Leveraging the Compiler

Static Analysis in Marine Robotic Systems

Toby Schneider

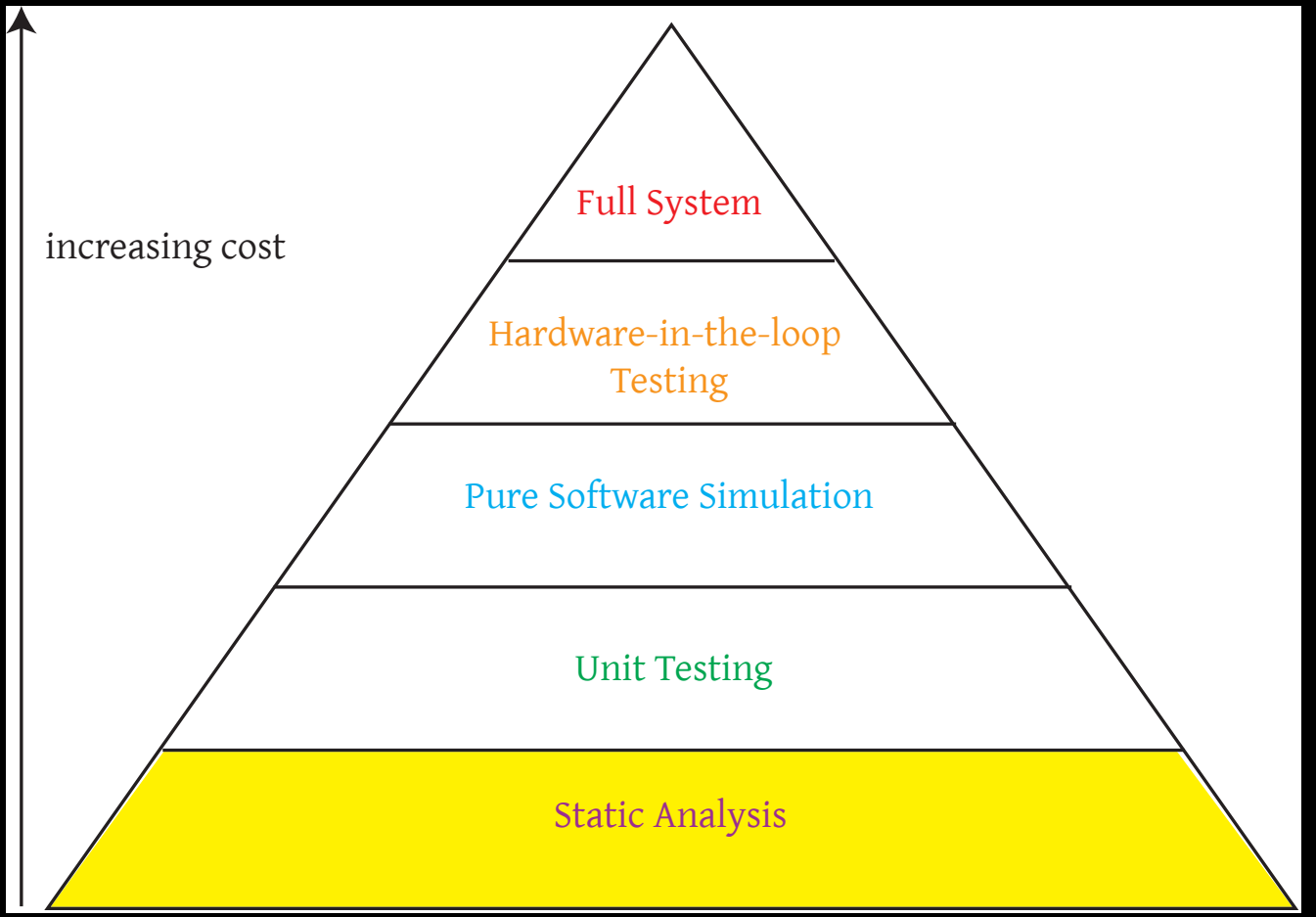
GobySoft, LLC

North Falmouth, MA, USA

MOOS-DAWG 2019, Cambridge, MA

Why?

- AUV field testing is incredibly expensive
- Bugs caught early cost less [money, time, frustration]



A collection of (static analysis) toys

Static analysis == tools run at compile time

- "Core" static code analysis (not marine specific)
  clang-analyzer, Coverity scan
- Compile-time dimensional analysis ("unit-safety")
  Boost Units
- Class-based state machines
  Boost Statechart, QP
- Automated publish/subscribe network analysis
  goby_clang_tool
- IvP Behavior domain completeness
  (conceptual)

Clang Analyzer

Wrapper “scan-build” can be used to run static analysis on existing codebase.

For example, for normal build of:

```
cmake .
make
```

You can run the clang-analyzer with:

```
scan-build cmake .
scan-build make
```

“Unit safety”

Compile-time dimensional analysis - using C++ type safety to enforce “unit safety” (demo based on boost::units)

Why? (all real cases I’ve seen)

- double roll = 0.5; // degrees or radians?
  -> auto roll = 0.5 * si::radians;
- float depth = 100; // meters, feet, or fathoms?
  -> auto depth = 100 * si::meters;
- double speed = 2.5; // m/s or knots?
  -> auto speed = 2.5 * knots;

Problem is amplified when projects grow and include more contributors.

Even enforcing SI use is suboptimal (latitude in radians...)

Units in DCCL

```
message PressureSample
{
   required double pressure = 1
      [(dccl.field) = { min: 0 max: 2e6 precision: 1
                        units { system: "si"
                                derived_dimensions:
                                        "pressure" }];

}
```

Creates "_with_units()" fields:

```
PressureSample sample;
// set
sample.set_pressure_with_units(100*si::deci*bars);
// get
quantity<si::pressure> value_pascals(
                          p.pressure_with_units());
```

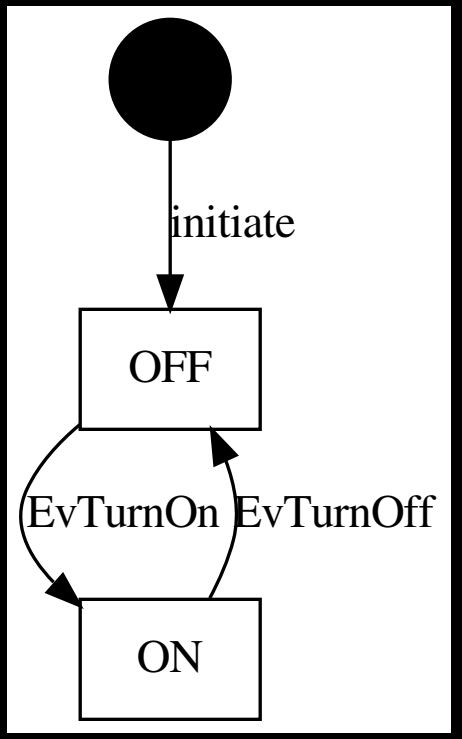
Class based state machines

Much of the code we write for robotics systems are asynchronously triggered state machines

- Publish/subscribe architecture provides the event triggers
- State machines in C++?
  - lots of booleans (very error prone)
  - enums and switch / case (blows up quickly with nested machines)
  - table driven (hard to add/remove states)
  - class based:

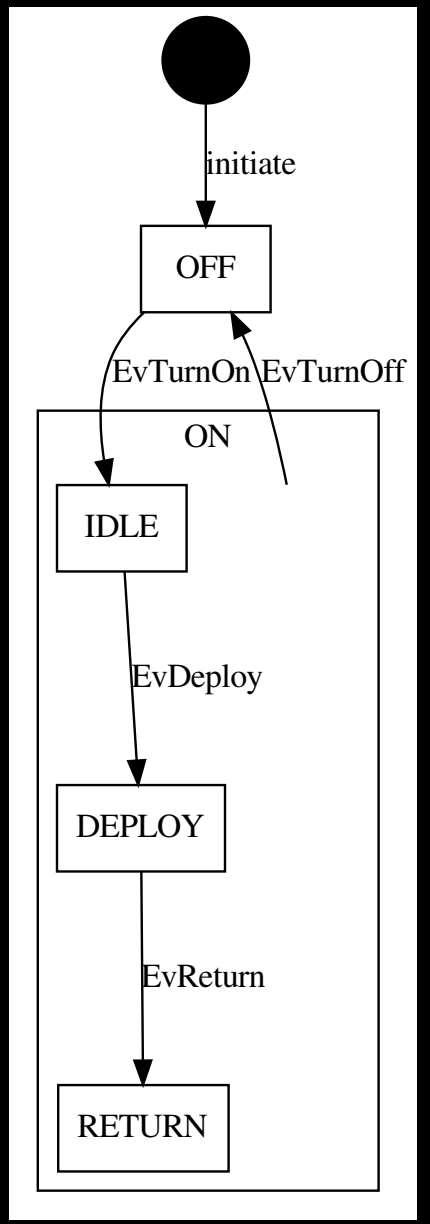
    Constructor = Entry action,
    Destructor = Exit action

A very simple machine




```
// events
struct EvTurnOn : event<EvTurnOn> {};
struct EvTurnOff : event<EvTurnOff> {};

struct On; struct Off;

// machine
struct Machine : state_machine<Machine, Off> {};

// states
struct On : simple_state<On, Machine>
{
    On() { std::cout << "Entering state:\tON" << std::endl; }
    ~On() { std::cout << "Leaving state:\tON" << std::endl; }
    using reactions = transition<EvTurnOff, Off>;
};

struct Off : simple_state<Off, Machine>
{
    Off() { std::cout << "Entering state:\tOFF" << std::endl; }
    ~Off() { std::cout << "Leaving state:\tOFF" << std::endl; }
    using reactions = transition<EvTurnOn, On>;
};
```

A hierarchical machine




```
struct EvTurnOn : event<EvTurnOn>{};
struct EvTurnOff : event<EvTurnOff>{};
struct EvDeploy : event<EvDeploy>{};
struct EvReturn : event<EvReturn>{};

struct Machine : state_machine<Machine, Off> {};
struct On : simple_state<On, Machine, on::Idle>
{
    On() { cout << "Entering state:\tON" << endl; }
    ~On() { cout << "Leaving state:\tON" << endl; }
    using reactions = transition<EvTurnOff, Off>;
};
namespace on {
struct Idle : simple_state<Idle, On>
{
    Idle() { cout << "Entering state:\tON::IDLE" << endl; }
    ~Idle() { cout << "Leaving state:\tON::IDLE" << endl; }
    using reactions = transition<EvDeploy, Deploy>;
};
struct Deploy : simple_state<Deploy, On>
{
    Deploy() { cout << "Entering state:\tON::DEPLOY" << endl; }
    ~Deploy() { cout << "Leaving state:\tON::DEPLOY" << endl; }
    using reactions = transition<EvReturn, Return>;
};
struct Return : simple_state<Return, On>
{
    Return() { cout << "Entering state:\tON::RETURN" << endl; }
    ~Return() { cout << "Leaving state:\tON::RETURN" << endl; }
};
} // namespace on

struct Off : simple_state<Off, Machine>
{
    Off() { cout << "Entering state:\tOFF" << endl; }
    ~Off() { cout << "Leaving state:\tOFF" << endl; }
    using reactions = transition<EvTurnOn, On>;
};
```

You could envision a compiler tool that draws the statechart from the code

Publish/Subscribe graph analysis

Why?

- Challenging to track publish/subscribe interfaces on AUVs with dozens of processes running.
- Verification that required subscriptions are being published by something.
- Goby3 extends to intervehicle/interthread transports and arbitrary marshalling languages which increases expressiveness but also increases complexity.
  - Ensure we're publishing and subscribing same group (~= MOOS variable name)
  - Ensure we're publishing and subscribing same type (~= double or string in MOOS)

Publish/Subscribe graph analysis

How?

- goby3 defines Groups (~=MOOS variable) as constexpr:
  `constexpr goby::middleware::Group nav("navigation")`
- "goby_clang_tool" based on Clang's libtooling
- While compiling, the tool searches through Clang AST:
  -> finds publish() and subscribe() method calls

```
interprocess().publish<groups::nav>(nav);
```

```
interprocess().subscribe<groups::nav, NavigationReport>(
    nav_callback);
```

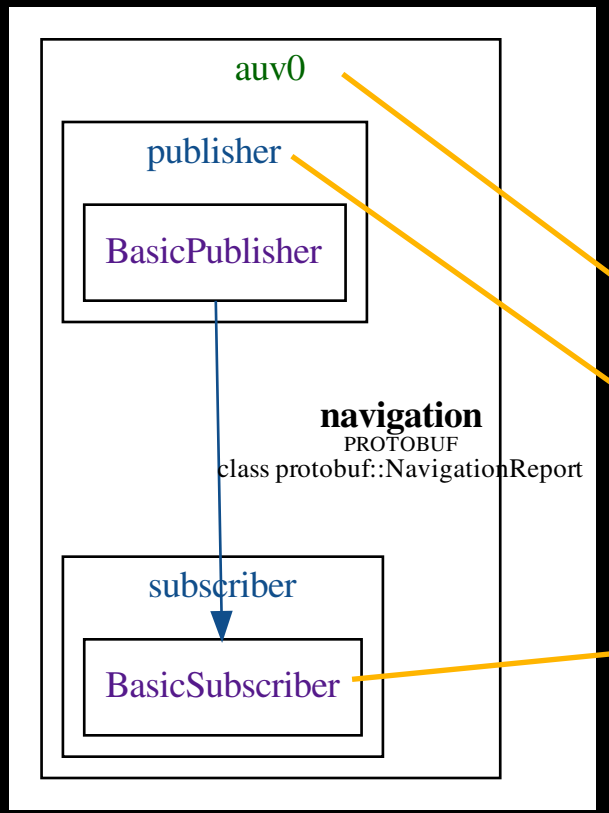
-> reads group, scheme, and type (AST declarations)
-> outputs an "interface" YAML file

```
application: publisher
interprocess:
  publishes:
    - group: navigation
      scheme: PROTOBUF
      type: "class protobuf::NavigationReport"
      thread: BasicPublisher
  subscribes:
    []
```

```
application: subscriber
interprocess:
  publishes:
    []
  subscribes:
    - group: navigation
      scheme: PROTOBUF
      type: "class protobuf::NavigationReport"
      thread: BasicSubscriber
```

Publish/Subscribe graph analysis

- Each interface YAML file defines the publish/subscribe interface for the application.
- In combination with a “deployment” file (showing which applications will be run where), a full graph can be produced (automatically, at compile time):




```
deployment: test
platforms:
  - name: auv0
    interfaces:
      - goby3_example_basic_interprocess_subscriber_interface.yml
      - goby3_example_basic_interprocess_publisher_interface.yml
```

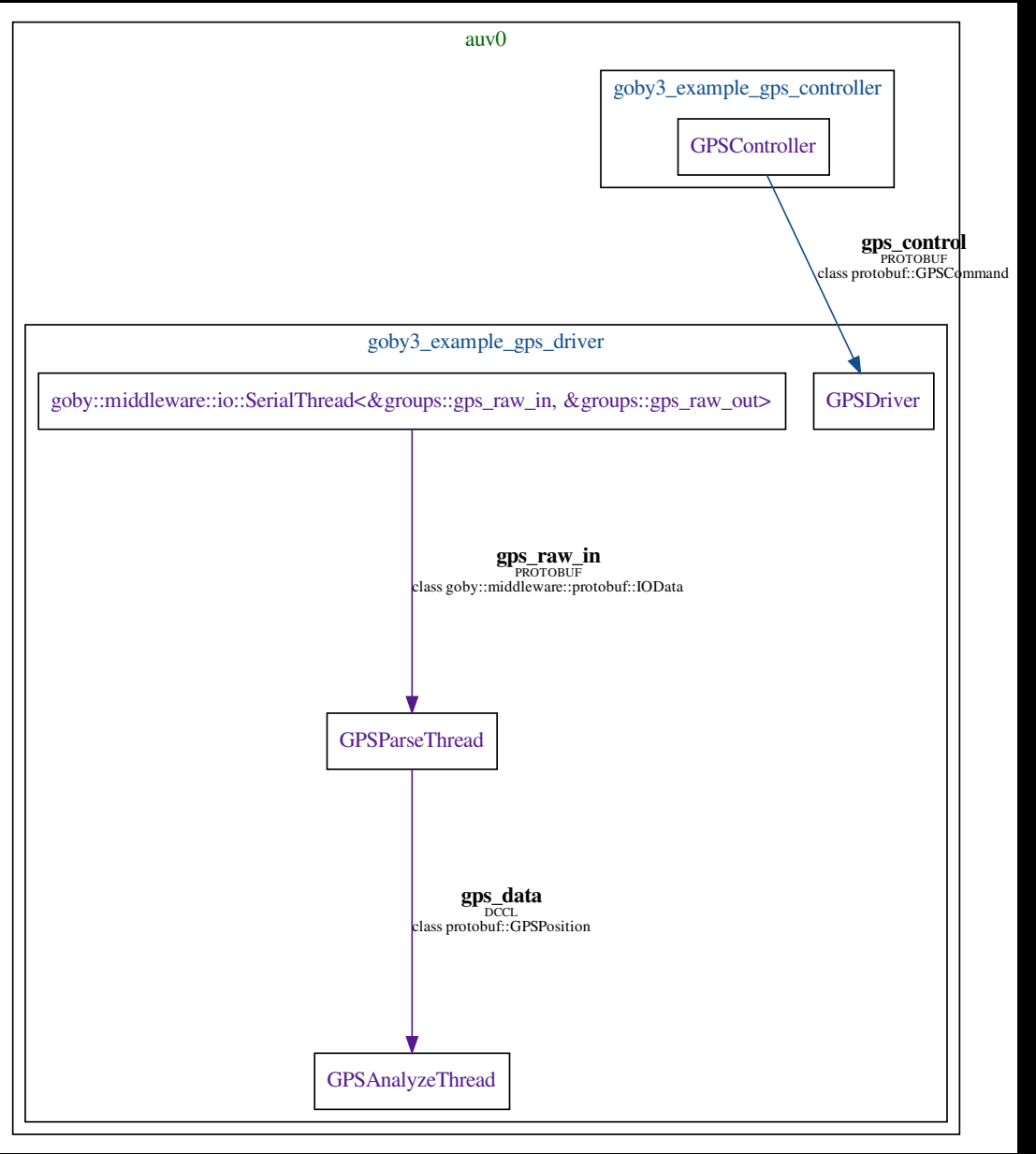
vehicle name

process name

thread name

Pub/Sub graph analysis: GPS Driver

- SerialThread reads serial port and publishes NMEA lines
- GPSParseThread subscribes for lines and writes parsed positions
- GPSAnalyzeThread subscribes for parsed positions.
- External GPSController can start/stop SerialThread



IvP Behavior completeness

Some ideas to ponder:

- It is easy to forget to include a behavior for each domain in every mode (e.g. forget a ConstantDepth behavior)
- If behaviors defined their domains statically (constexpr or some such) like Goby3 groups, this information could be fed into a tool that reads the .bhv
  -> ensure every mode has at least one behavior for each domain

Final thoughts

Play with the code yourself:

- https://github.com/GobySoft/moos-dawg-static-analysis-examples.git

Let me know if you're interested in getting involved with any of this:

- toby@gobysoft.org