Julia's adventures with Why3

Julia Lawall, Gilles Muller (Inria/LIP6), Virginia Aponte (CNAM), and others

April 2019

Problem setting

Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston,
Fabien Gaud, Vivien Quéma, Alexandra Fedorova:
The Linux scheduler: a decade of wasted cores.
EuroSys 2016: 1:1-1:16

Work conservation: Various bugs in the Linux kernel scheduler (load balancer) made it possible that some cores could be overloaded while other cores are idle.

- Wastes energy.
- May increase latency.
- Bugs hard to detect and linger for years.

Verification?

How can we prove that these bugs will never occur?

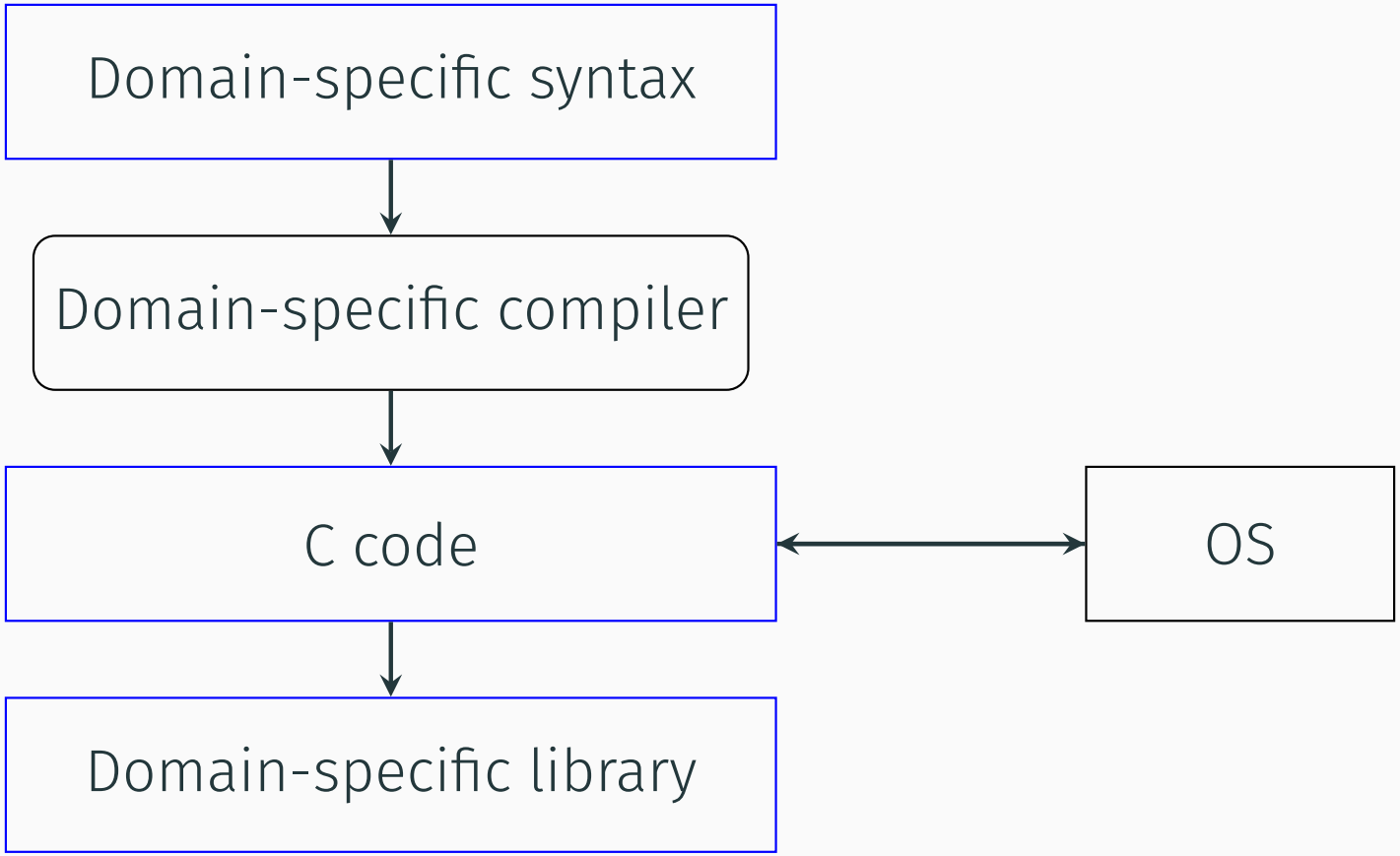
- Scheduler code is written in C.
- Concepts not well isolated from implementation details.
- Over 20 000 LOC a few years ago, more now.
- Concurrency.

DSL approach

Domain-specific syntax

Domain-specific library

DSL approach



Our case: Ipanema

Extension of Bossa single-core scheduling DSL to multicore.

Domain-specific declarations:

```
thread = { int load; time vruntime; system core cpu;}
core = {
    threads = {
      shared RUNNING thread current;
      shared READY set<thread> ready: order = {lowest vruntime};
    ... };
    set<domain> sd;
    time min_vruntime;
    system shared int cload; ...
}
```

Our case: Ipanema

Domain-specific operations critical for work conservation:

```
steal(core dst) = {
    can_steal_core(core src, core dst) {
        (valid(src.current) ? 1 : 0) + count(src.ready) > 1
    } => stealable_cores
    do {
        select_core() {
            first(stealable_cores order = {highest cload})
        } => src
        steal_thread(core dst, thread t) {
            if (src.cload - dst.cload > t.load) {
                t.vruntime -= src.min_vruntime;
                t.vruntime += dst.min_vruntime;
                t => dst.ready;
            }
        } until (src.cload == dst.cload);
    } until (runnable(dst) != 0);
}
...
```

Observations: Traditional motivations for a DSL

DSL makes it easy to script complex processing

- $\Rightarrow$ for state changes
- `first`/`highest` for selecting prioritized items in a list.

DSL controls how operations are composed, enabling verifications

- $\Rightarrow$ may allow only certain kinds of state changes, depending on the context (block vs unblock event)

Generated code (extract)

```
if ((busiest->cload - _fresh_23) > t->load) {
        t->vruntime = t->vruntime - busiest->min_vruntime;
        t->vruntime = t->vruntime + _fresh_24;
        list_add(&pos->ipanema.ipa_tasks, &ready_tasks);
        ipa_change_queue_and_core(t, NULL, MIGRATING_STATE,
                                  thief);
        busiest->cload = busiest->cload - t->load;
        _fresh_23 = _fresh_23 + t->load;
}
```

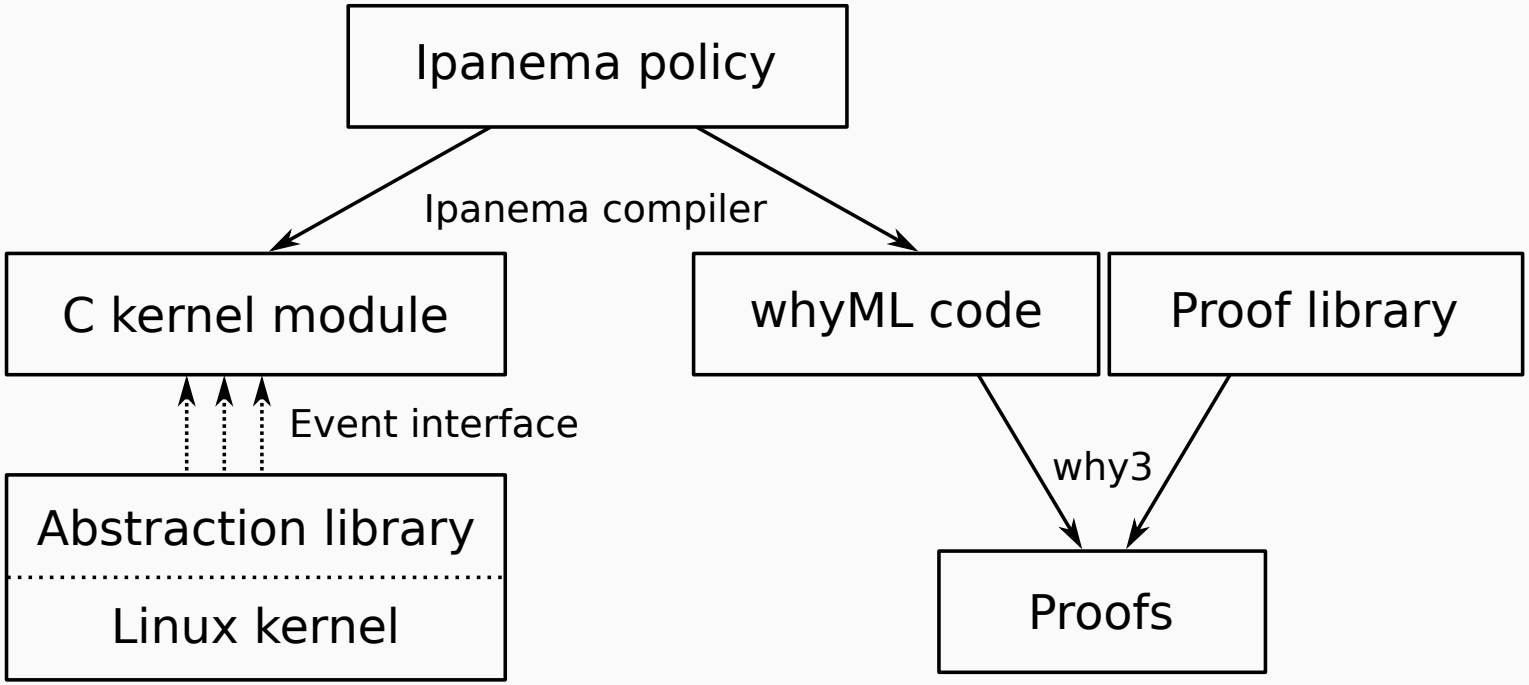
Generated code (extract)

```
if ((busiest->cload - _fresh_23) > t->load) {
        t->vruntime = t->vruntime - busiest->min_vruntime;
        t->vruntime = t->vruntime + _fresh_24;
        list_add(&pos->ipanema.ipa_tasks, &ready_tasks);
        ipa_change_queue_and_core(t, NULL, MIGRATING_STATE,
                                  thief);
        busiest->cload = busiest->cload - t->load;
        _fresh_23 = _fresh_23 + t->load;
}
```

Observations: Our motivation for a DSL

- DSL compilation enforces the barrier between the scheduling operations (policy) and the library (mechanism).
- C code structure is predictable.
- Mechanized verification of the algorithmic properties of the code that is actually executed becomes possible.
- Enables targeting an existing OS.

Current status



Why Why3?

Imperative programming language based on ML.

- ML is not C, but permits line by line translation.
- Avoid pointer correctness issues, focus on the algorithm.
- Pointer correctness ensured by the DSL/library.

Interface to many SMT solvers

- We mostly used Alt-Ergo and CVC4

Easy-to-use, entertaining IDE

- Moderately fast
- Actively supported

Why not Coq?

Tedious to construct the verification conditions.

- Why3 takes care of what to prove automatically.
- Why3 allows interfacing with Coq, but the resulting proofs are very fragile.

Still, Coq would provide control over what proof steps to take

- Solvers may take undesirable directions incurring a huge performance overhead.

Another problem with solvers

$$\text{false} \rightarrow \text{anything}$$

Another problem with solvers

$$\text{false} \rightarrow \text{anything}$$

- Solvers don't always discover this shortcut.
- Solvers don't give any feedback when they do discover this shortcut.
- Small thoughtless changes can introduce this issue at any time.

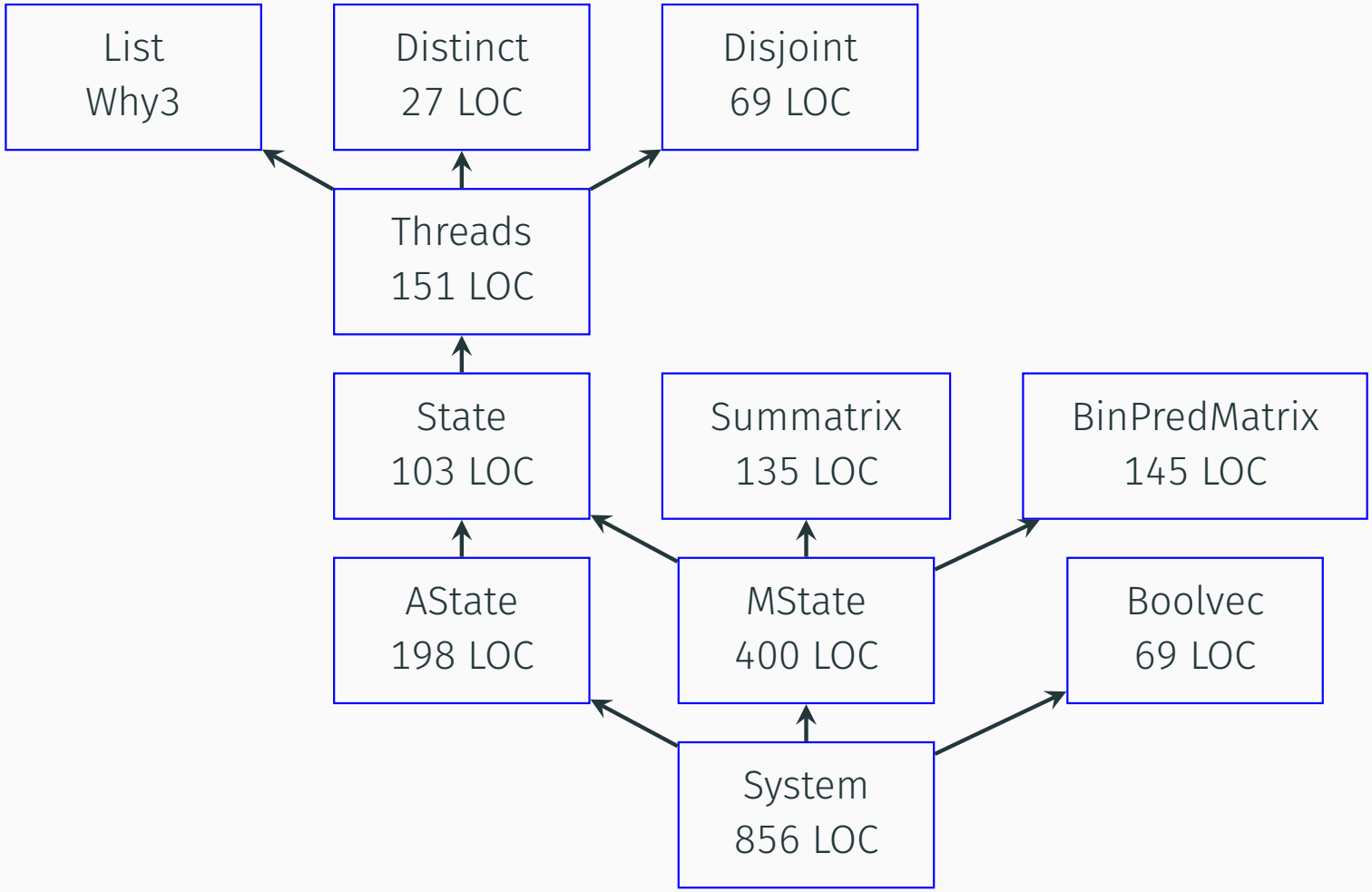
Another problem with solvers

$$\text{false} \rightarrow \text{anything}$$

- Solvers don't always discover this shortcut.
- Solvers don't give any feedback when they do discover this shortcut.
- Small thoughtless changes can introduce this issue at any time.
- Why3 **smoke detection** can help.

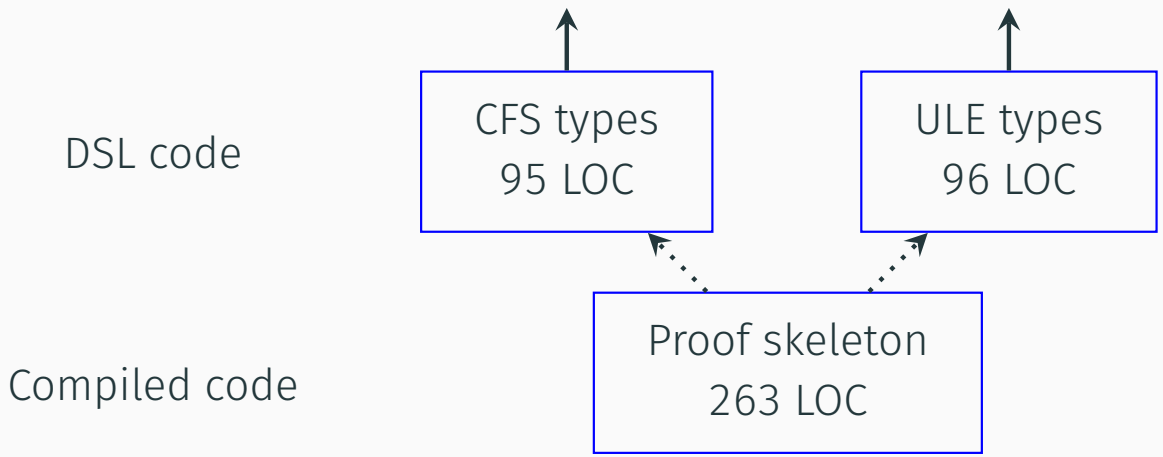
Architecture

Abstraction library: proved once



Architecture

Policy-specific code: proved once per policy



- Variants without and with concurrent scheduling events.

Proving work conservation (non concurrency case)

If any core is idle, no core is overloaded:

```
forall co:int. idle p.cores co →
    forall co1:int. not(overloaded p.cores co1)
```

Proving Work Conservation (non concurrency case)

If any core is idle, no core is overloaded:

```
forall co:int. idle p.cores co →
    forall co1:int. not(overloaded p.cores co1)
```

We have also proved Weak Work Conservation that takes into account the impact of concurrent scheduling events.

Demo

Assessment and Conclusion

- 6.5 minutes proving time on 2 cores
- Policy-developer facing code (CFS types and ULE types) proved fully automatically
  - One click, no asserts.
- More labor needed for library and proof skeleton.
  - Asserts, transformations, choice of solver, etc.
- Robust to the introduction of small policy variants.
- Fully proved (weak) work conserving load balancing algorithms.