School of Computer Science & Engineering

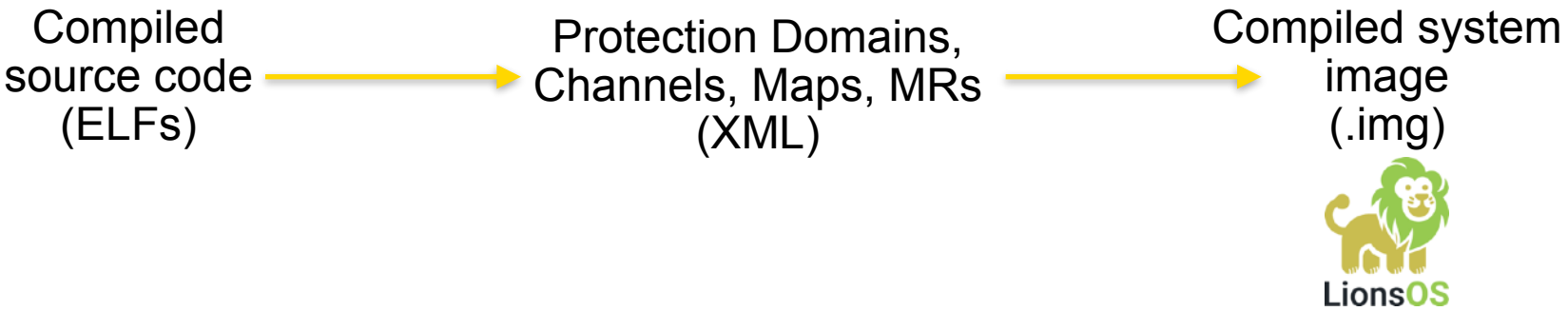
Trustworthy Systems Group

Simplifying Microkit System Composition with Acacia

seL4 Summit 2026

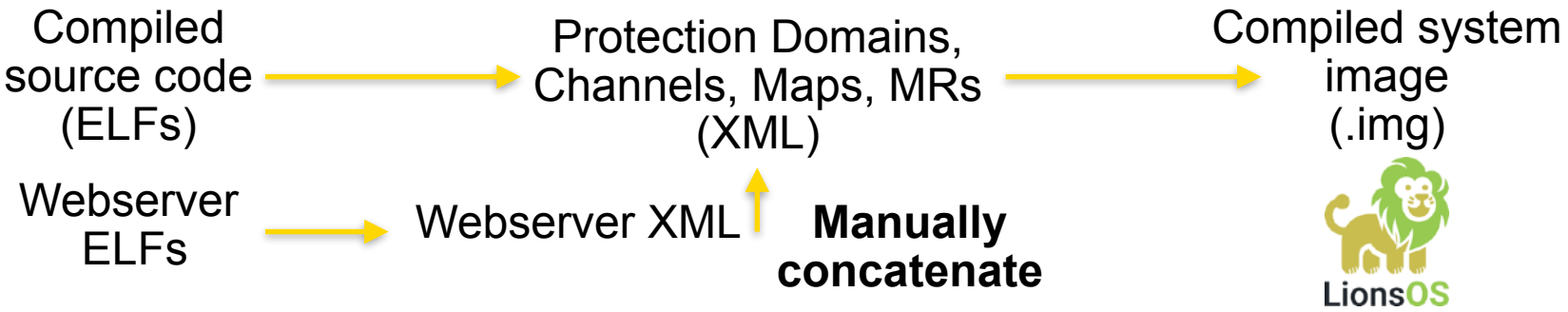
Lesley Rossouw

So, you want to make a Microkit system



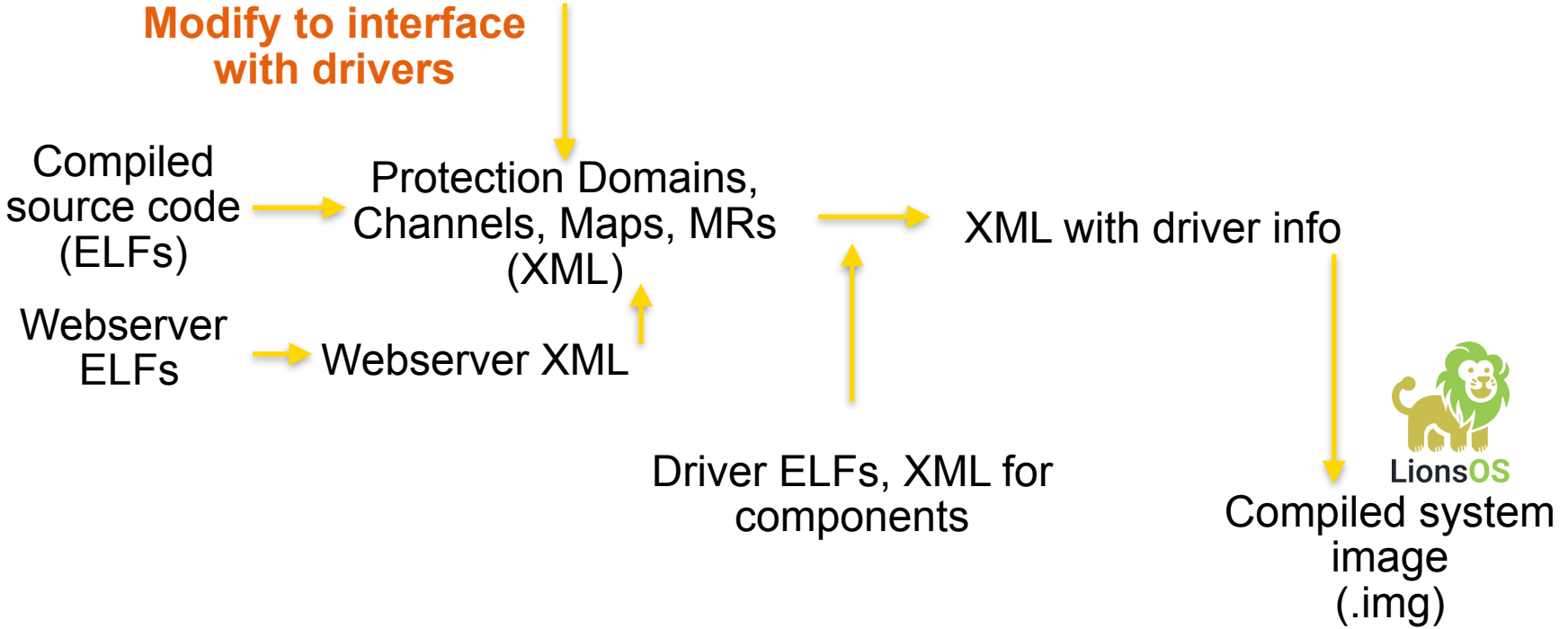
Wow! That was easy!

But: what happens if you want to reuse some existing code?

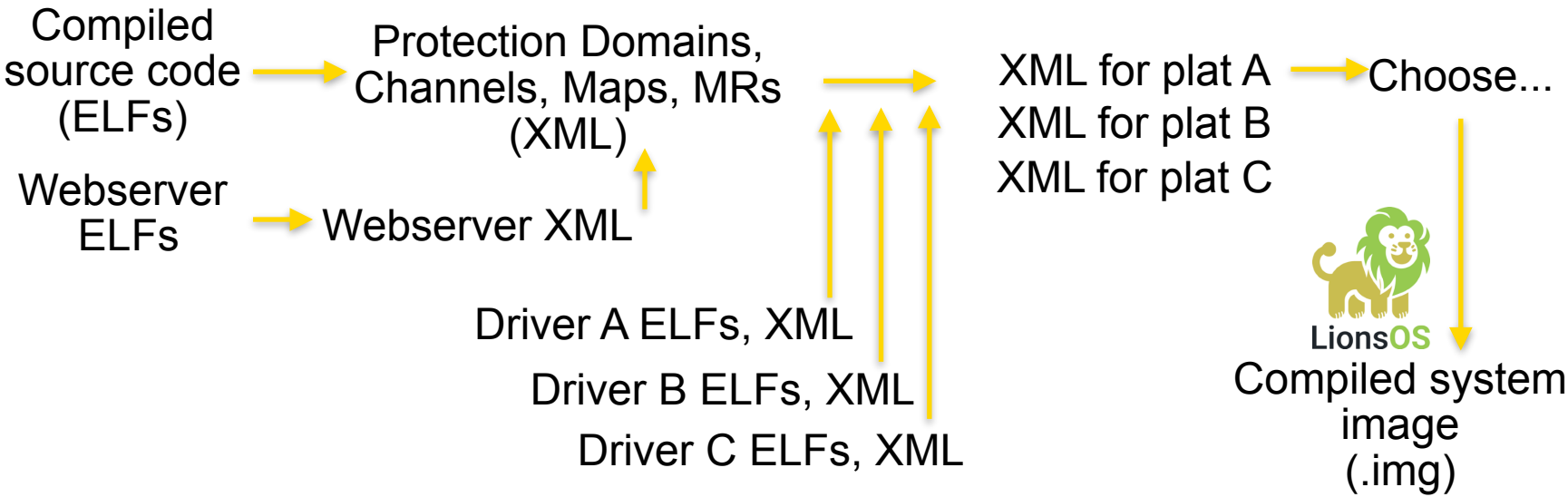


Okay, a little less easy...

What about drivers?



What if you need drivers for **incompatible** platforms??



Okay, not good!

We want something that is ...

Flexible

Friendly

Easy to extend

Powerful

General

A solution ... **Acacia**

- A **Radically Simple** system composition tool
- Implemented purely in (typed) Python3
- **Designed for extension!**

This talk

- An introduction to Acacia
  - Motivation
  - Conceptual problems
  - How does it work?
- A guided tour: creating components, using the sDDF
- How does it fit into the seL4 ecosystem?
- What's next?

Introduction to Acacia

Context, design, philosophy

The big picture

Point of seL4 Microkit…

(p)“The seL4 ecosystem lacked a framework was easy to use for engineers building embedded systems on seL4.”

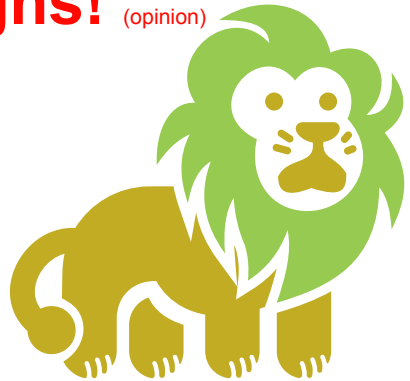
(p)“The Microkit provides abstractions that lend themselves to efficient implementation … and rule out a number of inappropriate design patterns”

Make seL4 **accessible!**

The big picture

… but Microkit only solves this problem for from-scratch system designs! (opinion)

(Specialised knowledge implied*)



Work like sDDF bridges expert knowledge gap to end-users

… if it's accessible!

In most cases, end-users are reusing lots of components!

The big picture

Great seL4 systems are built on the shoulders of giants

The big picture

Great seL4 systems are built on the shoulders of (minimal) giants

The big picture

Barriers to architecting systems and reusing prior art **may discourage using Microkit and seL4!**

(hopefully) we can be approachable to typical embedded systems engineers by **solving this**!

The big picture

Everybody benefits from easier composition

Research:
faster prototyping and experimentation

Industrial:
+ reduce engineering effort
+ easier adoption

Hobbyist:
+ accessibility

So: how can we make things easier?

What issues do we need to compose static systems?

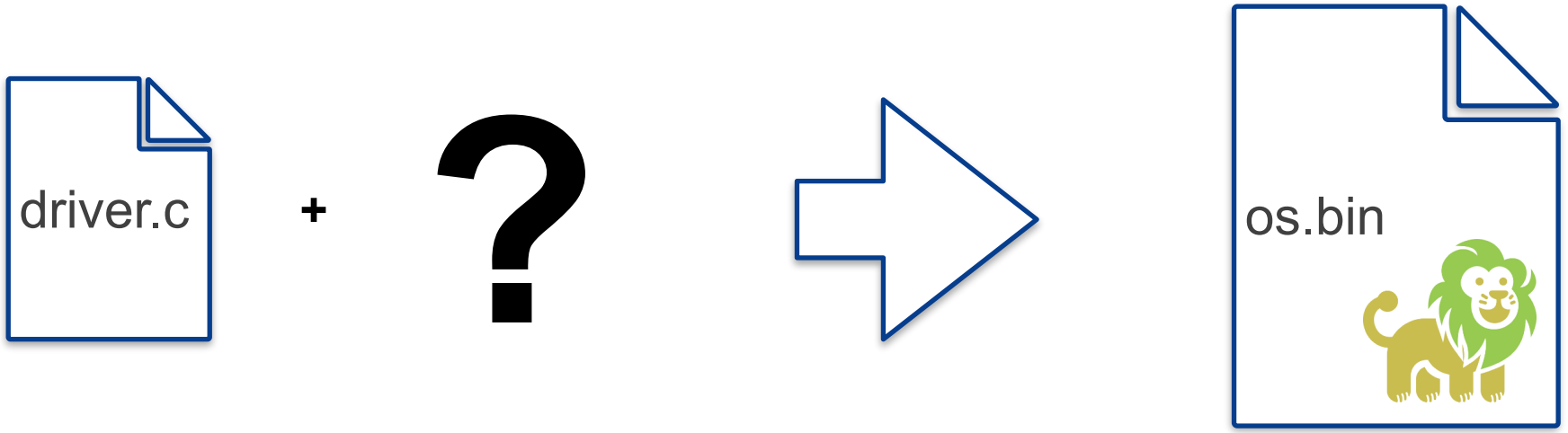
Problem 1

Supporting mutually incompatible platforms

Supporting reusable drivers

Case: drivers. Assume we have a driver for some device foo

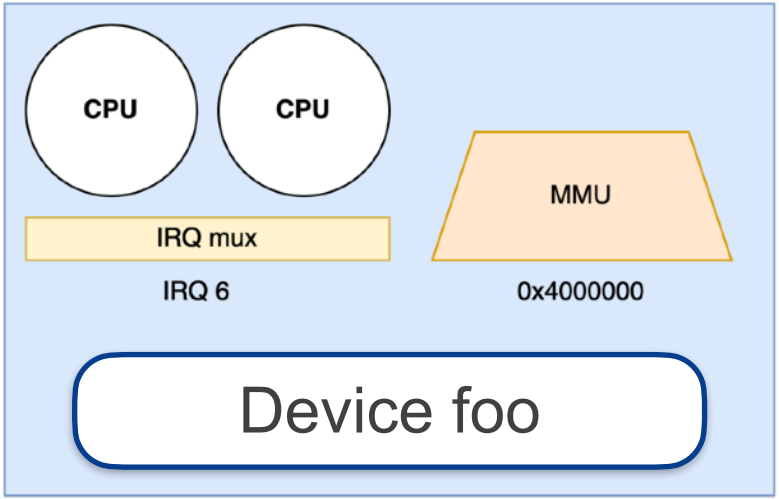
To target a specific platform, need device dependent info…



Supporting reusable drivers

Using the same device on different platforms implies…

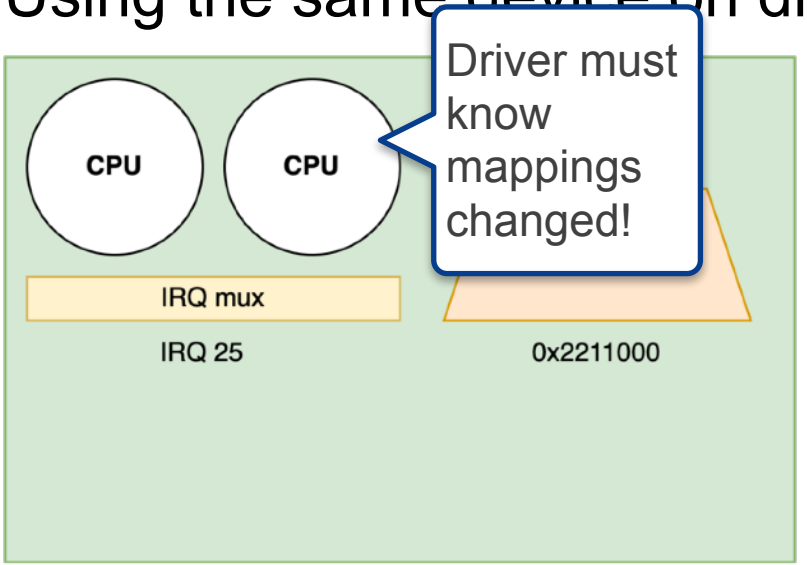
- Different memory mappings
- Different interrupt numbers
- (possibly) different bus, different clock setup etc.



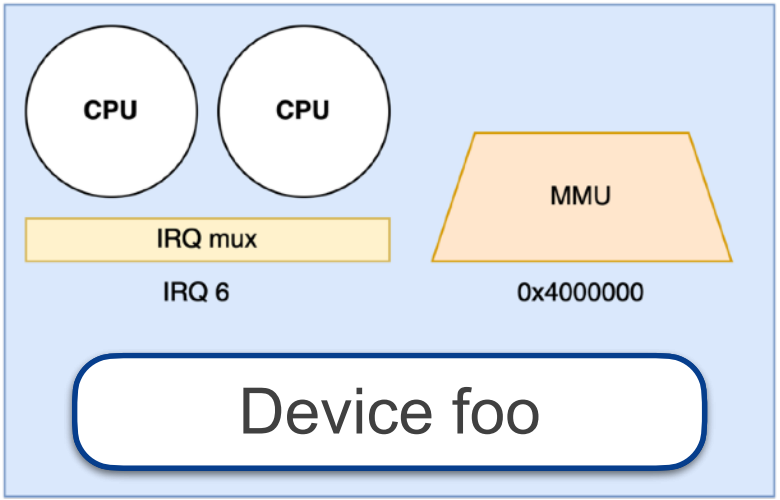
Fooboard

Supporting reusable drivers

Using the same device on different platforms implies…



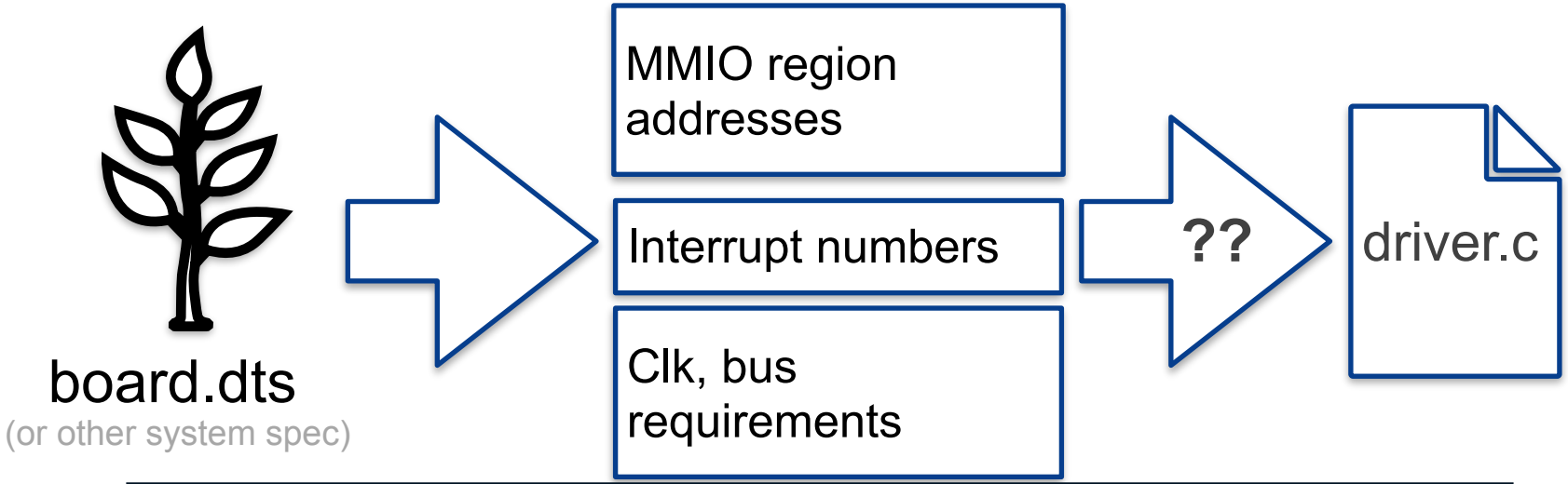
Barboard



Fooboard

Supporting portable drivers

Easy solution: device trees! (x86 gets trickier…)



… but how do we configure the driver? Static OS!

Problem 2

Configuring components based on system architecture

Configuring components

Problem: we [illegible] **post-compile**

Can't simply do this without tooling support!

MMIO region addresses

Interrupt numbers

Protocol config

??

driver.elf

SDK model! Avoid messing with verified binaries

Configuring components

Solution: config structs.

```c
typedef struct device_resources {
    char magic[DEVICE_MAGIC_LEN + 1];
    uint8_t num_regions;
    uint8_t num_irqs;
    device_region_resource_t regions[DEVICE_MAX_REGIONS];
    device_irq_resource_t irqs[DEVICE_MAX_IRQS];
} device_resources_t;
```

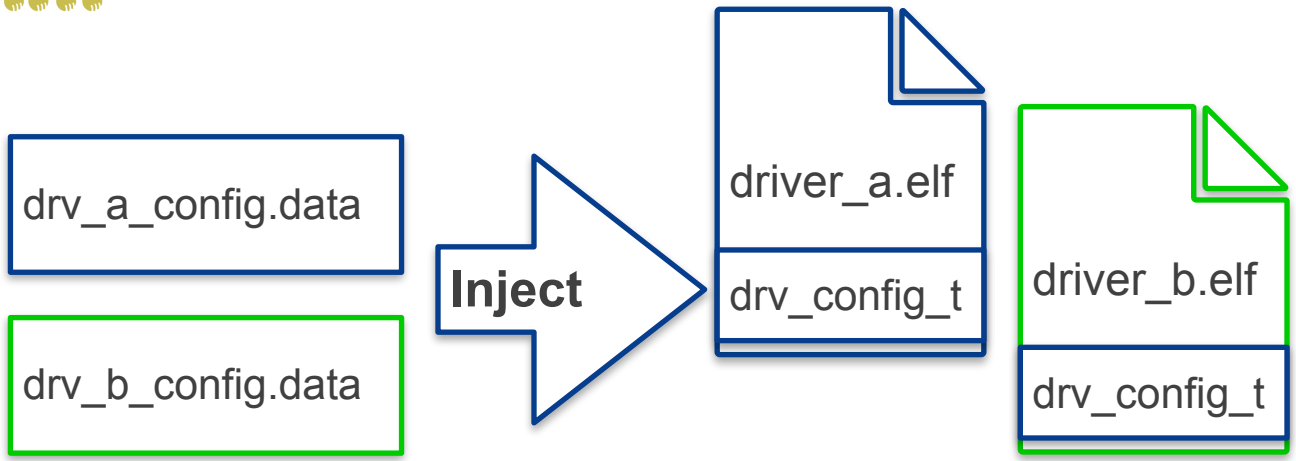
Define struct types with fields for expected config

```c
__attribute__((__section__(".serial_driver_config"))) serial_driver_config_t config;
__attribute__((__section__(".device_resources"))) device_resources_t device_resources;
```

Driver leaves an empty **section** typed as that struct

Configuring components

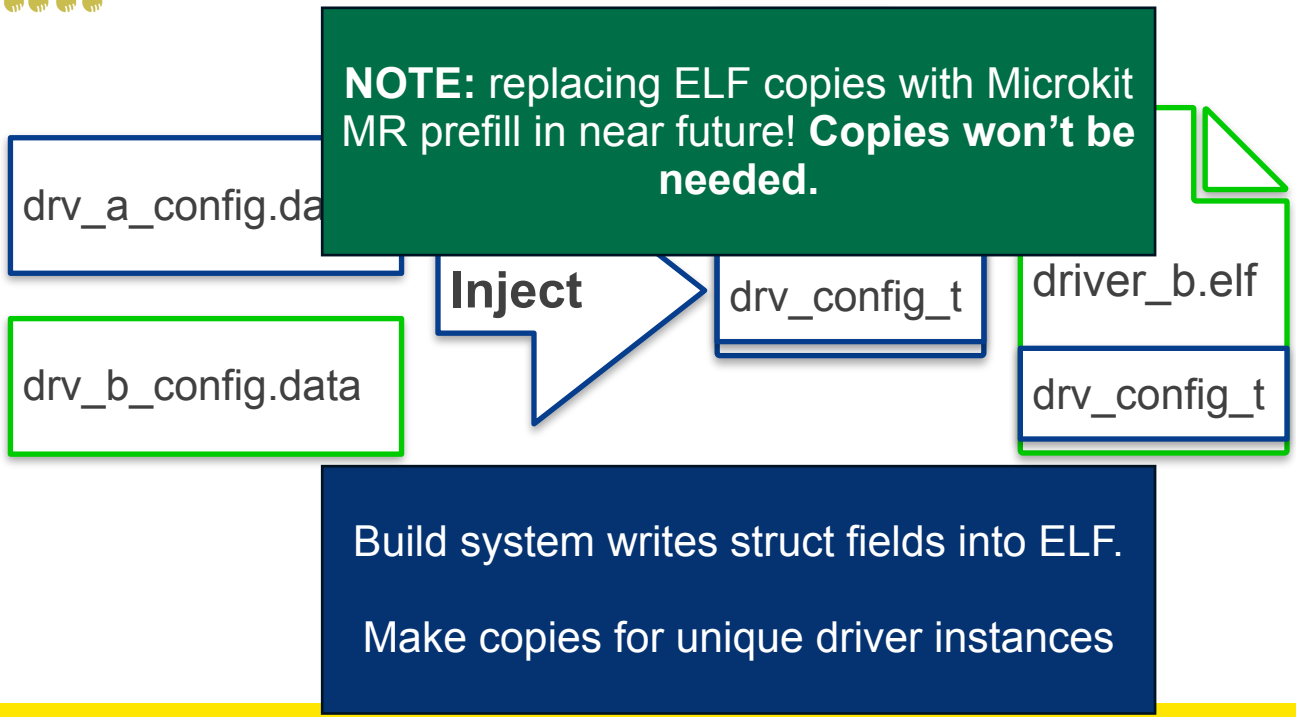
Solution: config structs.



Make copies for instances

Configuring components

Solution: config structs.



Make copies for instances

Configuring components

Solution: config structs.

```
assert(device_resources_check_magic(&device_resources));
assert(device_resources.num_irqs == 1);
assert(device_resources.num_regions == 1);

uart_base = (uintptr_t)device_resources.regions[0].region.vaddr;
```

Driver accesses struct as though it was populated pre-build!

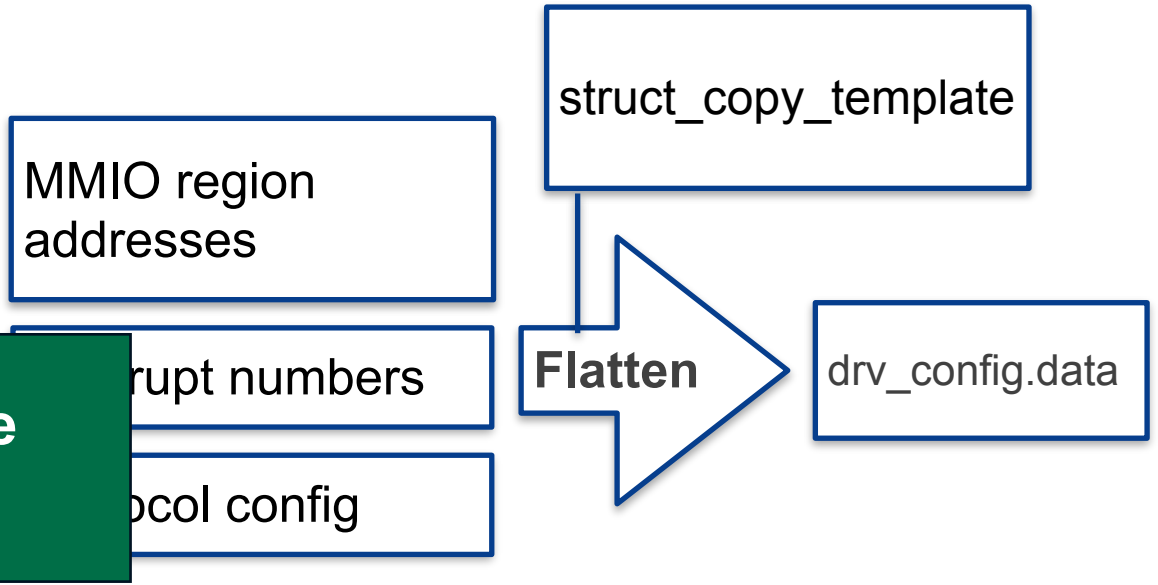
Configuring components

But: how do we make our data to patch?
Struct layout depends on target system…

Naïve: replicate struct in another tool, pack it there…

Bad! DRY!

How can we have one source of truth?



Configuring components

If only our compiled code could somehow tell us what the struct looks like…

DWARF debug symbols to the rescue!

https://dwarfstd.org

DWARF debug symbols

DWARF: standard architecture and OS independent debug symbols for many languages!

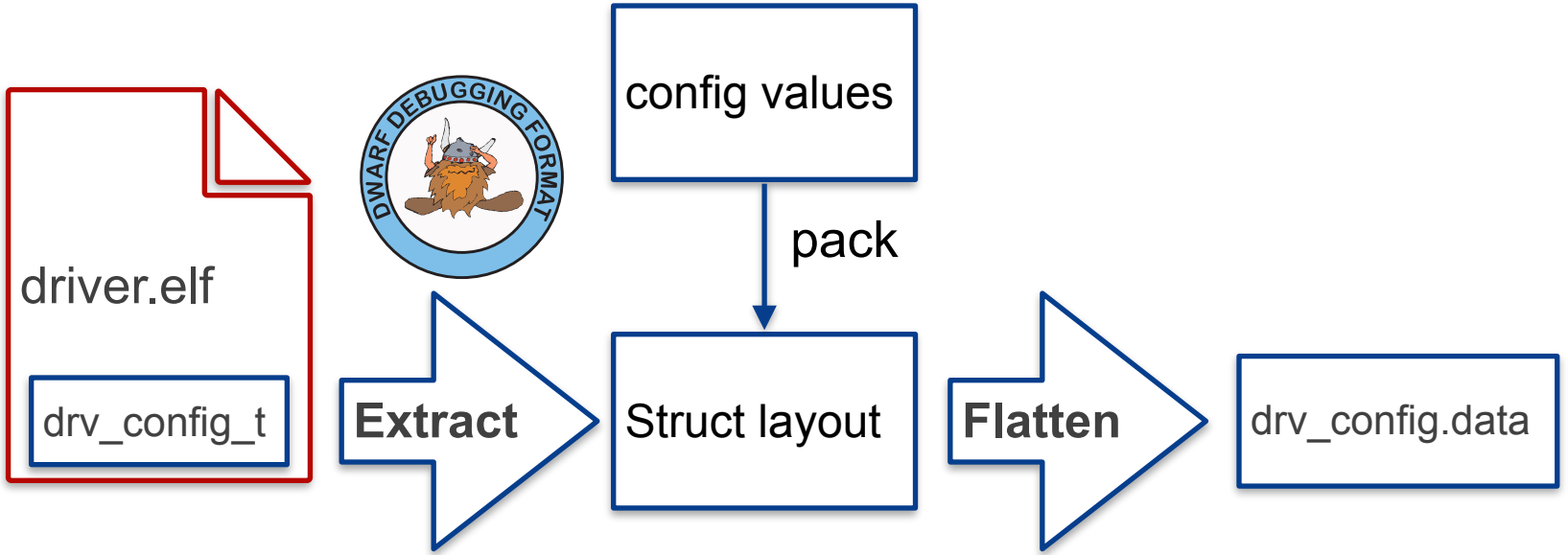
Includes:

- Names, types, sizes of all structs and struct members
- Type mappings (e.g. foo_t -> uint32_t -> unsigned long)
- **Already there for most Microkit builds!***

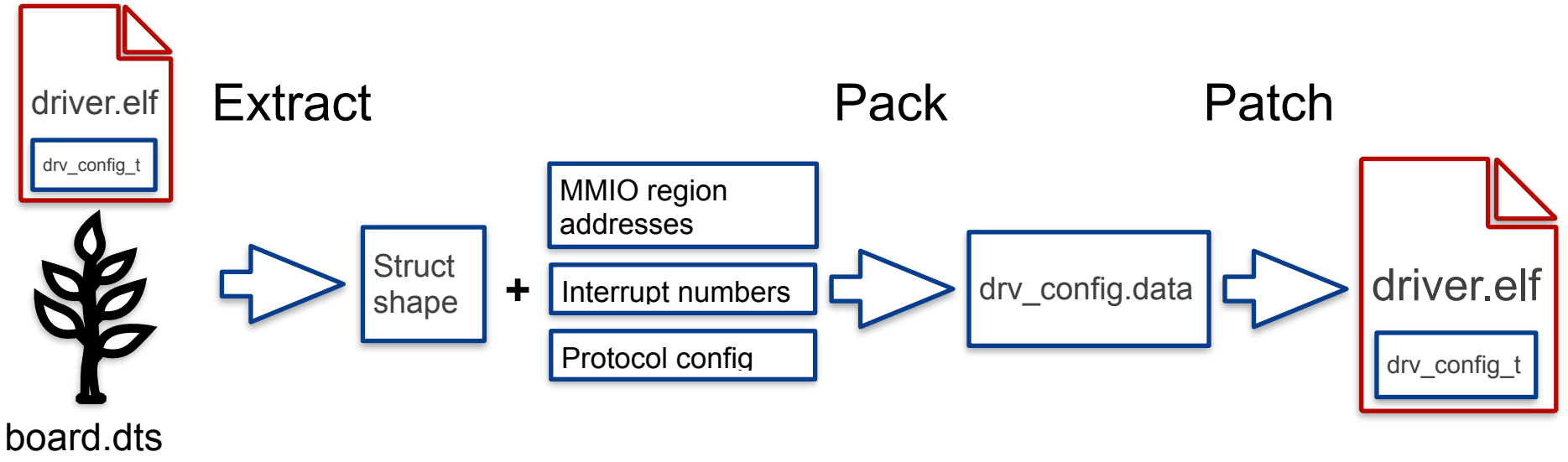
Tells us everything we need to repack struct!

Configuring components

Better: extract struct shape from ELF!



Overview (so far)



We can now configure generic programs that depend on the platform!

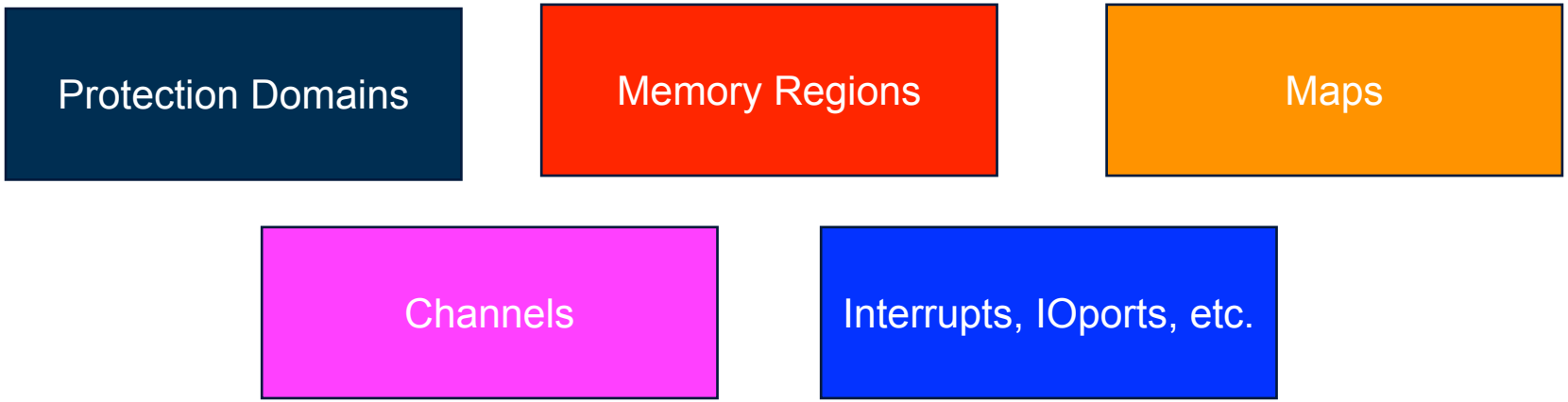
But: how do we represent and generate modules?

Problem 3

Modularisation and reuse of Microkit components

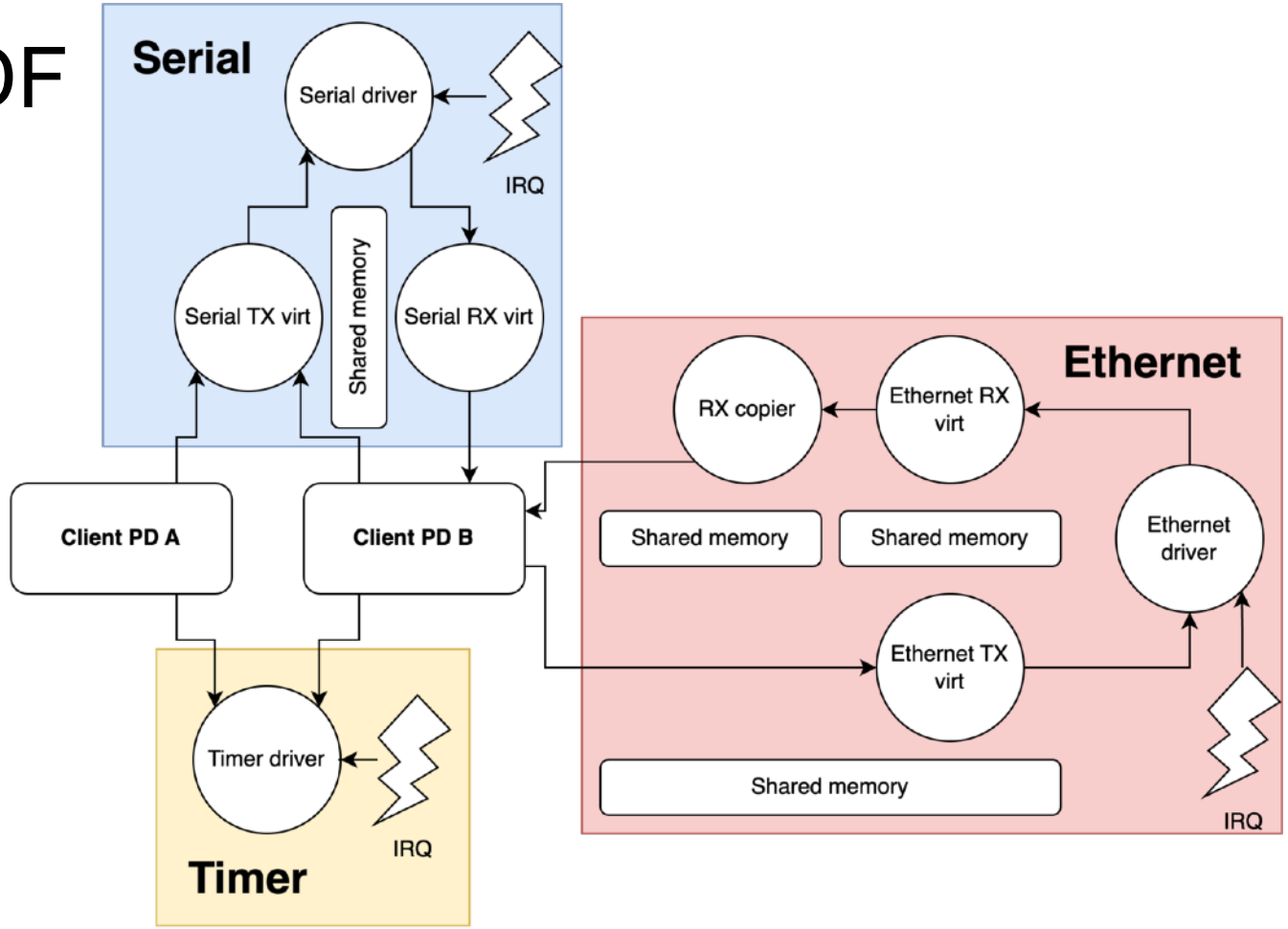
Modularisation

Goal: come up with an idiom for representing modules consisting of:

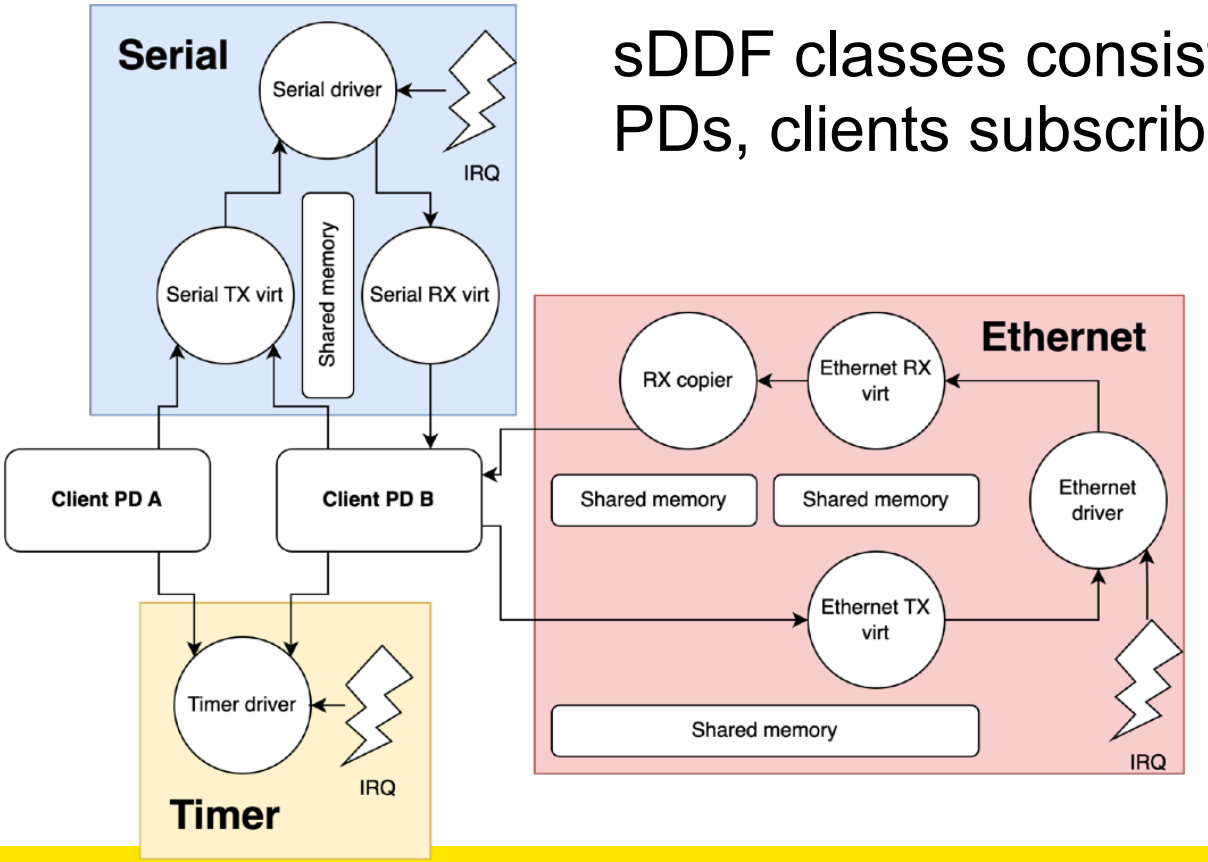


We can take some ideas from existing Microkit work

sDDF



sDDF



sDDF classes consist of groups of PDs, clients subscribe to class

sDDF

This is a client-server model! (or publisher-subscriber)

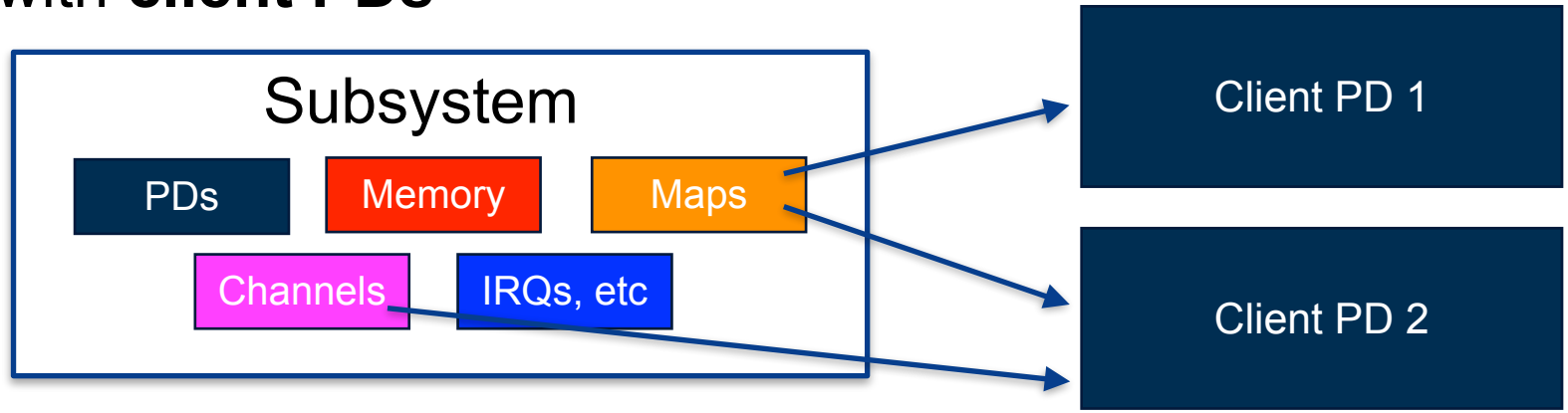
Servers are groups of Microkit objects with **clients**.

- Set up server first, connect clients later.
- Config structs…
  - Tell servers about their MRs, channels, config data
  - Tell clients how to talk to server

Client-server is a natural model

Modularisation

Subsystems: collections of microkit objects which interacts with **client PDs**

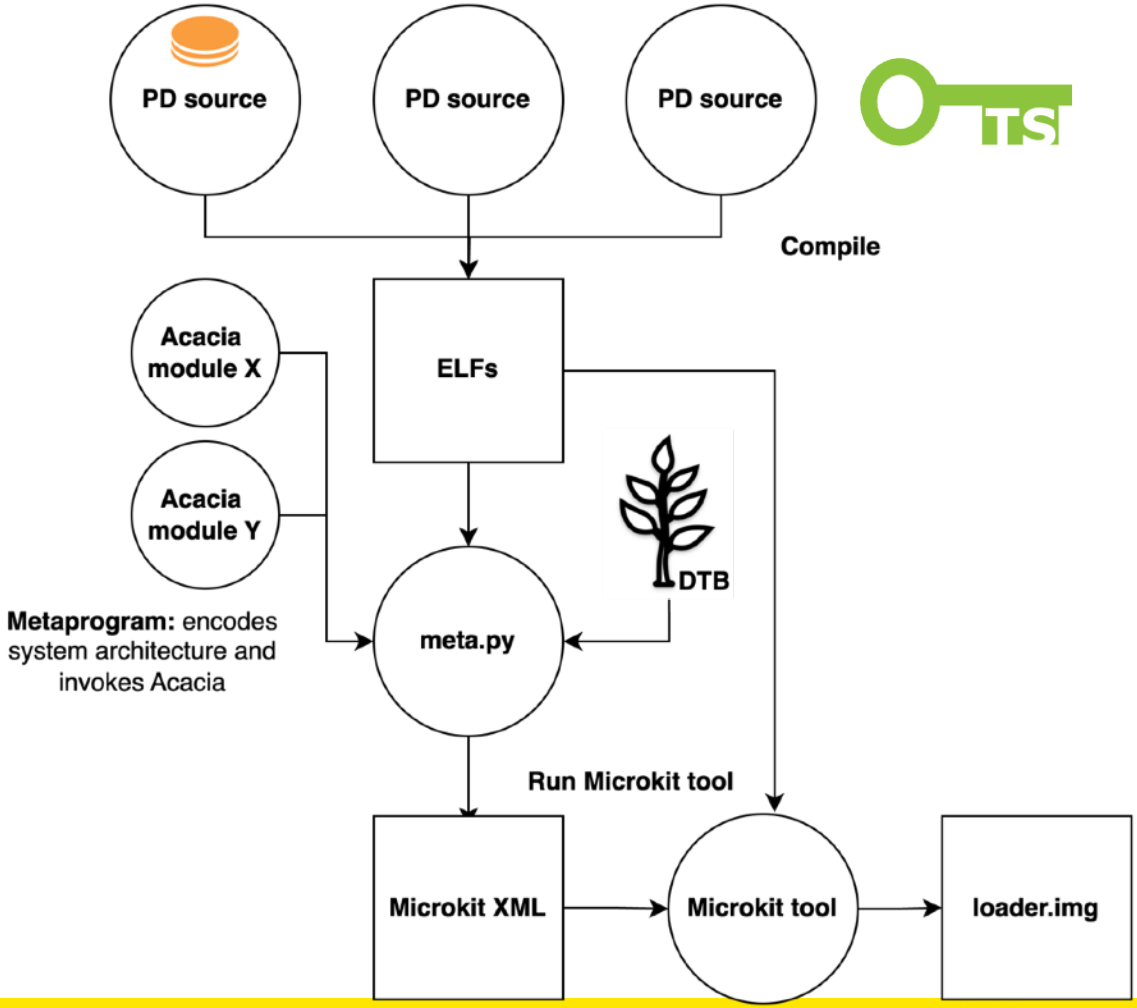


Finally … **Acacia**

- A **Radically Simple** system composition tool leveraging **device trees**
- Enables design and reuse of portable components with **subsystems**
- Generates and injects configuration data using **config structs**

How does it work?

- Runs **post-compile*** but **before Microkit runs**
- Acacia accepts a **system description** - “metaprogram” - defines project
- **Generates Microkit XML and config structs!**



How does it work?

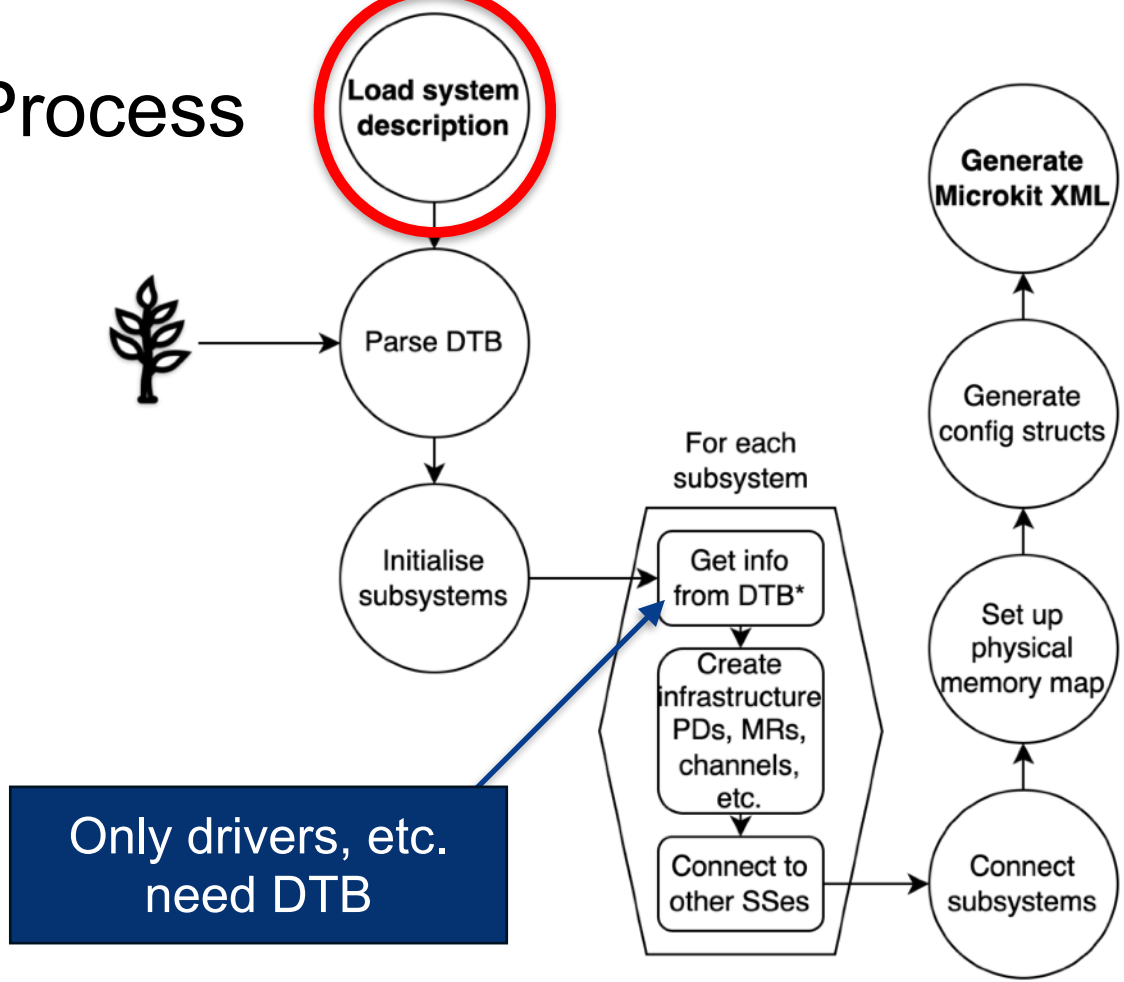
- Python API
  - Imperative interface for creating all Microkit objects
  - Conveniences for creating channels, maps
  - Auto-allocation of vaddrs of maps, paddrs of MRs (when needed)
  - Define subsystems w/ inheritance
    - … **or make your own modularisation as a Python module!**
- **Critical design goal**: prevent programming errors, give descriptive diagnostics to guide implementation

System description

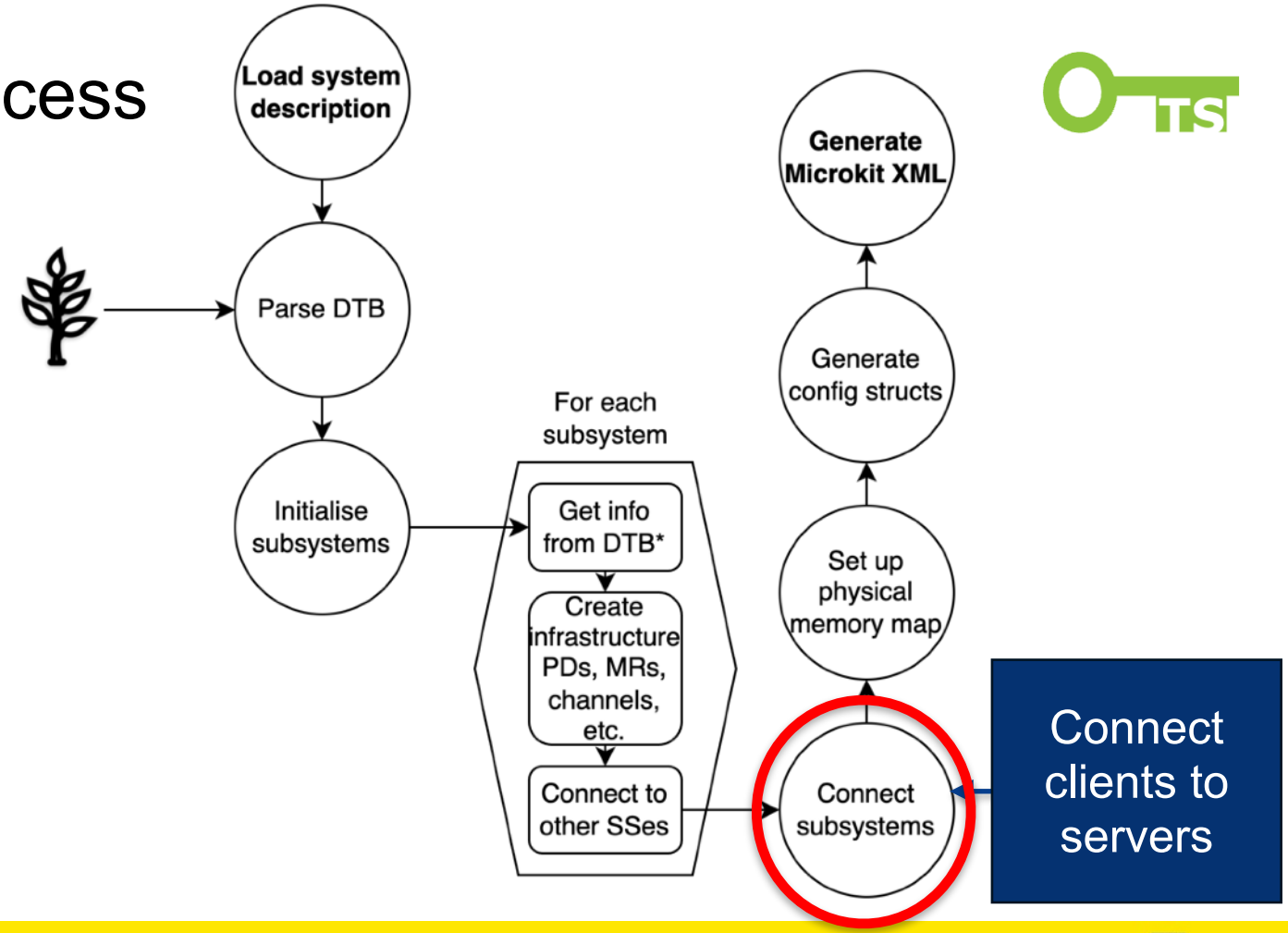
- LionsOS default: **metaprogram**
  - Python script declares all needed Acacia elements
  - **Flexible**
- … not ideal in all cases
  - WIP: declarative API using YAML
  - “Just give me subsystem X, Y and Z for my client!”

```python
# Copyright 2025, UNSW
# SPDX-License-Identifier: BSD-2-Clause
import os, sys
import argparse
from typing import List
from dataclasses import dataclass
from acacia import System, ProtectionDomain, MemoryRegion, Channel, DeviceTreeBlob, Map

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)), "../../"))
from acacia_sddf import BOARDS, sDDFI2C, sDDFSerial, sDDFTimer


def generate(sdf_file: str, output_dir: str, dtb: DeviceTreeBlob):
    client_pn532 = ProtectionDomain(sdf, "client_pn532", "client_pn532.elf", priority=1)
    client_ds3231 = ProtectionDomain(
        sdf, "client_ds3231", "client_ds3231.elf", priority=1
    )

    i2c = sDDFI2C(
        sdf, board.i2c.compatible, board.i2c.node_path, driver_prio=200, virt_prio=199
    )
    i2c.add_client(client_ds3231)
    i2c.add_client(client_pn532)

    timer = sDDFTimer(sdf, board.timer.compatible, board.timer.node_path)
    timer.add_client(client_ds3231)
    timer.add_client(client_pn532)

    serial = sDDFSerial(
        sdf,
        board.serial.compatible,
        board.serial.node_path,
        driver_prio=201,
        virt_tx_prio=200,
        allow_rx=False,
        enable_color=False,
        baud_rate=board.baud_rate if board.baud_rate else 115200,
    )
    serial.add_client(client_ds3231)
    serial.add_client(client_pn532)

    out_file = f"{output_dir}/{sdf_file}"
    sdf.make_config_structs()
    print(f"Saving to {out_file}")
    sdf.write_xml_file(out_file)
```

Acacia Process



Acacia Process



Walkthrough

Let’s make a system!

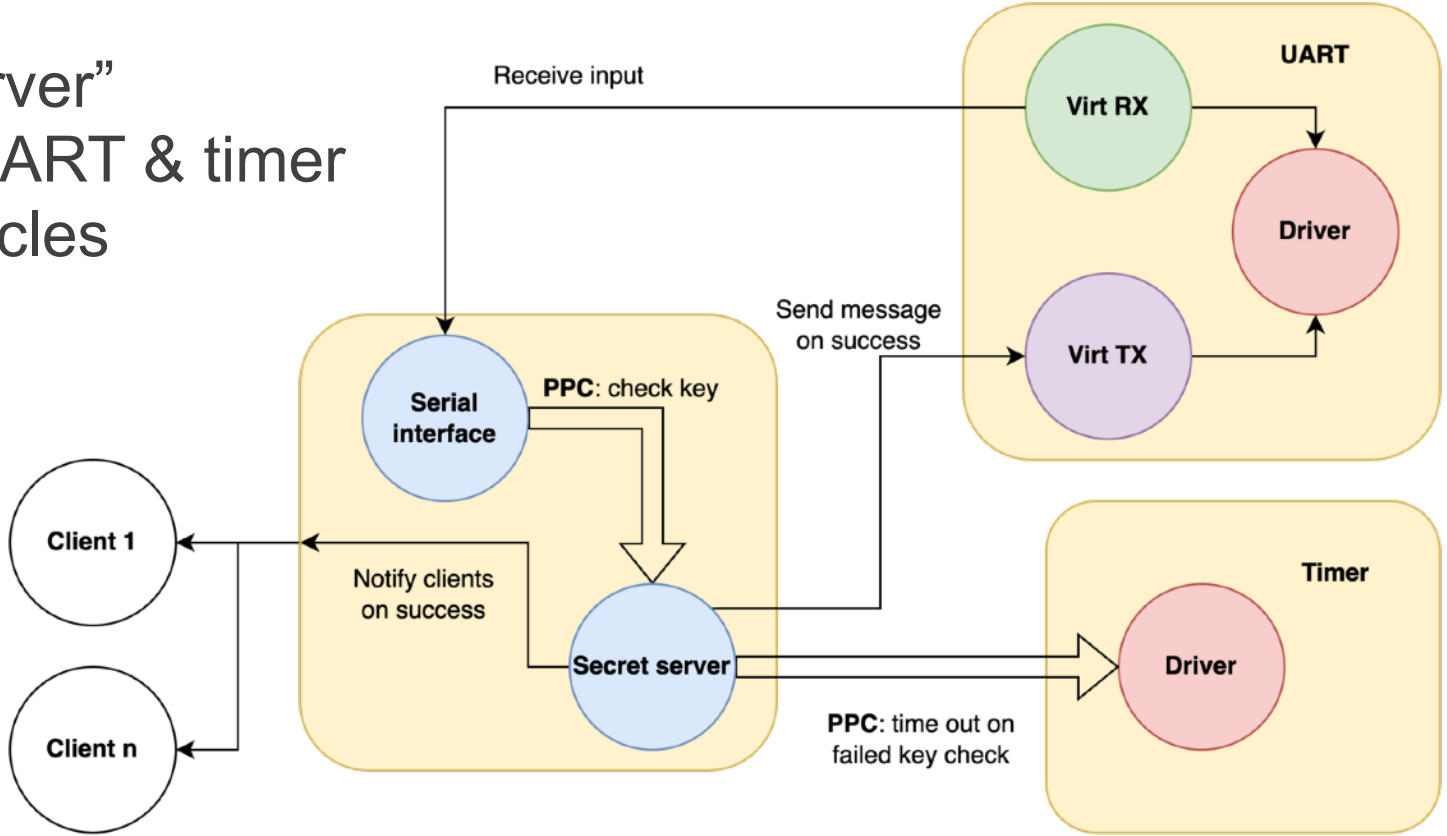
Far easier to show rather than tell

Walkthrough

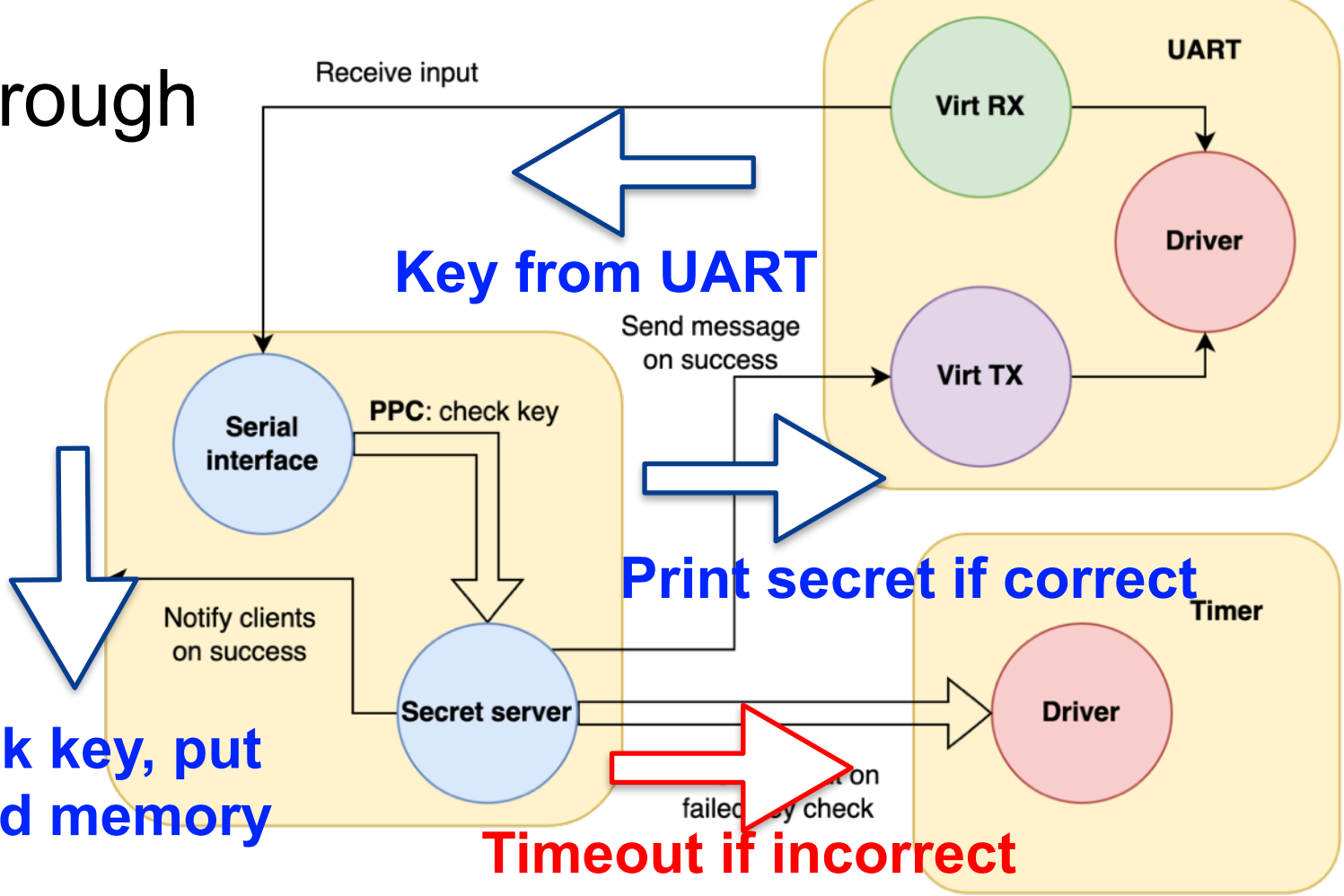
- Let's make a demo system!
  - **Goal 1:** exercise most of the Microkit
  - **Goal 2**: showcase how to modularise your code AND use existing modules
  - **Goal 3**: demo a client-server relationship
- What shall we make?

Walkthrough

- “Secret server”
  - sDDF UART & timer
  - PDs: circles

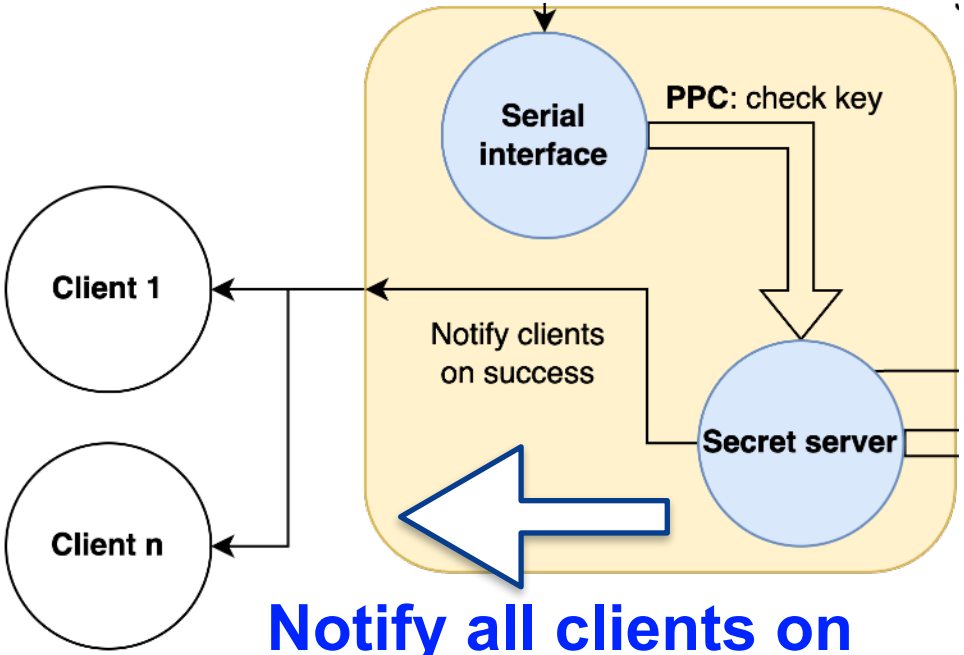


Walkthrough



PPC to check key, put key in shared memory

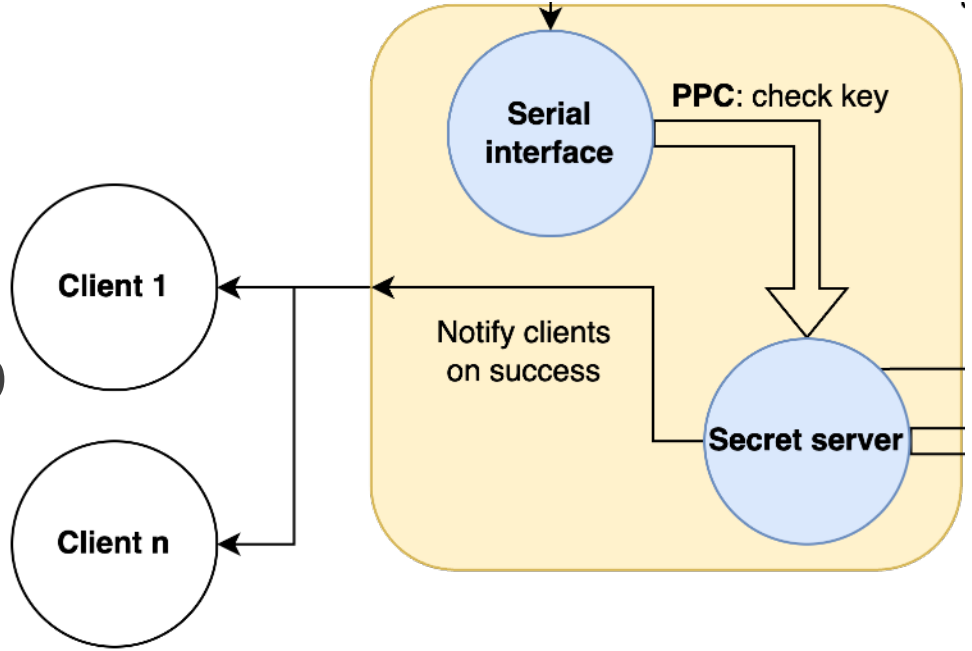
Walkthrough



Notify all clients on success

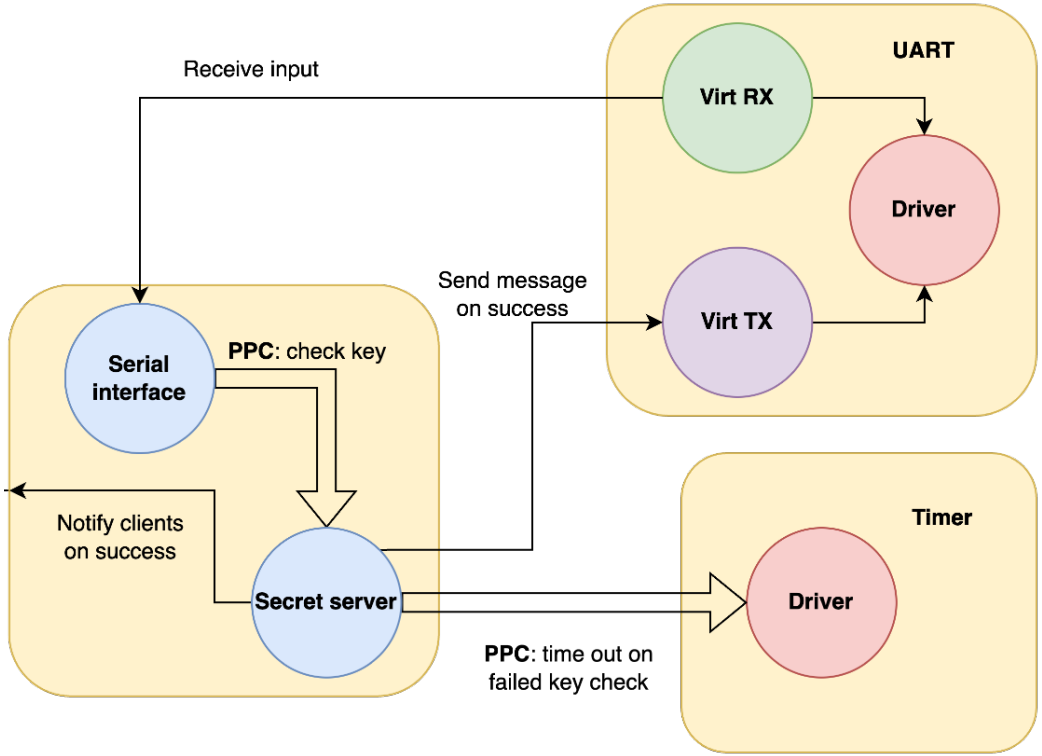
Walkthrough: Microkit objects?

- Subsystem needs…
  - Two Protection Domains
  - One channel
  - One memory region, mappings into each PD
- Each client needs…
  - One channel



Walkthrough: config data?

- Both PDs:
  - Channel ID
  - Address of shared memory region
- Secret server
  - Timeout duration
  - Key
  - Secret
  - Channel ID of each client



Walkthrough code

Demo is on GitHub

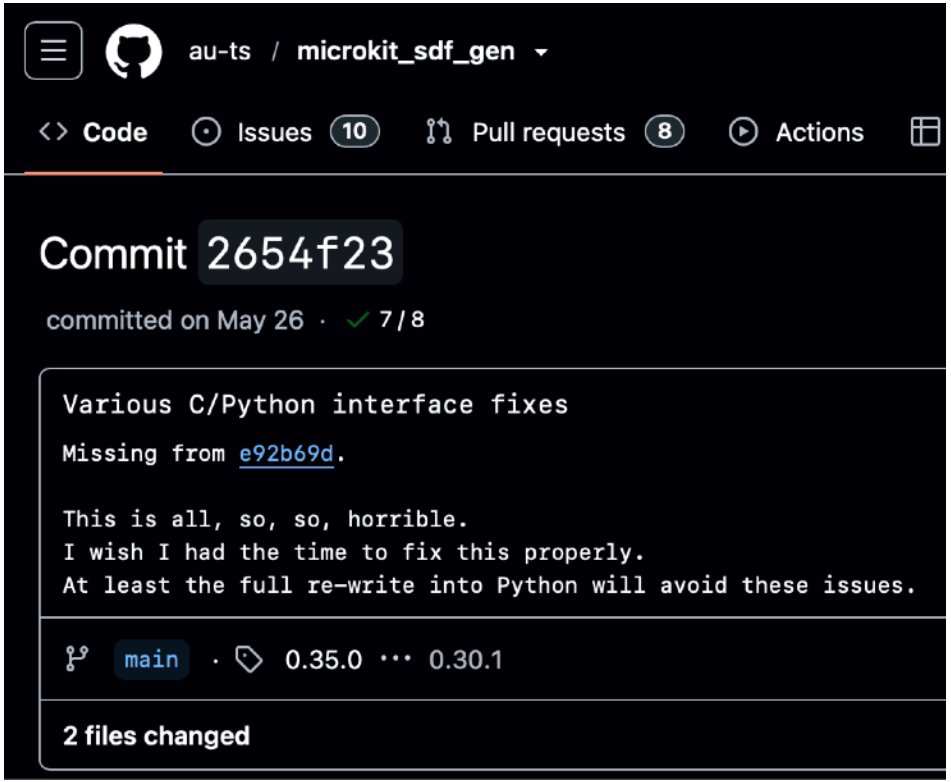
Acacia and the seL4 ecosystem

Where does it fit?

CAmkES

First attempt: sdf_gen

- **Zig** tool with Python bindings; generates Microkit primitives from Python file
- **Turned out inflexible**: monolithic design, no module abstraction, no DWARF parsing

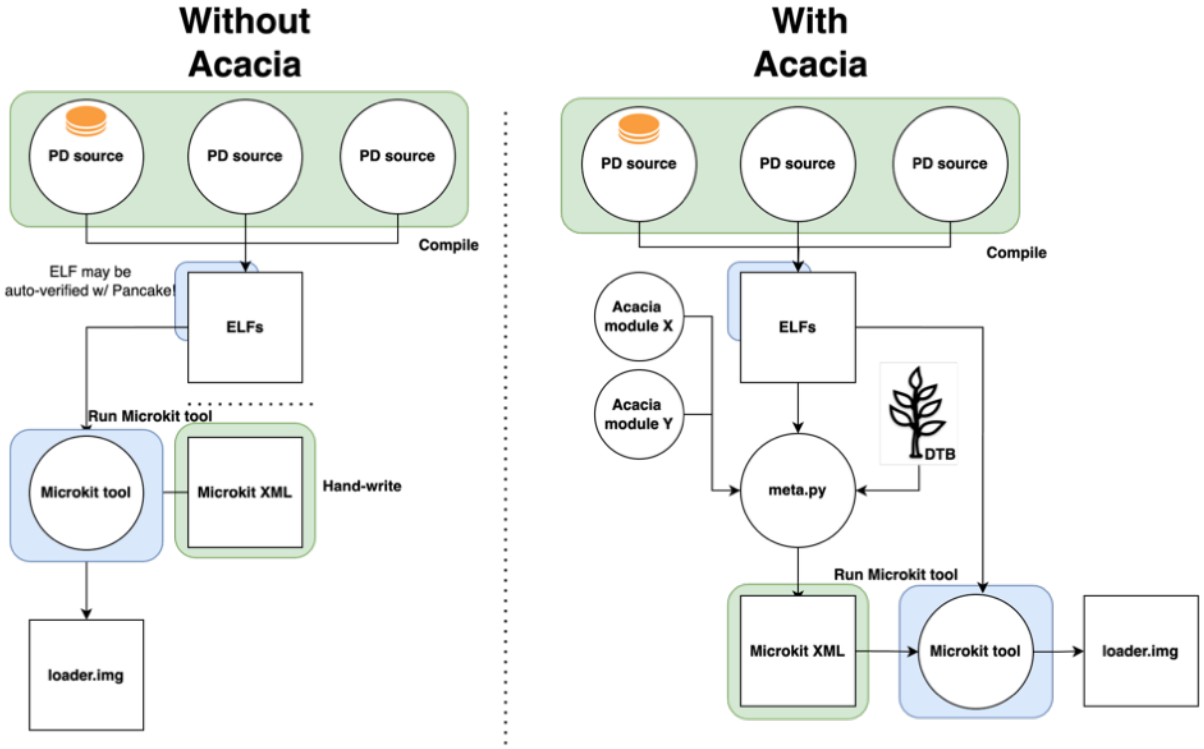


One of the final commits to sdf_gen

What does Acacia mean for the TCB and verification?

Tl;dr: not much!

- **Without Acacia**: XML files need human review
- **With Acacia**: XML files still need review
  - … but are far easier to work with!
  - May reduce (minor) bugs … reuse of known-good code rather than bespoke



What’s next?

Current state

- Currently:
  - ✅ Subsystem builder capable of managing ALL current LionsOS code reuse
  - ✅ Acacia can fully replace sdf_gen
- In progress
  - ➡️ SOON Support for all sDDF drivers used in LionsOS (merging now)
  - ➡️ SOON Support for LionsOS firewall & KITTY
  - 🏗️ YAML declarative interface
  - ✅ Dependency graph resolver (not merged yet)
- Future
  - Build system integration (avoid Make snippets?)

Current state

- ✅ Acacia CAN:
  - Generate (valid) Microkit XML easily
  - Enable frictionless use of device trees for Microkit projects
  - Significantly simplify configuring components
  - Allow modularisation of almost anything you can express in Microkit
- 🙅 Acacia CAN'T (yet):
  - Build your source code for you
  - Generate config structs for Pancake (no DWARF!)

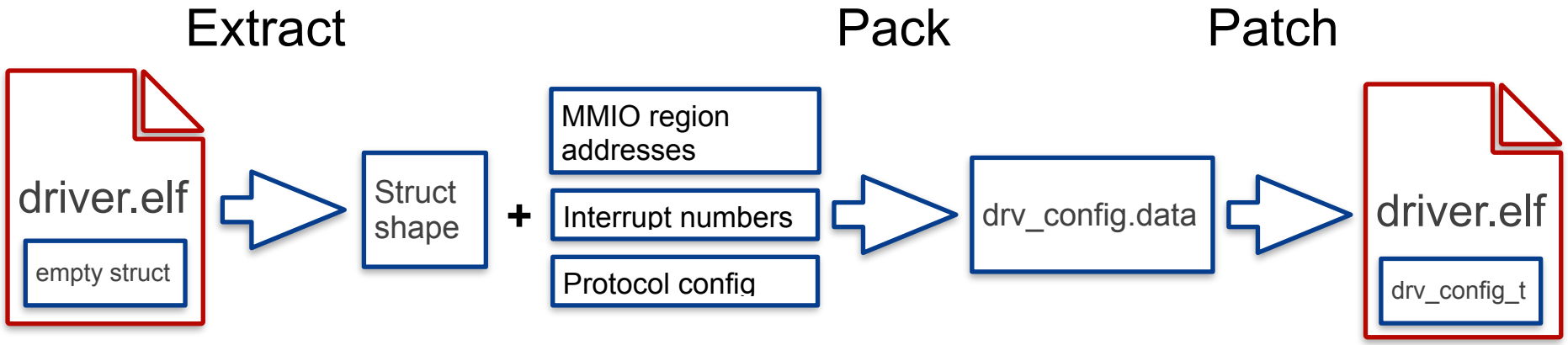
Questions?

Have thoughts on Acacia, Microkit, sDDF or LionsOS tooling? Send me an email!

lesley.rossouw@unsw.edu.au

Walkthrough

Configuring components



Complete flow, without replication!

Microkit strips debug symbols when assembling ELFs