Automated Fault Tree Generation for Rust Programs

Dominic Zimmer
Saarland University
Saarland Informatics Campus
dominic.zimmer@cs.uni-saarland.de

Andreas Schmidt
Saarland University
Saarland Informatics Campus
andreas.schmidt@cs.uni-saarland.de

***Abstract*—With software driving more and more of our world, dependability assurance becomes increasingly essential. While *safe-by-design* is desirable, there are always failure conditions that cannot be avoided and must be handled (or deemed acceptable in review). However, this requires awareness and explicitness of failure conditions as well as their impact on the overall system. Fault trees are commonly used in dependability assurance, up to the point that their creation and properties are mandated by safety standards. As of today, these documents are created manually by safety engineers—a process that lacks traceability between the software source code and associated fault trees.**

Recent advances in programming languages have led to stronger type systems and more modern ways to express failure conditions. One example is the Rust language, which uses monadic types to make potential failures explicit. In this paper, we present our tool `craft`, which uses static program analysis to generate fault trees of functions for a subset of the Rust language. We showcase both capabilities as well as limitations of our approach and give directions for future work.

***Index Terms*—fault trees, qualitative, dependability, Rust**

I. Introduction & Background

We investigate how traditional formal models for safety-critical systems can be automatically generated for software systems written in modern languages.

A. Fault Trees

A widely established toolkit to analyze safety-critical systems are fault trees [1], [2]. Fault trees relate the failure of a system to the failure of its individual components in terms of boolean connectives, all the way down to so-called basic events. Basic events typically describe individual *faults*, that is failures of small components of a system in terms of a Poisson distribution that captures the expected time-to-fail of the component. Used by the largest organizations for safety-critical systems, such as NASA [3], fault trees enjoy support by numerous analysis tools [2] and serve as a fundamental formalism to express different dependability aspects of critical systems [4], [5].

Typically, fault trees are constructed automatically [6], [7] from system models or generated from failure reports obtained by profiling a safety-critical system. In systems where failure is catastrophic, it can be difficult or unreasonable to obtain even approximates for the desired failure conditions. Manually describing a system's structure as a fault tree, however, is prone to human errors introduced when modelling the system.

While fault trees have been widely (and successfully) employed for hardware, the focus of our work are software programs. This implies that—unless an abstract model of software systems [8] is considered—we are mostly interested in a system's qualitative failure structure, rather than in quantitative measures that are near impossible to obtain for software [9]. To the best of our knowledge, there are no static analysis tools available that automatically generate fault trees for software from their source code. We want to address this lack of tooling and framework and shine light into the widely unexplored corner of *automatically generated program fault trees* and *qualitative analysis* thereof.

B. Dependable Software and Rust

While our approach is applicable to many modern programming languages, the focus of our work is on Rust. With both academical research (e.g. RustBelt [10]) as well as industrial activities (e.g. Ferrocene[1]), Rust is foreseen to become an addition to the limited set of language ecosystems suitable for development of dependable and efficient software. Besides the guarantees provided by Rust's unique ownership system, it serves as a modern, high-level language that is starting to get more support every day, e.g. in the Linux kernel.[2]

Rust's failure (or exception) handling is different to most imperative programming languages used in critical software (e.g. C). These languages express failure implicitly by functions throwing exceptions (or returning well-known values) of which a calling function may decide to catch *some*—whilst rethrowing others. Rust makes the possibility of failure explicit in its type system by widely supporting `Result<T, E>` types. In Rust, a Result is either a value `Ok(t: T)` or an error `Err(e: E)`—both being variants of the same enumeration. The explicit usage of `Result`s in the type system make Rust more explicit and verbose with respect to the fallibility of individual components: A developer writing a function `f` that calls a fallible function `g` (i.e. a function that returns a `Result`) must explicitly make a decision what to do with the result of `g` in both cases of `Ok(v)` and `Err(e)`. If desirable, the `#[must_use]` attribute can be used to issue a diagnostic compiler warning in case a result is ignored,

[1] https://github.com/ferrocene
[2] https://lwn.net/Articles/908347/

```rust
enum Option<T> {
    Some(T),
    None,
}
```

```rust
enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

Fig. 1. Recoverable failures are typically represented using either of the two standard library types `Option` or `Result`.

encouraging the developers to make an explicit decision what to do with the result.

C. Contribution

The contributions of this paper are threefold:

1) We propose a *qualitative failure abstraction* for a subset of Rust programs.
2) We implement this abstraction in a prototypical tool that automatically extracts *fault trees* from Rust code.
3) We discuss *future work*, particularly outlining the idea of future extensions to our fault trees and analyses.

II. Concept

A. Types of Failures

Our tool `craft` is aimed at analyzing when Rust programs can *recoverably fail*.[3] Recoverable failure in Rust is typically[4] encoded in the return type, using either of the two container types `Option<T>` or `Result<T,E>` defined in Fig. 1. We say that a variable of type `Option<T>` or `Result<T,E>` *fails*, if it is constructed using the `None` or `Err(e)` constructor, respectively. Thereby, we only discriminate failure from non-failure, ignoring possible information about the fault that the `Err(e)` constructor might bear. Consequently, for the purpose of this work, we will from now on treat values of type `Option<T>` and `Result<T, ?>` equally and without loss of generality talk about `Option<T>` and its constructors `Some(t: T)` and `None`, assuming that the same conceptual work would apply to `Result<T, ?>`.

B. Failures and Fault Trees

In order to qualitatively analyze the failure (i.e. the return of a non-`Some` value) of a single component (i.e. a program), we relate its failure to the failures of subordinate components of the program using static fault trees. Fault trees are trees bearing *(failure) events* as leaf nodes, and boolean connectives as intermediate nodes. We use green circles as leaf nodes to depict *basic events* that might fail. Beige boxes are used to insert names of *intermediate events* anywhere in the tree, allowing us to easily refer to subtrees. Blue triangles at root (leaf) position of a fault tree depict transfer-out (-in) gates, allowing composition of fault trees along transfer-in and -out gates. The root node of a fault tree describes the failure of the entire component as a boolean connective consisting of basic events, transfer gates, $\wedge$, $\vee$, and $\neg$. By supporting negations of events, our fault trees typically belong to the fragment of *non-coherent* fault trees [11].

[3] *Recoverable failure* refers to the program execution being able to recover from individual failures, not that the failure could be un-done

[4] Mutably borrowed Result arguments are possible, but uncommon.

```rust
fn read_int() -> Option<i32> { ... }

fn read_int_or_default(x: Option<i32>)
     -> Option<i32> {
    read_int().or(Some(0))
}
```

Fig. 2. Failure of the function `fn read_int_or_default()` is related to the failure of the callee `read_int()`.

In the scope of static code analysis for Rust, basic events are expressions and functions whose type (or return type) are fallible data types. We want to express the failure of a function in terms of (a) failure of its arguments, (b) failure of nested function calls, and in terms of (c) failure of expressions.

III. Automated Fault Tree Extraction

A. Motivational Example

Using the example given in Fig. 2, we motivate the approach of our tool by first manually and intuitively analyzing the system. The function `read_int_or_default()` uses the function `Option::or` to combine the results of `read_int()` and `Some(0)` into one value. `Option::or` only fails, if both of its arguments fail, in this case the results of `read_int()` and `Some(0)`. We can express this observation in the language of fault trees as shown in Fig. 3. Obviously, subsequent fault tree analysis will show that the component `fn read_int_or_default()` cannot fail.

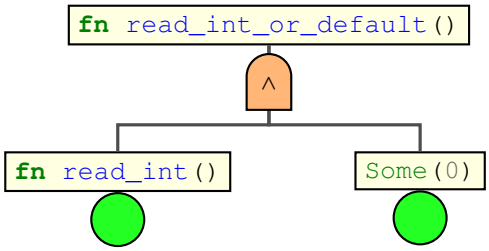

Fig. 3. The fault tree expressing the possible failures of `fn read_int_or_default()` from Fig. 2 in terms of failure of its function calls to `fn read_int()` and `Some(0)`.

Using the same intuition used to derive the fault tree in Fig. 3, we will develop a failure abstraction for a subset of Rust programs, allowing us to automatically generate fault trees from Rust code, formalizing how a function can fail.

B. Failure Abstraction

We implement static analysis of Rust programs using two functions defined on the set of Rust expressions $Expr$. Let $\mathcal{F} : Expr \to FT$ be the *failure function* that assigns an expression e a fault tree $\mathcal{F}(e)$ overapproximating the expression's failure. Furthermore let $\mathcal{T} : Expr \to Ty$ be the function that assigns every expression e its type $t \in Ty$, as it would be inferred by the Rust compiler.

We now briefly discuss the fault tree abstraction given to the most interesting Rust language constructs covered:

Expressions Since we are only concerned with *fallible* expressions, we define the failure of any non-fallible expression

e (that is $\mathcal{T}(e) \neq$ `Option<T>`, for some `T`) as $\mathcal{F}(e) =$ false. Furthermore, in the absence of symbolic analysis techniques, and if none of the following rules yield further insight, we overapproximate an expression's failure to be $\mathcal{F}(e) =$ true.

Functions The most fundamental building blocks of the fault trees we generate are fallible functions, i.e. functions returning `Option<T>`. We want to relate the failure of a function call $f(a_1, \ldots, a_m)$ to the failure of its arguments $a_1, \ldots, a_m$ and the structure of the function itself. In the resulting fault trees, we represent fallible function arguments by transfer-in gates.

We overapproximate the failure of a function by the fact that, for the function application to fail, some return location of the function must have failed. Without extensive control flow analysis, we cannot statically decide which **`return`** statement caused the function to fail, so we approximate the failure $\mathcal{F}(f) = \bigvee_i \mathcal{F}(r_i)$ where $r_1, \ldots, r_n$ are the **`return`** and tail expressions of the function body.

Function applications are either abstracted as a) instantiations of a fault tree by substituting its transfer-in gates by the roots of the arguments' fault trees, or as b) basic events that are to be included as part of the final analysis, e.g. **`fn`** `foo()` in Fig. 5.

If / Match Expressions As for control flow within functions, we cannot generally, statically infer, but must overapproximate the failure of **`if`** and **`match`** expressions. Let $e_1, \ldots, e_n$ be the corresponding branches of the **`if`**-expression or match arms of the **`match`**-expression. For such an expression e, we approximate its failure as $\mathcal{F}(e) = \bigvee_i \mathcal{F}(e_i)$.

In the case of a match expression $m =$ **`match`** `x {...}`, where the matched variable `x` is of fallible type, we obtain a more granular analysis: Let e_S and e_F be the match arms corresponding to the success and failure constructors of `x`, respectively. We can then define $\mathcal{F}(m) = (\mathcal{F}(x) \rightarrow \mathcal{F}(e_F)) \wedge (\neg\mathcal{F}(x) \rightarrow \mathcal{F}(e_S))$, capturing the essence of the if-then-else operator.

Well-Known Functions Instead of conducting a repetitive fault tree analysis for all encountered functions, `craft` uses a lookup table of well-known functions, including many member functions of the namespace `std::option::Option` of the Rust standard library. For example, we can identify `std::option::Option::map(o, •)` with the identity function on `o`, as `map` preserves the `Some`-ness and `None`-ness of any `Option` value. Similarly, the family of combinators covered by the standard library lets us identify `std::option::Option::or` with logical conjunction on fault trees. Most notably, we associate the `Option<T>` constructors `Some` and `None` with false and true, respectively.

C. Example

Let the function **`fn`** `foo() -> Option<`**`i32`**`>` and variable `x: Option<`**`i32`**`>` denote basic events, in relation to which we want to express the failure of the function **`fn`** `combine()` depicted in Fig. 4. **`fn`** `combine()` has a single return location: the immediate tail expression `or(x, map(foo(), |s| s + 1))`. The functions

```
fn combine(x: Option<i32>)
           -> Option<i32> {
  x.or(
    foo().map(|s| s + 1)
  )
}
```

Fig. 4. A small rust program, in which the failure of **`fn`** `combine()` is to be related to its argument `x` and subordinate function call to **`fn`** `foo()`

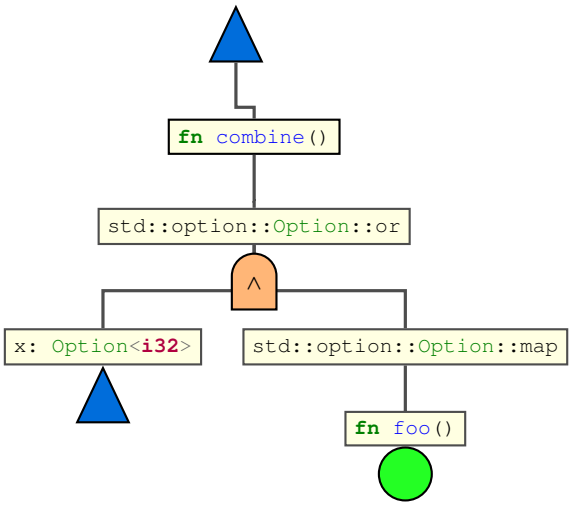

Fig. 5. The fault tree expressing the possible failures of **`fn`** `combine()` from Fig. 4 in terms of failure of its argument `x` and the call to **`fn`** `foo()`.

`or` and `map` can be statically identified to be methods of the `std::option::Option` enum and are thus subject to simplifications of well-known functions. As shown in the fault tree in Fig. 5, we can express the failure of **`fn`** `combine()` in terms of the failure of its argument `x` and the failure of **`fn`** `foo()`. The intermediate events `std::option::Option::or` and `std::option::Option::map` show the partial compilation results from individual function calls to subordinate fault trees. Using the blue transfer-in and -out gates, we can connect fault trees resulting from compilation of subexpressions to existing fault trees, linking function composition in Rust to composition of fault trees.

D. Implementation

The foundation of our static analysis tool is `rust-analyzer` [12], the currently best maintained compiler frontend for Rust, which provides a stable API and tools from syntax parsing to type checking. Using `rust-analyzer` to obtain the program's abstract syntax tree, we structurally generate fault trees during traversal.

IV. Discussion

Authorities concerned with safety, e.g. TÜV [13], suggest fault trees to assess when a system is expected to fail. The literature is abundant with fault trees describing the risks of physical systems, though our literature research showed that fault trees have been largely undiscussed for program

failures. Outlining how fault trees can be used to express the *recoverable failures* of individual functions of a code base, we laid the cornerstone towards a novel framework allowing safety assessment of software systems by the developer.

Contrary to the *recoverable* failures expressed as elements of the type `Result<T, E>`, Rust also features *irrecoverable* failures: `panic!`s. A thread `panic!`s, signalling it has run into an unintended state from which recovery is impossible, initiating termination. Contrary to recoverable failures, numerous tools to detect irrecoverable failures are readily available, e.g. `rustig!`[5] and `no-panic`.[6]

A. Limitations

As the aim of our analysis is the detection of recoverable failures, we assume the programs analyzed by `craft` to be `panic!`-free. Moreover, the language fragment covered by our prototypical tool is restricted to **`macro`** and **`loop`** free Rust programs.

As of now, our approach is limited by the fact that the fault trees we generate significantly overapproximate the failure of function **`return`**s and **`if`** expressions. To refine their abstractions, control flow analysis needs to be integrated into our tool, enabling more insight into the interplay of a function's failure and its arguments' failures.

B. Future Work

As discussed previously, we want to express failure of functions returning `Option<T>` in terms of their arguments. When discussing well-known functions, we have seen that to a degree, this idea is applicable to higher-order functions such as `map(self: Option<T>, f: F) -> Option<U>` whose type signature expects an infallible function `F: FnOnce(T) -> U`. Hence, we could express the failure of `map` in terms of its lone fallible, first argument. To generate fault trees for higher-order fallible functions whose arguments are themselves fallible functions (e.g. `and_then` only differs in `F: FnOnce(T) -> Option<U>` from `map`), a notion of higher-order fault trees supporting function application needs to be developed. As a result, more higher-order function failures could be analyzed.

Moreover, once failure analysis of a function has been conducted, it needs not be computed again until the function changed. As soon as this project is mature enough, we plan to store conducted failure analyses in a central, versionized platform, for example in ClearlyDefined.[7] By granting easy access to failure analysis results, we can facilitate the development of safe software, generating demand for tooling, such as IDE support for safety analyses.

Through means of traditional program analysis techniques, we plan to include approximative analysis of **`loop`**s in `craft`. Furthermore, using symbolic execution we intend to provide more explicit failure conditions, such as `rhs == 0` in the case of the *fallible* safe division function `fn checked_div(self: i32, rhs: i32)`.

[5]https://github.com/Technolution/rustig
[6]https://github.com/dtolnay/no-panic
[7]https://clearlydefined.io/

V. Conclusion

In this work, we present an automated procedure to generate fault trees from Rust source code. Through qualitative analysis of non-coherent fault trees, explicit failure conditions for the failing of Rust functions can be obtained. We implement our approach in a prototypical tool written in Rust, employing the established `rust-analyzer` library.

Future work is needed to address current issues regarding the embedding of control flow into our model.

Acknowledgment

This research is supported by the German Research Foundation (DFG) grant 389792660 as part of TRR 248 – CPEC (see https://perspicuous-computing.science). Additionally, it is supported as part of STORM_SAFE, an Interreg project supported by the North Sea Programme of the European Regional Development Fund of the European Union.

References

[1] S. Kabir, "An overview of fault tree analysis and its application in model based dependability analysis," *Expert Syst. Appl.*, vol. 77, pp. 114–135, 2017. [Online]. Available: https://doi.org/10.1016/j.eswa.2017.01.058

[2] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Computer Science Review*, vol. 15-16, pp. 29–62, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013715000027

[3] P. N. Scientific and T. Information, "Fault tree analysis - a bibliography," 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:59640969

[4] K. J. Sullivan, J. B. Dugan, and D. Coppit, "The galileo fault tree analysis tool," *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pp. 232–235, 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:15870778

[5] W. Vesely, *Fault Tree Handbook*, ser. NUREG-0492. Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, 1981. [Online]. Available: https://books.google.de/books?id=JXX9vgEACAAJ

[6] C. E. Dickerson, R. Roslan, and S. Ji, "A formal transformation method for automated fault tree generation from a UML activity model," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 1219–1236, 2018. [Online]. Available: https://doi.org/10.1109/TR.2018.2849013

[7] F. Mhenni, N. Nguyen, and J.-Y. Choley, "Automatic Fault Tree Generation from SysML System Models," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, Jul. 2014.

[8] J. Parri, S. Sampietro, and E. Vicario, "FaultFlow: a tool supporting an MDE approach for Timed Failure Logic Analysis," in *17th European Dependable Computing Conference, EDCC 2021, Munich, Germany, September 13-16, 2021*. IEEE, 2021, pp. 25–32. [Online]. Available: https://doi.org/10.1109/EDCC53658.2021.00011

[9] N. G. Leveson, *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995.

[10] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, 2018. [Online]. Available: https://doi.org/10.1145/3158154

[11] J. D. Andrews, "The use of not logic in fault tree analysis," *Quality and Reliability Engineering International*, vol. 17, no. 3, pp. 143–150, 2001.

[12] "rust-analyzer," https://rust-analyzer.github.io/, accessed: 2023-09-19.

[13] "Functional safety of electrical/electronic/programmable electronic safety-related systems - part 1: General requirements," International Electrotechnical Commission, Geneva, CH, Standard ISO/IEC TR 61508-1:2010, 2010. [Online]. Available: https://www.iec.ch/functional-safety