

モデル検査を使ったコードレビュー

2025-04-26

長谷川 央

背景 (1/2)

割り込み可能なシステムの非同期処理のコードは、
振る舞いを目で追うことが困難である

例えば...

OSのスケジューラでは、以下の2つの処理が存在する

1. タスクを実行開始
（Running 状態に移行）
2. I/O処理を待つために処理を中断
（Waiting 状態に移行）
3. タスクをキューに戻し、
代わりに次のタスクを実行開始

1. タスクを実行開始
（Running 状態に移行）
2. 高優先度のタスクがキューに到達
3. タスクを一時中断
（Preempted 状態に移行）
3. 実行していたタスクをキューに戻し、
高優先度のタスクを実行開始

背景 (2/2)

左側2の処理が行われたタイミングで、
高優先度のタスクがキューに到達したら
どのような処理が行われるのか

そのとき、コード上では、
どのように実行箇所が遷移するのか

1. タスクを実行開始
（Running 状態に移行）
2. I/O処理を待つために処理を中断
（Waiting 状態に移行）
3. タスクをキューに戻し、
代わりに次のタスクを実行開始

1. タスクを実行開始
（Running 状態に移行）
2. 高優先度のタスクがキューに到達
3. タスクを一時中断
（Preempted 状態に移行）
3. 実行していたタスクをキューに戻し、
高優先度のタスクを実行開始

従来手法

前ページの疑問を解決するために、以下のような手法が考えられる

- 目視でのコードレビュー
 - 目でコードを追って、どのように動いているのかを確かめる方法
 - しかし、実行され得る経路が多すぎて、把握しきれない
- モデル検査
 - 処理を抽象化してモデルとして書き表し、
モデル検査を使ってどのような振る舞いが起こり得るのかを確かめる方法
 - この方法では、モデルの記述とコードの対応関係が分かりづらい

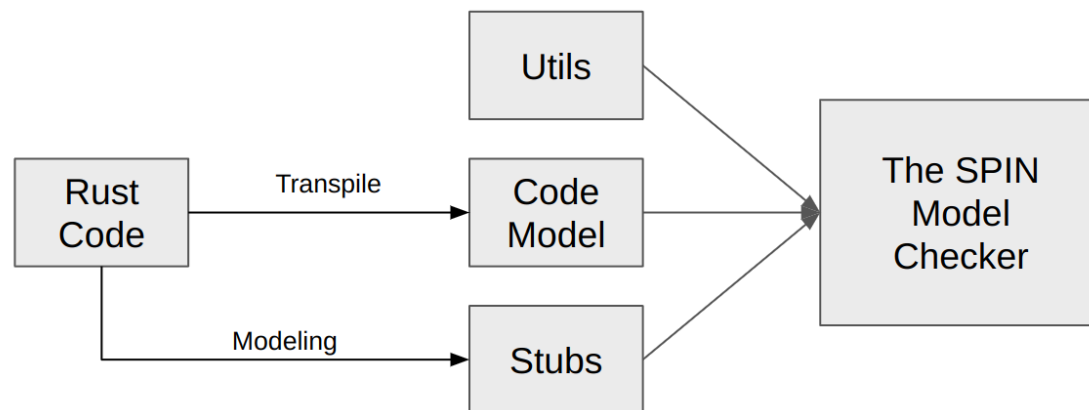
提案手法

コードとモデルを1対1対応させながらモデル検査を行う方法

(ただし、完全に1対1対応させることはできない)

アプローチ：

1. Base Modelの作成
2. コードを1行ずつモデルへ変換
(Code ModelとUtilsの作成)
3. 動かすために必要な機構を
モデリングして追加 (Stubsの作成)
4. 動かしながら、動きを確認



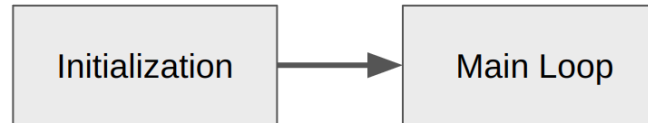
Base Modelの作成

Base Modelでは、
コードの遷移条件を整理し、モデルとして表す

OSのスケジューラは、以下の2つの要因で処理が遷移する

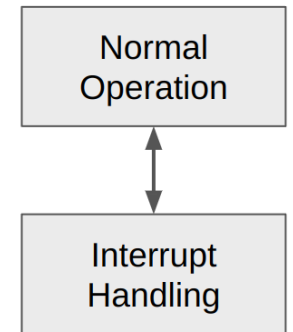
1. 関数呼び出しによる遷移：

初期化処理が終わると、
メインループの関数に処理が移行する

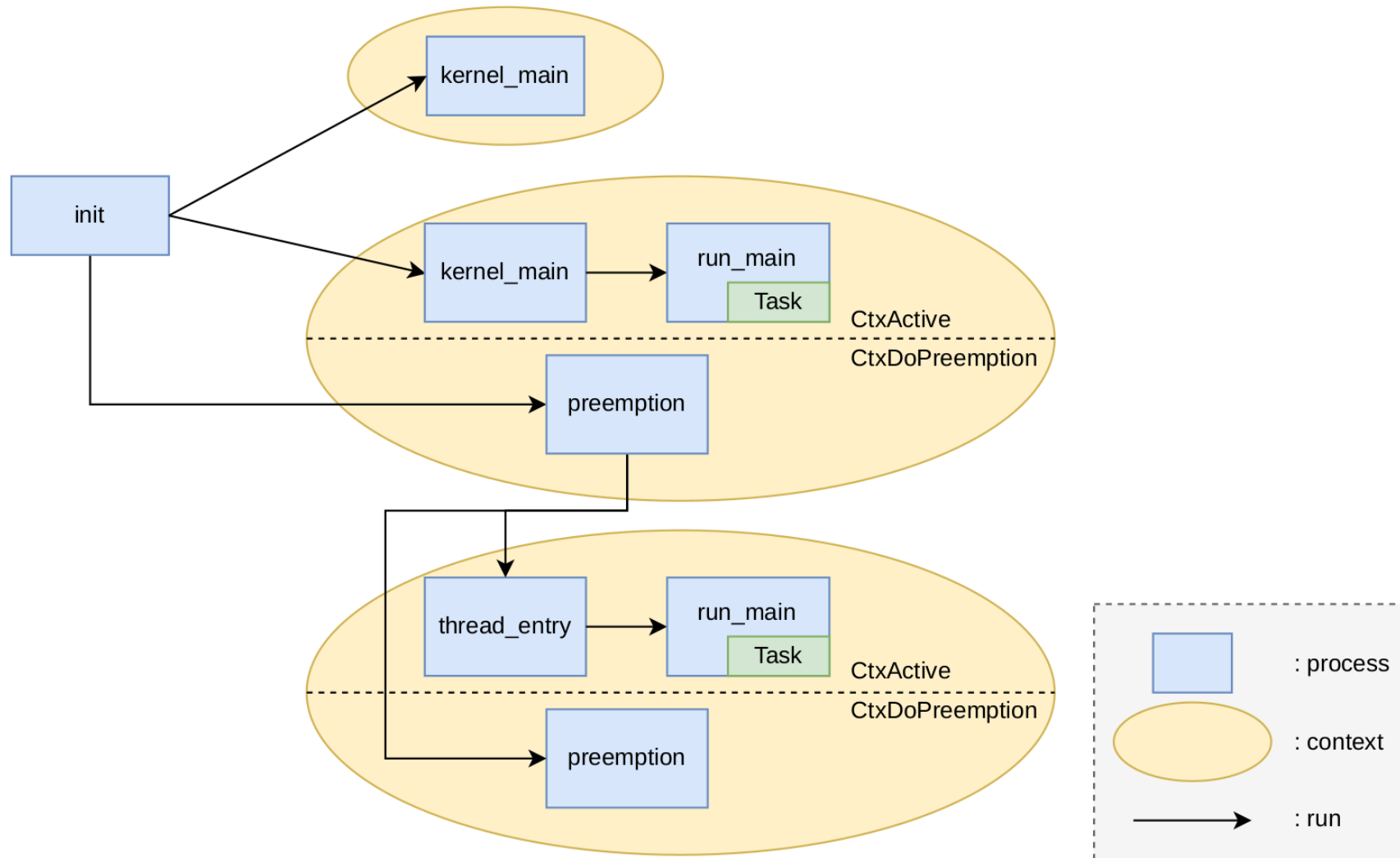


2. 割り込みの発生による遷移：

割り込まれたタイミングで、
割り込み処理のコードの実行が
開始される



Base Modelの例（自動運転車用のOS）



コードの変換

```
1 pub fn run_main() {
2   CPUID_TO_RAWCPUID[awkernel_lib::cpu::cpu_id()]
3     .store(awkernel_lib::cpu::raw_cpu_id(),
4       Ordering::Relaxed);
5   .....
6   loop {
7     if let Some(task) = get_next_task() {
8       #[cfg(not(feature = "no_preempt"))]
9       {
10         let mut node = MCSNode::new();
11         let mut info = task.info.lock(&mut node
12 );
13
14         if let Some(ctx) = info.
15 take_preempt_context() {
16           info.update_last_executed();
17           drop(info);
18
19           unsafe { preempt::yield_and_pool(
20 ctx) };
21
22           continue;
23         }
24       }
25     }
26     .....
27 }
```

```
1 proctype run_main(byte __ctx_id) provided (
2   CONTEXT_STATE[__ctx_id] == CtxActive) {
3   .....
4   RAWCPUIDS[awkernel_lib_cpu_id()] =
5     awkernel_lib_raw_cpu_id();
6
7   main_loop:
8   do
9     :: get_next_task(option_task_id) ->
10       assert(interrupt_flag[awkernel_lib_cpu_id()]
11 == false);
12     if
13       :: OPT_IS_SOME(option_task_id) ->
14         _tid = option_task_id.value;
15         take_preempt_context(_tid, option_context)
16
17     ;
18
19     if
20       :: OPT_IS_SOME(option_context) ->
21         yield_and_pool(option_context.value);
22         goto main_loop
23       :: OPT_IS_NONE(option_context)
24     fi;
25     .....
26 }
```


Utilsの作成

Promelaは、
Rustのようなリッチな型を持っていない

- Vec型
- Option型
- Result型
- ...

これらの処理を変換できるように、
マクロを書き、
Utils として追加する

```

1 pub fn run_main() {
2   CPUID_TO_RAWCPUID[awkernel_lib::cpu::cpu_id()]
3   .store(awkernel_lib::cpu::raw_cpu_id(),
4   Ordering::Relaxed);
5   .....
6   loop {
7     if let Some(task) = get_next_task() {
8       #[cfg(not(feature = "no_preempt"))]
9       {
10        let mut node = MCSNode::new();
11        let mut info = task.info.lock(&mut node
12        );
13
14        if let Some(ctx) = info.
15        take_preempt_context() {
16          info.update_last_executed();
17          drop(info);
18
19          unsafe { preempt::yield_and_pool(
20          ctx) };
21
22          continue;
23        }
24      }
25      .....

```

```

1 proctype run_main(byte __ctx_id) provided (
2   CONTEXT_STATE[__ctx_id] == CtxActive) {
3   .....
4   RAWCPUIDS[awkernel_lib_cpu_id()] =
5   awkernel_lib_raw_cpu_id();
6
7   main_loop:
8   do
9     :: get_next_task(option_task_id) ->
10      assert(interrupt_flag[awkernel_lib_cpu_id()]
11      == false);
12     if
13     :: OPT_IS_SOME(option_task_id) ->
14      _tid = option_task_id.value;
15      take_preempt_context(_tid, option_context)
16
17     ;
18     if
19     :: OPT_IS_SOME(option_context) ->
20      yield_and_pool(option_context.value);
21      goto main_loop
22     :: OPT_IS_NONE(option_context)
23     fi;
24     .....

```

Stubのモデリング

今回は、「タスクの状態遷移を引き起こす部分のコード」に注目しており、それ以外のコードの遷移には関心がない

前ページの「コードの変換」では、タスクスケジューリングのみの変換を行う

しかし、**I/Oイベント** や **割り込み** が発生しなければ、レビューしたい振る舞いがモデル上で起こらない

前ページで作成したモデル（Code Model）が、自分が確認をしたい処理をしてくれるように処理を追加する

出来上がるモデルの例（自動運転車のOS）

- The Code Model

- `kernel_main.pml`
- `preempt_part1.pml`
- `preempt_part2.pml`
- `task_part1.pml`
- `task_part2.pml`
- `thread.pml`
- `rr.pml`
- `scheduler.pml`

- Utils

- `verification.pml`
- `rust_primitives.pml`

- Stubs

- `delta_list.pml`
- `awkernel_lib.pml`
- `mutex.pml`
- `task_stub.pml`
- `interrupt_stub.pml`

モデル検査の実行

確かめたい振る舞いを実行させるようにStubsをいじりながら、
モデル検査を実行する

例えば...

- Preemptionが単独で実行されるとき処理
- I/Oイベント待ちが発生するときの処理
- PreemptionとI/Oイベント待ちが同時に発生するときの処理
- ...

Case Study

実際に、

自動運転車用に開発中のOSのコードに本手法を適用した

割り込み発生時にイベント待ちが発生すると、

意図しない割り込みが発生することがあるというバグを発見し、修正を行った

正常な処理

[CPU 1]

1. Round-robinスケジューラ下でTask Aを実行開始

[CPU 0 (Scheduler)]

2. Task Aがtime sliceを超過していることを確認

3. CPU 1にInter-processor interrupt (割り込み) を送信

4. IPIを受け取り、Preemptionを実行

5. Task Aをキューに戻し、次のTask B (次のタスク) を実行開始

Case Study

実際に、

自動運転車用に開発中のOSのコードに本手法を適用している

割り込み発生時にイベント待ちが発生すると、

意図しない割り込みが発生することがあるというバグを発見し、修正した

バグ発生時の処理

[CPU 1]

1. Round-robinスケジューラ下でTask Aを実行開始

[CPU 0 (Scheduler)]

2. Task Aがtime sliceを超過していることを確認

2. イベントを待機するために、Task AをWaitingに移行

3. Task Aをキューに戻し、Task Bの実行を開始

4. CPU 1にInter-processor interrupt (割り込み) を送信


5. IPIを受け取り、Preemptionを実行

6. Task Bをキューに戻し、次のTask C (次のタスク) を実行開始


arXiv

作業中に書いた報告書のようなものは、arXivで公開中

<https://arxiv.org/abs/2503.09282>

 Cornell University

We gratefully acknowledge support from the Simons Foundation, [member institutions](#), and all contributors. [Donate](#)

 > cs > arXiv:2503.09282

Search... All fields [v](#) Search

[Help](#) | [Advanced Search](#)

Computer Science > Software Engineering

[Submitted on 12 Mar 2025]


A Case Study on Model Checking and Runtime Verification for Awkernel

Akira Hasegawa, Ryuta Kambe, Toshiaki Aoki, Yuuki Takano

In operating system development, concurrency poses significant challenges. It is difficult for humans to manually review concurrent behaviors or to write test cases covering all possible executions, often resulting in critical bugs. Preemption in schedulers serves as a typical example. This paper proposes a development method for concurrent software, such as schedulers. Our method incorporates model checking as an aid for tracing code, simplifying the analysis of concurrent behavior; we refer to this as model checking-assisted code review. While this approach aids in tracing behaviors, the accuracy of the results is limited because of the semantics gap between the modeling language and the programming language. Therefore, we also introduce runtime verification to address this limitation in model checking-assisted code review. We applied our approach to a real-world operating system, Awkernel, as a case study. This new operating system, currently under development for autonomous driving, is designed for preemptive task execution

Access Paper:

[View PDF](#)
[HTML \(experimental\)](#)
[TeX Source](#)
[Other Formats](#)

 [view license](#)

Current browse context:
cs.SE
[< prev](#) | [next >](#)
[new](#) | [recent](#) | [2025-03](#)
Change to browse by:
[cs](#)

References & Citations

[NASA ADS](#)

まとめ

- コードレビューでモデル検査を利用する手法を提案（予定）
- 実際のOSで実行し、バグを発見し修正した
- 多分、スケジューラ以外の処理のレビューにも使えるので使ってみてね！