Universität Rostock

Traditio et Innovatio

Masterarbeit

Eine Petrinetzsemantik für Rust

Vorgelegt von:

Tom Meyer

Matrikel-Nr.: 8200839

Eingereicht am:

datum

Betreuer:

Prof. Dr. Karsten Wolf

UNIVERSITÄT ROSTOCK | FAKULTÄT FÜR INFORMATIK UND ELEKTROTECHNIK

Lehrstuhl für Theoretische Informatik

Albert-Einstein-Straße 22, 18059 Rostock | Tel: +49 381 498 7641

Titel

Titel english

Betreuer: Prof. Dr. Karsten Wolf

Tag der Ausgabe:
Tag der Abgabe:

Contents

1 Introduction

1.1 Motivation

High quality work is the desire of a professional; Consequently, high quality software is the desire of a professional software engineer.

However, achieving high quality is an ambiguous task, since it is rather context dependent. There are some methods though, that address common issues, like bug reduction or code readability. A noneexhaustive list might be:

- make use of programming patterns
- establish a clear development cycle
- use a bug tracker
- write tests
- make use of static analysis
- make use of model checking

The last two are particularly interesting because they are executed on the source code itself. No human error possible.

Static analysis is typically part of programming languages. One popular invariant this approach guarantees, is type safety. But some recent programming languages like Rust[] make use of additional invariants. In the case of Rust, the ownership model is the most notable additional concept. With strict mutability and aliasing rules, the ownership model avoids a variety of memory errors. The price is a restricted expressiveness, that manifests in compiler warnings that are unknown to other languages.

Model checking is typically not a feature of programming languages, as it requires a mapping to a formalism. But if such a mapping exists, desireable properties can be proven mathematically for all possible runs of the program. A formalism that is especially suited to analyse concurrent flow of execution is Petri-nets.

In this work we will try to find a mapping between rust and Petri-nets to analyse properties of concurrency in rust programs. We will check for deadlocks as an example property in small programs to prove the concept.

1.2 Outline

Lets see what we can expect in the following chapters.

Before we can dive into the specifics we need to establish the basics. In the background section we will first take a closer look on Rust. Next, we look at the challenges of parallel programming and why they are important. This will also cover the relationship of rust with concurrency. And the last significant topic we have to establish is Model Checking.

After we understand those principles, we can move on to the approach taken. We will again start with the specifics of Rust. How can we use generic code to translate that into something we need, and what language concepts do we need to understand for translation. We will see that the rust language has an intermediate representation that simplifies our situation. The most important part is probably the translation of the language features to a petri net. It will contain a detailed description of our mapping and its limits. The generated petri net has to be analysed by a Model Checker. How this is done und on which test programs we will discuss next.

Finally wie will discuss the results and infer future work from the implementations shortcomings.

```
Und nun ein toller Satz zur Überleitung.
```

2 Background and related work

2.1 Rust

```
- rust empowers bla bla \cite{}
- low level with high level abstractions
- small runtime
- no garbage collection
- controlled memory access
  - reduced expressiveness in exchange for increased memory safety
  - features to tackle memory safety issues
  - borrow checker
  - no data races etc
- fast vs safe vs compile time effort \cite{}
- deadlocks are considered safe in rust terms \cite{}
```

2.2 Compilers

The goal of this work is to combine the benefits of the Rust ownership system with the benefits of Petri-Net model checking.

To achieve this goal we have to translate from Rust to Petri-Nets, and we want to do it programmatically. This is basically the definition of a compiler[1, Chapter 1.1]:

> "Simply stated, a compiler is a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language;"

Compilers underwent heavy research and development in the past. Nowadays the structure of a compiler can be summarized into well defined phases[1, Chapter 1.2]:

1. During the **Lexical Analysis**, the character stream of a source file is converted into a token stream. Tokens are all significant language components like keywords, identifiers and symbols ('=', '+', '{', etc.).

2. During **Syntax Analysis** (parsing) the token stream is structured into a tree, typically a syntax tree, where each node represents an operation with its children as operation arguments.

3. The following **Semantic Analysis** checks that the syntax actually matches the requirements; The grammar that the language is based on.
   Additional static analysis – like type checking – is done in this phase as well.

4. Further representations might be produced in the **Intermediate Code Generation** phase. An intermediate representation can be everything that helps. A low level representation that is close to machine code is a common case. Examples are Java Bytecode or the LLVM intermediate representation

5. The intermediate representation can be used for further analysis and optimization in the **Code Optimization** phase. Executable size or execution speed might be improved here. Multiple intermediate representations might be generated and optimized before entering the final phase.

6. **Code Generation**; Which generates another representation. The only difference is that it is the final one – the target representation. Thus it often produces executable machine code.

These phases should clarify the general concept of a compiler but in practice phases might be less distinct. They can blend together and some can be skipped entirely. In the end however, we have a mapping from the source representation to the target representation.

2.3 Parallel Programs

```
- the problem with parallel programs
  - need to communicate data
    - messages
    - shared memory
  - data needs to be
    - consistent
    - synchronized (wait for each other)
    - up to date
  - deadlocks can easily be introduced with
    synchronisation flaws \cite{}
- what are deadlocks
  - synchronisation
  - mutex/semaphore
  - threads
  - dining philosophers
- rust and parallel programs
- how can deadlocks be introduced in rust
  - rust and deadlocks -> considered safe code
```

2.4 Model Checking

```
- tests vs Model Checking
  - tests
    - only work for specific cases\cite{}
    - done by program execution
  - Model Checking
    - works in the general case\cite{}
    - done by program analysis
- different approaches (BDDs etc.)
- petri nets!!
```

2.4.1 Petri-Nets

-

2.4.2 CTL*

-

```
  - other verification implementations
    - (verification by language?)
      - functional programming invariants?
      - prolog invariants?
      - languages with verification methods in its design?
    - c verification
      - valgrind?
    - rust verification
  - petri net verification
    - bpel
```

3 Approach

After we learned the underlying concepts, we can go on with the actual task.

In this chapter it is shown how the initial rust code is translated and analysed. We will see that the rust compiler has interfaces that we can use to reduce our effort, and that it creates an intermediate representation that fits our needs nicely. We will work out a translation for the elements of this intermediate representation, and discuss the format that we will translate into. Finally we will discuss how we can use a model checker to find deadlocks in some simple test programs.

3.1 Rust Compiler

- what is a compiler
- where can rustc be found
- other rustc information
    - maintaining structure
    - development process
    - release strategie
    - main releases
    - cargo integration

- steps
    - Parsing input
    - Name resolution, macro expansion, and configuration
    - Lowering to HIR
    - Type-checking and subsequent analyses
    - Lowering to MIR and post-processing
    - Translation to LLVM and LLVM optimizations
    - Linking

3.2 Interception strategie

- try to use as many language features as possible
- with least possible effort
- after name resolution
- after code generation

```
    - avoids complicated language features
- after borrow checking
    - increases the assumptions we can make
    - reduces cases we have to consider
- after static optimization
    - constant propagation
- before loosing rust features
- before being machine dependent
-> seemingly best solution: mir
```

3.3 Mid-level Intermediate Representation (MIR)

```
- control flow graph

other stuff from rustc book

      - how to traverse
      - single elements
```

3.4 Petri Net Representation

```
    - petri net formalism and format (high level? edge descriptions)
```

3.5 Translation

```
- entry point
    - using main
    - ignoring pre main
        - os specific
        - independent from program semantics
        - not all mir available
- elements
    - programm
        - starts
        - ends
    - memory
        - constants
        - statics
        - places
        - locals
```

```
    - function
        - no translation memory (multiple calls = multiple translations)
        - stack like structure
        - no recursion possible with this approach
        - can panic
        - can diverge
        - can be empty (in theorie)
    - basic blocks
    - statement
        - rvalues and operands
        - assign
        - etc
    - terminator
        - goto
        - switch int
        - call
        - etc
- panic handling
    - do not catch by other threads
    - distinguish between successful and unsuccessful termination
- which parts can or must be excluded from translation
    - std implementations
    - part between std::mutex and pthread mutex
- emulating features
    - missing mir parts
    - external libraries
    - intrinsics and platform specific behavior
    - mutexes
        - high level and low level interfaces
        - choosing high level
            - harder model checking for more general Mutexes
            - os independent
            - less elements need to be translated
                - smaller net
                - worse mapping
        - need to track associated locals

- joining the translations
    - joining basic elements
    - joining basic blocks
    - joining function calls
    - recursion limits
```

3.6 Model Checking

- many tools
- pnml as common representation language
    - finding tools on standard site
- lola integration (format of a petri net)
  - own format
- CTL formula
    - expected deadlocks (program end)
    - difference between deadlock in petri net and deadlock in program
        - try to stay analogous
    - difference between p = 0 and EF(p = 0)
    - try to force witness paths

3.7 Test Programs

- stay simple to prove concept
    - empty program
    - programs which cover the basic language features
    - program with a deadlock
    - similar program without a deadlock

3.8 Debugging

- dot file for small programms
- finding unconnected nodes
    - possible bug found
    - marking live places as result
    - uninitialized places still have to be marked
      (or removed entirely)
- witness path
    - reducing the complicated net to witness path nodes
      and its neighbors

4 Result

- expected deadlocks (expected program termination)
- fitting formulas
  - exkurs in temporal logic
- structure of results
- examples
  - minimal deadlock
  - fixed minimal deadlock
  - deadlock with multiple threads
    - dining philosophers?
  - data dependent deadlock
  - maybe a real world example
    - some distributed system?
      - mqtt?
      - multiagent system?

4.1 Analysis

without the use of some data even most flow releated properties are not discoverabl
this makes emulation necessary
a new local <-> memory model makes sense
makes emulation easier but does not resolve the problem
high lvl nets might resolve this issue at an unknown performance cost

4.2 Translation

- how easy/mapping quality
    - flow is easy
    - data is difficult
        - data is moved between many locals
        - moving quickly masks semantic e.g. for mutexes
- rusts borrow checking rules can probably be exploited better
- foreign functions cannot be translated and must be emulated
    - special emulation makes sense for some of them

- especially for flow relevant functions like threads and mutexes

4.3 Verification

- data representation
- connectiveness of the graph
- possibility of panics can be detected
  - little usefulness in complex programs
- simple deadlocks can already be detected
  - finding a correct formula is not trivial
    - panic paths mask normal execution deadlocks
    - splitting paths can mask deadlocks from single paths
- more sophisticated approach is necessary

5 Conclusion

What was the goal:
- find a mapping between rust and petri-nets
- find deadlocks in rust programs

What was done:
- find a fitting program representation to translate (mir)
- develop a set of rules to translate from representation to formalism
- develop rules for interfacing program logic (non-mir)
- alter mir representations connected to deadlocks (mutexes)
  for flow analysis
- check the results with simple tests programs

What was found:
- some kinds of deadlocks can be found
- complex deadlocks need further improvement
- data representation can be improved

6 Future Work

6.1 Verification

- deadlocks
    - introduction of multi threading
        - paths have to join or
        - paths need own termination places
    - what cannot be detected
- mir generation tests
    - unconnected nodes
    - flow analysis
- automatic evaluation of wittiness path
- high lvl Verification
    - requires other formulas

6.2 Translation optimization

- track moved variables
    - refine the concept of locals and memory
- high lvl nets
- map the source to the nodes similar to the mir approach
- analyse complex programs

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” Addison wesley, vol. 7, no. 8, p. 9.

Eidesstattliche Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Rostock, 02. März 2018

_____

Tom Meyer