Compile-time Deadlock Detection in Rust using Petri Nets

Horacio Lisdero Scaffino

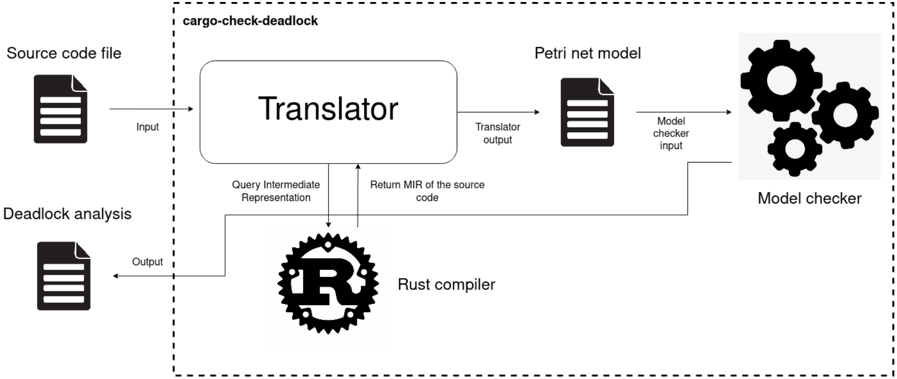
October 9, 2025

Agenda

Agenda

A bird's-eye view of the tool

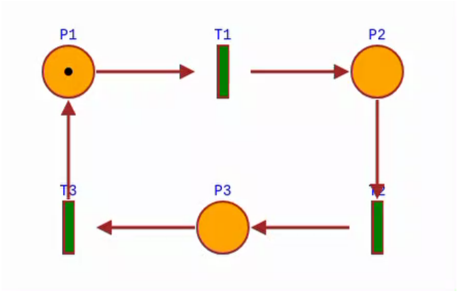
The Translator is the core component. The model checker and the Rust compiler, *rustc*, are dependencies.



Agenda

Informal definition

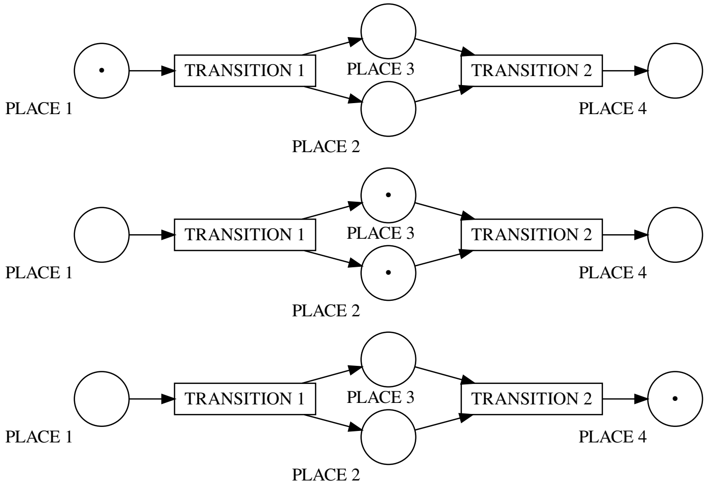
A Petri net is a mathematical modeling tool used to describe and analyze the behavior of concurrent systems. It is a directed, bipartite graph with a *marking*, i.e., an initial assignment of tokens to places.



- Places: Represent states in the system (*circles*)
- Transitions: Represent usually events or actions that occur in the system (*rectangles*)
- Tokens: Marks inside of places that are created and consumed by transitions (*points inside of places*)

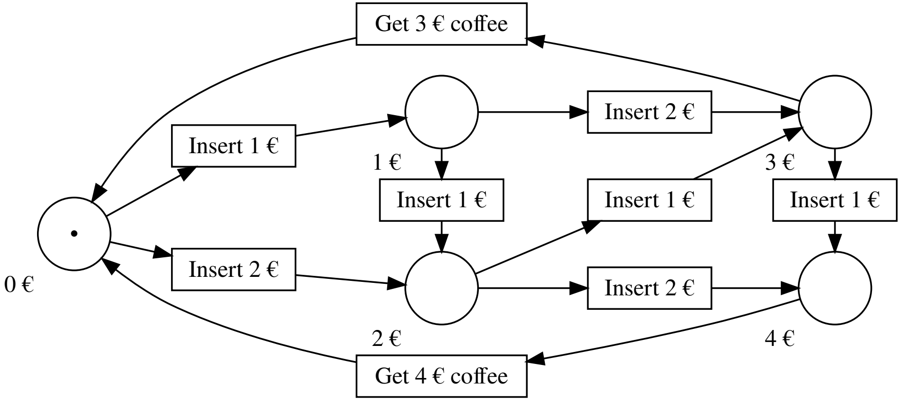
Transition firing rule

A transition may only fire if it is *enabled*: All places pointing to it have a token.



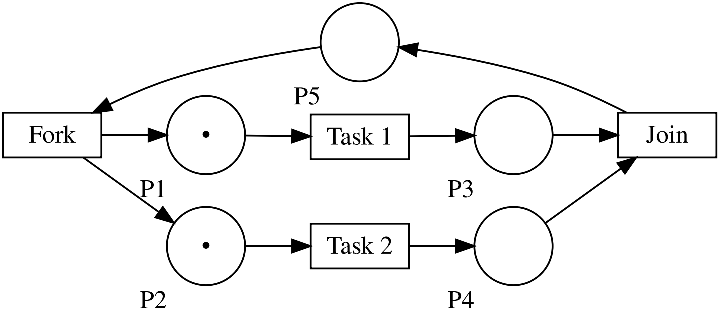
Vending machine

This is a finite-state machine (FSM), a subclass of Petri nets.



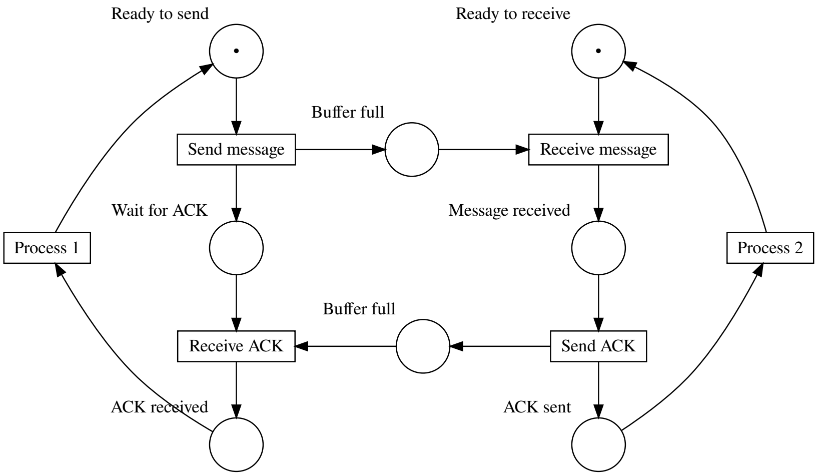
Parallel activities: Fork/Join

This is a marked graph (MG), a subclass of Petri nets. Observe the concurrency between Task 1 and Task 2. This cannot be modeled by a single finite-state machine.



Communication protocols: Send with ACK

A simple protocol in which Process 1 sends messages to Process 2 and waits for an acknowledgment to be received before continuing. For simplicity, no timeout mechanism was included.



Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Several model checkers are being developed and there is even a Model Checking Contest that takes place every year. State-of-the-art tools can handle Petri net models with more than **70 000 transitions** and **one million places**.

Reachability analysis

Petri nets can be analyzed using formal methods to conclude whether the net can reach a deadlock or not. There is a notion of liveness analogous to the one found in computer systems.

Several model checkers are being developed and there is even a Model Checking Contest that takes place every year. State-of-the-art tools can handle Petri net models with more than **70 000 transitions** and **one million places**.

Translating source code to a Petri net has been done before for other programming languages [1, 2] and also for Rust [3, 4]. The difficulty lies in supporting more synchronization primitives than simple mutexes and translating code from real-world applications.

Agenda

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: `while` and `for` to simple `loop`.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (`impl`) points to a valid trait.
  - Type checking.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: **`while`** and **`for`** to simple **`loop`**.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (**`impl`**) points to a valid trait.
  - Type checking.
- Mid-level Intermediate Representation (MIR):
  - Checking of exhaustiveness of pattern matching.
  - Desugar method calls to function calls (`x.method(y)` becomes `Type::method(&x, y)`).
  - Add implicit dereferencing operations.
  - Borrow checking.

Compilation stages in *rustc*

- Lexing: The source text is turned into a stream of atomic source code units known as tokens.
- Parsing: The stream of tokens is converted into an **Abstract Syntax Tree**.
- High-level Intermediate Representation (HIR):
  - Desugar loops: `while` and `for` to simple `loop`.
  - Type inference: The automatic detection of a type of an expression.
  - Trait solving: Ensuring that each implementation block (`impl`) points to a valid trait.
  - Type checking.
- Mid-level Intermediate Representation (MIR):
  - Checking of exhaustiveness of pattern matching.
  - Desugar method calls to function calls (`x.method(y)` becomes `Type::method(&x, y)`).
  - Add implicit dereferencing operations.
  - Borrow checking.
- Code generation:
  - *rustc* relies on LLVM as a backend.
  - It leverages many optimizations of the LLVM intermediate representation.
  - LLVM takes over from this point on.
  - At the end, object files are linked to create an executable.

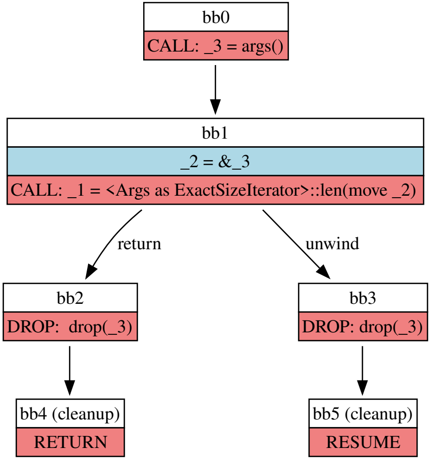
Hello World in MIR

BB means “basic block”. Each one is formed by statements and one terminator statement. The terminator statement is the only place where the control flow can jump to another basic block.

```
fn main() -> () {
    let mut _0: ();
    let _1: ();
    let mut _2: std::fmt::Arguments<'_>;
    let mut _3: &[&str];
    let mut _4: &[&str; 1];

    bb0: {
        _4 = const _;
        _3 = _4 as &[&str] (Pointer(Unsize));
        _2 = Arguments::<'_>::new_const(move _3) -> bb1;
    }

    bb1: {
        _1 = _print(move _2) -> bb2;
    }

    bb2: {
        return;
    }
}
```

MIR as a graph that shows the flow of execution

The MIR is a form of control flow graph (CFG) used in compilers. In this form, the translation to a Petri net becomes evident.



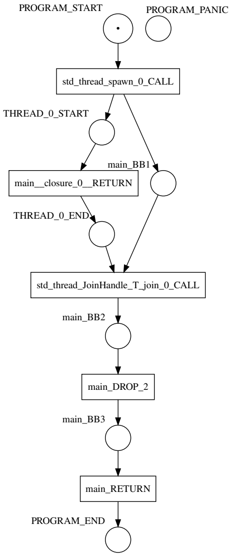
Example program

Let's consider a trivial program that spawns a thread that does nothing and immediately joins it.

```rust
fn main() {
  let thread_join_handle = std::thread::spawn(move || {
      // some work here
  });
  // some work here
  let _res = thread_join_handle.join();
}
```

- `std::thread::spawn` should create an additional token that models the program counter of the second thread.
- The joining thread should wait until the spawned thread finishes.

Petri net model for a thread



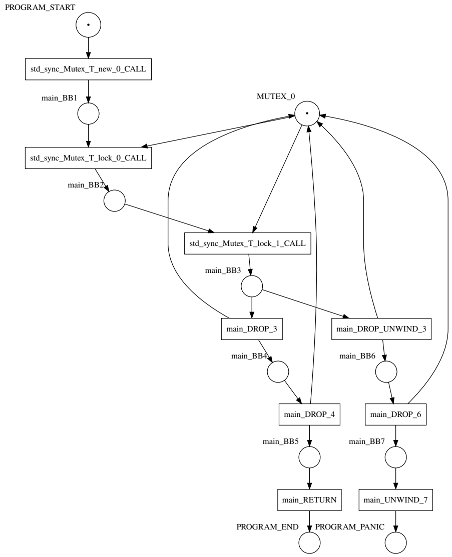
Example program

Consider a simple program that locks a mutex twice. The second lock operation will deadlock because the lock handle returned by the first call to `std::sync::Mutex::lock` is not dropped until it falls out of scope.

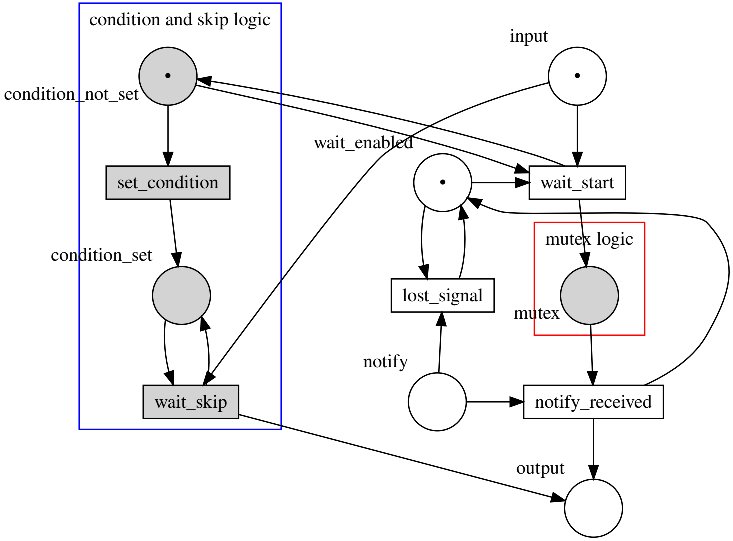
```rust
fn main() {
  let data = std::sync::Mutex::new(0);
  let _d1 = data.lock();
  let _d2 = data.lock(); // cannot lock, since d1 is still active
}
```

- There should be a single place that models the mutex.
- Locking the mutex is taking the token from the mutex place.
- Unlocking the mutex is setting the token back in the mutex place.

Petri net model for a mutex



How to model a condition variable



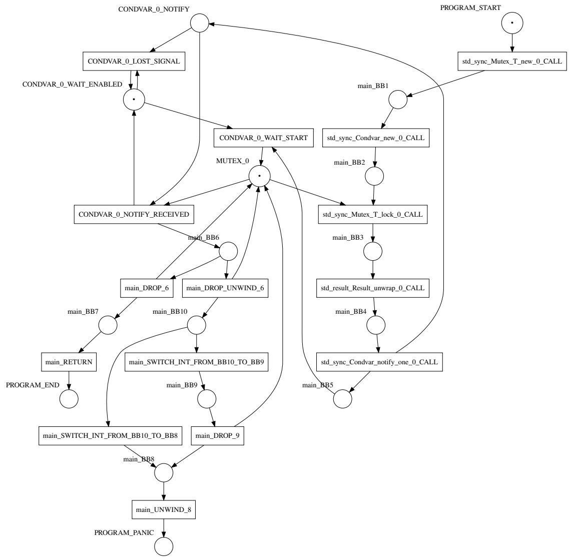
Example program

We have to use a very simple example program to keep the net small. In this case, the thread is trying to notify itself, which leads to a lost signal.

```rust
fn main() {
  let mutex = std::sync::Mutex::new(false);
  let cvar = std::sync::Condvar::new();
  let mutex_guard = mutex.lock().unwrap();
  cvar.notify_one();
  let _result = cvar.wait(mutex_guard);
}
```

- The model for the condition variable should appear in the Petri net.
- The notify place should be set.
- But the signal gets consumed because `std::sync::Condvar::wait` was not called.

Petri net model for the example program



Agenda

Limitations of the current version

It cannot translate everything possible in Rust / MIR...

- Closures outside of `thread:spawn` are not supported.
- Using arrays, `Vec`, and other data structures may cause the translation to give false results.
- `Channels`, `RwLock`, `Barrier` are not supported.
- Async is not supported.
- Synchronization mechanisms from external crates such as `tokio` or `semaphore` are not supported.

But we could add these features in the future :)

Limitations of the approach

The state explosion problem: The number of possible program states grows very fast!

Limitations of the approach

The state explosion problem: The number of possible program states grows very fast!

Data values are not modelled...

- Deadlocks caused by certain user input
- Deadlocks that appear only when a specific number of threads is started
- Dynamic code, foreign function calls, etc

Limitations of the approach

The state explosion problem: The number of possible program states grows very fast!

Data values are not modelled...

- Deadlocks caused by certain user input
- Deadlocks that appear only when a specific number of threads is started
- Dynamic code, foreign function calls, etc

Internals of used crates are not analyzed...

- Deadlocks in calls to libraries or due to them
- Deadlocks in `std` or `core`
- Deadlocks in underlying syscalls

Pros of the approach

It is a graphical approach and therefore the model is easier to debug.

There is a clear correspondence between the control flow in the Rust program and the Petri net. The model checkers provide the sequence of transition firings that lead to the deadlock.

We are far from reaching a performance bottleneck or size limit for this relatively simple approach. Model checkers get better every year and we can put them to use.

The tool shows how rustc could be extended to detect more bugs at compile-time and make Rust even safer!

Agenda

Quick demo

Bibliography

K. M. Kavi, A. Moshtaghi, and D.-J. Chen, “Modeling multithreaded applications using petri nets,” *International Journal of Parallel Programming*, vol. 30, pp. 353–371, 2002.

A. Moshtaghi, “Modeling Multithreaded Applications Using Petri Nets,” Master’s thesis, The University of Alabama in Huntsville, 2001.

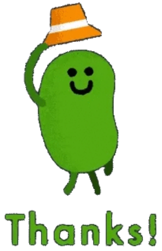
T. Meyer, “A Petri Net semantics for Rust,” Master’s thesis, Universität Rostock — Fakultät für Informatik und Elektrotechnik, 2020.
`https://github.com/Skasselbard/Granite/blob/master/doc/MasterThesis/main.pdf`.

K. Zhang and G. Liua, “Automatically transform rust source to petri nets for checking deadlocks,” *arXiv preprint arXiv:2212.02754*, 2022.

Online Petri net simulators

- A simple simulator by Igor Kim can be found on `https://petri.hp102.ru/`. A tutorial video on Youtube and example nets are included in the tool.
- A complement to this is a series of interactive tutorials by Prof. Wil van der Aalst at the University of Hamburg. These tutorials are Adobe Flash Player files (with extension `.swf`) that modern web browsers cannot execute. Luckily, an online Flash emulator like the one found on `https://flashplayer.fullstacks.net/?kind=Flash_Emulator` can be used to upload the files and execute them.
- Another online Petri net editor and simulator is `http://www.biregal.com/`. The user can draw the net, add the tokens, and then manually fire transitions.

Questions?



Links

Tool	https://github.com/hlisdero/cargo-check-deadlock
Presentation	https://github.com/hlisdero/thesis/tree/main/presentation_eurorust_2025
Published crate	https://crates.io/crates/cargo-check-deadlock
Thesis	https://github.com/hlisdero/thesis