Universität Rostock

Traditio et Innovatio

Masterarbeit

Eine Petrinetzsemantik für Rust

Vorgelegt von:

Tom Meyer

Matrikel-Nr.: 8200839

Eingereicht am:

13. März 2020

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

Eine Petrinetzsemantik für Rust

Es wird ein allgemeiner Ansatz gezeigt Rustprogramme in ein Petrinetzmodel zu überführen. Die Übersetzung eines Beispielprogramms wird als Eingabe in einem Model-Checker verwendet um ein Deadlock zu finden, der durch mehrfaches blockieren eines Mutex verursacht wir. Ein Prototyp der Übersetzung ist in der Lage den Deadlock im Beispielprogram zu finden und einen Zeugenpfad zu generieren. Anschließend werden einige Vorschläge zur Verbesserung der gezeigten Übersetzung diskutiert.

A Petri-Net semantics for Rust

We show a general approach to translate a Rust program into a Petri-Net model. An example program is used as input for a model checker to find a contained deadlock. The deadlock is caused by locking a mutex multiple times. A prototype of our concept is able to find the deadlock from the example program and produces a witness path to the corresponding part of the translation. In the last part of this work we discuss how our approach can be improved further.

Betreuer: Prof. Dr. Karsten Wolf

Tag der Ausgabe: 27.09.2019
Tag der Abgabe: 13.03.2020

Contents

1 Introduction

Here is a simple Rust[1] program that stops execution before it can terminate successfully:

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

pub fn main() {
    let data = Arc::new(Mutex::new(0));
    let _d1 = data.lock();
    let _d2 = data.lock();
}
```

Listing 1.1: A deadlock!

The reason is a deadlock caused by locking a mutex twice without releasing it in the meantime. Rust is a language that is highly concerned with memory safety and concurrency[2] but the detection of deadlocks is explicitly excluded from the design[3, Chapter 8.1](for good reasons). Nevertheless, it could be invaluable to detect such a situation automatically. A proven method to do so is model checking[4] there we check a model of our program against certain properties.

In this work we will:

- develop a Petri-Net[5] semantics for Rust programs to serve as a model in chapter 3 (especially chapter 3.4) ,
- find a mutex semantics for our model in chapter 3.6
- detect the deadlock from listing 1.1 with a model checker in chapter 4.2 and 4,
- investigate the result and the shortcomings of our approach in chapter 4
- and show a list of improvements that could lift our approach to be usable in realistic use cases in chapter 8

But first, we take a look on some important concepts in the next chapter.

2 Background

Before we start there, are some basics that are useful to know about. In this chapter we introduce the formalism for our model and for our properties to verify. We also explain the ideas behind mutexes, deadlocks and compilers. And we begin our background chapter with a summary of the programming language Rust.

2.1 Rust

By trend programming languages can be divided into fast or safe[6]: either a language is used to produce highly optimized code that runs fast on the targeted system, or a language makes use of sophisticated safety features that prevent inconsistencies at the cost of runtime performance.

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” [7]. To ensure a minimal performance overhead Rusts runtime was kept small[1, Chapter 16.1] and features like garbage collection where neglected[1, Chapter 4]. Instead, safety issues are addressed with Rusts **ownership model**[2].

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. When handing over ownership it is lost for the previous owner (to unsure unique ownership). However, references can be ‘borrowed’ without loosing ownership. These borrows come in two flavors:

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 while a mutable borrow is active.

References that go out of scope are ensured to be deconstructed by Rusts **borrow checker**. Programs that do not 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[2] 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[3, Chapter 8.1] which we want to address with this work.

2.2 Deadlock and Mutex

Typically, deadlocks are a problem of concurrent systems where resources are shared or a section of a program must only be entered once at a time (a critical section). To assure sequential access those resources are often guarded by a locking mechanisms like Dijkstras semaphores[8] or the simpler form: a mutex. Mutex stands for mutual exclusion, and they are used for exclusive access to a critical section. If a process of a program wants to enter this section, it has to acquire a mutex lock. This is only possible if no other process currently has acquired the lock, otherwise the former process has to wait (or try later). If the critical section is left by the locking process, the lock has to be released to unblock waiting processes. If the locks are not released correctly it can lead to a situation where all process are waiting to acquire a mutex lock that will never be released and the whole execution comes to a halt: a deadlock. A deadlock situation often depends on nondeterministic behavior (for example process/thread scheduling or network communication) which can make debugging rather difficult. Therefore, having proof of deadlock absence can be powerful information especially in highly parallel systems.

2.3 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[9, 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[9, 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. The **Code Generation** phase, 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 resemble 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.

2.4 Verification

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. A **system is correct** if its specification satisfies all its properties [4, 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 for properties in 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, firstly the behavior of a system needs to be represented in a (name giving) model and secondly the properties that the system needs to fulfill have to be specified; Usually in a formal logic. Having both, all possible relevant model states are explored to verify that the given properties hold in all state. Unfortunately the amount of states is typically exponential in the system size; A phenomenon that is known as the **state explosion problem**[10, Introduction]. This is the most outstanding problem of model checking, but fortunately there are methods to weaken the impact of the explosion. Some major techniques are symbolic model checking, partial order reduction and abstraction refinement[11, Chapter 5]. However, the important information here is that model checking nowadays, can tackle systems with a realistic amount of states.

In this work, we use Petri-Nets as a formalism for the model and CTL* to express properties to check.

2.4.2 Petri-Nets

Petri-Nets where developed in the mid 1900s by Carl Adam Petri[5]. It is a formalism that is well suited to model concurrent behavior, since it does not model each state explicitly.

A Petri-Net is 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 the own kind. 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. A disabled transition is **dead** if there is no reachable state in that it is enabled again. Additionally the whole net is **dead** if every transition in the net is dead. In that case a **terminal state** is reached.

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 $W : F \to \mathbb{Z}$
- and an initial marking $m_0 : P \to \mathbb{Z}$

Other important definitions are:

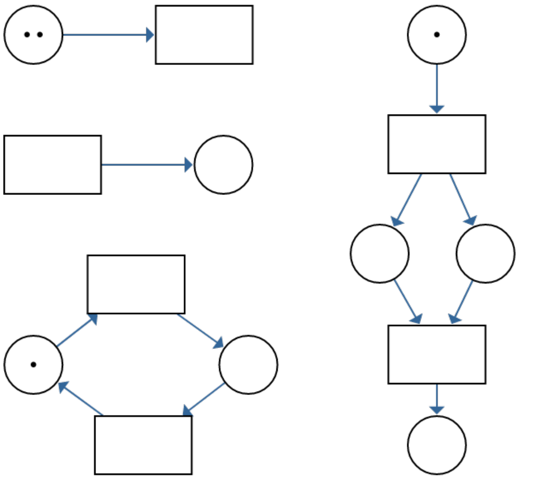


Figure 2.1: Petri-Net examples

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

Figure 2.1 shows four example nets. In the top left corner is a net with one place and one transition. The place is marked with two tokens and is the only place in the transitions preset. This means that the transition is enabled and can fire. Since every firing consumes a token (no edge annotations mean an edge weight of one), the transition can fire exactly two times. After that, the net will be dead. The net below looks similar, only the edge direction is flipped. Now the preset of the transition is empty. The semantics here is that all places in the preset are properly marked resulting in a transition that can always fire. As a result, the connected place can accumulate an unbounded amount of tokens. The third net in the bottom left corner has a circular shape. At any given time one of the two transitions can fire, virtually moving the token back and forth. The token count in this net has an upper bound of one since both transitions always consume as many tokens as they produce and it initial marking has one token. It also has no terminal state and can always fire one of the two transitions. And the last net on the right shows that tokens indeed are produced and consumed, not moved. The top transition will consume a token from the top place and produce two tokens, one on each place below. After that the second transition will consume both tokens (one token from each place in the preset) and produce a token on the bottom most place. It is not possible to consume only one of the two tokens – every place from the preplace is involved in the firing of a transition! And after the second transition has fired the net will be dead in a terminal state.

There is a large formal background to Petri-Nets that makes it possible to check for a variety of properties[12]. For example, it can be analyzed 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 Computational Tree Logic*

To articulate the properties we want to verify, we need a precise formalism for both state properties and properties of timing. However, low-level Petri-Net have no time associated with transition firing; Neither is there a defined duration between the firing of two transitions, nor a duration of a single firing. So in our context the concept of time is not concerned with duration, but with the order of events. Which transition fired before another; Or from which state can we reach another state.

These properties are commonly expressed in a **temporal logic**. The primary ones used in model checking are linear temporal logic (LTL) from Pnueli[13] computational tree logic (CTL) from Clarke and Emerson[14] and CTL* (the '*' is pronounced 'star') from Emerson and Halpern[15], a super set which includes both. Since we will later use CTL* to formulate our properties, we will introduce its formula construction here. A CTL* formula can be structured with the following elements:

- The most basic CTL* formula is an **atomic proposition** (AP) which describes an atomic part of the system state, hence it is a **state formula** Φ. In our Petri-Nets this is typically a marking of a single place like p_1 is marked with 5 tokens: $p_1 = 5$, or p_3 has less than 3 tokens: $p_3 < 3$.

- Every state formula is also a **path formula** φ which describe a sequence of events in a system. A plain state formula like from above describes the path to the initial state of the system. Such a formula is satisfied if the first state of the system satisfies the property (and makes no statement on the following states).

- Formulas can be combined to larger formulas with logic operators. For example: $!(p_1 = 5 \ \& \ p_3 < 3)$.

- The **Next** operator $X\varphi$ describes a path to the successor state. For example: $p_1 = 5$ is satisfied if p_5 is marked with 5 tokens in the first state, $X(p_1 = 5)$ is satisfied if p_5 is marked with 5 tokens in the second state, $XX(p_1 = 5)$ is satisfied if p_5 is marked with five tokens in the third state and so on.

- The **Eventually** operator $F\varphi$ describes a path formula that is satisfied if the enclosed formula is satisfied in some successor state or – to formulate it differently: if it is reachable from the formula it is applied on.

- The **Always** operator $G\varphi$ describes a path formula that is satisfied if the enclosed formula is satisfied in every successor state – or if it is always satisfied in the enclosed formula.

- The **Until** operator $\varphi_1 U \varphi_2$ describes a path formula that is satisfied if the formula φ_1 is satisfied in every state until a state is reached which satisfies φ_2 at least once. For example there is always an enabled transition in a circle (of the form $t_1 \to p_1 \to ... \to t_n \to p_n \to t_1$) as long as at least one of the places is marked with a token. But it can be marked again later if all tokens where lost.

- And finally there are the **Path quantifiers** $E\varphi$ and $A\varphi$ which can be used to make a statement for the branching behavior of the system. Depending on a given state there might be multiple possible following states. In Petri-Nets that means there are multiple enabled transitions that can fire next. So depending on the actually firing transition, the resulting sequence of following states (the paths) can differ. If the **Universal path quantifier** $A\varphi$ is applied on a path formula the resulting formula is only satisfied if the path formula is satisfied in **all** branching paths. In contrast, the **Existential path quantifier** $E\varphi$ only requires the associated path formula to be satisfied in **one** of the successive paths.

The AP's and operators can be combined to express complex properties which serve as an input for model checkers.

3 Approach

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

In this chapter we will introduce an idea to translate Rust code into Petri-Nets. We will see that the Rust compiler creates an intermediate representation that fits our needs, we will work out a translation for each of the elements in this intermediate representation, and discuss the format that we will translate into. Finally, we will show how we can use a model checker to find a deadlock in a simple test program.

3.1 Rust Compiler

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

1. write a translator from scratch,
2. or use something existing.

Writing an own translator means total control over the process. Features can be added iterative as needed and data structures can be designed efficiently for our 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[16], with a maintained documentation[17][18]. If we want to use this compiler we have to learn how it works and which concepts are important. This can be very time-consuming and difficult, but it seems to be time worth spend if we are able to use difficult features right away. For this reason this project is based on the Rust compiler *rustc*. So let's see how its basic structure looks like.

The Rust compiler has several phases and a variety of intermediate representations [17, Chapter 2.1]:

1. In the first phase Rust source files are parsed to an abstract syntax tree (**AST**) that matches the original Rust syntax closely.
2. In the second phase implicit information is expanded. This includes 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 normalized so that code analysis is easier. For example for loops are rewritten to simple endless loops with break conditions in if-statements.
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 are leaving the tree structure and switching to a graph. MIR is based on a control-flow graph[19] and 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 like Rusts borrow checking as well as further optimization is done on the MIR level.
6. Phase six lowers the MIR to the Rust independent LLVM[20] intermediate representation. LLVM IR is a low level representation that is close to assembly language. Additional optimization are done for this representation and the result is translated into several object files.
7. Finally, in the last phase the object files are linked into a complete binary executable.

This leaves us with several options to intercept and translate the current intermediate representation into Petri-Nets.

3.2 Interception strategy

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 most existing compiler features and minimizes the own translation effort.

We want to intercept after basic features like name resolution as we need them ourselves and surely won't improve them in a reimplementation. Also, code generation, including macro and generics expansion, should be handled by the stock compiler. They just produce more Rust code that we will treat anyway. Additionally, after expansion, no code generation syntax will remain, so we do not have to deal with it at all.

More optional for our use are the static analysis features like type checking and borrow checking. Though, we want to use their assumptions, we do not depend on them actually being checked, since nobody will ever run a Rust program that did not go through the full 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 analysis anyway and abort on errors. This prevents time-consuming runs for programs that cannot be build.

A less optional compiler feature that will greatly ease our effort, is the lowering of the representation. This way we can exploit that several high level language features are reduced to significantly less low level features. Which, in turn, means fewer 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 from the higher level representations. We also probably want to avoid machine dependent representations. After all we want to verify generic Rust programs and not the peculiarities of a single machine. This might be debatable though.

Finally, we want to make use of all optimizations we can. Especially optimizations that reduce the code (and net) size. So features like constant propagation and dead code elimination would be nice to have to reduce the impact of state explosion.

Reminding the Rust compiler phases with those thoughts, the best place we can intercept is between phase five and six: after borrow checking and optimizations on the MIR. Here, all code was expanded, all Rust specific static analysis and optimization was done and all unnecessary language features where reduced to a minimal set of instructions. And since both MIR and Petri-Nets are graph representations, a mapping should be relatively straight forward. More so if we consider that MIR represents control flow quite closely. Also, we avoid the even 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. Here is a simple Rust program:

```rust
pub fn main() {
    let x = 5;              // x is an integer
    let _y = get_result(x); // call a function with x
}
// a function that takes an integer and returns an integer
pub fn get_result(x: usize) -> usize {
    // check the value of x
    match x {
        1 => 1, // if it is one: return one
        2 => 2, // if it is two: return two
        _ => 5, // if it is something else: return 5
    }
}
```

Listing 3.1: A Rust program. And there is no deadlock!

And this is a textual representation of the corresponding MIR generated by the compiler:

```
fn  main() -> () {
    // Declaration of locals.
    let mut _0: ();     // the return local: no return => null-tuple
    let _1: usize;      // local _1 has type usize
    let _2: usize;
    let mut _3: usize;  // local _3 is mutable so it can change its value
    bb0: {                  // BasicBlock 0: execution starts here
        StorageLive(_1);    // initialize local
        _1 = const 5usize;  // assign constant value 5 to local _1
        StorageLive(_2);
        StorageLive(_3);
        _3 = _1;            // assign value of local _1 to local _3
        // Call function get_result with local _3 as argument
        // _3 moves into the scope of the function.
        // After that it can not be used in this scope anymore.
        // Continue execution in BasicBlock1.
        _2 = const get_result(move _3) -> bb1
    }
    bb1: {               // BasicBlock1
        StorageDead(_3); // local is not used again in this function
        StorageDead(_2);
        StorageDead(_1);
        return; // return from main, program finished
    }
}
fn  get_result(_1: usize) -> usize {
    // Declaration of locals.
    // The argument _1 is an additional local that can be used.
    let mut _0: usize; // same return type as in the function declaration
    bb0: {  // BasicBlock 0: execution starts here
        // Continue depending on the value of _1.
        // If it is one continue in BasicBlock2.
        // If it is two continue in BasicBlock3.
        // otherwise continue in BasicBlock1
        switchInt(_1) -> [1usize: bb2, 2usize: bb3, otherwise: bb1];
    }
    bb1: {
        _0 = const 5usize; // assign the return value
        goto -> bb4;       // continue in BasicBlock4
    }
    bb2: {
        _0 = const 1usize;
        goto -> bb4;
    }
    bb3: {
        _0 = const 2usize;
        goto -> bb4;
    }
    bb4: { return; } // return to the calling function
```

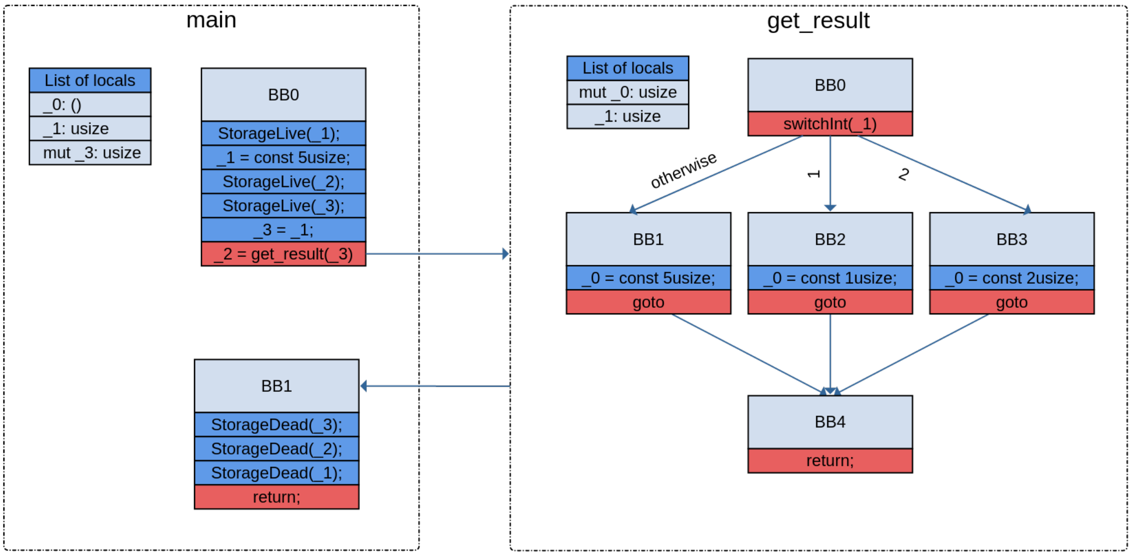


Figure 3.1: Graphical MIR from listing 3.2.

```
}
```

Listing 3.2: Generated MIR to program from listing 3.1

As mentioned before, MIR is derived from control flow graphs[17, chapter 2.17]. Consequently, it can also be represented graphically as shown in figure 3.1. It consists of several **basic blocks** which are interconnected with directed edges. Each basic block consist of any amount of non branching **statements** (dark blue in figure 3.1) and a single – possibly branching – **terminator** (red in figure 3.1). 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 introduce 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 locals are conceptually always located on the stack. This means that a location can also be represented as a place, but places are not necessarily locations. In our example all places are also locals.

Statements work on these data representations. The most prominent type of statement is the assignment that assigns an **rvalue** derived from an expression to a memory location – meaning a place (the **lvalue**). For example the second statement in *BB0* from *main* is an assignment statement that assigns the constant value *5* (a unary expression with only one constant operator) to local *_1*. All other statements in our example are *StorageLive* and *StorageDead* statements. Locals cannot be used before they were set live or after they were set dead – except for the return local (always local *_0*) and function arguments (always the first locals after local *_0*). They are logically linked to locals in the calling function where the initialization was already handled.

Terminators can direct control flow for example with a function-call-terminator (like the one in *BB0* of *main*. These redirect control flow to a subroutine that is executed until a return-terminator is hit. This one marks the end of a function and let the flow continue in the calling function. The return terminator in the main function marks the end of the program and results in its successful termination.

Terminators can also split control flow as the *SwitchInt* terminator in *get_result* shows. This terminator redirects control flow depending on the value of local *_1*. In the example this results in three possible paths in which the function can be traversed. All of which join in a basic block with a single *return* terminator.

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 cases we have to consider.

3.4.1 Entry Point

We will start with the most abstract view on our translation: the whole program. A program is something that typically has a beginning and an ending. We can model this with a **program start** and a **program end** place. Depending 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 the end place will not be connected to the graph (which will have no negative effect on the verification process). 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 distinguish 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). Figure 3.2 shows the three resulting places.

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. In the runtime static memory and language specific features are initialized (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 of the actual program semantics. And there is another problem in starting

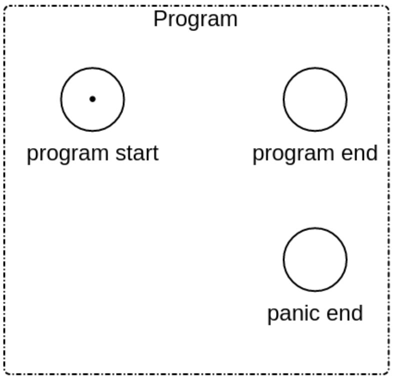


Figure 3.2: Features that every program shares: the start and end places.

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 but it complicates the translation process unnecessarily. Additionally, we previously already argued for a platform independent approach 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 at the main method.

3.4.2 Functions

The next important parts of a program are the executed 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 Petri-Net function model would be of no help, leaving just the start place as a feasible similarity between functions as pictured in figure 3.3: 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.

Now we can almost model the first complete program: the empty program:

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

Listing 3.3: The empty program

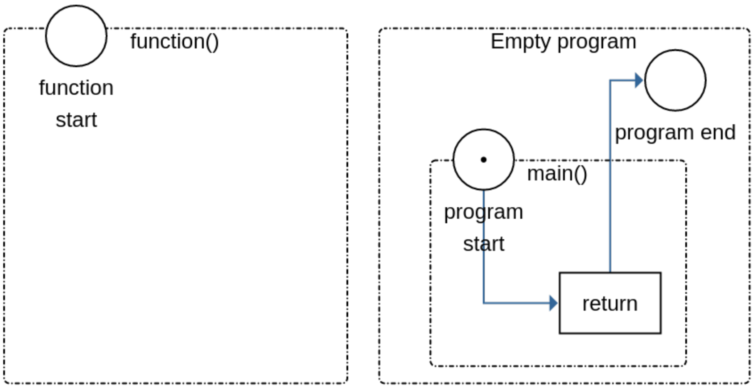


Figure 3.3: A conceptual view on a function on the left. The Empty program on the right.

The translated Petri-Net for the empty program can be seen in figure 3.3. Its program start place is also the start place of the main function. We anticipate the return terminator here (discussed in chapter 3.4.6) to connect program start and program end place. The result is a net that consumes a token from the program start place and produces a token on the program end place. Afterwards the net is in its terminal state.

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 not obviously) to get the MIR from a whole program. Just function per function. 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 basic blocks. Local variables are also indexed from zero on for each function, with some of them having a special meaning: the first local is always the functions return value and it is followed by locals 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 with 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 after we are done. In a 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, the translation process 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 properly. 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 a recursive function is just pushed on a new stack frame again and again, until it can be resolved (the base case can be called for all recursive levels) 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 (this would mean a dynamic set of places in a static graph). All memory we have is the memory 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 depth. 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 execution 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 then: modeling every state of every possible variable would bloat our net terribly.

To stay in the realm of low-level nets we need to abstract from variable states. In principle a Petri-Net place is already something that holds data: a set of tokens. For our purposes we can interpret a place as a memory location and a token as arbitrary data. Though we loose the information for the actual variable value, we still can analyze 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. 'Reading' from and 'writing' to memory can be modeled with a transition that consumes the token, while also producing a token. In theory the consumed and produced token in a read has to be the same while writing can change the token. But again this is only relevant for high-level nets.

Now, in many programming language there are different concepts of memory that behave differently, and Rust is no exception. Basically we can distinguish between four different concepts[1, Chapter 3.1, Chapter 4.1, Chapter 19.1]:

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 in 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 to 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**: where 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 can be reused later. Both allocation and deallocation are on demand, which 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 (constraint by its type) 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 these states with three Petri-Net places for each local as shown in figure 3.4: one place for the uninitialized state, one for the living state there 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

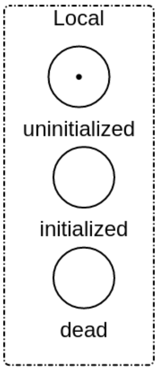


Figure 3.4: Places of a local in a function.

of a struct, which index of an array or if we dereference the local. This information is stored implicitly as MIR-place on which statements can operate. Unfortunately this projection is hard to impossible 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 a projection is used in a statement we can interpret this as access to the initial local. Again we loose some information here in exchange for a manageable design.

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. There 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 annotated with the corresponding constant values.

In conclusion, we need two models for memory access: a single place that represents the whole constant memory space and a set of places for each local variable with uninitialized, live, and dead place. If we have to handle heap space, we can hide the access behind a local variable (ignoring the projection).

3.4.4 Basic Blocks

A basic block 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 will again start at the entry point of 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 block's entry place corresponds to the first statement's entry place and the basic block's exit place

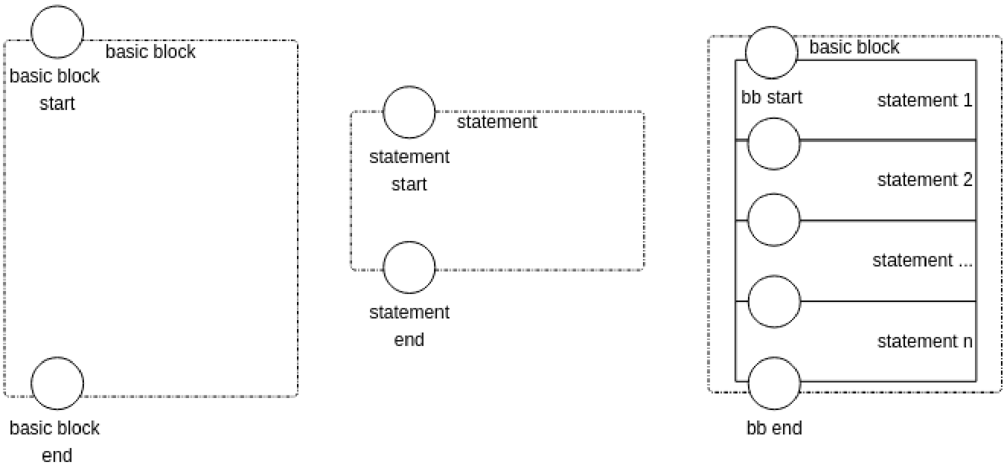


Figure 3.5: The structure of basic blocks and statements.

corresponds to the last statement's exit place. The terminator is modeled by one or more transitions that consume the token from the end place and produce a token on another basic blocks start place.

3.4.5 Statements

Statements are the part of the MIR graph that change data. And there are several kinds of them, with some of them not used in every compiler phase. For example a MIR statement named 'FakeRead' is used for static analysis but is removed in a later optimization phase when the sanity was satisfied.

However, the general structure of statements, 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 statements end place will be the seconds statements 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. Figure 3.5 illustrates this connection. 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 a token on data places to resemble an interaction with 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[1]: 'StorageLive', 'StorageDead' and 'Assign'.

[1] A complete list can be found in the documentation:

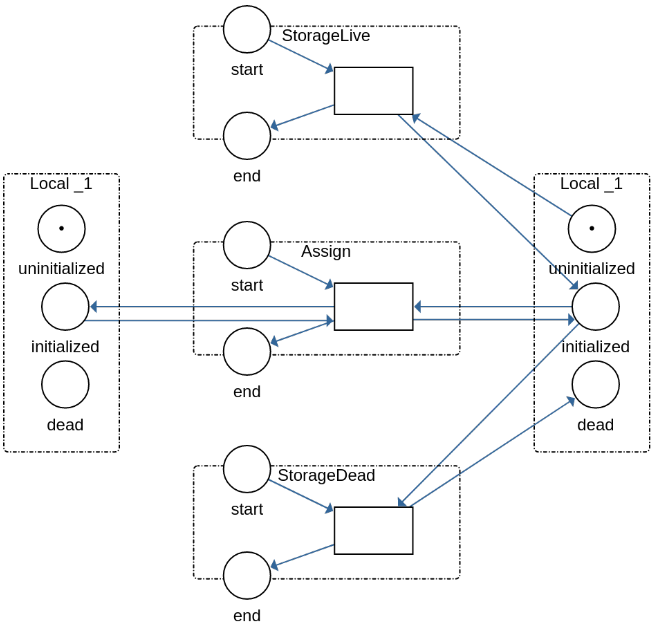


Figure 3.6: Three kinds of statements (unconnected)

- 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 rvalue 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 a statement's 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, 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. A property that is very close to a *program counter* in CPU's which stores the address of the instruction that will be

https://doc.rust-lang.org/nightly/nightly-rustc/rustc/mir/enum.StatementKind.html executed next. This next instruction might just increment the counter (in sequential parts) to mark the subsequent instruction as active or manipulate it to jump to a totally different instruction. This is a convenient similarity since it increases confidence that the Petri-Net actually models something similar to an execution semantics of a real program.

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 blocks end place and produce a token on a basic blocks start place.

The MIR representation again has different kinds of terminators[2]. Here is a list with the most important ones:

- The most basic *Goto* terminator has a single transition. It connects two basic blocks inside a single function.
- The *SwitchInt* terminator implements a conditional branch to multiple other basic blocks inside a single function. A branching path can only be taken if its condition is met. Only the last branching path acts as an *otherwise* branch. It will be taken if there is no other valid path where the condition is met. Since 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. In that case 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

[2] A complete list can be found in the documentation: https://doc.rust-lang.org/nightly/nightly-rustc/rustc/mir/enum.TerminatorKind.html

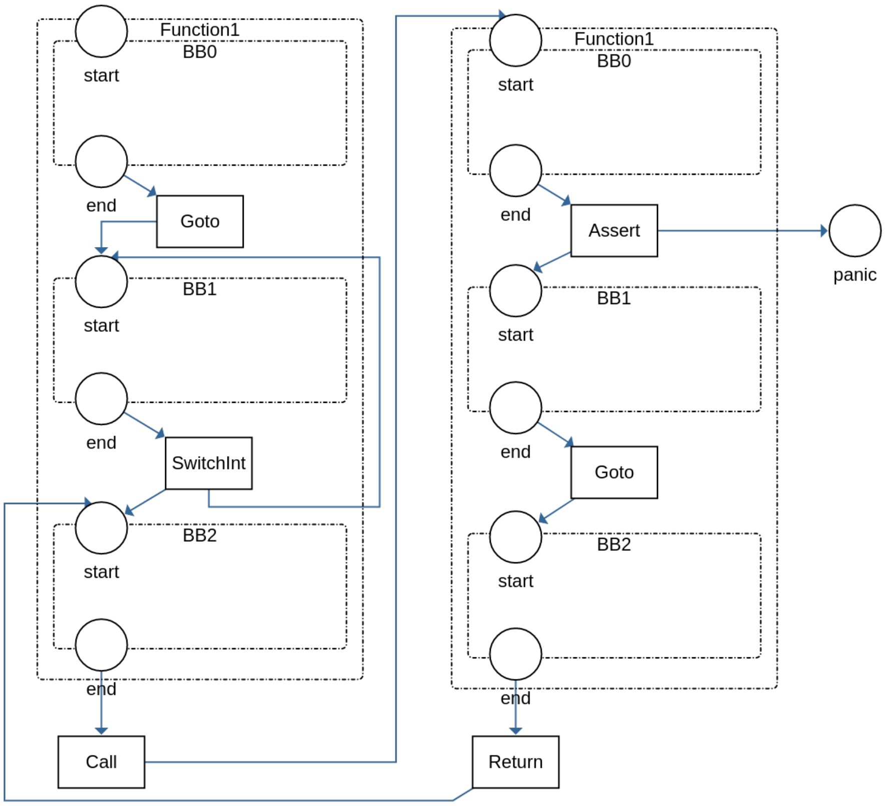


Figure 3.7: Connection of terminator transitions.

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.

With an entry point, basic blocks, a representation of locals and constants, statements and terminators we can build a more complex program. Consider this simple program with a function call:

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

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

Listing 3.4: A simple function call

The generated MIR for listing 3.4 (shown in listing 3.5) is a bit more complex. Not only because of the *StorageLive* and *StorageDead* statements, but also because of additional copies of locals. Additionally, both *main* function and *call* function are split into two basic blocks. The *main* function has to be split because the function-call-terminator redirects control flow to another function. And the *call* function has to be split because of a multiplication that might overflow. An assert-terminator guarantees that this behavior is detected and treated.

```
fn  main() -> () {
    let mut _0: ();
    let mut _1: usize;
    let mut _2: usize
    let mut _3: usize
    bb0: {
        StorageLive(_1);
        _1 = const 5usize;
        StorageLive(_2);
        StorageLive(_3);
        _3 = _1;
        _2 = const call(move _3) -> bb1;
    }

    bb1: {
        StorageDead(_3);
        _1 = move _2;
        StorageDead(_2);
        StorageDead(_1);
        return;
    }
}
fn  call(_1: usize) -> usize {
    let mut _0: usize;
    let mut _2: usize;
    let mut _3: (usize, bool);
    bb0: {
        StorageLive(_2);
        _2 = _1;
        _3 = CheckedMul(move _2, const 2usize
        assert(!move (_3.1: bool), "attempt to multiply with overflow"
) -> bb1
    }
    bb1: {
        _0 = move (_3.0: usize);
        StorageDead(_2);
        return;
    }
}
```

Listing 3.5: Generated MIR for a simple function call

Figure 3.8 shows how the translated net should look for the *main* function. The program starts in *BB0*, where all locals are set *StorageLive* and the value of local *_1* is set to 5. At the end of the basic block, the subroutine is called with local *_2* as return value and local *_3* as an argument. To represent that their values might change in the subroutine they are connected with it. After the call returns, its return value is assigned to local *_1* and all locals are set *StorageDead*. Only the panic place is unconnected since there is no panic path generated in the MIR representation.

But it could as well have been. And since this is an important part of program execution it demands a closer look.

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 ungracefully.

Rust has a different approach: normally functions are expected not to fail. Functions that can fail will return a *Result* type that can hold 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 unwound 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 generates 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 modeled this event with the separate ‘panic end’ place in our Petri-Net. 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 awful errors

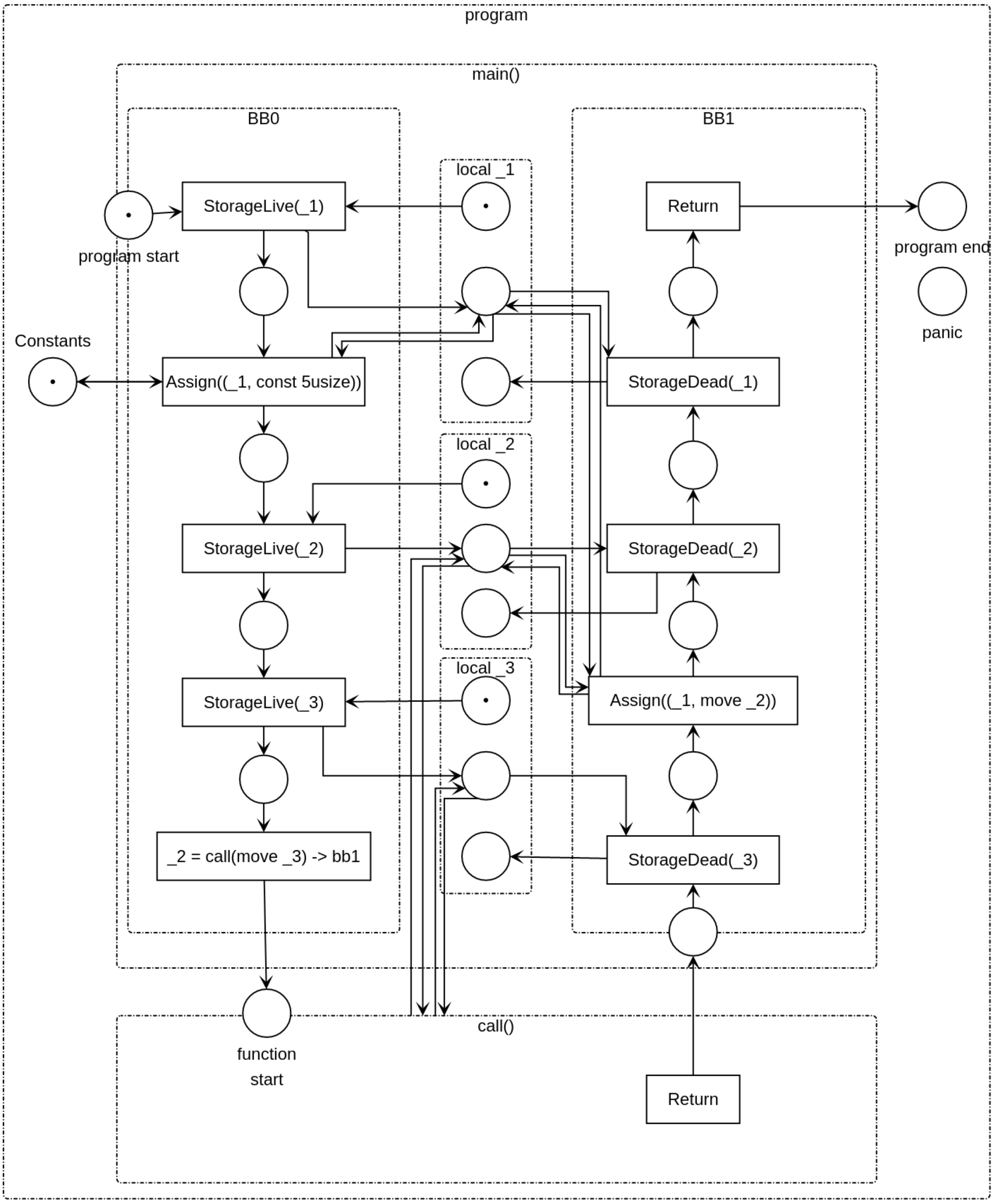


Figure 3.8: Main function for listing 3.4

.

like dereferencing pointers to inaccessible 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 memory 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 implementer. 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 languages is always a desirable feature to have. Unsurprisingly, Rust implements some interfaces to access functionality that is not native to itself. The two most important ones are compiler *intrinsic functions* where the implementation is hidden by the compiler; And the *Foreign Function Interface* (FFI) 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. So, if we use threads, we use foreign code. 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.

A generic FFI-call can be thought of as a combination of a statement and a terminator. The locals for the arguments have to be accessed by a transition like we did for statements. Afterwards we branch depending on the return value, like we did in the terminators. This model is sufficient to 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 that are vital 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 as figure 3.9 shows.

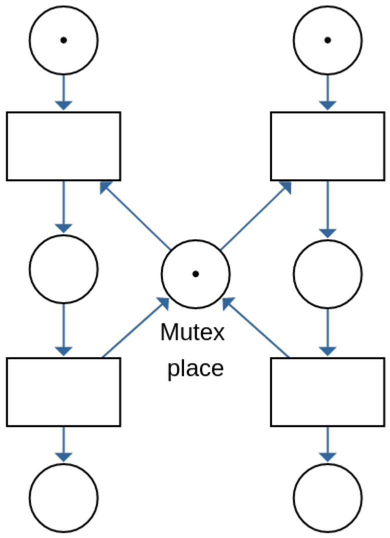


Figure 3.9: Two flows guarded by a mutex.

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 preplaces 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 3.6 shows an example usage of a mutex.

```rust
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.
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
    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 3.6: Mutex in Rust

Now, the call to mutex *lock()* and the release of 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 way that ignores the implementation. When a mutex is called we will stall normal translation 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 3.7, 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, that 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 by the mutex guard in local *_3*.

```
fn main() -> () {
    let mut _0: ();
    let _1: Arc<Mutex<i32>>;
    let mut _2: Mutex<i32>;
    let mut _4: &Mutex<i32>;
    let _5: &Mutex<i32>;
    let mut _6: &Arc<Mutex<i32>>;
    scope 1 {
        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 3.7: 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 in our model; 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 (in a separate data structure). Locking and unlocking mutexes then just check the marking 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 terms of storage consumption.

4 Demonstration

After developing a translation concept, it is time to use it in practice. In this chapter we will discuss how to develop the deadlock property we want to check and the results of our verification run. Furthermore, we examine the used net representations and the test programs that were used in the development process.

4.1 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**[21][22] 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**[23], 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 were all integrated as target representation in our prototype. And with a finished translation we can finally feed a model checker with our Petri-Net.

4.2 Model Checking

To test and inspect our results we have the choice between different tools. An inspiration of performant tools like TAPAAL[24], ITS-tools[25] or LoLA[26][27] can be found in the results of the 'Model Checking Contest'[28]. 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 to a dead Petri-Net where no transition is enabled 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 there execution stops. This means that if we reach either the *program end* or the *panic end* 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 keyword $DEADLOCK$. So the state we want to discover would be $\Phi = DEADLOCK\ \&(program\_end = 0\ \&\ panic = 0)$. 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 operators $EF\varphi$ in combination. Its semantic is that the given property is satisfied in any of the successive paths at some point (or state). So our final formula would look like this:

$$\Phi = EF(DEADLOCK\ \&(program\_end = 0\ \&\ panic = 0))$$

And having that, lets use it in a test program.

4.3 Test Programs

If we want to get some confidence in our translation process we have to see if it behaves as expected. Initially, the basic functionality should be tested against the simplest programs to fail early. And what could be simpler than the empty program from listing 3.3? Beside giving a good starting point for a testable implementation, 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 our simple function call from listing 3.4 or an endless program (which actually is completely deadlock free):

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

Listing 4.1: An endless program

However, non of these programs are significant for what we really want to achieve: deadlock detection. For this, we use our very first example program from listing 1.1. If our Petri-Net model is worth something the model checker should detect a deadlock for this program and none if we remove the last line (lock the mutex only once).

4.4 Translation target

In figure 4.1 we can see the generated net for the function call program from listing 3.4 (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 where rearranged so that the called function is on the left and the main

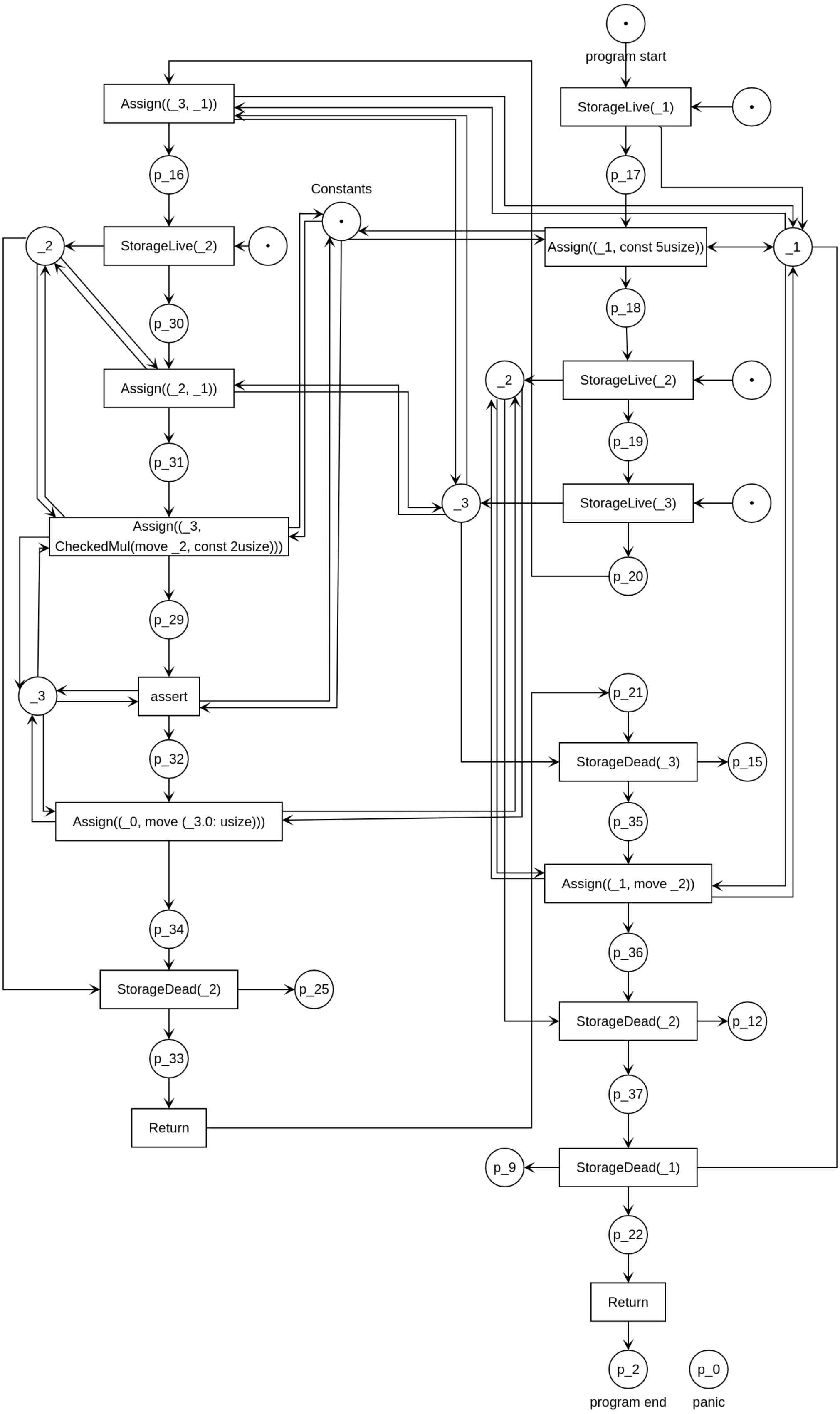


Figure 4.1: Translated Petri-Net

function on the right. The initialized places of the locals where renamed to show their MIR 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. In contrast, variable manipulation always has parallel incoming and outgoing edges. On a closer look we can see that local _3 of the called function has no storage live or storage dead 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[1] has yet to be solved. Unfortunately this behavior introduces an unintended deadlock into our translation since depending transitions can only fire if the initialized places where 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* places where marked as well. 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.

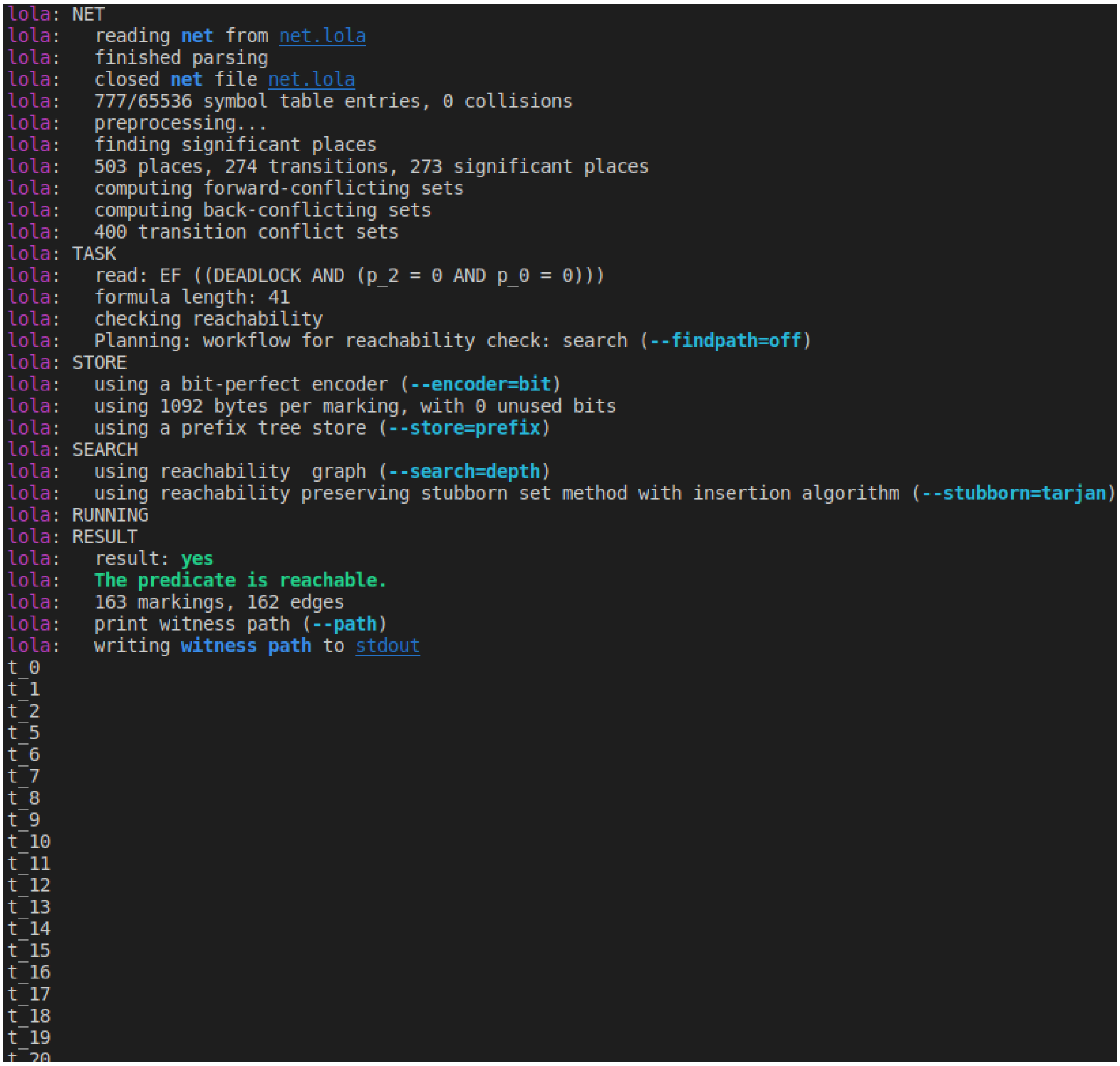
A second detail that the net shows is that the function call transition is implicit; The last statement of the main functions first basic block (*StorageLive(_3)*), is directly connected to the first statement of the callees first basic block (*Assign(_3, _1)*). This is an implementation detail of the 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 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.

4.5 Verification results

Of course we want to use our translation further for verification. Using our formula on the deadlock program from listing 1.1, 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

[1] https://github.com/rust-lang/rust/issues/67400



```
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

program, no deadlock is detected. Following the witness path in the deadlock case, 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 discussed in chapter 3.6 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 another formula can be found, to verify that leaving the waiting section is always possible. But again this most likely would require to consider the values that variables can hold and therefore high-level semantics.

5 Evaluation

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 some lessons learned that we can discuss.

5.1 The Model

The main draw back that followed us for the entire process is the abstraction of data in low-level Petri-Nets. Advantages of the low-level model are not only 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 workaround for compensating the missing storage-statements, the token count on every place cannot exceed a maximum of one. 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 moves into another local (like a field of a struct), we completely loose this information in our translation. Although, we can probably exploit Rusts 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 a move of a value from one local to another (which is encoded in MIR with a keyword), could be to connect the places with a transition right away.

Another disadvantage of our current model is function inlining. 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 catches the translation process in an endless loop 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.

5.2 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. Unfortunately virtually every program with a realistic size can panic. So this property is 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 data values.

6 Related work

Rusts design principles strongly include memory safety and other safety properties. And there is an effort in the language community to formalize and proof these properties. A core part of Rusts memory management was modeled in a formalism named Patina by Reed[29] in 2015. Patinas statements satisfy memory safety properties like initialization before use or aliasing bounds for mutable memory. λ_{Rust} by Jung et al.[30] extends safety statements to unsafe code (where the Rust borrow checker does not enforce its strong rules) and was verified to hold the formulated safety guarantees. Recently Jung et al. published another approach to minimize undefined behavior (where compiled code can be unpredictable due to different compiler implementations) in unsafe code. These are important approaches to proof the guarantees that the language claims to give. However, guarantees outside these boundaries have to be verified by other means. Besides regular methods like unit and integration tests there is a model checking effort by Toman et al. [31] to give further memory safety guarantees especially on unsafe code.

Despite Petri-Nets models are seemingly not used for verification of traditional programming languages there was some effort to model general concurrent programs with the Basic Petri Net Programming Notation B(PN)^2 by Best et al. [32]. They used multilabled nets (M-nets)[33], a class of high-level Petri-Nets for their approach. Fleischhack et al. extended B(PN)^2 with procedures – including recursion[34]. There is also research on Petri-Net semantics for description languages like the commonly used Specification and Description Language (SDL)[35] (also based on M-nets) or the Business Process Execution Language for Web Services (BPEL)[36][37]. Both are used to verify properties of processes that are formulated in their description language. Also, the π-calculus is backed by a Petri-Net semantics[38] based on low-level Petri-Nets with inhibitor arcs (inhibitor arcs require the connected preplace to be empty to activate a transition).

7 Conclusion

The main goal of this work was finding a mapping from Rust programs to Petri-Nets. A translated net then was intended to be used 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 cover all necessary features. Yet, the general approach seems to be applicable and can be refined further to deal with complex scenarios.

8 Future Work

Although our approach produces Petri-Nets that are close to the Rust semantics, there is a lot of space to improve. First, 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 spawning and joining them. In a Petri-Net that maps simply to a transition that produces a token on two separate places or consuming from two places respectively. On the other hand, 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 where 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, in turn, masks its semantic connection. The movement might be modeled in a Petri-Net by separating the data from locals. A move 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 model with a reasonable control flow emulation, more complex scenarios should be tested. This could include artificial ones like Dijkstras dining philosophers [39] or real life programs. This would be critical to decide if the verification process is efficient enough to be used in authentic use cases. Test analysis 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 compiler 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] 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

[3] Rust-Team. (2019) Nomicon. [Online]. Available: https://web.archive.org/web/20191013065400/https://doc.rust-lang.org/nomicon/

[4] 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

[5] C. A. Petri, "Kommunikation mit automaten," 1962.

[6] S. Klabnik. (2019) Rust, webassembly, and the future of serverless. [Online]. Available: https://youtu.be/CMB6AlE1QuI?t=369

[7] Rust-Team. (2020) Rust website. [Online]. Available: https://web.archive.org/web/20200119174448/https://www.rust-lang.org/

[8] E. W. Dijkstra, "Cooperating sequential processes," in The origin of concurrent programming. Springer, 1968, pp. 65–138.

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

[10] K. L. McMillan, "Symbolic model checking," in Symbolic Model Checking. Springer, 1993, pp. 25–60.

[11] 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.

[12] T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[13] A. Pnueli, “The temporal logic of programs,” in 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). IEEE, 1977, pp. 46–57.

[14] E. M. Clarke and E. A. Emerson, “Design and synthesis of synchronization skeletons using branching time temporal logic,” in Workshop on Logic of Programs. Springer, 1981, pp. 52–71.

[15] E. A. Emerson and J. Y. Halpern, “Decision procedures and expressiveness in the temporal logic of branching time,” Journal of computer and system sciences, vol. 30, no. 1, pp. 1–24, 1985.

[16] Rust-Team. (2020) Rust-lang project. [Online]. Available: https://github.com/rust-lang/rust

[17] ——. (2020) Rustc guide. [Online]. Available: https://rust-lang.github.io/rustc-guide/

[18] ——. (2020) Rustc documentation. [Online]. Available: https://doc.rust-lang.org/nightly/nightly-rustc/rustc/

[19] F. E. Allen, “Control flow analysis,” in Proceedings of a Symposium on Compiler Optimization. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19. [Online]. Available: https://doi.org/10.1145/800028.808479

[20] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004, pp. 75–86.

[21] 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.

[22] 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.

[23] E. Koutsofios and S. C. North, “Drawing graphs with dot,” 1996.

[24] Aalborg-University. (2019) Tapaal model checker. [Online]. Available: http://www.tapaal.net/

[25] Y. Thierry-Mieg, “Symbolic model-checking using its-tools,” in Tools and Algorithms for the Construction and Analysis of Systems, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 231–237.

[26] Universität-Rostock. (2019) Lola model checker. [Online]. Available: http://service-technology.org/lola/

[27] K. Schmidt, "Lola a low level analyser," in International Conference on Application and Theory of Petri Nets. Springer, 2000, pp. 465–474.

[28] (2020) Model checking contest. [Online]. Available: https://mcc.lip6.fr/

[29] E. Reed, "Patina: A formalization of the rust programming language," University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02, 2015.

[30] R. Jung, J.-H. 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, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3158154

[31] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for rust (n)," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015, pp. 75–80.

[32] E. Best and R. P. Hopkins, "B(pn)2 - a basic petri net programming notation," in PARLE, 1993.

[33] E. Best, H. Fleischhack, W. Fraczak, R. P. Hopkins, H. Klaudel, and E. Pelz, "A class of composable high level petri nets," in International Conference on Application and Theory of Petri Nets. Springer, 1995, pp. 103–120.

[34] H. Fleischhack and B. Grahlmann, "A petri net semantics for b (pn)/sup 2/with procedures," in Proceedings of PDSE'97: 2nd International Workshop on Software Engineering for Parallel and Distributed Systems. IEEE, 1997, pp. 15–27.

[35] ——, "A compositional petri net semantics for sdl," in International Conference on Application and Theory of Petri Nets. Springer, 1998, pp. 144–164.

[36] C. Stahl, A Petri net semantics for BPEL. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät . . . , 2005.

[37] N. Lohmann, "A feature-complete petri net semantics for ws-bpel 2.0," in International Workshop on Web Services and Formal Methods. Springer, 2007, pp. 77–91.

[38] N. Busi and R. Gorrieri, "A petri net semantics for π-calculus," in International Conference on Concurrency Theory. Springer, 1995, pp. 145–159.

[39] E. W. Dijkstra, "Hierarchical ordering of sequential processes," in The origin of concurrent programming. Springer, 1971, pp. 198–227.

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, 13. März 2020

Tom Meyer