount of time taken for writing the SA significantly by reducing the number of build cycles and documentation accesses needed.

6.3.1 The problem

It is not generally hard to write a new analysis, but doing so from scratch without tooling support can be tedious. Creating the boilerplate, implementing the appropriate `LateLintPass` method and destructuring the compiler data structures (steps 2 through 4 from Section 5.1.1) are the most mechanical and will make up the largest amount of code for smaller analyses. Although the boilerplate can be copied from a template, it is easy to accidentally forget to edit parts of the generic template. When implementing the `LateLintPass` trait methods, a common mistake is to forget appropriate `macro` checks, which significantly reduce false positives or reports that the user cannot act upon, because the error is occurring outside their code. Also, destructuring a Rust syntax component requires either enormous experience in writing SA for Rust, or studying the compiler internals documentation. Since most destructuring is done in multiple steps, going back and forth between accessing the documentation and writing the next destructuring takes up most of the time.

6.3.2 The new tool and its usage

In order to reduce the burden on lint authors, a tool was developed as part of this thesis. The tool allows annotating examples of the offending code and then generating the boilerplate and destructuring code. The tool has been published as a component of clippy in order to increase the usage within the Rust community.

The tool itself is a lint that triggers on any code construct which has the special attribute `#[clippy::author]` attached. As an example consider the detection of a useless `+` operation where one argument is a `0`. This situation may occur after a half automated refactoring or due to a typographical error. The lint author would create an example code showing the issue

```
let x = variable + 0;
```

In order to apply the attribute to the expression `variable + 0` the expression is placed within parentheses and preceded with `#[clippy::author]`

```
let x = #[clippy::author](variable + 0);
```

The reason for the parentheses is that operator precedence rules also apply to attributes. Attributes bind stronger than any binary operation, which would mean that `#[clippy::author] variable + 0` would apply the attribute just to the `variable`. Unnecessary parentheses are ignored in Rust.

The above example produces the following output:

```
if_chain! {
    if let ExprKind::Binary(ref op, ref left, ref right) = expr.node;
    if BinOpKind::Add == op.node;
    if let ExprKind::Path(ref path) = left.node;
    if match_qpath(path, &["variable"]);
    if let ExprKind::Lit(ref lit) = right.node;
    if let LitKind::Int(0, _) = lit.node;
    then {
        // report your lint here
    }
}
```

which is a simplified version of

```
if let ExprKind::Binary(ref op, ref left, ref right) = expr.node {
    if BinOpKind::Add == op.node {
        if let ExprKind::Path(ref path) = left.node {
            if match_qpath(path, &["variable"]) {
```

```
                if let ExprKind::Lit(ref lit) = right.node {
                    if let LitKind::Int(0, _) = lit.node {
                        // report your lint here
                    }
                }
            }
        }
    }
}
```

To reduce the visually unhelpful nesting, the `if_chain` macro allows a linear list of conditions, assignments and conditional destructurings.

As a reminder, the expression started out with as `variable + 0`, which is destructured from the outside in. The outermost expression is a binary expression, having an operator, an expression to the left of the operator and an expression to the right. The pattern

```
ExprKind::Binary(ref op, ref left, ref right)
```

extracts each of these three components *if and only if* the expression is actually a binary expression. The `ref` keyword states that ownership of the components is not taken but they are borrowed instead. Since mutation or ownership is not of interest for observing the components, this suffices for the purpose of this example.

The operator of the function is a `+` or an addition. Thus the operator needs to be checked either via comparison

```
if BinOpKind::Add == op.node
```

or via destructuring

```
if let BinOpKind::Add = op.node
```

The left hand side of the expression is a variable name. The compiler does not treat variable names and other paths like function names (`some_module::some_function`) separately. Thus it can only be a single element path. First the left hand side expression is destructured if it is a path:

```
if let ExprKind::Path(ref path) = left.node
```

extracting the path in the process, and then a helper function for checking whether a path matches a given path element list is called:

```
if match_qpath(path, &["variable"])
```

In the `some_module::some_function` this check would be

```
if match_qpath(path, &["some_module", "some_function"])
```

The right hand side of the expression is a literal `0` expression. Again, we first destructure if it is a literal and extract the inner value.

```
if let ExprKind::Lit(ref lit) = right.node
```

Instead of further extracting the literal as a number, one can use a non-extracting pattern to directly check the value

```
if let LitKind::Int(0, _) = lit.node
```

It would have also been valid to extract and check in separate steps,

```
if let LitKind::Int(int_value, _) = lit.node
if int_value == 0
```

but the more compact form is generally preferred in simple cases. More complex cases benefit from multi-line destructuring, but in this case using the concise pattern benefits the readability.

Handling the by default overly constrained patterns

Even if the example expression is `variable + 0`, the author might have desired to match any expression which adds zero to any other expression and not just variables named `variable`. Fortunately loosening the rules on what is detected is as simple as removing the overly restrictive lines.

In this case, the relevant lines are

```
if let ExprKind::Path(ref path) = left.node;
if match_qpath(path, &["variable"]);
```

Which restrict the left hand side of the operation to paths which have a single element called `variable`. Removing those two lines will remove said rule and cause any addition whose right hand side is a literal zero to be remarked upon.

Detecting variations of the pattern

It suggests itself that not only `x + 0` but also `0 + x` should be linted due to the commutativity of integral addition. This requires duplicating the conditions to allow multiple options to trigger the lint.

```rust
if_chain! {
    if let ExprKind::Binary(ref op, ref left, ref right) = expr.node;
    if BinOpKind::Add == op.node;
    then {
        // iterate over all possible options (in this case two)
        for operand in &[left, right] {
            if_chain! {
                if let ExprKind::Lit(ref lit) = operand.node;
                if let LitKind::Int(0, _) = lit.node;
            then {
                // report your lint here
            }
        }
    }
}
```

Instead of continuing the `if_chain`, one iterates over all the possible operands (in the case of binary addition, two) and applies the check in each iteration.

6.4 Community driven database

> *Even the solitary genius programmer doesn't hack alone. They use code written by their younger inexperienced selves and write code that will be (ab-)used by their forgetful future selves.*
>
> – *free interpretation of The Unix Koans of Master Foo*

Static analyses are the vaccines of the software development industry. If a bug appears in multiple independent projects and reoccurs in seemingly harmless code written by new and veteran developers, the developers start writing code style guidelines. But guidelines have an inherent flaw: they aren't automatically enforced. Instead a second pair of eyes is required to detect violations of said guidelines. Thus, after the bug proves resistant to eliminating it via social means, authors of static analysis tools develop a new analysis detecting this bug. After the new analysis makes it into a release and is spread through the various projects, the bug is eliminated once and for all.

Similar to vaccinations, there are few static analyses for rare bugs that are easily fixed. Even if the bug itself is severe and might exist for a long time before being fixed, due to its low occurrence it will not become prominent enough to warrant designing a new static analysis for the developers of a commercial static analysis tool. Since some potential static analyses are project or library specific, there is no motivation to create a static analysis for something that doesn't benefit a significant percentage of the users. If the library is used throughout the community - e.g. the

boost[1] library in C++ - the existence of such an analysis can become important enough to encourage the development of a new analysis.

The flaw in this analogy with vaccinations is that there's no "health insurance" for software. This means that projects with a large business behind them can afford static analysis tools, while small companies cannot afford to pay the equivalent of an extra developer in the project.

Fortunately there are open source solutions which, due to their nature of open contribution, are often easy to integrate in existing projects and can contain precisely targeted static analyses that only apply to certain libraries or projects. At the same time, this dependence on contribution from users causes projects to stagnate when they become hard to use or fail to update to newer version of the programming languages they try to analyze.

The same issue exists for commercial static analysis tools, but there it is more serious, since the users cannot continue maintaining the tool themselves (since the source code is not available), but need to move on to another tool entirely. Furthermore, false positives and feature requests in open source projects can be addressed by the users themselves, speeding up process from issue detection to elimination significantly [141]. Bugfixes in the clippy project require changing between 100 and 500 lines of code and most pull requests are merged in under a week [58]. Writing a new analysis is more involved usually, and can take up to a year, depending on the complexity of the analysis and the availability of the lint author.

Licensing

A major worry in commercial projects is the use of free licenses within their own projects. GPL[2] licensed libraries may only be used within a project that is itself GPL licensed. There are various exceptions and modifications like the LGPL[3] license, but the rules can be complex enough to scare off potential users.

Static analysis tools do not suffer from this issue, since popular licenses like GPL, Apache2[4], MIT[5] and CC0-1.0 allow using tools in projects that do not fit their license, since the tool itself is not included in the final product. This is analogous to using a GPL licensed compiler (e.g. the GCC[6]) in a commercial closed source project. The compiler cannot be integrated into the final product, but it may generate the product without violating the license.

New contributions to an open source static analysis tool require the contribution to be licensed under the same license as the rest of the tool. This means that developers of a commercial project may produce open source code within the project if they themselves tackle an issue or feature in the static analysis tool.

[1] peer-reviewed portable C++ libraries
[2] GNU general public license
[3] Lesser General Public License
[4] an open source license
[5] open source license created by the Massachusetts Institute of Technology
[6] Gnu Compiler Collection, often also just refers to the Gnu C Compiler

Many companies forbid such contributions, since it allows an information flow from inside the company to the open internet. The perceived danger of disclosing information to or producing benefits for competitors can often be a show-stopper due to the increased effort required for developers to obtain permission to export expertise to outside the company.

Chapter 7

Evaluation

7.1 Analysis of the goals of this thesis

The new concept proposed in this thesis

- increases the quality of software projects by automating flaw detection,
- allows encoding developer experience in tooling to ensure it is not lost if the developer leaves the project,
- takes review and teaching load off experienced developers allowing new developers to quickly and automatically have their code reviewed without having to consult another developer, and
- creates a work flow and software engineering methodology that enables and encourages the introduction of static analysis and models at any point in the project's lifetime.

Furthermore, as part of the work of this thesis the author

- implemented prototypes and tools to employ the new methodology in real life projects, and
- contributed to open source projects related to compiler extensions and static analysis. This ensured that the work done for this thesis directly benefits existing projects and indirectly validates the introduced concepts.

Measuring the **quality of software projects** is in itself a heavily researched topic. While no empirical data was collected for showing that the concepts in this thesis help to increase software quality, it goes without saying that automating software quality checking will make it easier and cheaper to keep or move a piece of software to a desired quality level. The alternative of manually performing quality checks frequently will put a strain on developers and likely be a repetitive and boring task, and will thus likely be performed with little enthusiasm. This thesis presents various methods with which to incrementally add more and more automation to quality checking and measurement.

The goal of **encoding developer experience in tooling** has been achieved by this thesis. Designing project specific static analyses is now possible without extensive manual work and even without requiring a deep understanding of compiler internals.

These project specific analyses allow **experienced developers to write static analyses for problems frequently encountered by less experienced developers**. Developers (especially inexperienced ones) often fear that they waste the time of more experienced developers with their questions. Additionally human reviewers can accidentally appear to "sneer" or "look down" at the "bad" code written by inexperienced developers. When confronted with diagnostics by compilers and static analyses, developers of any experience level are much more comfortable with taking the advice, even criticizing it or asking for help if they don't understand the compiler messages.

The concepts of this thesis neither require a lock-in to just these concepts nor a large initial cost for introducing a minimal version of it. Every concept can be introduced incrementally to an existing project **without interrupting the existing structures and workflows**. It can even be introduced on a per module, function, file, or other arbitrary separation unit. Once it has proven its worth within a unit, it can be expanded to others or incrementally strengthened with more strict or new analyses. A survey gauging the importance of usability issues[1] was performed by the author [166]. The raw survey results can be found in Appendix I. While the usability issues appear to be superficial, they are the major blockers for adoption of compiler extensions. If the setup requires any manual intervention or more than the execution of a single command or installer, most potential users are already discouraged from attempting to install the extension. Further manual intervention like the installation of additional dependencies is disliked, but not deemed to be a blocker.

To showcase the concepts on real world projects, **implementations were prototyped on the Rust compiler** as well as on the clang C++ compiler. These prototypes where then applied to research projects, student projects and open source projects.

Once the Rust compiler prototypes reached a certain level of maturity, **they were upstreamed to the Rust project**, where the new analyses and static analysis design aids are now part of the static analysis toolbox nicknamed "clippy". The author furthermore enabled clippy to become trivially usable in any Rust project, allowing it to become officially distributed together with the Rust compiler and becoming essentially standard in many Rust projects.

[1] see Section 2.2.9 for an introduction of the usability issues

7.2 Usability study

A user study has been performed at the RustFest 2018 conference in Paris [113]. During a 6 hour workshop with 50 participants, the participants were asked to report their experience levels and their status during the workshop. During the workshop the participants wrote their own static analysis either for their own projects or for erroneous code examples.

7.2.1 Collected data

The participants self-reported their experience levels, which showed a large variance:

- one participant had never programmed until the day before the workshop
- five participants had never programmed Rust until the day before the workshop
- about two thirds of the participants had a year or more of experience with Rust
- less than a third of the participants had previous experience working with any compiler

After a short introduction into test driven development and a step-by-step walk-through for designing a very simple static analysis, each participant chose a problem to be detected either from a list given by the workshop conductor or came up with their own problems for which to write a static analysis. The time required for developing these analyses varied enormously:

- the first analysis was implemented after 30 minutes
- the longest development time was 5 hours
- most initial implementations were finished after around 3 hours

7.2.2 Analysis

The workshop showed that developers of any experience level can write their own static analysis within the time frame of a classical 8 hour work day by applying the methodology introduced in this thesis to the Rust language and compiler.

7.2.3 Limits of this study

The developer experience of the workshop participants was given by the participants themselves, and is thus highly influenced by whether the participants over- or under-evaluate themselves. Furthermore the benefits of the concepts introduced will only be measurable in real world projects with significantly more complexity than example code snippets. In a workshop environment, participants can freely help each other out, which may end up not reflecting the ability of individual developers. The

study only analyzed the case of whether developers were able to develop static analyses with the methodology introduced in this thesis. Without repeating the study with other approaches, no comparisons are possible. No detailed per-participant data has been collected, thus allowing no insights into correlations between previous experience, complexity of analysis and development speed.

Chapter 8

Conclusion

The present thesis aims at increasing the safety and security of software in all categories of software projects. The inclusion of any software project and its developers as the target audience instead of a focus on specific areas like safety critical systems is important, because it enables companies that are only now looking into software development to move to a safe and secure software development model incrementally [77]. Such companies include for example energy system component engineers who historically designed safety critical electrical systems, but are now required to expose software components for the energy system of the future. If there were an immediate requirement to move to the software safety and security level of e.g. the aerospace industries, this would be an enormous hurdle to overcome for these companies. Even though research on modern software development practices like model driven development or formal analyses is still moving forward actively, these practices have seen little adoption outside sectors where they are required by regulations [4, 142]. High upfront costs for employing these practices both in training and actual implementation time are frequent causes of this lack of adoption [77, 104].

This thesis gives the project developers themselves the capabilities to incrementally introduce static analyses, code checking, code generation and abstract interpretation schemes into their project as these schemes become effective or even required. To achieve this goal, both writing extensions and using them is made economical [7, 32, 123]. Classically, compilation systems are extended by creating new tools and integrating them in the toolchain. This requires duplicating significant parts of the work that the compiler already does, which is not just expensive, but can also lead to discrepancies between the various components of the toolchain. Instead, in this thesis a new concept is developed, which allows analyses to be integrated directly with the compiler. For the realization of the new concept existing compilers and build systems are analyzed for their extensibility as well as modified to improve the support for extensibility. Chapter 3 discusses new concepts that enable developers to write compiler extensions themselves. The list of the aforementioned extensions include

- static analyses for detecting API misuse and automating frequently occurring

simple but time consuming tasks like enforcing style guides and best practices,
- static analyses for checking whether code adheres to a model,
- automating manual tasks like generating code from models and integrating them in the main code base without the user having to concern themselves with generated code,
- abstractly interpreting test suites in order to catch problems that are infeasible to detect via static analysis, and
- inventing further, entirely new processes that are uneconomical without having access to compiler internal state.

The scheme of automatically checking whether code adheres to a model is a novel one introduced in this thesis. Without the compiler extensions from this thesis, such a scheme is impractical both in the amount of work and in the developer experience. This code checking scheme takes an existing model and directly compares it to a reference implementation. It can either be implemented in a holistic way, requiring that the code covers all the model's constraints, or introduced to an existing project, incrementally synchronizing the code with the model.

Classical model driven development with the use of code generation can also be employed, and keeps the user-written source code free of any automatically generated code. Instead of writing generated code to disk, including it in the compilation process, and losing all links between the model and any issues that may occur in the development, only the compiler sees a temporary version of the generated code in memory. Instead of encoding the model in pure code, it is annotated beyond what is possible in code. Often, the abstract syntax tree or lower level intermediate representations can be generated directly, skipping the source representation entirely. Any of these abstract annotations allow diagnostic messages and developer tools to link code constructs with model constructs. Through the links, developers can use their IDE or other tooling to traverse, analyze and modify the entire project easily, instead of having to look up information through different, often incompatible systems. Chapter 4 compares this new concept with the classical methodologies.

All of this is not done by extending build systems with more tools, but instead integrating with the modular compiler infrastructure seamlessly. This means that there is never an inconsistency between the different views (model, code, requirements, …) on the entire software project. It also aids in tracking down issues, as the central unit of this new scheme - the compiler - knows all components and diagnostic messages can thus be adjusted accordingly. By including all relevant information in a diagnostic message and often even generating a solution suggestion, experienced developers in projects can encode their experience programmatically, offloading mentoring and teaching work that they normally have to perform to allow less experienced developers to participate [8]. Methodologies for designing high quality diagnostic messages and automatically applicable resolution suggestions were introduced.

While the implementation work in Chapter 5 focuses on the Rust compiler, experimental prototypes for the clang C++ compiler were implemented to showcase the

portability of the concepts. Various integration techniques are demonstrated with example implementations showing the ease at which complex compiler extensions can be built by starting out with a minimal viable prototype and building up incrementally to a powerful tool.

8.1 Contributions

Throughout the process of this thesis the author has

- implemented many static analyses and shared them with the Rust community,
- developed a system that automatically generates static analyses from offending code examples, which is used in practice by inexperienced and experienced developers alike to create new static analysis without a steep learning curve,
- blurred the boundary between constant evaluation and virtual machines by integrating the latter (miri, an abstract interpreter) into the official Rust compiler, fully replacing the existing constant evaluator,
- extended miri to allow arbitrary computations like regular expression matching, parsing of json or binary data, UTF8-checking or arbitrary precision math, and
- made a static analysis collection (clippy) trivially usable for Rust programmers, directly causing widespread adoption as observable in
  - the Fuchsia project (by Google) [43],
  - the Azure Pipelines Software (by Microsoft) [93],
  - the libra project (by Facebook, Lyft, Spotify, Uber, etc.) [33],
  - the servo browser renderer (by Mozilla) [96],
  - the parity bitcoin client [102],
  - a python interpreter [107],
  - the ripgrep project [37],
  - as well as many others.

The author has observed that sharing modular components of and with a compiler not just works in practice, but raises developer and user contentment. This helps to create an influx of new (even inexperienced) developers contributing to projects which commonly have had a high barrier of entry. By mentoring new developers and posting surveys, the author has collected information about stumbling stones that still persist and either acted on this feedback by adjusting documentation or software, or by incorporating new insights into this thesis.

While the assumption, that implementing static analyses and other compiler extensions is too expensive for non-safety-critical applications, is widespread, it is the author's opinion, that failing to address issues proactively at best moves the costs into the testing and production phase. Often the costs are even shifted to the end-user (requiring them to implement workarounds) and to the maintenance phase, increasing the costs enormously due to patching or recalls [60]. This shifting of the

costs happens in an unstructured and unplanned manner, making it hard to trace the costs. In contrast to that, the new concept helps projects recover from situations that usually would have lead to a complete rewrite and allows already stable projects to work towards full model driven development and formal analyses. This makes the costs very obvious, as there is explicit work being done.

8.2 Limits

The concepts introduced in this thesis require every compiler that uses these concepts to implement APIs for developers to consume. Different compilers can have very different designs for these APIs and are sometimes programmed in a language different from the target programming language. This makes sharing analyses between compilers hard or essentially impossible. Furthermore this requires a lot of duplication of similar constructs, since each compiler will require its own implementation.

Many programming languages do not consider the compiler as part of their specification. This by itself makes it hard to create a standardized interface even within a single language. Even if compiler aspects are part of the language standard, the standardization of compiler aspects are often kept as a compiler-specific extension, reducing the portability of projects using these APIs. The languages that have stabilized such systems naturally have a very high bar for introducing new concepts, since they will need to support a stabilized system indefinitely. These restrictions can severely limit the application of the concepts from this thesis.

It is inherently hard to perform controlled studies in topics where there are as many uncontrollable and unmeasurable variables as there are in a hypothetical study that could be performed to analyze the effectiveness of the concepts introduced in this thesis.

8.3 Outlook

As with all theses, the present thesis not only answers scientific questions, but opens new scientific questions that need to be addressed in future works. For instance, the designs of this thesis can be applied to other language's compilers, which has been experimentally done as part of this thesis. An interesting avenue for further work would be to attempt to find common ground between the compilers to share APIs, code and thus standardized data formats. Such a scheme could allow compilers to share entire analyses and other extensions.

Even without looking at compilers other than the Rust compiler, there is work to be done to fully standardize an API that end users can use directly from inside the language instead of having to resort to the inofficial schemes introduced in this thesis. The high standards that the Rust community requires from APIs accessible from

the language itself make this a nontrivial undertaking, requiring further research. Solely looking at the implementation perspective, the additional work is tractable.

While some anecdotal evidence has been collected in university student practica and in conference workshops, controlled studies are expected to yield new insights into the effect of self improving software development processes on programmer satisfaction, error rates and software sustainability via the new, scalable processes for reduction of technical debt.

Abbreviations

Abbreviation	Long Description
AADL	Architecture Analysis & Design Language
Ada	a programming language for safety critical systems
Apache2	an open source license
API	application programming interface
ASIS	Ada Semantic Interface Specification
ASLR	address space layout randomization
AST	abstract syntax tree
BASF	a german chemical company
boilerplate	supporting code that does not contribute to the logic, but is still required
boost	peer-reviewed portable C++ libraries
C	a programming language popular in systems programming
C++	an object oriented language based on C
C/C++	The family of C-like languages
CI	continuous integration
clang	a C language family frontend for LLVM
CLI	command line interface
clippy	a static analysis tool for the Rust language
DSL	domain specific language
EBNF	extended Backus-Naur form
gcc	the GNU Compiler Collection
GCC	Gnu Compiler Collection, often also just refers to the Gnu C Compiler
ghc	Glasgow Haskell Compiler
Go	a programming language developed by google as a replacement for C++
GPL	GNU general public license
GUI	Graphical User Interface
haskell	a popular functional language
HIR	high-level intermediate representation
IDE	integrated development environment
IR	intermediate representation

Abbreviation	Long Description
Java	a safe garbage collected programming language for object oriented programming
Javascript	a programming language for interactive webpages
JIT	Just In Time compilation
JML	Java Modelling Language
JSON	a self-describing data storage and transfer format
LGPL	Lesser General Public License
lint	either a single static analysis or its diagnostic message
linter	a tool containing one or many lints that can be run on program code
LSP	Language Server Protocol
matlab	a prototyping programming language and its corresponding IDE
MDD	model driven development
MIR	medium-level intermediate representation
MISRA-C	a subset of the C programming language used in the automotive industry
MIT	open source license created by the Massachusetts Institute of Technology
NASA	North American Space Agency
PHP	a programming language for web-backends
python	a programming language
RFC	Request For Comment, a peer reviewed process for changing the Rust language
rls	Rust Language Server
Ruby	a dynamic programming language
Rust	a systems programming language
rustc	the Rust Compiler
SA	Static Analysis
servo	a browser engine written in Rust
SPARK	a subset of Ada which allows formal proofs
Swift	successor of Objective-C for safe iOS app development
typestate	a concept for explicitly listing discrete states of objects
UML	Unified modeling language
VM	Virtual Machine
WCET	worst case execution time
XML	a self-describing data storage and transfer format

Bibliography

1. AdaCore; Altran UK Ltd (2017) SPARK 2014 Reference Manual.

2. Aldrich, J.; Sunshine, J.; Saini, D.; Sparks, Z. (2009) Typestate-oriented programming. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, pp 1015–1022.

3. Alhazmi, OH.; Malaiya, YK.; Ray, I. (2007) Measuring, analyzing and predicting security vulnerabilities in software systems. computers & security 26:219–228.

4. Anderson, B.; Bergstrom, L.; Goregaokar, M.; Matthews, J.; McAllister, K.; Moffitt, J.; Sapin, S. (2016) Engineering the servo web browser engine using Rust. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp 81–89.

5. Ashcraft, K.; Engler, D. (2002) Using programmer-written compiler extensions to catch security holes. In: Proceedings 2002 IEEE Symposium on Security and Privacy. IEEE, pp 143–159.

6. Babati, B.; Pataki, N. (2019) Static analysis of functors' mathematical properties in C++ source code. In: AIP Conference Proceedings. AIP Publishing, p 350002.

7. Bai, J-J.; Wang, Y-P.; Lawall, J.; Hu, S-M. (2018) DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp 587–600.

8. Balachandran, V. (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp 931–940.

9. Barik, T. (2016) How should static analysis tools explain anomalies to developers? In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp 1118–1120.

10. Barnes, J. (2005) Ada Rationale. http://www.adaic.org/resources/add_content/standards/05rat/html/Rat-5-4.html. Accessed 3 Oct 2018

11. Batcheller, A.; Fowler, SC.; Cunningham, R.; Doyle, D.; Jaeger, T.; Lindqvist, U. (2017) Building on the Success of Building Security In. IEEE Security & Privacy 15:85–87.

12. Becker, BA.; Glanville, G.; Iwashima, R.; McDonnell, C.; Goslin, K.; Mooney, C. (2016) Effective compiler error message enhancement for novice programming students. Computer Science Education 26:148–175.

13. Becker, BA.; Murray, C.; Tao, T.; Song, C.; McCartney, R.; Sanders, K. (2018) Fix the first, ignore the rest: Dealing with multiple compiler error messages. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. ACM, pp 634–639.

14. Bloch, M.; Barros, J. (2011) Physical-layer security: from information theory to security engineering. Cambridge University Press.

15. Brandt, T. (2015) SpaceX: Bringing Agile & BDD to the Final Frontier. https://www.linkedin.com/pulse/spacex-bringing-agile-bdd-final-frontier-timothy-brandt. Accessed 20 Mar 2016

16. Briand, L.; Labiche, Y.; Liu, Y. (2012) Combining UML sequence and state machine diagrams for data-flow based integration testing. In: Modelling Foundations and Applications. Springer, pp 74–89.

17. Broy, M.; Reussner, R. (2010) Architectural concepts in programming languages. Computer 43:88–91. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2010.277

18. Burns, A. (1999) The ravenscar profile. ACM SIGAda Ada Letters 19:49–52.

19. Buxton, JN.; Randell, B. (1970) Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee. NATO Science Committee/Scientific Affairs Division.

20. Campbell, GA. (2017) Cognitive Complexity-A new way of measuring understandability. Technical Report. SonarSource SA, Switzerland.

21. Chess, B.; McGraw, G. (2004) Static analysis for security. IEEE security & privacy 2:76–79.

22. Chiba, S. (2000) Load-time structural reflection in Java. In: European Conference on Object-Oriented Programming. Springer, pp 313–336.

23. Conley, CA.; Sproull, L. (2008) Design for quality: The case of open source software development. New York University, Graduate School of Business Administration.

24. Cox, JW. (2016) MedStar Health turns away patients after likely ransomware cyberattack. The Washington Post 29.

25. CRISP (2016) CAST-Workshop: Smart Metering Security.

26. crumblingstatue (2015) identity op lint triggers for constants. https://github.com/rust-lang/rust-clippy/issues/183. Accessed 19 Nov 2015

27. Cuoq, P.; Kirchner, F.; Kosmatov, N.; Prevosto, V.; Signoles, J.; Yakobowski, B. (2012) Frama-C. In: International Conference on Software Engineering and Formal Methods. Springer, pp 233–247.

28. Demuth, A.; Riedl-Ehrenleitner, M.; Egyed, A. (2016) Efficient detection of inconsistencies in a multi-developer engineering environment. In: 2016 31st ieee/acm international conference on automated software engineering (ase). IEEE, pp 590–601.

29. Dig, D.; Johnson, R. (2005) The role of refactorings in API evolution. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp 389–398.

30. Dyer, JW.; Stafford, LGTP.; Miller, JD. (2011) International Space Station Advisory Committee and Aerospace Safety Advisory Panel - Open Meeting Report.

31. Elsayed, EA. (2012) Reliability engineering. John Wiley & Sons.

32. Engler, D.; Chelf, B.; Chou, A.; Hallem, S. (2000) Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4. Usenix Association, p 1.

33. Facebook (2019) Libra. https://github.com/libra/libra. Accessed 15 Dec 2019

34. Fatin, P. (2015) Typing with pleasure. https://pavelfatin.com/typing-with-pleasure/. Accessed 30 May 2018

35. Feiler, PH.; Gluch, DP.; Hudak, JJ. (2006) The architecture analysis & design language (AADL): An introduction. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

36. Galinina, O.; Andreev, S.; Balandin, S.; Koucheryavy, Y. (2019) Internet of things, smart spaces, and next generation networks and systems: 19th international conference, new2an 2019, and 12th conference, ruSMART 2019, st. Petersburg, russia, august 26–28, 2019, proceedings. Springer Nature.

37. Gallant, A. (2017) Ripgrep recursively searches directories for a regex pattern. https://github.com/BurntSushi/ripgrep. Accessed 29 May 2020

38. Gancarz, M. (2001) Unix philosophy. 株式会社オ-ム社.

39. Garrido, J.; Zamorano, J.; Puente, JA de la.; Alonso, A.; Salazar, E. (2015) Ada, the programming language of choice for the upmsat-2 satellite. Data Systems in Aerospace—DASIA 2015.

40. Gilchrist, A. (2016) Industry 4.0: The industrial internet of things. Springer.

41. Goldman, M.; Little, G.; Miller, RC. (2011) Real-time collaborative coding in a web IDE. In: Proceedings of the 24th annual ACM symposium on User interface software and technology. ACM, pp 155–164.

42. Goldsborough, P. (2017) Emitting Diagnostics in Clang. http://www.goldsborough.me/c++/clang/llvm/tools/2017/02/24/00-00-06-emitting_diagnostics_and_fixithints_in_clang_tools/. Accessed 24 Feb 2017

43. Google (2019) Fuchsia. https://fuchsia.googlesource.com/fuchsia/. Accessed 15 Dec 2019

44. Goregaokar, M.; Schneider, O. (2018) Rust RFC 2476: Stabilize the Clippy static analyzer for Rust. https://github.com/rust-lang/rfcs/pull/2476. Accessed 8 Aug 2018

45. Graf, J.; Hecker, M.; Mohr, M.; Snelting, G. (2016) Sicherheitsanalyse mit JOANA. Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit.

46. Greenberg, A. (2015) Hackers remotely kill a jeep on the highway - With me in it. Wired 7:21.

47. Guth, D.; Hathhorn, C.; Saxena, M.; Roşu, G. (2016) Rv-match: Practical semantics-based program analysis. In: International Conference on Computer Aided Verification. Springer, pp 447–453.

48. Hagenmeyer, V.; Kemal Çakmak, H.; Düpmeier, C.; Faulwasser, T.; Isele, J.; Keller, HB.; Kohlhepp, P.; Kühnapfel, U.; Stucky, U.; Waczowicz, S.; others (2016) Information and communication technology in energy lab 2.0: Smart energies system simulation and control center with an open-street-map-based power flow simulation example. Energy Technology 4:145–162.

49. Hammer, C.; Schaade, R.; Snelting, G. (2008) Static path conditions for Java. In: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security. ACM, pp 57–66.

50. Hathhorn, C.; Ellison, C.; Roşu, G. (2015) Defining the undefinedness of C. In: ACM SIGPLAN Notices. ACM, pp 336–345.

51. Hatton, L. (2004) Safer language subsets: an overview and a case history, MISRA C. Information and Software Technology 46:465–472.

52. Head, A.; Sadowski, C.; Murphy-Hill, E.; Knight, A. (2018) When not to comment: questions and tradeoffs with API documentation for C++ projects. In: Proceedings of the 40th International Conference on Software Engineering. ACM, pp 643–653.

53. Holzmann, G. (2002) Static source code checking for user-defined properties. Proc. IDPT 2.

54. Hüttermann, M. (2012) Infrastructure as code. In: DevOps for developers. Springer, pp 135–156.

55. Ichbiah, JD.; Firth, R.; Hilfinger, PN.; Roubine, O.; Woodger, M.; Barnes, JG.; Abrial, J-R.; Gailly, J-L.; Heliard, J-C.; Ledgard, HF.; others (1983) Reference manual for the Ada programming language. Castle House Publications Tunbridge Wells, Kent.

56. Iyengar, SS.; Lepper, MR. (2000) When choice is demotivating: Can one desire too much of a good thing? Journal of personality and social psychology 79:995.

57. Johnson, SC. (1977) Lint, a C program checker. Citeseer.

58. Jones, P. Pull Request and Issue statistics. http://pgjones.github.io/push-pull/. Accessed 10 Jun 2019

59. Jongeling, R. (2019) How to live with inconsistencies in industrial model-based development practice. In: 2019 acm/ieee 22nd international conference on model driven engineering languages and systems companion (models-c). IEEE, pp 642–647.

60. Jorgensen, M.; Shepperd, M. (2006) A systematic review of software development cost estimation studies. IEEE Transactions on software engineering 33:33–53.

61. Jung, R. (2018) Pointers Are Complicated, or: What's in a Byte? https://www.ralfj.de/blog/2018/07/24/pointers-and-bytes.html. Accessed 24 Jul 2018

62. Jürjens, J. (2002) UMLsec: Extending UML for secure systems development. UML 2002 – The Unified Modeling Language 1–9.

63. Kahn, D. (1996) The codebreakers: The comprehensive history of secret communication from ancient times to the internet. Simon; Schuster.

64. Karlesky, M.; Williams, G.; Bereza, W.; Fletcher, M. (2007) Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In: Proc. Emb. Systems Conf, CA, USA. pp 1518–1532.

65. Keller, HB.; Schneider, O.; Matthes, J.; Hagenmeyer, V. (2016) Zuverlässige und sichere Software offener Automatisierungssysteme der Zukunft – Herausforderungen und Lösungswege. at-Automatisierungstechnik 64:930–947.

66. Kersten, M.; Matthes, J.; Manga, CF.; Zipser, S.; Keller, H. (1999) Customizing UML for the development of distributed reactive systems and code generation to Ada 95. Ada User Journal 23.

67. Kim, CHP.; Batory, D.; Khurshid, S. (2010) Eliminating products to test in a software product line. In: Proceedings of the ieee/acm international conference on automated software engineering. pp 139–142.

68. Kim, T.; Bierhoff, K.; Aldrich, J.; Kang, S. (2009) Typestate protocol specification in JML. In: Proceedings of the 8th international workshop on Specification and verification of component-based systems. ACM, pp 11–18.

69. Kinder, J.; Katzenbeisser, S.; Schallhart, C.; Veith, H. (2005) Detecting malicious code by model checking. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp 174–187.

70. King, S.; Hammond, J.; Chapman, R.; Pryor, A. (2000) Is proof more cost-effective than testing? IEEE Transactions on software Engineering 26:675–686.

71. Kiometzis, M.; Ullmann, M. (2017) Fahrdaten für alle? Car-2-car kommunikation und die folgen.

72. Klabnik, S. (2018) Declarative Macros with macro rules for General Metaprogramming. https://doc.rust-lang.org/book/ch19-06-macros.html#declarative-macros-with-macro_rules-for-general-metaprogramming. Accessed 3 Mar 2019

73. Klabnik, S. (2018) Function-like macros. https://doc.rust-lang.org/book/ch19-06-macros.html#function-like-macros. Accessed 3 Mar 2019

74. Kleinert, DT. (2016) Ausgewählte Entwicklungsthemen der sicherheitsgerichteten Automatisierung bei BASF.

75. Kochhar, PS.; Wijedasa, D.; Lo, D. (2016) A large scale study of multiple programming languages and code quality. In: 2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner). IEEE, pp 563–573.

76. Kopetz, H.; others (2016) Software reliability. Macmillan International Higher Education.

77. Krishnamurthy, R.; Heinze, T.; Haupt, C.; Schreiber, A.; Meinel, M. (2019) Scientific Developers v/s Static Analysis Tools.

78. Lattner, C.; Adve, V. (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, p 75.

79. Lee, Robert M. and Assante, Michael J. and Conway, Tim (2016) Analysis of the Cyber Attack on the Ukrainian Power Grid.

80. Limoncelli, TA. (2018) Manual Work is a Bug. Queue 16:20.

81. Litecky, CR.; Davis, GB. (1976) A study of errors, error-proneness, and error diagnosis in Cobol. Communications of the ACM 19:33–38.

82. Lodderstedt, T.; Basin, D.; Doser, J. (2002) SecureUML: A UML-based modeling language for model-driven security. UML 2002 – The Unified Modeling Language 426–441.

83. Löffler, M.; Decker, R. (2017) „Connected car “und customer experience management–unlösbare herausforderung oder gemeinsame chance für hersteller und händler? In: Innovative produkte und dienstleistungen in der mobilität. Springer, pp 521–533.

84. Matusiewicz, D.; Pittelkau, C.; Elmer, A. (2018) Die digitale transformation im gesundheitswesen. MWV Medizinisch Wissenschaftliche Verlagsgesellschaft.

85. Mårtensson, T.; Ståhl, D.; Bosch, J. (2017) Continuous integration impediments in large-scale industry projects. In: 2017 ieee international conference on software architecture (icsa). IEEE, pp 169–178.

86. McCluskey, G. (1998) Using java reflection. Java Developer Connection.

87. McConnell, S. (1996) Software quality at top speed. Software Development 4:38–42.

88. McGinniss, I.; Gay, S. (2011) Hanoi: A typestate DSL for Java.

89. McKeeman, WM.; Horning, JJ.; Wortman, DB. (1970) A compiler generator. Prentice-Hall Englewood Cliffs, NJ.

90. McKinney, W. (2012) Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. O'Reilly Media, Inc.

91. McNally, R.; Yiu, K.; Grove, D.; Gerhardy, D. (2012) Fuzzing: The state of the art. Defence, science and technology organization Edinburgh (Australia).

92. Menzies, T.; Nichols, W.; Shull, F.; Layman, L. (2017) Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle. Empirical Software Engineering 22:1903–1935.

93. Microsoft (2019) Azure. https://github.com/Azure/iotedge/search?q=clippy. Accessed 15 Dec 2019

94. Min, C.; Kashyap, S.; Lee, B.; Song, C.; Kim, T. (2015) Cross-checking semantic correctness: The case of finding file system bugs. In: Proceedings of the 25th Symposium on Operating Systems Principles. ACM, pp 361–377.

95. Mohurle, S.; Patil, M. (2017) A brief study of Wannacry Threat: Ransomware Attack 2017. International Journal 8.

96. Mozilla (2019) servo. https://github.com/servo/servo. Accessed 15 Dec 2019

97. Møller, A.; Schwartzbach, MI. (2012) Static program analysis. Aarhus University.

98. Object Management Group (2007) XML Metadata Interchange Specification.

99. Olson, S. (2016) Miri: An interpreter for Rust's mid-level intermediate representation. Master's thesis, University of Saskatchewan

100. OMG (2011) UML 2.4.1 Superstructure Specification. OMG.

101. OMG (2017) UML 2.5 Specification. OMG.

102. Parity (2019) parity bitcoin client. https://github.com/paritytech/parity-bitcoin. Accessed 29 May 2020

103. Pataki, N.; Szűgyi, Z.; Dévai, G. (2011) Measuring the overhead of C++ standard template library safe variants. Electronic Notes in Theoretical Computer Science 264:71–83.

104. Paulk, MC. (2005) An empirical study of process discipline and software quality. PhD thesis, University of Pittsburgh

105. Peachock, P.; Iovino, N.; Sharif, B. (2017) Investigating Eye Movements in Natural Language and C++ Source Code-A Replication Experiment. In: International Conference on Augmented Cognition. Springer, pp 206–218.

106. Preston-Werner, T. (2013) Semantic Versioning 2.0.0. In: línea. http://semver.org. Accessed 29 May 2020

107. RustPython (2020) Python interpreter written in Rust. https://github.com/RustPython/RustPython. Accessed 29 May 2020

108. Rybin, S.; Fofanov, V. (2000) Building Ada Development Tools with ASIS-for-GNAT. SIGAda.

109. Sadowski, C.; Aftandilian, E.; Eagle, A.; Miller-Cushon, L.; Jaspan, C. (2018) Lessons from building static analysis tools at Google. Communications of the ACM 61:58–66.

110. Santamarta, R. (2014) A wake-up call for SATCOM security. Technical White Paper.

111. Sarkar, A. (2015) The impact of syntax colouring on program comprehension. In: Proceedings of the 26th annual conference of the psychology of programming interest group (ppig 2015). pp 49–58.

112. Scherer, O. (2018) Compiler-internal lints. https://github.com/rust-lang/rust/issues/49509. Accessed 25 Apr 2020

113. Scherer, O. (2018) Fixing bugs once and for all. Rustfest Paris.

114. Schlich, B.; Kowalewski, S. (2009) Model checking C source code for embedded systems. International journal on software tools for technology transfer 11:187–202.

115. Schneider, O. (2016) Clippy-internal lints. https://github.com/rust-lang/rust-clippy/pull/1093. Accessed 20 Jun 2016

116. Schneider, O. (2017) Refactor suggestion diagnostic API to allow for multiple suggestions. https://github.com/rust-lang/rust/pull/41876. Accessed 18 Nov 2017

117. Schneider, O. (2017) Include rendered diagnostics in json. https://github.com/rust-lang/rust/pull/46052. Accessed 8 Nov 2017

118. Schneider, O. (2017) Change some notes into suggestions. https://github.com/rust-lang/rust/pull/43929. Accessed 21 Aug 2017

119. Schneider, O. (2018) Replace all const evaluation with miri. https://github.com/rust-lang/rust/pull/46882. Accessed 8 Mar 2018

120. Schneider, O. (2018) A graphical debugger for Rust MIR. https://github.com/oli-obk/priroda. Accessed 31 May 2020

121. Schneider, O.; Keller, HB. (2016) Integrierte Entwicklung zuverlässiger Software. Softwaretechnik-Trends 36.

122. Sedano, T.; Ralph, P.; Péraire, C. (2017) Software development waste. In: 2017 ieee/acm 39th international conference on software engineering (icse). IEEE, pp 130–140.

123. Shastry, B.; Yamaguchi, F.; Rieck, K.; Seifert, J-P. (2016) Towards vulnerability discovery using staged program analysis. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp 78–97.

124. Siminiceanu, RI.; Cataño, N.; Madeira, I.; CMU-Portugal, C da P. (2011) Automated Verification of Specifications with Typestates and Access Permissions. National Aeronautics; Space Administration, Langley Research Center.

125. Singer-Heinze, R.; Rajasekaran, PK.; Gupta, N. Smarter Compiler Messages for Semicolons in Eclipse.

126. Stach, C.; Alpers, S.; Betz, S.; Dürr, F.; Fritsch, A.; Mindermann, K.; Murthy Palanisamy, S.; Schiefer, G.; Wagner, M.; Mitschang, B.; others (2018) The avare patron-a holistic privacy approach for the internet of things. In: Proceedings of the 15th international joint conference on e-business and telecommunications. p 538.

127. Stackoverflow (2020) Developer survey. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved.

128. Stefik, A.; Siebert, S.; Stefik, M.; Slattery, K. (2011) An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In: Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools. ACM, pp 3–8.

129. Strom, RE.; Yemini, S. (1986) Typestate: A programming language concept for enhancing software reliability. Software Engineering, IEEE Transactions on 157–171.

130. Subramaniam, C.; Sen, R.; Nelson, ML. (2009) Determinants of open source software project success: A longitudinal study. Decision Support Systems 46:576–585.

131. thepowersgang (2018) Alternative rust compiler (re-implementation). https://github.com/thepowersgang/mrustc. Accessed 18 Oct 2018

132. The Rust Language Developers (2019) Build Scripts. https://doc.rust-lang.org/cargo/reference/build-scripts.html. Accessed 15 Aug 2019

133. Traver, VJ. (2010) On compiler error messages: what they say and what they mean. Advances in Human-Computer Interaction 2010.

134. TrustInSoft (2016) An interpreter for finding subtle bugs in programs written in standard C. https://trust-in-soft.com/tis-interpreter/. Accessed 14 Dec 2016

135. Unruh, E. (1994) Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.

136. Visser, E.; others (1997) Syntax definition for language prototyping. Eelco Visser.

137. Voelter, M. (2011) Language and IDE Modularization and Composition with MPS. In: International Summer School on Generative and Transformational Techniques in Software Engineering. Springer, pp 383–430.

138. Vokáč, M.; Tichy, W.; Sjøberg, DI.; Arisholm, E.; Aldrin, M. (2004) A controlled experiment comparing the maintainability of programs designed with and without design patterns – a replication in a real programming environment. Empirical Software Engineering 9:149–195.

139. Volanschi, N. (2008) A portable compiler-integrated approach to permanent checking. Automated Software Engineering 15:3–33.

140. Volanschi, N. (2012) Pattern matching for the masses using custom notations. Science of Computer Programming 77:609–635.

141. Vonnegut, S. (2015) Open Source vs. Commercial Tools: Static Code Analysis Showdown. https://www.checkmarx.com/2015/03/17/open-source-vs-commercial-tools-static-code-analysis-showdown-2/. Accessed 12 May 2015

142. Vyatkin, V. (2013) Software engineering in industrial automation: State-of-the-art review. IEEE Transactions on Industrial Informatics 9:1234–1249.

143. Waite, WM.; Goos, G. (2012) Compiler construction. Springer Science & Business Media.

144. Wolff, R.; Garcia, R.; Tanter, É.; Aldrich, J. (2011) Gradual typestate. In: ECOOP 2011–Object-Oriented Programming. Springer, pp 459–483.

145. Xu, G.; Mitchell, N.; Arnold, M.; Rountev, A.; Sevitsky, G. (2010) Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, pp 421–426.

146. Zhu, KX.; Zhou, ZZ. (2012) Research note—lock-in strategy in software competition: Open-source software vs. Proprietary software. Information Systems Research 23:536–545.

147. (2017) Expressive Diagnostics (clang). http://clang.llvm.org/diagnostics.html. Accessed 4 Jul 2017

148. (2017) Comparison of Diagnostics between GCC and Clang. https://gcc.gnu.org/wiki/ClangDiagnosticsComparison. Accessed 4 Jul 2017

149. The Rust Language. http://www.rust–lang.org/.

150. First Go Commit. https://go.googlesource.com/go/+/7d7c6a97f815e9279d08cfaea7d5efb5e90695a8. Accessed 12 Jul 2019

151. First Rust Commit. https://github.com/rust-lang/rust/commit/c01efc669f09508b55eced32d3c88702578a7c3e. Accessed 12 Jul 2019

152. Swift. http://www.nondot.org/sabre/. Accessed 12 Jul 2019

153. Tools supporting MISRA C and MISRA C++. https://misra.org.uk/Links/tabid/63/Default.aspx. Accessed 31 May 2020

154. GCC 4.5.0 plugin support. https://gcc.gnu.org/wiki/plugins. Accessed 9 Mar 2016

155. Extending and using GHC as a Library - Compiler Plugins. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/compiler-plugins.html. Accessed 9 Mar 2016

156. Clang Plugins. http://clang.llvm.org/docs/ClangPlugins.html. Accessed 9 Mar 2016

157. Rust ‘plugin‘ feature. https://github.com/rust-lang/rust/issues/29597. Accessed 9 Mar 2016

158. GHC/As a library. https://wiki.haskell.org/GHC/As_a_library. Accessed 12 Aug 2016

159. https://github.com/rust-lang/rfcs/pull/1681. Accessed 12 Aug 2016

160. The Disadvantages of UML. https://www.techwalla.com/articles/the-disadvantages-of-uml. Accessed 12 Mar 2018

161. NDepend Visual Studio extension. https://www.ndepend.com/. Accessed 7 Mar 2018

162. semmle engineering analysis platform. https://www.semmle.com/. Accessed 7 Mar 2018

163. CppDepend Code analysis tools. https://www.cppdepend.com/. Accessed 7 Mar 2018

164. custom notations for data structures. http://mypatterns.free.fr/. Accessed 7 Mar 2018

165. Comparing trait object references for equality gives different results based on the number of codegen units. https://github.com/rust-lang/rust/issues/46139. Accessed 30 Mar 2020

166. A survey about usability issues. https://users.rust-lang.org/t/survey-what-discourages-you-from-using-a-tool/14960. Accessed 31 May 2020

Appendices

Appendix A

Semantic versioning

Semantic versioning [106] was designed by Tom Preston-Werner as a solution to the ongoing problems with breaking changes and arbitrary versioning schemes. Breaking changes are discussed in detail in Appendix B. When a breaking change is applied to an API, users of the API need to update their code if they use the changed API. When updating a project dependency, it is not clear to the user whether the update contains a breaking change or just security fixes, bug fixes or completely new features. Semantic versioning signals these different update reasons via a three component version. The version is formatted `MAJOR.MINOR.PATCH`, where incrementing

- `PATCH` means bug fixes that do not cause breaking changes
- `MINOR` means new features which do not cause breaking changes
- `MAJOR` means you are doing breaking changes

This allows users of a dependency that follows the semantic versioning scheme to update to versions only changing `PATCH` or `MINOR` without worrying about breaking changes.

Semantic versioning is not transitive. A dependency may update one of its dependencies across a `MAJOR` version without updating its own `MAJOR` version, as long as no breaking changes occur due to the dependency update. This commonly happens when a dependency is just an implementation detail that is not exposed via the public API.

Appendix B

Forward compatibility guarantees

As programming languages and their compilers evolve, the internal structure and API of the compilers changes significantly. This means that any code written to interact with a specific version of a compiler, will stop working with newer versions and not work with older versions of the compiler.

The API surface of a library consists of functions, types and values (constants and statics). There are several classes of changes that can be done to an API. The most benign change is to strictly increase the surface of the API without touching the existing API surface. This kind of change includes:

1. Adding new items
2. Adding private fields to a struct that already has private fields
3. Renaming a type but adding a type alias with the old name
4. Renaming a function but adding a forwarding function with the old name
5. Increasing the set of input accepted by a function
6. Performance improvements of code
7. Fixing a crash

Many changes perceived by library authors as benign are in fact breaking changes for library users and often perceived as highly disruptive [29]. These changes include

1. Change the name of an item
2. Add a new field to a struct
3. Fix a bug in function code that library users might have worked around
4. A bug fix that causes previously accepted function input to be revoked

Finally there are various changes which clearly fall into the category of breaking changes. These include any kind of change that completely changes the behavior, values or types of items or simply removes items that library users might be using.

1. Decrease the Set of valid input to a function

2. Change a constant/static's value
3. Remove an item
4. Change a struct field's type
5. Change the number of arguments of a function
6. Change the type of a function argument
7. Change the behavior of a function for a valid input

Appendix C

The Rust programming language

This section covers several constructs used throughout this thesis. For more in depth documentation see https://www.rust-lang.org/learn.

C.1 Basics

Variables in Rust are declared with the `let` keyword. This is comparable to using the `auto` keyword in C++ to declare variables:

```
let x = 42;
```

It is not necessary to specify a type in the majority of cases, as the compiler can infer the correct type from the use sites of `x`. Once a type has been inferred, it cannot be changed, so the variable `x` may not be used to store strings or other non-integral data types.

By default, a variable in Rust is immutable. In order to be able to change a variable, it must be declared as `mut`:

```
let mut x = 42;
```

C.2 Constructors

The Rust language has no custom constructors. Instead each type offers a built-in syntax for constructing instances of it. Where a constructor is desirable, the C++ concept "named constructor idiom" should be used. This means that one needs to create a static function that returns an instance of the type. Internally such a static function will use the built-in syntax for creating the value. This is not just a convenience scheme, but allows fields of the type to be private, only permitting the named constructor functions to create values of that type.

The following table depicts the most common constructors as used throughout this thesis.

Type	Constructor
tuple	`()`, `(a,)`, `(a, b)`, …
array	`[]`, `[a]`, `[a, b]`, …
uniform array	`[value; length]`
struct	`Name { field_a: value, field_b: value }`

C.3 Destructuring

Destructuring is the inverse operation of creating objects (see Appendix C.2). For example, in the case of a tuple of type `(i32, f32, &str)` which is created with

```
let x = (42, 5.0, "foo");
```

destructuring is performed by assigning new variables for each field of the tuple:

```
let (the_int, the_float, the_str) = x;
```

What happens in detail becomes more obvious if the tuple has also been constructed with variables instead of literals:

```
let an_int = 42;
let a_float = 5.0;
let a_str = "foo";
let x = (an_int, a_float, a_str); // construct tuple from parts
let (a, b, c) = x; // destruct tuple into its parts
```

`a` is a new variable that contains the same value as `an_int`, `b` has `a_float`'s value and `c` has `a_str`'s value.

It is also possible to destructure a value "partially". This means instead of binding the destructured components to a variable, one can bind just parts of each component.

```
let (a, .., b) = (5, 6, 7, 8);
```

This only extracts the first and last tuple element, ignoring all the others.

C.4 Refutable destructuring

One can use destructuring patterns to only execute some code conditionally:

```
if let (a, 42) = some_tuple {
    // access to first tuple field only
    // if the second tuple field is `42`
}
```

There are more advanced patterns than just literals, e.g the following is valid Rust:

```
if let (a, 5..20) = x {
    // access to first tuple field only if
    // the second field is in the range `[5, 20)`
}
```

Often it is desirable to extract a variable only under certain conditions on the variable itself. In C and C++ a common pattern is to assign a variable inside an `if` condition and check it for non-null:

```
if (ptr = a_function()) {
    // do something with ptr
} else {
    // didn't get a ptr, do a backup operation
}
```

Unfortunately this allows accidentally writing

```
ptr = some_function();
// do something with ptr
```

while forgetting to check for null-ness. In Rust both actions can be combined without compromising safety:

```
if let Some(ptr) = a_function() {
    // do something with ptr
} else {
    // didn't get a ptr, do a backup operation
}
```

In case one tries to directly work with the result of `a_function` one would get a type-mismatch. This results from a possibly-`NULL` pointer having a different type than a guaranteed-not-null pointer. One can convert a guaranteed-not-null pointer to a possibly-`NULL` pointer by reversing the `Some` destructuring shown above and

using it as a constructor (`let x = Some(ptr);`). The other direction requires the conditional destructuring.

Note that possibly-`NULL` pointers are just an instance of the following generic `enum`, where the generic parameter of the guaranteed-not-null pointer can be `Box<SomeType>` for a heap allocation or `&SomeType` for any borrowed value.

```
enum Option<T> {
    Some(T),
    None,
}
```

A possible instance would be `Option<&SomeType>`.

It follows that this conditional destructuring is not special to pointer types, but can be used with any arbitrary object of enum, integer, array, tuple, … type.

C.5 Ownership

This section covers the basic typestate scheme available in Rust today, which consists of

- uninitialized,
- initialized, and
- moved-out-of.

Take the uninitialized variable `s`:

```
let s;
```

Any attempted access to the variable, will result in an error, informing the user that the variable has not yet been initialized.

```
let n = s.len(); // ERROR: `s` has not been initialized
```

The user can initialize the variable at any point

```
s = String::from("foo");
```

after which it may be accessed (note the exact same code that failed to compile above)

```
let n = s.len(); // `n` == 3
```

Assignments in Rust “move out” of the right hand side of the assignment, thus invalidating everything on the right hand side. In the following case, the variable `s` will be uninitialized again, while a new variable `t` is introduced and initialized at the same time.

```
let t = s;
```

Thus, the variable `t` can be accessed, but the variable `s` behaves as if it were never initialized save for the exact wording of the error message:

```
let n = t.len(); // `n` == 3
let n = s.len(); // ERROR: `s` has been moved out of
```

User-defined typestates can be emulated through leveraging the type system and ownership semantics by creating a type for every typestate and passing `self` by value to all typestate-changing methods. An example of the file handle typestate is given below

```
pub struct Error {
    pub error_kind: String,
}
pub struct Open {
    os_file_handle: OsFileHandle,
}
pub fn open(filename: &str) -> Result<Open, Error> { ... }
pub fn create(filename: &str) -> Result<Open, Error> { ... }
impl Open {
    pub fn write(self, text: &str) -> Result<Open, Error> { ... }
    pub fn close(self) -> Result<(), Error> { ... }
}
```

The `Result` type yields either a successful value of the first generic type, or an error value of the second generic type. It is an `enum` provided by the Rust standard library for convenience, but more complex scenarios can be handled by creating a custom `enum` to handle more result-variants than just binary success/failure.

Appendix D

Rustc internals

D.1 AST Items

- an `extern crate $cratename`
- an `use` declaration
- a definition of a `static`
  - containing an expression as its value
- a definition of a `constant`
  - containing an expression as its value
- a definition of a function
  - containing a block expression as its body
- a submodule
  - containing another list of items
- a foreign module
  - containing functions imported from other languages
- a type alias
  - containing a type reference
- an `enum` type definition
  - containing type references in the fields of its variants
- a `struct` type definition
  - containing type references in the fields' type
- a `union` type definition
  - containing type references in the fields' type
- a `trait` definition

    - containing declarations of associated constants, associated types and methods

- a default `trait` implementation (`impl Trait for .. {}`)
- an implementation (of a type or a trait for a type)
    - containing definitions for associated constants, associated types and methods

- an unexpanded macro invocation
    - can expand to zero or many items

D.2 AST Types

- a slice
    - containing a type reference for the element type
- an array
    - containing a type reference for the element type and
    - an expression for the array length
- a raw pointer
    - containing a type reference for the type of the pointed-to value
- a reference
    - containing a type reference for the type of the referenced value
- a function signature
- the never type (a type without values)
- a tuple type
    - containing a list of the tuple fields' type references
- a type name (or a full path to a type)
- a trait object type
- an impl trait type
- a not further specified type (`_`) that should be inferred by the compiler
- the implicit `Self` type of the `self` argument in a method
- a macro that expands to exactly one type

D.3 AST Expressions

- a `box a` expression
    - containing the expression to be stored in a heap allocation
- an `a <- b` inplace assignment expression

- containing the lvalue expression for the assignment target and
- the expression of the value to be stored

- an `a = b` assignment expression
  - containing the lvalue expression for the assignment target and
  - the expression of the value to be stored
- a shorthand assignment and binary operation expression (e.g. `a += 1`)
  - containing the lvalue expression for the assignment target and
  - the expression of the value to apply the operation on
- an array expression
  - containing a list of expressions for the values of the array's elements
- a function call
  - containing an expression resolving the function to be called, and
  - a list of expression for the function arguments
- a method call
  - containing a list of expressions for the function arguments
- a tuple expression
  - a list of expressions for the tuple's fields
- a binary operator expression
  - containing an expression for each of the left and right hand side of the operation
- an unary operator expression
  - containing an expression for the value to apply the unary operation to
- a numeric or string literal
- a cast expression
  - containing an expression for the value to be cast, and
  - a type reference for the target type.
- an if expression
  - containing an expression for the condition,
  - a block expression for the body, and
  - optionally an expression for the else body.
    * it is an expression, because it might contain further `if` conditions to create `else if` chains
- an if let expression
  - containing a condition pattern,
  - a value to check the pattern against,

  - a block expression for the body, and
  - optionally an expression for the else body.
- a while loop expression
  - containing a condition expression, and
  - a block expression for the loop body.
- a while let loop expression
  - containing a condition pattern,
  - a value to check the pattern against, and
  - a block expression for the loop body.
- a for loop expression
  - containing a pattern for the current loop value,
  - an expression for the value to iterate over, and
  - a block expression for the loop body.
- a loop expression
  - containing a block expression for the loop body
- a match expression
  - containing an expression to match against, and
  - a list of matches to try to match
    * the matches consist of a test-pattern,
    * optional match guards, and
    * an expression to evaluate if the match succeeds.
- a closure definition
  - containing the closure body expression
- a block expression
  - containing a list of items,
  - statements, and
  - optionally an expression to evaluate against.
- a field expression (`a.field_name`)
  - containing an expression to evaluate as a struct value, and
  - the name of the field
- a tuple field expression (`a.0`)
  - containing an expression to evaluate as a tuple value, and
  - the index of the field to access
- an index expression (`a[i]`)
  - containing an expression to index into and
  - an expression that evaluates to an index value to be used for indexing

- a range expression
  - containing either zero sub expressions (an infinite range `..`),
  - a start expression for ranges without an end,
  - an end expression for ranges without a start,
  - or both a start and end expression for closed ranges.
- a variable name or a path to a constant/static value
- a referencing operation
  - containing the expression to take the address of
- a `break` expression
  - containing an optional expression to return from the loop
- a `continue` expression
- a `return` expression
  - containing an optional expression to return from the function
- inline assembly
- a macro invocation expanding to an expression
- a struct literal expression
  - containing a list of expressions for the fields of the struct, and
  - optionally an expression of another struct value which is used to fill in all fields that are not specified in this expression
- an array literal filling an array with instances of the same element
  - containing an expression for the value to repeat, and
  - an expression for the length of the array (and thus the number of times the value is repeated).
- a "try" expression (for the `?` error forwarding operator)
  - containing an expression to check for errors

Appendix E

Pointer equality operation during abstract interpretation

The following table depicts the behavior of comparisons of pointers to different kinds of memory. Note that the lower half of the table is empty because comparison is commutative.

	heap	static	const	local	mut local	heap (dead)	local (dead)	fn
heap	`==`	`false`	`false`	`false`	`false`	nondet	`false`	`false`
static		`==`	`false`	`false`	`false`	`false`	`false`	`false`
const			maybe	maybe	`false`	`false`	nondet	`false`
local				maybe	`false`	`false`	nondet	`false`
mut local					`==`	`false`	nondet	`false`
heap (dead)						nondet	`false`	`false`
local (dead)							nondet	`false`
fn								nondet

- `==` means that the result is the comparison of the identity of the allocation
- `false` means that the result will always be `false` because the two can never be equal
- "nondet" means that there exist examples where the comparison does not yield deterministic results and should thus must be handled depending on the use case of the interpreter
- "maybe" means that if the value of the memory differs, the result is `false`, otherwise the comparison is "nondet", because the memory might have been duplicated/deduplicated or not depending on optimization parameters.

Appendix F

Code generation from model

The demo can also be found at github.com/oli-obk/model-demo

```
// lib.rs
extern crate proc_macro;

use crate::proc_macro::TokenStream;
use quote::*;
use std::collections::HashSet;
use std::str::FromStr;

#[proc_macro]
pub fn state_machine(input: TokenStream) -> TokenStream {
    let input = input.to_string();
    let mut lines = input.split(',');
    let type_name = lines
        .next()
        .expect("first line must be the name of the type")
        .trim();
    let type_name = Ident::new(type_name);

    let mut names = Vec::new();
    let mut sources = Vec::new();
    let mut targets = Vec::new();
    let mut states = HashSet::new();

    for line in lines {
        if line.trim().is_empty() {
            continue;
        }
        let mut split = line.split(':');
        names.push(Ident::new(split.next().unwrap().trim()));
```

```
            let mut split2 = split.next().unwrap().split("->");
            assert!(split.next().is_none(), "only one colon is allowed per line");
            let source = split2.next().unwrap().trim();
            sources.push(Ident::new(format!("{}::{}", type_name, source)));
            let target = split2.next().unwrap().trim();
            targets.push(Ident::new(format!("{}::{}", type_name, target)));
            states.insert(target);
            states.insert(source);
            assert!(split2.next().is_none(), "multiple `->` found in one line")
        }

        let states: Vec<_> = states.into_iter().map(Ident::new).collect();

        let gen = quote! {
            enum #type_name {
                #(#states),*
            }
            impl #type_name {
                #(
                    fn #names(&mut self) {
                        match self {
                            #sources => *self = #targets,
                            _ => panic!("invalid transition"),
                        }
                    }
                )*
            }
        };
        TokenStream::from_str(&gen.to_string()).unwrap()
    }

    # Cargo.toml
    [package]
    name = "model-demo"
    version = "0.1.0"
    authors = ["Oliver Scherer <github35764891676564198441@oli-obk.de>"]
    edition = "2018"

    [lib]
    proc-macro = true

    [dependencies]
    syn = "0.14.4"
    quote = "0.3"
```

```rust
// tests/phone.rs
use model_demo::state_machine;

state_machine!{
    Phone,
    begin_incoming_call: HangedUp -> Ringing,
    accept_call: Ringing -> Speaking,
    end_call: Speaking -> HangedUp,
    begin_outgoing_call: HangedUp -> Dialing,
    finished_dialing: Dialing -> Waiting,
    call_accepted: Waiting -> Speaking,
}

#[test]
fn succeed() {
    let mut phone = Phone::HangedUp;
    phone.begin_incoming_call();
    phone.accept_call();
    phone.end_call();
    phone.begin_outgoing_call();
    phone.finished_dialing();
    phone.call_accepted();
    phone.end_call();
}


#[test]
#[should_panic]
fn fail() {
    let mut phone = Phone::HangedUp;
    phone.accept_call();
}
```

Appendix G

Custom driver for project specific static analyses

The declare_lint macro creates the relevant data structures from the lint name and description.

```
declare_lint! {
    // a capitalized version of the lint name
    pub SOME_LINT_NAME,
    // the default diagnostic level (Allow, Warn, Deny, Forbid)
    Forbid,
    // a short message explaining the gist of the lint
    "the interns keep taking shortcuts that bite us later"
}
```

A data structure to hold any additional lint information is created via the declare_lint_pass macro, although it can also be manually created.

```
declare_lint_pass!(SomeLintPass { extra_info: String } => [SOME_LINT_NAME]);
```

The analysis logic and resulting diagnostic message is developed inside an implementation of the EarlyLintPass trait or the LateLintPass trait.

```
impl EarlyLintPass for SomeLintPass {
    fn check_ident(
        &mut self,
        cx: &EarlyContext,
        ident: Ident,
    ) {
        if ident.to_string().contains("foo") {
            cx.span_lint(
                SOME_LINT_NAME,
```

```
                ident.span,
                "Message goes here",
            );
        }
    }
}
```

Finally, the compiler API is invoked and all the new lints are added to the compilation process.

The arguments to the custom driver need to be forwarded to the compiler library call.

```
let args: Vec<_> = std::env::args().collect();
```

The compiler is invoked by creating a default compiler driver

```
let mut compiler = driver::CompileController::basic();
```

and modifying a specific stage in the compiler pipeline. In this case, the analysis is registered immediately after parsing the source file has completed.

```
compiler.after_parse.callback = Box::new(move |state| {
    let mut ls = state.session.lint_store.borrow_mut();
    // insert the new pass
    ls.register_early_pass(None, false, box Pass);
});
```

Now all that is left is to forward the program arguments and the modified compiler driver to the compiler library.

```
rustc_driver::run_compiler(
    &args, Box::new(compiler), None, None,
)
```

In order to run it on a single source file one can invoke it as `your_binary some_source.rs`. It can also be run on entire crates by setting the `RUSTC_WRAPPER` environment variable as the path to the custom driver. Any further calls to `cargo` will now use the custom driver instead of the original rustc.

Appendix H

init function call before foo function call demo

This section contains the full source code of the demo for analyzing a program for mis-use of an API that requires an initialization before further use of the API. For the given API

```
fn foo() {}

fn init() {}
```

It is specified that `init` must be called before `foo`. Instead of just documenting the fact and leaving the correct usage to the user, the API can be enriched with annotations and an analysis can prove that the API is used incorrectly (or in a way that is too complex for the analysis to analyze).

The annotated API looks as follows:

```
#[rustc_diagnostic_item = "foo"]
fn foo() {}

#[rustc_diagnostic_item = "init"]
fn init() {}
```

The example can be replicated by using the nightly Rust compiler

> rustc 1.41.0-nightly (ff15e9670 2019-12-13)

and the clippy static analysis tool at commit d82debbd01847dcc3e11abb9f9f3fb48b70b6845

The following first code segment is generic and does not contain any interesting parts of the static analysis.

```
use crate::utils::span_lint;
use rustc::declare_lint_pass;
use rustc::hir::def_id::DefId;
use rustc::hir::Crate;
use rustc::lint::{
    LateContext, LateLintPass, LintArray, LintPass,
};
use rustc::mir;
use rustc::ty;
use rustc::ty::TyCtxt;
use rustc_data_structures::fx::FxHashSet;
use rustc_index::bit_set::BitSet;
use rustc_index::vec::Idx;
use rustc_mir::dataflow::{
    do_dataflow, BitDenotation, BottomValue,
    DataflowResultsCursor, DebugFormatted, GenKillSet,
};
use rustc_session::declare_tool_lint;
use syntax::source_map::Span;
use syntax_pos::symbol::Symbol;

declare_clippy_lint! {
    /// Checks whether the init/foo API is used correctly
    pub INIT_BEFORE_FOO,
    correctness,
    "must call the `init` function before the `foo` function"
}

declare_lint_pass!(Pass => [INIT_BEFORE_FOO]);

impl<'a, 'tcx> LateLintPass<'a, 'tcx> for Pass {
    #[allow(clippy::too_many_lines)]
    fn check_crate(
        &mut self,
        cx: &LateContext<'a, 'tcx>,
        _: &'tcx Crate,
    ) {
        // Only trigger the lint if this function has a main function
        let check_main = Symbol::intern("check_main");
        if let Some(main_fn) = cx.tcx.get_diagnostic_item(check_main) {
            #[allow(clippy::default_trait_access)]
            let mut call_stack: FxHashSet<DefId> = Default::default();
            let init = check_init(cx.tcx, main_fn, &mut call_stack);
            if let InitState::NeedsInit(span) = init {
```

```
                span_lint(
                    cx,
                    INIT_BEFORE_FOO,
                    span,
                    "call to `foo` not preceded by call to `init`",
                );
            }
        }
    }
}
```

The next code segment contains the main analysis checking for each function whether it (recursively) calls `init`, does not call `init` or in fact requires `init` to have been called already.

```
enum InitState {
    Init,
    NotInit,
    NeedsInit(Vec<Span>),
}

fn check_init(
    tcx: TyCtxt<'_>,
    def_id: DefId,
    call_stack: &mut FxHashSet<DefId>,
) -> InitState {
    // Bail out on recursion
    // (the stack already contains a call to this function)
    if !call_stack.insert(def_id) {
        return InitState::NotInit;
    }
    let result = check_init_inner(tcx, def_id, call_stack);
    call_stack.remove(&def_id);
    result
}
```

The actual logic is implemented in the `check_init_inner` function. We are using two functions here in order to be able to use early returns in the inner function while still doing an operation on the `call_stack` variable afterwards. While this could likely be implemented with a marker type and a destructor, that would be more complex and less obvious than the given solution.

```
fn check_init_inner(
    tcx: TyCtxt<'_>,
```

```
    def_id: DefId,
    call_stack: &mut FxHashSet<DefId>,
) -> InitState {
    // FIXME: functions calling `init` on all
    // code paths should be treated just like `init`.
    if tcx.is_diagnostic_item(Symbol::intern("init"), def_id) {
        return InitState::Init;
    }

    // MIR from other crates may not be available,
    // so we won't be able to detect anything there
    if !tcx.is_mir_available(def_id) {
        return InitState::NotInit;
    }

    // We just want the MIR of the function and
    // don't care about any other information
    let mir = tcx.optimized_mir(def_id);

    let dead_unwinds = BitSet::new_empty(mir.basic_blocks().len());
    let seen_init = do_dataflow(
        tcx,
        mir,
        def_id,
        &[],
        &dead_unwinds,
        SeenInit {
            tcx,
            mir,
            call_stack: call_stack.clone(),
        },
        |_bd, _p| DebugFormatted::new(&"no id"),
    );
    let mut cursor = DataflowResultsCursor::new(seen_init, mir);

    for (block, bbdata) in mir.basic_blocks().iter_enumerated() {
        let terminator = bbdata.terminator();

        let loc = mir::Location {
            block,
            statement_index: bbdata.statements.len(),
        };
        // If init has not been called before reaching
        // this source location, then we must report an
```

```
        // error on all `foo` calls encountered
        cursor.seek(loc);

        let func = match &terminator.kind {
            mir::TerminatorKind::Call { func, .. } => func,
            // We only care about function calls
            _ => continue,
        };
        let callee_id = match func.ty(&**mir, tcx).kind {
            ty::FnDef(def_id, _) => def_id,
            // Function pointer calls aren't implemented
            // in this simple analysis, so we assume
            // any dynamic call to require init to have been called.
            _ => return InitState::NeedsInit(
                vec![terminator.source_info.span],
            ),
        };
        let is_foo = tcx.is_diagnostic_item(
            Symbol::intern("foo"),
            callee_id,
        );
        if !cursor.contains(NoIdx) && is_foo {
            return InitState::NeedsInit(
                vec![terminator.source_info.span],
            );
        }
        let init = check_init(tcx, callee_id, call_stack);
        if let InitState::NeedsInit(mut span) = init {
            span.push(terminator.source_info.span);
            return InitState::NeedsInit(span);
        }
    }
    InitState::NotInit
}
```

The next section contains a dataflow helper data structure that is used to check whether each function call is guaranteed preceded by a call to `init`.

```
/// Determines whether `init` has been
/// called at a specific point in the code
struct SeenInit<'a, 'tcx> {
    mir: &'a mir::Body<'tcx>,
    tcx: TyCtxt<'tcx>,
    call_stack: FxHashSet<DefId>,
```

```
}

#[derive(
    Copy, Clone, Debug, Eq,
    PartialEq, Hash, Ord, PartialOrd,
)]
/// Only one bit per block, so we need
/// no information stored in the bit index.
struct NoIdx;

impl Idx for NoIdx {
    fn index(self) -> usize {
        0
    }
    fn new(i: usize) -> Self {
        assert_eq!(i, 0);
        NoIdx
    }
}
```

A dataflow analysis is implemented that informs the main analysis whether `init` has been guaranteed to be called at any particular point in the program. This is achieved by marking each BasicBlock as having seen `init` if and only if all predecessor blocks have seen `init`. This means that

```
if some_condition {
    init();
    // init has been called here
}
// init may not have been called here
```

The following code snippet contains the implementation of this analysis.

```
impl<'a, 'tcx> BitDenotation<'tcx> for SeenInit<'a, 'tcx> {
    type Idx = NoIdx;
    fn name() -> &'static str {
        "seen init"
    }

    fn bits_per_block(&self) -> usize {
        1
    }

    fn start_block_effect(&self, on_entry: &mut BitSet<NoIdx>) {
```

```
        on_entry.clear();
    }

    fn statement_effect(
        &self,
        _trans: &mut GenKillSet<NoIdx>,
        _loc: mir::Location,
    ) {}

    fn terminator_effect(
        &self,
        trans: &mut GenKillSet<NoIdx>,
        loc: mir::Location,
    ) {
        let func = match &self.mir[loc.block].terminator().kind {
            mir::TerminatorKind::Call { func, .. } => func,
            // We only care about function calls
            _ => return,
        };
        let callee_id = match func.ty(self.mir, self.tcx).kind {
            ty::FnDef(id, _) => id,
            // Function pointer calls aren't implemented
            // in this simple analyses, so we assume
            // any dynamic call to require init to have been called.
            _ => return,
        };
        let init = check_init(
            self.tcx,
            callee_id,
            &mut self.call_stack.clone(),
        );
        if let InitState::Init = init {
            trans.gen(NoIdx);
        }
    }

    fn propagate_call_return(
        &self,
        _in_out: &mut BitSet<NoIdx>,
        _call_bb: mir::BasicBlock,
        _dest_bb: mir::BasicBlock,
        _dest_place: &mir::Place<'tcx>,
    ) {
        // Nothing to do when a call returns successfully
```

```
    }
}

impl<'a, 'tcx> BottomValue for SeenInit<'a, 'tcx> {
    /// bottom = not seen
    const BOTTOM_VALUE: bool = true;

    // Return true if `inout_set` changed
    fn join<T: Idx>(
        &self,
        inout_set: &mut BitSet<T>,
        in_set: &BitSet<T>,
    ) -> bool {
        // This is the opposite of what normal dataflow does.
        // Normal dataflow is `true` if *any*
        // predecessor block is `true`.
        // We want to be `true` only if *all*
        // predecessor blocks are `true`
        inout_set.intersect(in_set)
    }
}
```

Appendix I

Survey about the ease of tool integration

Survey results of a survey asking tool users what generic usability problems affect them the most [166].

	No influence	Open issue	Depends on Tool Importance	Won't use	Respondents
Frequent work needed to keep the tool working	2%	7%	41%	50%	54
Frequent adjustments to configuration needed	16%	18%	43%	23%	51
First time configuration required	55%	15%	30%	0%	54
Dependencies that must be manually installed	19%	9%	63%	9%	54
Manual setup required	17%	7%	72%	4%	54

Appendix J

Full list of related publications

J.1 Peer reviewed conferences and journals

- Zuverlässige und sichere Software offener Automatisierungssysteme der Zukunft, Keller, Schneider, Matthes, Hagenmeyer (2016)
  - at-Automatisierungstechnik 64, Seiten 930-947
- Integrierte Entwicklung zuverlässiger Software, Schneider, Keller (2016)
  - Softwaretechnik-Trends 36
- Programmiersprachen und Konzepte zur Entwicklung zuverlässiger und sicherer Automotive Software, Schneider (2017)
  - VDI-Berichte 2310
- Reaching const evaluation singularity, Schneider (2018)
  - FOSDEM 2018
- Learn to stop fixing bugs, Schneider (2016)
  - Rust Belt Rust Pittsburgh

J.2 Language design and standardization (peer reviewed white papers)

- RFC 1229 warn on statically known erroneous code, Schneider (2015)
  - https://github.com/rust-lang/rfcs/pull/1229
- RFC 2043 abstraction of non-determinism, Schneider (2018)
  - https://github.com/rust-lang/rfcs/pull/2043
- RFC 2341 variables in constant evaluation, Schneider (2017)
  - https://github.com/rust-lang/rfcs/pull/2341

- RFC 2342 control flow in constant evaluation, Schneider (2018)
  - https://github.com/rust-lang/rfcs/pull/2342
- RFC 2344 loops in constant evaluation, Schneider (2018)
  - https://github.com/rust-lang/rfcs/pull/2344
- RFC 2345 user controlled abortion of constant evaluation, Schneider (2018)
  - https://github.com/rust-lang/rfcs/pull/2345
- RFC 2476 stabilize static analyzer for Rust, Goregaokar, Schneider (2018)
  - https://github.com/rust-lang/rfcs/pull/2476
  - Now used in various prominent projects, e.g.
    * fuchsia (Google),
    * libra (Facebook, Lyft, Spotify, Uber and others),
    * azure pipelines (Microsoft).

J.3 Peer reviewed open source

Only significant contributions are listed here. Contributions that solely took implementation work without relevant research work are left out.

- Implementation of compile-time function calls for the Rust language (2015)
  - https://github.com/rust-lang/rust/pull/26848
- Project specific static analyses (2016)
  - https://github.com/rust-lang/rust-clippy/pull/1093
- Automatically resolvable static analysis results (2017)
  - https://github.com/rust-lang/rust/pull/43929
  - https://github.com/rust-lang/rust/pull/46052
  - https://github.com/rust-lang/rust/pull/41876
- Integration of a virtual machine for Rust MIR into the Rust compiler (2018)
  - https://github.com/rust-lang/rust/pull/46882
  - Used for all constant evaluation
- Analysis and prototype of existential types for the Rust language (2018)
  - https://github.com/rust-lang/rust/pull/52024
  - Base feature enabling `async-await` support in the Rust language
- Stabilization of minimal compile-time function call feature for the Rust language (2018)
  - https://github.com/rust-lang/rust/pull/54835