

Masterarbeit

Eine Petrinetzsemantik für Rust

VORGELEGT VON:

Tom Meyer

MATRIKEL-Nr.: 8200839

EINGEREICHT AM:

datum

BETREUER:

Prof. Dr. Karsten Wolf

Titel

Titel english

Betreuer: Prof. Dr. Karsten Wolf

Tag der Ausgabe:

Tag der Abgabe:

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Outline	1
2	Background and related work	3
2.1	Rust	3
2.2	Compilers	4
2.3	Parallel Programs	5
2.4	Verification	5
2.4.1	Model Checking	5
2.4.2	Petri-Nets	6
2.4.3	CTL*	7
3	Approach	8
3.1	Rust Compiler	8
3.2	Interception strategie	9
3.3	Mid-level Intermediate Representation (MIR)	10
3.4	Translation	11
3.4.1	Entry Point	11
3.4.2	Functions	14
3.4.3	Memory	16
3.4.4	Basic Blocks	18
3.4.5	Statements	18
3.4.6	Terminators	21
3.5	Panic Handling	23
3.6	Interface Emulation	24
3.7	Petri Net Representation	26
3.8	Model Checking	27
3.9	Test Programs	27
4	Result	29
4.1	Translation target	29
4.2	Verification results	31

<i>Contents</i>	iii
4.3 Discussion	31
4.3.1 The Model	31
4.3.2 Verification	33
5 Conclusion	34
6 Future Work	35
Bibliography	37

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, desirable 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

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.[1]

2.1 Rust

Typically programming languages can be divided into fast or safe: To increase software speed, developers have to optimize specific system details. This micro management easily leads to mistakes as it is hard to keep track of the complexity of the whole system. Mistakes that can be avoided with safety features like garbage collection – compromising execution speed.

The Rust project is an attempt to build a language that is both: fast and safe, as the official slogan indicates: “A language empowering everyone to build reliable and efficient software.”[2]. To ensure a minimal performance overhead Rusts runtime was kept small[1, Chapter 16.1] and features like garbage collection were neglected[1, Chapter 4]. Instead, safety issues are addressed with Rusts **ownership model**[3].

In Rust, a memory resource (object) is associated by one unique owning variable. The owner can mutate the object, reference it or hand over ownership. By handing over ownership it is lost for the previous owner, to ensure unique ownership. However, references can be ‘borrowed’ in two flavors without losing ownership:

1. There can be an arbitrary amount of **immutable references** at a given time.
2. But only one active **mutable reference**. No immutable references can be active at the same time and the owner is prohibited from mutating.

References that go out of scope are ensured to be deconstructed by Rusts **borrow checker**. Programs that don’t meet the ownership requirements will not compile and raise an appropriate error message.

By enforcing the ownership rules, Rust programs avoid common problems like dangling pointers, double frees and data races[3] with no impact on execution speed. The cost is transferred to compile time, where additional errors complicate development and established programming patterns have to be revised. And while eliminating all these errors can be invaluable, Rust cannot prevent all mistakes. One of which are deadlocks[4, Chapter 8.1] which we want to address with this work.

[Überleitung]

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[5, 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[5, 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. And **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 Verification

Every software engineer who worked on reasonable complex systems probably admits that initial software versions are full of inconsistencies and bugs. To achieve resilient and correct software a detailed understanding of the system and careful reviews of the implementation is needed. If this process is done systematically it is called verification.

To verify that software operates correctly it is required to know what ‘correct’ means. Correctness is no intrinsic property of a system; It has to be defined in its context. For this, a description of the system – a **specification** is required, to infer the **properties** it should fulfill. % example for specification and properties % A **system is correct** if its specification satisfies all its properties[6, Chapter 1].

Among important approaches to verify software are code reviews and testing. Both techniques are valuable to find different kinds of errors. But in this work we will focus on **Model Checking**, an approach to search the complete state space of a system.

2.4.1 Model Checking

Model checking tries to solve the ambitious problem to check a property for every possible system configuration. To do that, the behavior of a system needs to be represented in a (name giving) model and separately the properties that the system needs to fulfill;

Typically in a formal logic. Having both, all possible relevant model states are explored to verify that the given properties hold in that state. Unfortunately the the amount of states is typically exponential in the system size; A phenomenon that is known as the **state explosion problem**[7, Introduction]. This is the most outstanding problem of model checking, but it would be less used if there where no way to weaken the explosions impact. Some of the major techniques are symbolic model checking, partial order reduction and abstraction refinement[8, Chapter 5]. However, the important information here is that model checking can tackle systems with a realistic amount of states.

What is more important to know is the model formalism we want to use in this work and how we will describe our properties; Namely Petri-Nets and CTL*.

2.4.2 Petri-Nets

Petri-Nets where developed in the mid 1900's by Carl Adam Petri[9]. It is a formalism that is well suited to model concurrent behavior, since it does not model each state explicitly.

Petri-Nets are a bipartite directed graph. That means that a Petri-Net has two kinds of nodes where one kind is always connected to the other but never with itself. One type of nodes are **places**, that can hold an arbitrary amount of **tokens**. In low level petri nets these tokens do not carry any information; Only the amount of tokens on each place determines the system state. The other node type are **transitions**. A transition is **enabled** when places that are connected with an incoming edge (from place to transition) have at least as many tokens as the corresponding edges **weight**; I.e. A place that is connected through an edge with weight three needs at least three tokens. A transition without incoming edges is always enabled. Enabled transitions can **fire**. A firing transition **consumes** tokens corresponding to the edge weight from places connected with incoming edges – the transition **preset** – and **produces** tokens corresponding to the edge weight on places with outgoing edges – the transition **postset**. If the postset is empty (no outgoing edges) no tokens are produced but the transition can still fire. Which enabled transition fires next is nondeterministic, so they fire randomly.

Formally that means a Petri-Net is a five-tuple (P, T, F, W, m_0) with

- the set of places P ,
- the set of transitions T ,
- the set of edges F with $F \subseteq (P \times T) \cup (T \times P)$,
- the edge weights $W : F \rightarrow \mathbb{Z}$
- and an initial marking $m_0 : P \rightarrow \mathbb{Z}$

Other important definitions are:

- a marking M maps all places $p \in P$ to a number of tokens $M : P \rightarrow \mathbb{Z}$
- the preset of a transition $t \in T$ is: $\bullet t = \{p \in P | (p, t) \in F\}$

- the postset of a transition $t \in T$ is: $t\bullet = \{p \in P \mid (t, p) \in F\}$
- the preset of a place $p \in P$ is: $\bullet p = \{t \in T \mid (t, p) \in F\}$
- the postset of a place $p \in P$ is: $p\bullet = \{t \in T \mid (p, t) \in F\}$
- a transition $t \in T$ is enabled if $\forall (p, t) \in \bullet t : M(p) \geq W(p, t)$

!!!

examples

!!!

Where is a large formal background to Petri-Nets that makes it possible to check for a variety of properties. For example it can be analysed if a particular marking can be reached from an initial marking or how many tokens a place may hold. And – important for this work – if the net can reach a **dead** state, where no transition can fire anymore. This state is equivalent to a deadlock. Another nice feature of Petri-Net-Models is that, if a state is found where a property is satisfied, it is typically possible to find a witness path from the initial marking.

A downside of the formalism is the difficulty to model data. This can be addressed with several additions that lead to **high level Petri-Nets**, where tokens are associated with additional properties that can represent data. Transition in high level Petri-Nets can also respect the data of tokens in their firing behavior. However, these additions to the formalism weaken the statements that can be derived from it, limiting the properties that can be checked for.

2.4.3 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

There are basically two options to translate rust into Petri-Nets:

1. write an own translator or,
2. use something existing.

Writing an own translator means total control over the process. Features can be added iteratively as needed and data structures can be designed efficiently for this special purpose. However, this would result in a new compiler for rust which eventually has to cover every language feature; Including some difficult ones like macro expansion and generics. Features that someone already implemented, spending a fair amount of thought in the process; Backed by a large community; Over several years. Resulting in a compiler that is openly available under an open source license[10], with a maintained documentation[11][12]. And even though learning how this complex compiler works, and where to find the relevant parts for this project is also difficult, it seems to be time worth spend if we are able to skip learning and implementing difficult features. For this reason this project is based on the rust compiler ‘rustc’. So lets see how its basic structure looks like.

The rust compiler does have several phases and a variety of intermediate representations [11, Chapter 2.1]:

1. In the first phase rust source files are parsed to an abstract syntax tree (**AST**) that matches the original syntax closely.
2. In the second phase implicit information is expanded. This includes for example macros and identifier names.

3. Phase three lowers the AST to a simpler form called high-level intermediate representation or **HIR**. The HIR still is quite similar to normal rust syntax but some structures are normalised so that code analysis is easier. For example for loops are rewritten to simple endless loops with break conditions.
4. The fourth phase executes some static analysis on the HIR. Things like type checking and encapsulation verification is done on this representation.
5. Another representation is generated in phase five: the mid-level intermediate representation (**MIR**). Now we leaving the tree structure and switching to a graph. MIR is based on a control-flow graph [?]nd language features are reduced to a minimum. There is only one type of loop and branches respectively (only gotos instead of ifs or pattern matching). Additional static analysis is done on the mir level, like rusts borrow checking, as well as further optimization.
6. Phase six lowers the MIR to the rust independant LLVM[?]ntermediate representation. LLVM IR is a low level representation that is close to assembly language. Optimization for this representation can now be done for the code resulting in several object files.
7. In the last phase the object files are linked into a complete binary executable.

But wich phase is best to intercept to translate the current intermediate representation into petri nets?

3.2 Interception strategie

After deciding to use the rust compiler as basis for this work, we need to determine the phase we want to intercept the default compilation to translate to our own target. A suitable location makes use of the most possible compiler features with the least amount of translation effort.

We want to skip basic features like name resolution as we need this feature ourselves and surely won't improve it by a reimplementaion. Also code generation, including macro and generics expansion, should be handled by the compiler. They just produce more rust code that we will treat anyway. But after expansion, no code generation syntax will remain and we can simply ignore this feature.

More optional for our use are the static analysis features like type checking and borrow checking. Though, we want to use their assumption, we do not depend on them actually being checked, since nobody will ever run a rust program that did not go through the complete compilation process. So it is quite safe to assume that nobody will do model checking on a program that won't compile. However, we still can enforce those invariants if we run the static analisis anyway and abort on errors. This prevents time consuming runs for programs that cannot be build.

A compiler feature that *can* greatly ease our effort is the lowering of the representation. After all that means that several high level language features are reduced to less low level features. Which in turn means less features we have to cover with our implementation. We don't want to go to low though, since lower representations might prevent us from exploiting assumptions that are hidden in the translation process. We also probably want to avoid a machine dependant representation. To verify generic rust programs and not the peculiarities of a single machine. This might be debateable though.

Lastly we want to make use of all optimizations we can. Especially optimizations that reduce the code (and resulting net) size. So features like constant propagation and dead code elimination would be nice to have to reduce the limit of state explosion. At least a little bit.

Looking back an the rust compiler phases with those thoughts in mind, the seemingly best place we can intercept is between phase five and six. After borrow checking and optimizations on the MIR. Here, all code was expended, all rust specific static analysis and optimization was done and all unnecessary language features where reduced to a minimal set of instructions. And since MIR is a graph representation as well as petri nets, a mapping should be relatively straight forward. More so if we consider that MIR represents control flow quite closely as petri nets do. Also we avoid the still lower level and rust independent LLVM IR and the machine dependent object files.

3.3 Mid-level Intermediate Representation (MIR)

Now where we pinned down MIR as the intermediate representation to use, it is helpful to understand how it is structured.

As mentioned before, MIR is derived from control flow graphs[11, chapter 2.17]. It consists of several **basic blocks** which are interconnected with directed edges. Each basic block consist of any amount of non branching **statements** and a single – possibly branching – **terminator**. All statements in a single basic block are executed sequentially. Between those, no branching can occur in or out. Only terminators can redirect the control flow. They are the ones that represents conditional execution (i.e. if-then-else constructs) or jumps to other basic blocks (including loops).

Data in MIR is represented as **locals** and **places** (not to be confused with petri-net places). Places represent any kind of memory location, whereas locations are conceptually located on the stack. This means that a location can also be represented as a place, but places are not necessarily locations. Statements work on these data representations. The most prominent type of statement is an assignment that assigns an **rvalue** derived from an expression to a memory location – meaning a place. Also based on the data terminators can direct control flow. Which branch the program takes might depend on the value a place currently holds.

It is also important to know that each function has a separate MIR representation. Calling other functions is an action done by a call-terminator. They can direct control

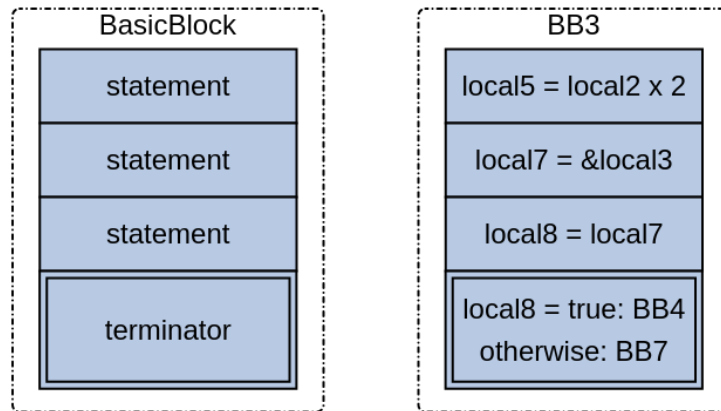


Figure 3.1: Structure of a basic block. Several chained statements with a single trailing terminator. The terminator can redirect control flow to other basic blocks. In this example BB3 branches to BB4 or BB7 depending on the value of local8

flow to a subroutine that is executed until a return-terminator is hit. After that the calling function resumes execution at a previously defined basic block. An important note is that the functions in a call are referenced by name and not with an edge in the graph (this results in the separate MIR representation for each function). This approach ensures that recursive function calls are kept simple.

Besides the specific kinds of statements and terminators, this is the general concept for the MIR graph. A more detailed view would be of little help since it would get very close to the actual implementation. Implementation details, however, are frequently changing; In fact during this work it changed multiple times breaking the code. A circumstance that is communicated by the compiler team and needed because a too stable API would restrain the compiler development process.

3.4 Translation

After we saw how the MIR graph is structured, we can try to find a translation to Petri-Nets. This will include some more details and edge case we have to consider.

3.4.1 Entry Point

The will start with the most abstract view on our translation: the whole program. A program is something that typically has a beginning end an ending. We can model this with a **program start** and an **program end** place that every program has. Depending

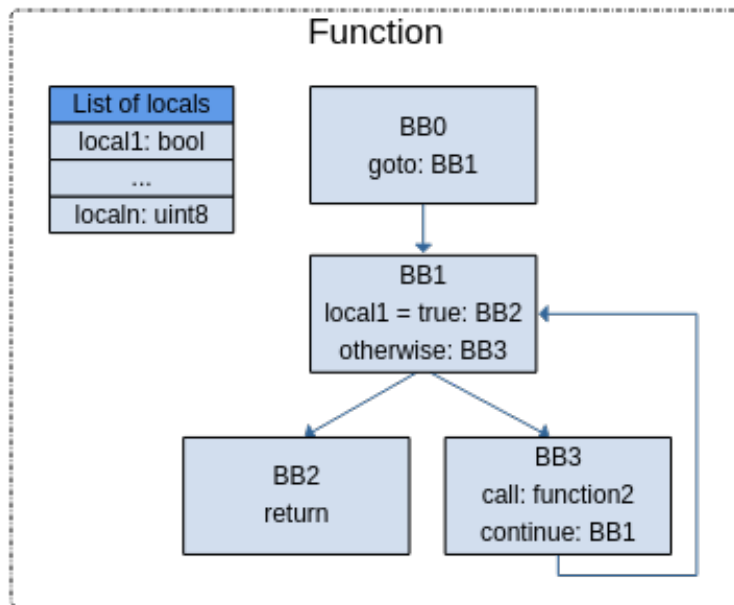


Figure 3.2: Example of a functions internal control flow. The statements of the basic blocks are omitted as they don't affect control flow. A function has a list of locals that are accessible on the stack. Each local has an associated type. Functions always start with the first basic block (BB0 in this case). This function either returns to the calling function in BB2 or calls 'function2' in BB3. When function2 returns, control flow continues in BB1.

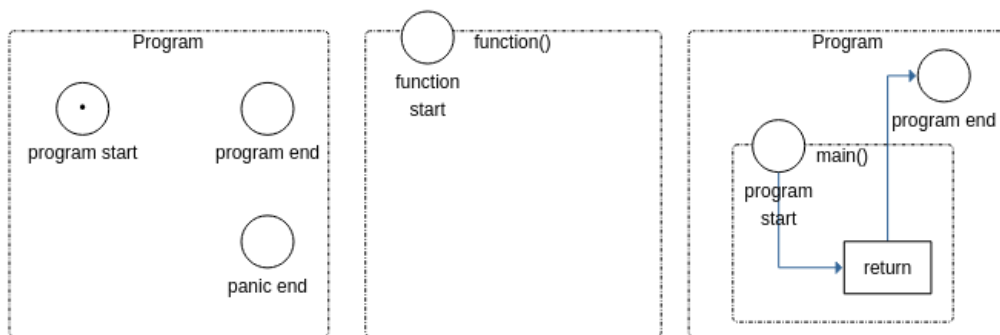


Figure 3.3: Stubs for programs and functions, and a minimal terminating program. A program has a start and an end place. The start place holds a token, as program execution begins there. If a program terminates successfully the program end place is marked. A termination caused by a panic marks only the panic place.

A function has a distinct starting place (it always starts at the first basic block). But there is no end place, as functions can be left by any terminator. The shortest possible (terminating) program has an empty main function. Program start place and main function start place are one and the same place. Empty functions have no basic block in MIR, so we have to add an artificial transition to leave the function.

on the program they are interconnected somehow. The sole exception are programs with a diverging main function – where the main function ends in an endless loop. In this case start and end place will not be connected, but an unconnected end place does not harm either. In rust a second end place for panics – the **panic end** place – is a helpful addition. This place will be marked then the program terminated unsuccessfully after a panic was raised. A circumstance that might be helpful to distinguishing in verification runs. Finally the program start place needs to be marked with a token. We will later see that this token also indicates the first statement to execute and that it virtually moves from statement to statement acting like a program counter (even though petri net tokens do not move semantically but rather be consumed and produced).

Although these are all the basic features shared by every program, we still have to look a little closer on the semantics of the program start place. This place is a bit ambiguous since, in practice, the main function is not what is executed first. Usually programs have a ‘runtime’ that is initialized before `main()` is called. These setup static memory and initialize language specific features, among other things. And even though low level languages like rust or c have a small runtime, they still have one. Now we have to decide if this runtime should be considered for Petri-Net translation. After all it is part of the finally executed binary. On the other hand it is platform dependent code that is independent from the actual program semantics. And there is another problem in starting in the pre main code. It turns out that the MIR for that part is not completely available in every compiler version. It is possible to get the missing parts subsequently but it complicates the translation process unnecessarily. Additionally, we previously argued for a platform independent approach already in chapter 3.2. It would be inconsistent to detach from that agenda now. So, because of these reasons we decided to skip the pre `main()` code and start translating programs with the actual main method.

3.4.2 Functions

After we modeled the important features of a program, we can look on the next important path that a program consists of. Control flow wise these are functions. As in most other imperative languages rust programs have a main function that wrap all its functionality (excluding the runtime as discussed in the previous section). The main function can call other functions that in turn can call additional functions, and so on. When executing a program the called functions are organized on a stack called ‘call stack’. When a new function is entered a new stack frame is pushed on the stack including information like function arguments and local variables. On leaving a function it is removed from the stack deallocating memory the frame occupied.

In MIR calling and returning from functions is done in a basic block terminator and can occur arbitrarily often. Considering this behavior in a function model would be of no help, leaving just the start place as a feasible similarity between functions. A function that is called always begins its execution there. And it will always be identical to the start place of the first basic block.

So, from a modeling perspective functions are not the decisive abstraction. But if we consider the translation process they get more important. The rustc interface for MIR works per function. This means that it is not possible (at least no obvious one) to get the MIR from a whole program but only from single functions. A likely explanation for this design choice is that a lot of context information switches with functions. For example every functions starts with basic block zero and increases the count for the following. Local variables are also indexed with some of them having a special meaning. The first being the functions return value followed by variables for all function arguments. If we want to translate a program we have to keep this structure in mind. In our implementation we have done this virtually by an own call stack. We traverse the program function by function. Every time we encounter a function call terminator we try to translate the called function and return where we left if we are done. On the new call the context is switched to the new variables and basic blocks, and the return semantics is stored.

However, this approach has some implications:

1. If the same function is called multiple times in the program, it is also translated multiple times. Although, this can probably be avoided with an intelligent function cache this approach is sufficient for a proof of concept.
2. The far more extensive implication are the voided recursion capabilities. If we encounter a recursive function with this strategy, we will be trapped in an endless loop.

Actually recursion is a feature that can never be achieved with low-level Petri-Nets, as its data model cannot be mapped correctly. How often a recursive function is called depends on the data it is called with and cannot be known at compile time. In normal program execution functions are just pushed on a new stack frame until it is resolved (the base case is called for all recursive calls) or the stack overflows. Most relevant programs will be still executable. But we cannot model this stack like behavior in low-level Petri-Nets because we cannot get additional memory. All memory we have is that we know of at compile time. And we cannot reuse the places of a function for different recursion levels as we cannot distinguish the tokens from the recursion levels. We could just remodel each recursion level again and again to a fixed recursion level. However, this would impact the verification results, since a property might change for programs with different maximum recursion depths.

A solution to this problem can be high-level petri nets, where we can distinguish between tokens. There we could reuse places from the same function by annotating tokens with the corresponding recursion level. We won't go into detail of this approach, though, since this work is based on the low-level semantics. We will just have to accept that we cannot translate recursive functions for the time being.

3.4.3 Memory

To translate a rust program we cannot focus only on flow. We also need a valid representation of data. Unfortunately, as mentioned before, data is complicated to model in low-level petri nets. With only the concept of bare tokens, we can only model finite memory space and even if we do, modeling every state of every possible variable would bloat our net terribly.

To stay in the realm of low-level nets we have need to abstract from variable states. In principle a Petri-Net place already is something that holds data: a set of tokens. If we interpret a place as a memory location and a token is data, though we loose a lot of information, however, we still can analyse the program flow. Additionally, this representation resonates well with a possible high-level extension where the token can be annotated with the current value of a variable. So, to keep it simple we represent a memory location as a single place with a single token. And we ‘read’ from memory with a transition that consumes the token, while also producing a token.

Now in many programing language there are different concepts of memory that behave differently, and rust is no exception. Basically we can distinguish between four different concepts:

1. **Constant memory space:** The size of the constant space is known at compile time and the values of this space do not change during runtime (hence the name). The values of all constants are compiled into the executable and no computation is needed to get them.
2. **Static memory space:** Like in the constant space, the size of the static memory space is known at compile time. The big difference is that data the static space can change during runtime. So during program startup, a fixed amount of memory is allocated and all static variables are initialized. Later they can be used rather similar as normal variables. With some rust specific constrains that we will not discuss here, as they are not important for Petri-Net translation.
3. The **Stack:** there the size is known for each stack frame (function call). This is the combined size of all local variables a function needs for execution. If a function is called a new frame is pushed to the stack and if the function terminates the frame is removed. Since the last pushed frame will be the first to be removed (LiFo - Last in, first out), the stack can be managed without data fragmentation.
4. The **Heap:** which is the place there every remaining memory goes. A dynamic variable, there the size cannot be known at compile time needs to allocate memory on demand at runtime. If the memory is not needed anymore, it will be deallocated and the again free space can be reused. Since both allocation and deallocation are on demand, this can introduce memory fragmentation, since the size of the holes is not uniform.

The most prominent concept on the MIR layer is the stack. Every part of the MIR is associated with a function which has a set of local variables (locals). The current state of a local variable is changed by statements. This includes the values the variable can be assigned to as well as information about variable liveness. Each local starts in an uninitialized state until a statement is called to set it ‘storage live’. A living variable can be assigned to arbitrary values (the type allows) as long as it is needed. Afterwards a statement sets the variable ‘storage dead’ to indicate that it will not be used again in this function call. We can model this behavior with three Petri-Net places for each local: one place for the uninitialized state, one for the living state where the variable can be used and one for the dead state. The first active state will be ‘uninitialized’ so this one must be marked with a token initially. The unique ‘storage live’ and ‘storage dead’ statements (which are special statements generated for the MIR) have to be called to traverse the three states. They will consume a token from the previous state and produce one on the next. This enforces that data access can only be done if a token is on the live place. This way we can later verify if the liveness invariants are met.

Heap and static space is hidden behind local variables. For example we can access a value on the heap by dereferencing a pointer we stored in a local. In MIR this is done with a projection from a local that describes which part of the local is used; Which field of a struct, which index of an array or if we dereference the local for example. This information is stored implicitly as MIR-place on which statements can operate. Unfortunately this projection is hard to model. Especially pointer dereferencing is a problem since the memory model is handled by the operating system at runtime. But for all these projections the source – the initial local – is known. If we use a projection in a statement we can interpret this as an access to the initial local. Again we lose some information here in exchange for a manageable design.

What is much easier to model is the space for constant variables. Their behavior is close to locals that live for the entire runtime. So they do not need any uninitialized or dead place, just a place that represents the current value. In fact we can model the whole constant space as a single Petri-Net place, where every access is done with different transitions. Where is no data manipulation anyway. And again this approach resonates well with a possible high-level petri net extension where every accessing transition produces tokens with the corresponding constant values.

In conclusion we need two models for memory access: a set of places for each local variable with uninitialized, live, and dead place, and a single place that represents the whole constant memory space. Memory is accessed and modified in statements and if we have to handle heap space we can hide this behind an access of a local variable which is projected to other memory space that we are unable to control (or model). Before taking a deeper look on the memory using statements next we will see how we can model the basic blocks that contain the statements.

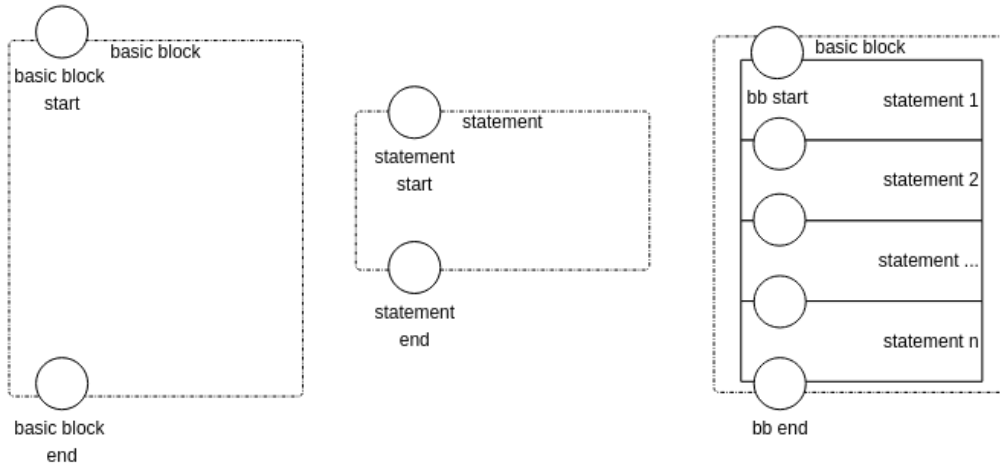


Figure 3.4: The structure of basic blocks and statements. Basic blocks as well as statements have a distinct start and end place. The start place of the first statement is identical with the basic block start place. The same is true for the last statements end place and the basic block end. Also the end place of a leading statement is the start place of the following statement (illustrating the sequential nature of statements). Terminators are represented entirely by transitions that are not pictured here, as they are typically linking to other basic blocks.

3.4.4 Basic Blocks

It is essentially a container for a sequential part of a program; With an entry point and an exit point. On exit, program flow can be redirected to one or more other basic blocks (redirections to other functions will again start at a basic block).

Consequently, in a Petri-Net model, a basic block has an entry place and an exit place. These are bound to the containing statements, where the basic blocks entry place corresponds to the first statements entry place and the basic blocks exit place corresponds to the last statements exit place. The terminator is modeled by one or more transitions that are connected to the end place to another basic blocks start place.

3.4.5 Statements

A basic block does not have many features of its own. Statements are the part of the MIR that change data. There are several different kinds of statements; How many and how they are structured depend on the exact kind of the statement. Additionally, some of them are not used in every compiler phase. For example a MIR statement named 'FakeRead' is used for static analysis but removed in a later optimization phase when

the sanity was satisfied.

The general structure of the different statements, however, is always rather similar. In Petri-Net terms, they have a starting place, a transition that manipulates some data places and an end place. If two statements succeed each other, the first transitions end place will be the seconds transition start place. Similarly, the start place of the first statement in a sequence will be shared with the start place of the surrounding basic block. Likewise the end place of the basic block is shared with the end place of the last statement. The statement transition will consume a token from the start place and produce a token on its end place. Additionally, the transition will consume and produce data places to resemble a manipulation of this data. As mentioned earlier the actual data is transparent in the low-level Petri-Net model but might be added in a possible high-level model.

As an example for the data manipulation we can look at the three most important statements: ‘StorageLive’, ‘StorageDead’ and ‘Assign’.

- The *StorageLive* statement consumes a token from the *uninitialized* place of a local variable, and produces one on its *live* place.
- The *StorageDead* statement consumes a token from the *live* place of a local variable, and produces one on its *dead* place.
- An *Assign* statement in the language semantics, evaluates an expression and assigns the result to a memory location (lvalue). In Petri-Net semantics it consumes a token from the lvalues live place and all live places that are involved in the expression. Simultaneously new tokens are produced on all of them (one per involved place). This means that no assign statement can be executed until the corresponding locals are initialized with a storage live statement or after the local was retired with a storage dead statement.

With the virtual movement of a token from one statements start place to another we modeled a marking mechanism for the currently active instruction (meaning a statement or terminator). If we would simulate the resulting Petri-Net (except at program start and end) we always have a token on exactly one start place of a single instruction. This instruction will be the next one to be executed and afterwards a new one will be marked. This property is very close to a *program counter* in CPU’s which stores the address of the instruction that will be executed next. This next instruction might just increment the pointer (in sequential parts) to mark the subsequent instruction as active or manipulates the pointer to jump to a totally different instruction. This is a convenient similarity since it increases confidence that the Petri-Net actually models something that is similar to an execution semantics of a real program.

Next we see how jumps work in our Petri-Net model.

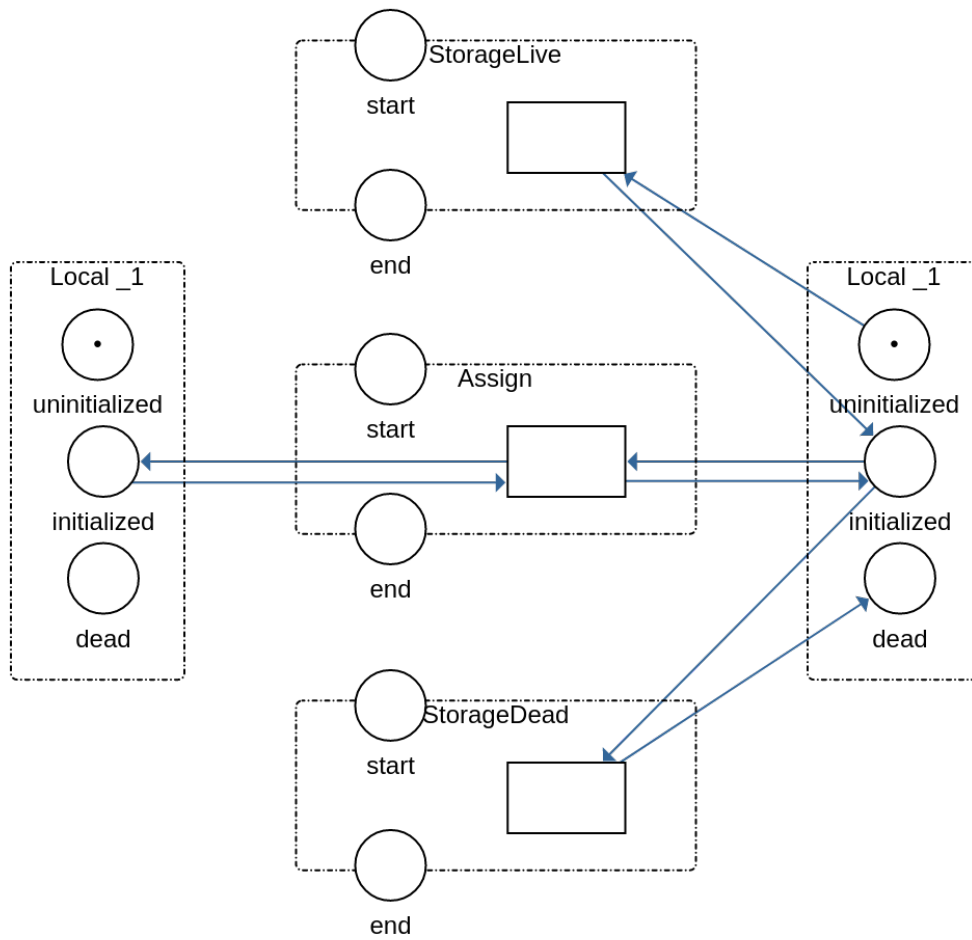


Figure 3.5: Three kinds of statements (unconnected)

3.4.6 Terminators

After we covered the strict sequential part of our program we will now discuss the jumping and branching terminators. They too have a common structure: Every terminator has at least one transition which consumes a token from a basic block ends place and produce a token on a basic blocks start place.

The MIR representation again has different kinds of terminators. Here is a list with the most important ones:

- The most basic *Goto* terminator has a single transition. It connects two basic blocks inside of a single function.
- The *SwitchInt* terminator implements a conditional branch to multiple other basic blocks. A branching path can only be taken if its condition is met. However, the last path acts as an *otherwise* branch that will be taken if there is no other valid path. As the Petri-Net representation is used for model checking we always consider every possible branch. As a result we can ignore the condition and just connect all involved basic blocks with a transition.
- The *Call* terminator connects basic blocks of different functions. It also encodes function arguments and the basic block to continue after the called function returns. We have to remember this information for the translation (especially the arguments), And have to make sure that we reuse their locals in the called function. Also function calls might panic and if they do execution continues on a separate basic block.
- The *Return* terminator is the return path from a successful function call. Here the current basic block is connected with the one that is encoded in the Call terminator.
- The *Resume* terminator is the return path from an unsuccessful function call. This terminator is connected with the panic basic block that is encoded in the function call.
- The *Assert* terminator branches depending on a condition. If the condition evaluates as expected – normal execution flow continues, and if it does not – a panic is started on a separate path. Again, in the Petri-Net we always consider both cases so we can safely ignore the condition and just model both paths with a transition.

We can see that terminators determine successful and unsuccessful execution causing panics which are a error handling mechanic of rust. This is an important part of program execution and demands a little closer look.

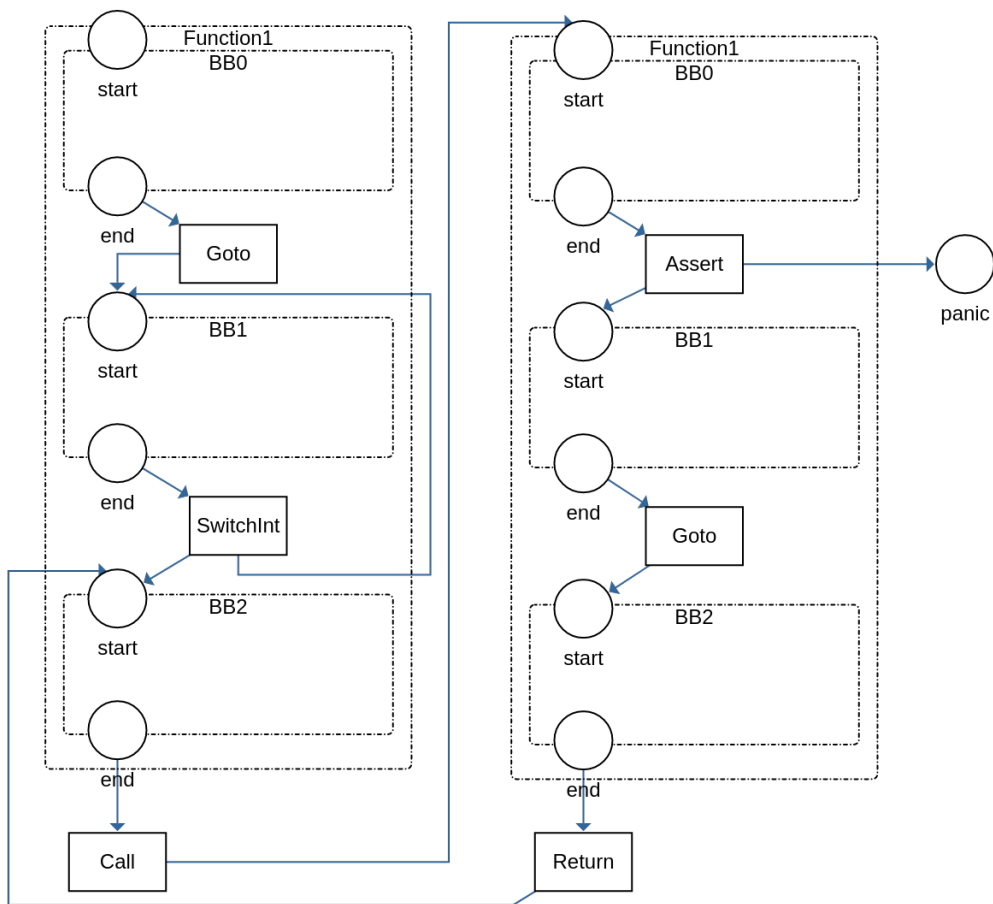


Figure 3.6: Connection of terminator transitions

3.5 Panic Handling

Error handling is a vital part in programming and typically is considered in program language design. Many languages implement an exception semantic with try and catch blocks. Exceptions can be thrown and if they are not caught the program terminates ungraceful.

Rust has a different approach: normally functions are expected to not fail. Functions that can failed will return a *Result* type that can be the result of the computation or an error with a description. The type system enforces handling of both cases and the language gives some mechanisms to do so. Of course the interesting part is the error case. This one can be escalated to previous function calls until a consistent program state can be restored.

But it might not always be possible to recover from an error state. In such a situation the program can be instructed to *panic* and shutting down ungracefully like with an uncaught exception in other languages. The program execution is aborted, the stack will be unwinded and an error message with the details of the panic is generated (stating the error message and the location of the error). Though, panics can be caught by a parent thread they typically lead to the termination of the current program. This panic structure ensures that the compiler always knows when a panic can happen so it can generate appropriate code.

Code that we can identify and handle in the MIR representation. We hinted this in the last chapter: some terminators can branch execution to a normal path or a panic path. Branches to a panic path lead to basic blocks that are marked as cleanup and handle stack unwinding. If execution enters a cleanup path, it cannot return to a normal path and will end up in an erroneous termination (unless caught by a parent thread). We can model a separate termination place in our Petri-Net for this event. This way we can check for this property later. However, stack unwinding will execute generated code that is not a distinct part of the program semantic. After all, what really matters is that an erroneous state was reached that we can't recover. So we can reduce the size of our net by skipping the cleanup paths and directly put a mark on the panic place.

Of course the rust compiler cannot cast black magic to prevent the really bad errors like dereferencing pointers to unaccessible memory. Such a miscalculation would lead directly to an os exception (or other undefined behavior) causing an error that cannot be caught by rust – or worse – a messed up state with no error at all! And since rust cannot detect those errors, we cannot model them either. We just have to assume that all compiled operations are valid in the execution context.

This kind of errors have to be avoided by the implementor. But rust aids avoidance by marking such operations and functions as *unsafe* to use. Most of the time it is possible to avoid unsafe code using abstractions and safe operations. Only a small code base needs to use unsafe code to create those safe abstractions and once the integrity there is kept, all depending higher level code benefits.

3.6 Interface Emulation

There are endless possibilities to implement an algorithm and not all of them depend on rust as description language. Still, using established algorithms from other source languages is a desirable feature to have. Unsurprisingly, rust implements some interfaces to access functionality that is not native to rust. The two most important are compiler *intrinsic functions* where the implementation is hidden by the compiler; And the *Foreign Function Interface* for interfacing C-Style exports. Both of them call instructions that are not represented in the MIR graph.

These calls leave the realm of rust code where no guarantees can be made (which makes most of them unsafe). But on lower level code we won't get around them. For example rusts thread interface in linux systems is an abstraction of the 'pthread' library which is written in C. To translate such items we will have to emulate them somehow. In case of compiler intrinsics this would be a finite amount of work but there is no upper bound for foreign functions that might be called. So we have to implement at least a generic translation that works for all calls. Additionally we can translate important functionality by hand.

A generic call can be thought of a combination of a statement and a terminator. The locals for the arguments have to be accessed by a transition like we did for the statements. Afterwards we branch depending on the return value, like we did in the terminators. This behavior describes all basic data manipulation functionality. But there are other functionalities that we need to address in a special way: some foreign functionality can influence the execution flow. For example mutexes and threads are significant for our purposes.

Mutexes

Mutexes guard execution flow depending on a data variable. If we just access the variable and continue like we do for statements, flow could always continue. However, flow should only be allowed to continue if the mutex variable can be acquired (and block otherwise). This can easily be modeled in Petri-Nets with a marked mutex place. Then, on every try to acquire the mutex a transition has to consume the token from the mutex place. The first try will do so without further problems but every following attempt will be blocked, since transitions can only fire when all pre-places are properly marked. When leaving the critical section a transition needs to reproduce a mark on the mutex place again, so that blocked processes can continue with their computation.

In rust Mutexes are part of the standard library. Before data guarded by a mutex can be mutated it has to be unlocked. If it is, the data cannot be accessed again until the lock is released. This is done automatically when the variable of the accessed data goes out of scope. listing 3.1 shows an example usage of a mutex.

```
1 use std::sync::{Arc, Mutex};
```

```

// Here we're using an Arc (reference counting smart pointer)
// to share memory among threads, and the data inside the Arc
// is protected with a mutex.
6 let data = Arc::new(Mutex::new(0));
{ // begin of a new scope
    // Mutex is acquired by the lock() method.
    // If the lock was previously acquired the call will block
    // until the lock is released
11 let mut data: Arc<Mutex<i32>> = *data.lock().unwrap();
    // mutate data
    data += 1;
    // the lock is released here when 'data' goes out of scope.
} // end of scope

```

Listing 3.1: Mutex in rust

Now, the call to mutex *lock()* and the release of the the lock will trigger several other functions that handle a thread safe access and the blocking behavior. For translation we can model the complete underlying behavior or abstract from it. Both approaches are perfectly valid: the former would be closer to the actual execution behavior while the latter would be closer to the basic program semantics (that does not care about the mutex implementation). But in favor of a small Petri-Net we stick to the abstract wa that ignores the implementation. When a mutex is called we will stall normal execution and just insert our own abstraction at the place of the corresponding function calls. But there is more we have to consider.

If we take a look on a pruned version of the MIR layer in listing 3.2 we can identify a Problem. Because mutexes are unlocked implicitly and often are wrapped in other types, we loose the information when a specific mutex is locked or unlocked. The mutex that is created in *bb0* is wrapped in a smart pointer in *bb2* masking the original mutex. In this simple program the smart pointer is unnecessary but usually mutexes guard critical sections to ensure thread safety. However to reference the mutex safely in multiple threads the reference has to be thread safe, which is ensured by the wrapping *Arc* smart pointer. By the time we lock the mutex in *bb4* we can not read directly from the local which mutex it belongs to (in case there are multiple mutexes in a program). The same problem is inherited bey the mutex guard in local *_3*.

```

fn main() -> () {
    let mut _0: ();
    let _1: Arc<Mutex<i32>>;
5   let mut _2: Mutex<i32>;
    let mut _4: &Mutex<i32>;
    let _5: &Mutex<i32>;
    let mut _6: &Arc<Mutex<i32>>;
    scope 1 {
10        let _3: Result<MutexGuard<i32>, PoisonError<MutexGuard<i32>>>;
    }
    // create the mutex first

```

```

bb0: {
    _2 = const Mutex::<i32>::new(const 0i32) -> bb2;
}
// move the mutex into the smart pointer
bb2: {
    _1 = const Arc::<Mutex<i32>>::new(move _2) -> bb3;
}
// dereference the smart pointer
bb3: {
    _6 = &_1;
    _5 = const <Arc<Mutex<i32>> as deref(move _6)
        -> [return: bb4, unwind: ...];
}
// lock the mutex
bb4: {
    _4 = _5;
    _3 = const Mutex::<i32>::lock(move _4)
        -> [return: ..., unwind: ...];
}
// mutex goes out of scope and destructor is called
bb6: {
    drop(_3) -> [return: ..., unwind: ...];
}
}

```

Listing 3.2: Pruned MIR for using a mutex

To connect our transitions to the correct mutex places we have to track or infer the corresponding mutexes when they are locked or unlocked. In theory the strict borrowing and aliasing rules of rust should ensure that the correct mutex can always be inferred. However, this is not trivial; So in the test implementation the local variables that are associated with a mutex (like the Arc in local `_1` and the MutexGuard in local `_3`) are marked with the original mutex. Locking and unlocking mutexes then just check the mark for the fitting mutex and connect its mutex places with the correct transitions. This is a simple approach for a proof of concept, but it probably can be improved especially in storage consumption.

3.7 Petri Net Representation

Implementing a basic graph structure to represent our Petri-Net is not very difficult. But to be compatible with a Model checker we have to use some interface or a standard that defines a commonly known structure. Luckily there exists an xml-based standard for Petri-Nets called **Petri Net Markup Language**[13][14] or **PNML**. However, LoLa – the model checker that we actually used – defines it own representation. And a third representation that comes in handy for visualizing and debugging is **DOT**[?, ?] a simple language for graph definitions. All languages are comparable in their core idea. They

all list nodes (places, transitions) and arcs of the graph. Additionally, the petri net representations encode information for token count and arc weight. Since all the three representation serve their own purpose, they are all integrated as target representation in our work. And with a finished translation we can finally feed a model checker with our Petri-Net.

3.8 Model Checking

To test and inspect our results we have the choice between different tools. There are jada jada [?] For a proof of concept it is not very important which model checker is used, since we will verify small test programs; Performance is not the biggest concern at this time. This is why LoLa was chosen by personal preference for this work.

Having a model checker and a rust program that is translated to a Petri-Net, the last thing we need is a property to check for. We want to search for deadlocks in the source program. That means that the program execution is blocked unexpectedly and no operation can be executed. This translates nicely a dead Petri-Net where no transition is active and the net reached a final state. We have to be careful though: there are states where the Petri-Net is expectedly dead; Program termination is by definition a state where execution stops. This means that if we reach either the *program_end* or the *panic* place, our net is expected to be dead. To check for an unexpected deadlock we need to make sure that our termination places are not marked: *program_end* = 0 & *panic* = 0. Additionally the net has to be dead. In LoLa this is expressed with the key word *DEADLOCK*. So the state we want to discover would be *DEADLOCK* & (*program_end* = 0 & *panic* = 0). The final part we have to consider is the temporal aspect. To specify that our state property holds eventually and to find an applicable path, we can use the *EF* operator. Its semantic is that the given property exists and will hold true at some point in the future. So our final formula would look like this:

$$EF(DEADLOCK \ \& \ (program_end = 0 \ \& \ panic = 0))$$

And having that, let's use it in a test program.

3.9 Test Programs

If we want to get some confidence in our translation we have to see if it behaves as expected. Initially, the basic functionality should be tested against the most simple programs to fail early. And what could be simpler than the empty program?

```
pub fn main() {}
```

Listing 3.3: The empty program

Beside giving a good starting point for a testable implementation, this program already revealed the edge case of empty functions. These function are not necessarily optimized away on the MIR level but also have no basic blocks as expected for every other function; A case that has to be considered. Additionally, the fact that all terminating programs have a deadlock should also get obvious here at the latest!

Other programs that can strengthen the confidence in the translation process include some important language features, like an endless program (which actually is completely deadlock free):

```
pub fn main() -> ! {
    loop {}
}
```

Listing 3.4: An endless program

Or a program with a function call:

```
pub fn main() {
2   let mut x = 5;
    x = call(x);
}

fn call(i: usize) -> usize {
7   i * 2
}
```

Listing 3.5: A simple function call

However, none of these programs are significant for what we really want to achieve: deadlock detection. For this, we need a program that forces a repeating lock of a mutex. This does not necessarily have to be in multiple threads (which is fortunate since we did not cover actual parallelism with threads yet). We simply have to lock the mutex twice in a row to create a deadlock. And since the rust compiler does no own deadlock analysis this code compiles successfully:

```
use std::sync::{Arc, Mutex};

2 pub fn main() {
    let data = Arc::new(Mutex::new(0));
    let data = Mutex::new(0);
    let _d1 = data.lock();
7   let _d2 = data.lock(); // cannot lock, since d1 is still active
} // unreachable end of main
```

Listing 3.6: A deadlock!

If our Petri-Net model is worth something it should detect a deadlock for this program and none if we remove the last line.

And with this we have all we need to evaluate our approach.

4 Result

Having a concept on how to translate a rust program into a Petri-Net, we can now inspect the actual results.

4.1 Translation target

In figure 4.1 we can see the generated net for the function call program 3.5 from the last chapter (since this is small enough to show and big enough to not be trivial). This is the true data that was produced for the dot target so we cannot immediately see the virtual boundaries for statements basic blocks and functions. To make the structure more clear the nodes were rearranged so that the called function is on the left and the main function on the right. The initialized places of the locals were renamed to show their name (locals are scoped by functions so their names are not unique).

We can see a path from program start to program end; The panic place is unconnected because the program cannot panic. Local life cycles are also visible as a single edged path from marked uninitialized place to unmarked dead places. Variable manipulation always has parallel incoming and an outgoing edges. On a closer look we can see that local _3 of the called function has no initializing and uninitializing statements. This is actually a correct representation of the MIR graph: for some reason the storage statements are not generated for some lvalues (in this case the lvalue from checked multiplications). If this is expected behavior is not known at the time of writing; A bug report [?] as yet to be solved. Unfortunately this behavior introduces an unintended deadlock into our translation since depending transitions can only fire if the initialized places were previously marked. To use the net for verification we have to work around this issue until it is fixed (or until the cause is modeled correctly). To be able to continue testing simply all initialized place were marked. This way the involved transitions can fire, but only if the previous transition produced a token on the connecting place (the program counter place). An additional token will also remain on the initialized places even after the storage dead transition fired. However execution flow, again, will not be affected because of the program counter places.

A second detail that the net shows is that the function call transition is implicit; The last statement of the main functions first basic block is directly connected to the first statement of the callees first basic block. This is an implementation detail of the translator. Since our model currently always inlines function calls (it generates a separate net for every call), these are entirely sequential. That means a missing transition does

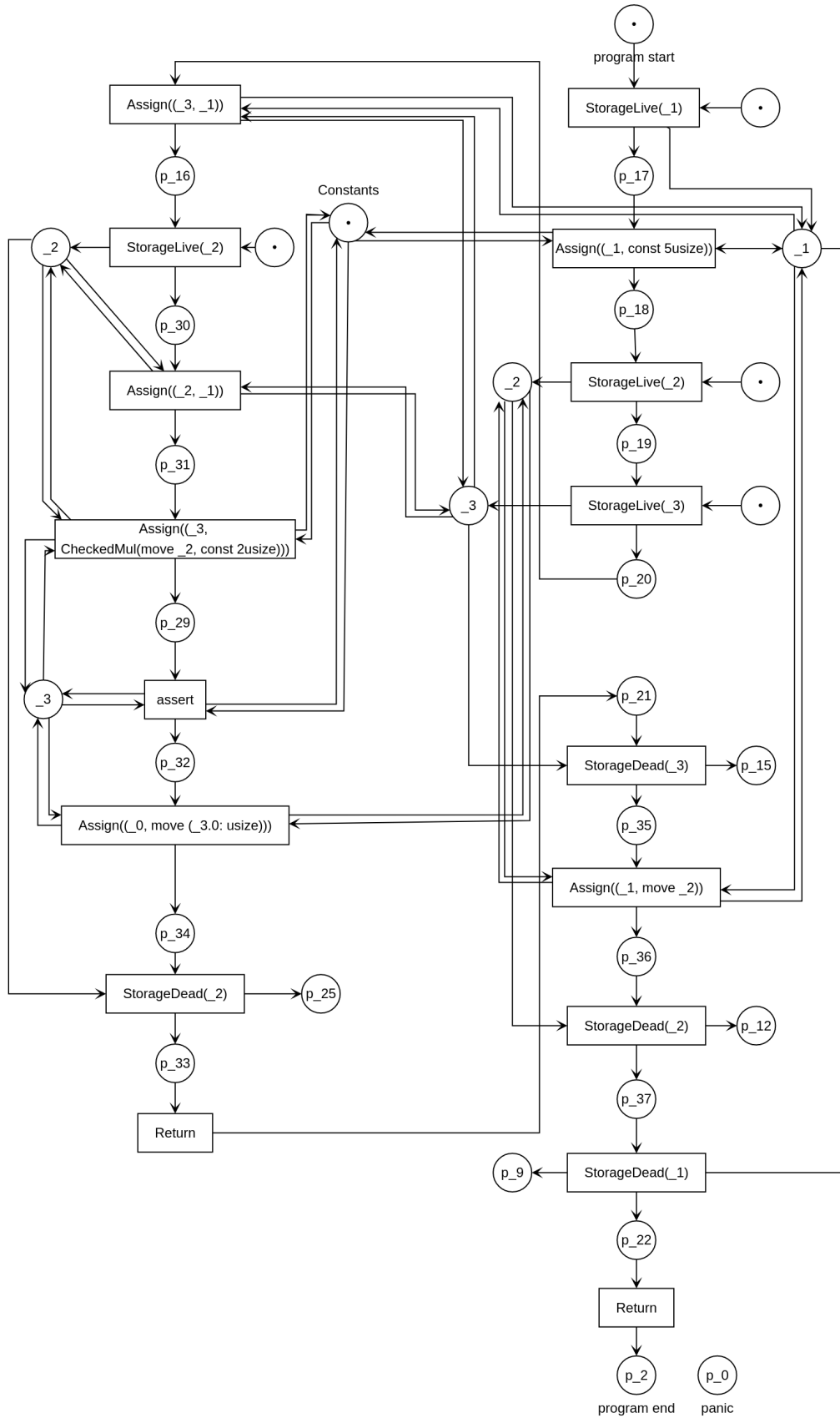


Figure 4.1: Translated Petri-net

not harm. If previously translated functions shall be reused though, this issue needs rework. But to be able to skip inlining we need high level semantics anyway.

An additional issue that can come to attention is the missing cleanup path of the assert terminator. This is the path that would lead to a panic. Logically the assert is inserted because the preceding checked multiplication can overflow, which is undefined behavior and by default should panic in rust semantics. This particular program cannot fail at this point, since the involved variables are constant and small enough to be multiplied. If this is the reason why the panic path is not generated (or optimized away) in the MIR representation however, shall be a question for the rust compiler team.

4.2 Verification results

Of course we want to use our translation further for verification. Using our formula on the deadlock program from listing 3.6, LoLa does find a deadlock and can produce a witness path as shown in figure 4.2. In contrast, if we remove the last line of the program, no deadlock is detected. But, following the witness path does indeed lead to the transitions that are involved in the mutex emulation (the mutex place is empty causing involved transitions to be dead). Earlier tests have also shown that the mutex emulation is necessary to produce deadlocks. Without the information on where and how execution should be blocked, the translation process cannot infer this behavior. This is a general problem for blocking behavior by external cause.

At this point it might be adequate to add that active waiting (looping until a guard has an expected value) is not a deadlock and also would not be detected with this approach. However, it is likely that a another formular can be found that verifies that leaving the waiting section is always possible. But again this most likely would require to consider the values that variables can hold.

4.3 Discussion

The results show that our basic approach can be used to verify some basic properties. And even though the state of the translation is nowhere near productive use there are still some lessons that can be learned that we can discuss.

4.3.1 The Model

The main draw back that followed us for the entire process is the abstraction of data in low-level petri nets. The advantages of the low-level model not only is the reduced complexity and higher verification performance, it can also produce stricter assertions. Additionally, our particular model most likely can exploit a petri-net property called safety. If we overlook the work around we discussed for compensating the missing storage-statements, the token count on every place cannot exceed a maximum of one.

```

lola: NET
lola:   reading net from net.lola
lola:   finished parsing
lola:   closed net file net.lola
lola:   777/65536 symbol table entries, 0 collisions
lola:   preprocessing...
lola:   finding significant places
lola:   503 places, 274 transitions, 273 significant places
lola:   computing forward-conflicting sets
lola:   computing back-conflicting sets
lola:   400 transition conflict sets
lola: TASK
lola:   read: EF ((DEADLOCK AND (p_2 = 0 AND p_0 = 0)))
lola:   formula length: 41
lola:   checking reachability
lola:   Planning: workflow for reachability check: search (--findpath=off)
lola: STORE
lola:   using a bit-perfect encoder (--encoder=bit)
lola:   using 1092 bytes per marking, with 0 unused bits
lola:   using a prefix tree store (--store=prefix)
lola: SEARCH
lola:   using reachability graph (--search=depth)
lola:   using reachability preserving stubborn set method with insertion algorithm (--stubborn=tarjan)
lola: RUNNING
lola: RESULT
lola:   result: yes
lola:   The predicate is reachable.
lola:   163 markings, 162 edges
lola:   print witness path (--path)
lola:   writing witness path to stdout
t_0
t_1
t_2
t_5
t_6
t_7
t_8
t_9
t_10
t_11
t_12
t_13
t_14
t_15
t_16
t_17
t_18
t_19
t_20

```

Figure 4.2: LoLa output

One obvious use for that property is the state encoding for every place, which can be done with a single bit this way. This might be helpful for very large programs with lots of places.

The greatest disadvantage of low level Petri-Nets is the reduced expressiveness of data. While flow related properties are easy to model with simple tokens, as soon as we enter the realm of data related properties, we have to make compromises. The approach we took models every interaction with data but we cannot facilitate their set of possible values for verification tasks. Another problem is that no moving data is modeled. If a previously initialized local becomes part of another local (like a field of a struct) we completely lose this information in our translation. Although, we can probably exploit Rust's strict borrow checking and aliasing rules to model a much closer relation between data, both locals are generated independently with their own places and life cycle. An improvement for example for a move of a value from one local to another (which is encoded in MIR), is to connect the places with a transition right away.

Another disadvantage of our current model is the inlining of functions. If a program calls the same function at different places a separate instance will be inserted at every call site. This not only makes the net larger, it also causes the translation to diverge in recursive functions.

A solution for most of these problems might be high-level Petri-Nets where we can model data verbosely. With them we could properly model data and detach function calls from the call site. Only the cost for verification performance remains unknown at the moment. Some problems, on the other hand, cannot be solved this way. For example, program parts that are not represented by MIR (like foreign functions and compiler intrinsics) cannot be translated and have to be emulated. Also information on blocking behavior is needed to model deadlocks appropriately. For mutexes we again worked around this issue with emulation. Additional blocking functionality (like waiting threads) have to be emulated separately.

4.3.2 Verification

The ability to discover deadlocks is already a useful property for model checking. But our model is not restricted to this single property. An easy addition is to check if the panic state is reachable. Since increased complexity also increases the likelihood of operation that can panic this might be of limited use; Unless variable data can be respected. However, more complex properties could deal with conditional reachability. For example if a function can be reached from a particular program state. Or if every execution of a program eventually visits a function or statement. But then again, our statements would be much stronger if we could consider if we consider data.

5 Conclusion

The main goal of this work was finding a mapping from rust programs to Petri-Nets. A translated net then was intended to use in a model checker to find deadlocks.

To reach that goal we searched for a suitable representation for rust programs and developed a set of rules to translate that representation into Petri-Nets. We did this for the basic components and constructed a complete model out of that components. Because some important flow related information – like blocking execution – is hard to detect with our approach, we also added an emulation for rust mutex locks. And finally we tested if a simple test program can be translated and verified with a model checker to find the expected deadlock.

An analysis of our translation showed that our data model is very abstract and probably can be further improved. However, the model of program flow seems to be close to the execution semantics of rust programs. Our test showed the expected behavior, but complex programs where not tested because the implementation does not yet cover all necessary features. Yet, the general approach seems to be applicable and can be refined further to deal with complex scenarios.

6 Future Work

Although our approach produces Petri-Nets that are close to the rust semantics there is a lot of space to improve. First of all necessary flow related properties have to be modeled or emulated. On the one hand a mechanism for splitting execution flow has to be integrated. Primarily that means appropriate handling of threads, most likely by emulating the functionality of starting and joining them. In a Petri-Net that maps simply to a transition that produces a token on two separat places or consuming from two places respectively. On the other the model for guarding critical sections has to be refined further. While the Petri-Net representation here is simple, a sound concept for the rust side has to be found. Emulation of mutexes probably already catches a lot of scenarios but others can be found where this not suffice. For example low-level *no_std* environments there the mutexes from the standard library cannot be used. Additionally, the current implementation actively marks locals to distinguish between mutex instances while this probably can be inferred.

The currently used data model can be improved as well. Data that moves between locals or moves into or out of structures is currently modeled independently for every local, which masks its semantic connection. The movement might be modeled in a Petri-Net by separating the data from locals. The movement then indicates that the previous local cannot access the data anymore, quite similar to the rust ownership model. If this can be done close to the rust semantics it might already fix the problem with marking mutexes.

Given a solid translation model with a reasonable control flow emulation, more complex szenarios should be testet. This could include artificial ones like Dijkstra's dining philosophers [?]r real life programs. This would be critical to decide if the verification process is efficient enough to be used in authentic use cases. Such tests would also improve from more sophisticated verification results. Currently the witness path is only a chain of transition ids. But the MIR stores source file location information that could be linked with the corresponding Petri-Net nodes. It is likely that this information can be used to map the witness path to the original program source code. This would improve usability a lot.

A graph representation of the MIR might also help in the development process for MIR generation. For example the missing storage statements we talked about in chapter 4 left the initialized and dead place unconnected in the Petri-Net (which was excluded from the image). This is a graph property that can be verified and might indicate a bug. If more graph properties should be met by the MIR graph, they could be included into a test case to improve the development process.

And finally, lifting the model to high-level petri nets could be a solution to some intrinsic shortcomings (like the recursion restriction) and open the door to data dependent verification properties.

Bibliography

- [1] S. Klabnik and C. Nichols, The Rust Programming Language. No Starch Press, 2018. [Online]. Available: <https://web.archive.org/web/20190929000131/https://doc.rust-lang.org/book/>
- [2] R. Team. (2020) Rust website. [Online]. Available: <https://web.archive.org/web/20200119174448/https://www.rust-lang.org/>
- [3] N. D. Matsakis and F. S. Klock, II, “The rust language,” Ada Lett., vol. 34, no. 3, pp. 103–104, Oct. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2692956.2663188>
- [4] R. Team. (2019) Nomicon. [Online]. Available: <https://web.archive.org/web/20191013065400/https://doc.rust-lang.org/nomicon/>
- [5] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” Addison wesley, vol. 7, no. 8, p. 9.
- [6] C. Baier and J.-P. Katoen, Principles of model checking. MIT press, 2008. [Online]. Available: https://www.academia.edu/download/30717533/_principles_of_model_checking.pdf
- [7] K. L. McMillan, “Symbolic model checking,” in Symbolic Model Checking. Springer, 1993, pp. 25–60.
- [8] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model checking and the state explosion problem,” in LASER Summer School on Software Engineering. Springer, 2011, pp. 1–30.
- [9] C. A. Petri, “Kommunikation mit automaten,” 1962.
- [10] R. Team. (2020) Rust-lang project. [Online]. Available: <https://github.com/rust-lang/rust>
- [11] ——. (2020) Rustc guide. [Online]. Available: <https://rust-lang.github.io/rustc-guide/>
- [12] ——. (2020) Rustc documentation. [Online]. Available: <https://doc.rust-lang.org/nightly/nightly-rustc/rustc/>

- [13] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, “The petri net markup language: Concepts, technology, and tools,” in Applications and Theory of Petri Nets 2003, W. M. P. van der Aalst and E. Best, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 483–505.
- [14] E. Kindler, “The petri net markup language and iso/iec 15909-2: Concepts, status, and future directions,” Entwurf komplexer Automatisierungssysteme, vol. 9, pp. 35–55, 2006.

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