

Combining Static and Dynamic Verification for the Analysis of OCaml Programs

Gonalo Pereira Duarte

Advisor: Mrio Pereira

February 21, 2025

► Introduction

- State of the Art
- Preliminary Results
- Work Plan

Motivation

Introduction

Programmers need to [test software](#) before release.

- ▶ Error-prone
- ▶ Time-consuming
- ▶ More demanding with increasing complexity of systems

Motivation

Introduction

Solutions:

- ▶ Extensive testing
- ▶ Type-checking (even before testing)
- ▶ Automatic runtime verification
 - ▶ ... and still all done separately may not be enough!
- ▶ what about proof of programs?
 - ▶ ... that might be too costly and time-consuming!

Research Questions

Introduction

- ▶ *Is it possible to [combine static and dynamic verification](#) for OCaml programs?*
 - ▶ Is it possible to create an [executable subset of GOSPEL](#)?
 - ▶ Should we use [Runtime Assertion Checking](#) (RAC) when deductive verification fails?

Research Questions

Introduction

- ▶ *Is it possible to combine static and dynamic verification for OCaml programs?*
 - ▶ Is it possible to create an executable subset of GOSPEL?
 - ▶ Should we use Runtime Assertion Checking (RAC) when deductive verification fails?
 - ▶ Perhaps...
 - ▶ They are not mutually exclusive, if applied correctly.

Research Questions

Introduction

- ▶ *Is it possible to combine static and dynamic verification for OCaml programs?*
- ▶ Is it possible to create an executable subset of GOSPEL?
- ▶ Should we use Runtime Assertion Checking (RAC) when deductive verification fails?

Work to be done... but E-ACSL has proven to be of great importance!
Why shouldn't E-GOSPEL follow the same path?

Research Questions

Introduction

- ▶ *Is it possible to combine static and dynamic verification for OCaml programs?*
- ▶ Is it possible to create an executable subset of GOSPEL?
- ▶ Should we use Runtime Assertion Checking (RAC) when deductive verification fails?

Monitor is the technique for the job!

Expected Contributions

Introduction

- ▶ Research and Identify an executable set of GOSPEL - [E-GOSPEL](#);
 - ▶ Translation of contracts defined by user into assertions that can be run
- ▶ Implement [Monitors](#) that can help with the verification process;
 - ▶ Bridge between static and dynamic schools of verification
- ▶ Evaluate effectiveness of the verification process with both [E-GOSPEL](#) and [Monitors](#);

- Introduction
 - ▶ State of the Art
- Preliminary Results
- Work Plan

Current Approaches

State of the Art

Divided into three sets:

- Combining static and dynamic verification
- Current tools
- Executable Specifications for RAC

Combining Static and Dynamic Verification

State of the Art

- [GOSPEL](#) - the best unification for [OCaml](#)
 - Previous works done by Soares, Chirica and Pereira in “*Static and Dynamic Verification of OCaml Programs: The Gospel Ecosystem*”
- [Cameleer](#) (deductive verification)
- [ORTAC](#) (runtime assertion checking)

Current Tools

State of the Art

For dynamic verification:

- ▶ [JML](#) - Java Modelling Language
- ▶ [ACSL](#) - ANSI/ISO C Specification Language
 - ▶ [E-ACSL](#) - executable subset of ACSL
- ▶ [SPARK](#) - Ada
- ▶ [ORTAC](#) - for OCaml

Current Tools

State of the Art

For static verification:

- ▶ [Frama-C](#) - C
- ▶ [Cameleer](#) - OCaml
- ▶ [SPARK](#) - Ada & C
- ▶ [KeY](#) - Java
- ▶ [VeriFast](#) - Java & C

- Introduction
- State of the Art
 - ▶ Preliminary Results
- Work Plan

Queue Example

Preliminary Results

- ▶ Simple implementation;
- ▶ Straight-forward specifications;
- ▶ Use of *List* from OCaml libraries - easy understanding of code;
 - ▶ Example becomes much more easier to understand.

Queue Example - Implementation

Preliminary Results

```
open List
```

```
type  $\alpha$  queue = {  
  mutable front :  $\alpha$  list;  
  mutable back :  $\alpha$  list;  
  mutable size : int;  
}
```

```
let[@logic] is_empty q = q.size = 0
```

```
let make () =  
  { front = []; back = []; size = 0 }
```

```
let pop a =  
  let x =  
    | [] → raise Not_found  
    | [ x ] →  
      a.front ← List.rev a.back;  
      a.back ← [];  
      x  
    | x :: xs →  
      a.front ← xs;  
      x  
  in  
    a.size ← a.size - 1;  
    x
```

```
let push a x =  
  if is_empty a then a.front ← [ x ] else a.back ← x :: a.back;  
  a.size ← a.size + 1
```

NOTE: The `Queue` is implemented using two elements of type `List`

Queue Example - Specification

Preliminary Results

```
type  $\alpha$  t
(*@ mutable model view:  $\alpha$  list *)

val is_empty :  $\alpha$  t  $\rightarrow$  bool
(*@ b = is_empty a
   ensures b  $\leftrightarrow$  t.view = []*)

val make : int  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$  t
(*@ t = make ()
   ensures t.view = []*)

val pop :  $\alpha$  t  $\rightarrow$   $\alpha$ 
(*@ a = pop t
   modifies t.view
   requires t.view  $\neq$  []*
   ensures t.view =
     if old t.view = []
     then []
     else List.tl (old t.view)
   ensures if old t.view = [] then false
     else a = List.hd (old t.view)*)

val push :  $\alpha$  t  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$ 
(*@ push t a
   modifies t.view
   ensures t.view = append_last a (old t.view)*)
```

modifies clause for identification of altered elements

Example: `pop` will modify the `t.view` element when the function ends.

```
val pop :  $\alpha$  t  $\rightarrow$   $\alpha$ 
(*@ a = pop t
   modifies t.view
```

requires handles pre-conditions of functions

Example: `pop` requires that `t.view` must not be empty.

```
val pop :  $\alpha$  t  $\rightarrow$   $\alpha$       requires t.view  $\neq$  []*
```

ensures clause for post-conditions of functions

Example: `make` **always** returns `t.view` as `[]`.

```
val make : int  $\rightarrow$   $\alpha$   $\rightarrow$   $\alpha$  t
(*@ t = make ()
   ensures t.view = []*)
```

Queue Example - Explanation

Preliminary Results

- ▶ Example verified and checked in both [Cameleer](#) and [ORTAC](#).

- ▶ Error cases:

1. Errors **in the code**

- ▶ [Cameleer](#) - inconclusive proof
- ▶ [ORTAC](#) - runtime error

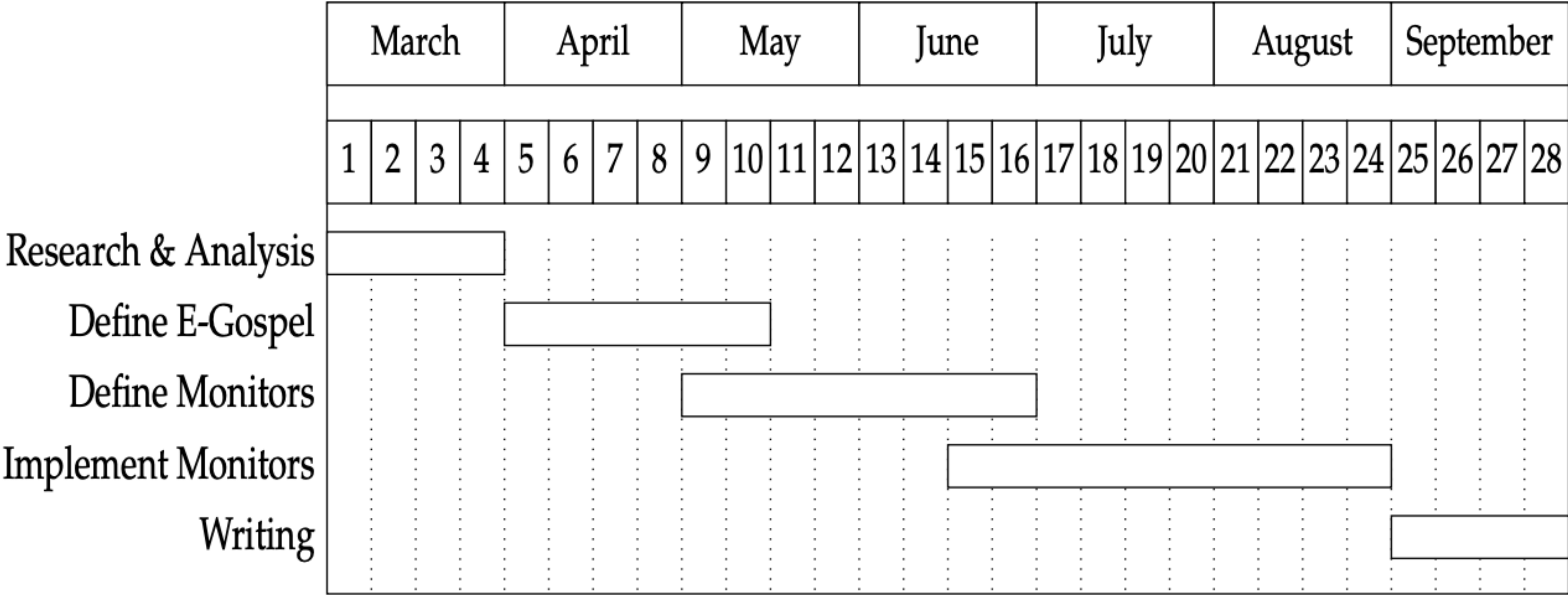
2. Errors **in the specification**

- ▶ [Cameleer](#) - anything can happen
- ▶ [ORTAC](#) - anything can happen

- Introduction
- State of the Art
- Preliminary Results
 - ▶ Work Plan

Gantt Chart

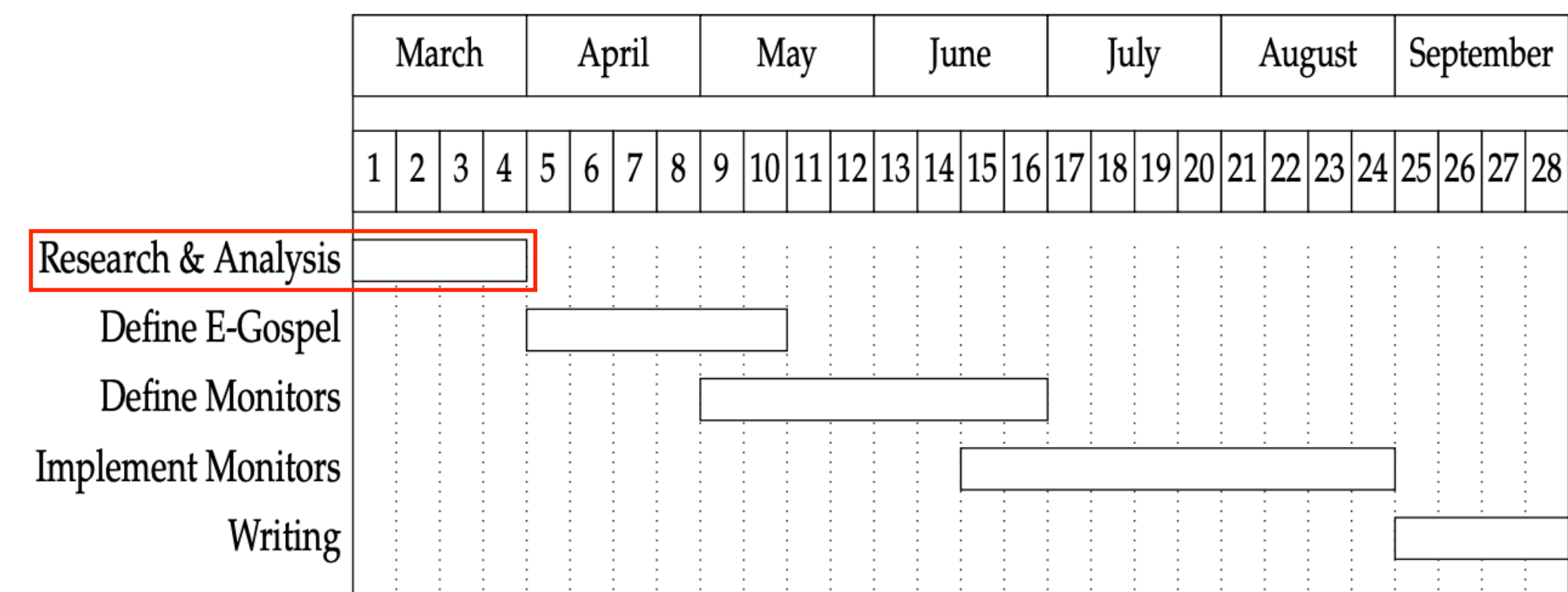
Work Plan



Research & Analysis

Work Plan

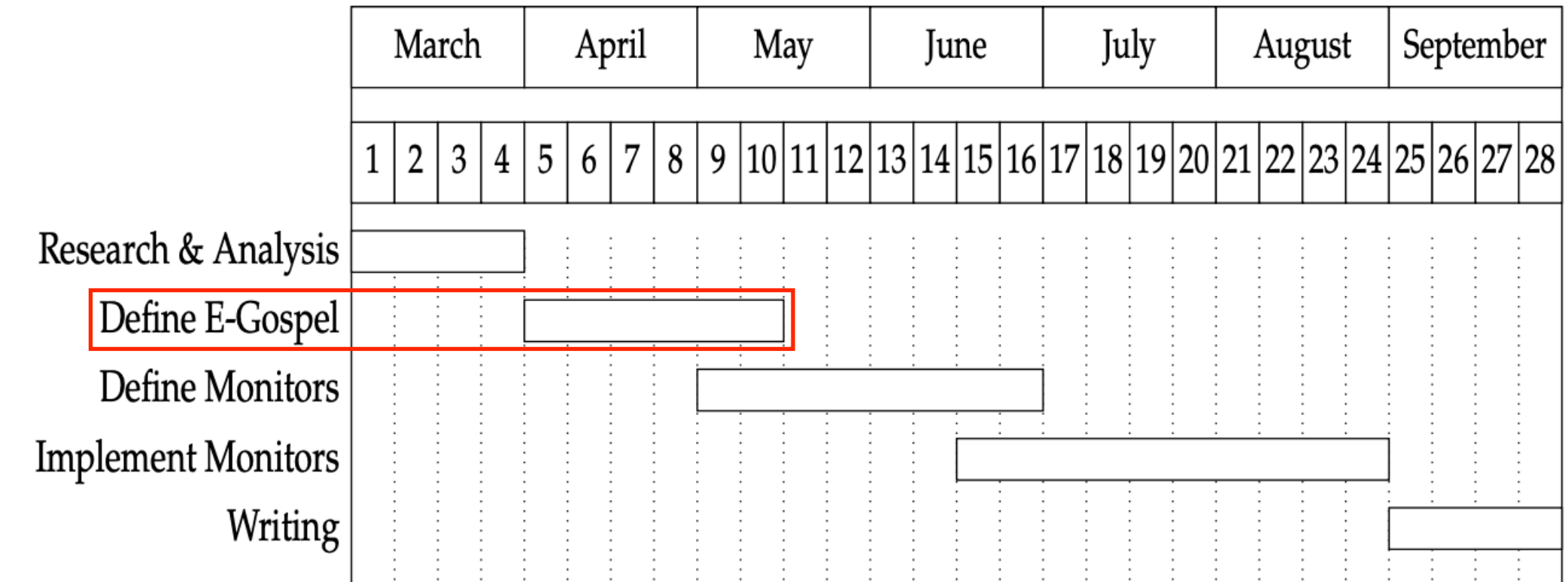
- Research study cases
- Analyse cases in ORTAC + Cameleer
- Gather data for future work
- Duration:
 - From first week of March until the end of the month
 - Approximately 4 weeks



Define E-GOSPEL

Work Plan

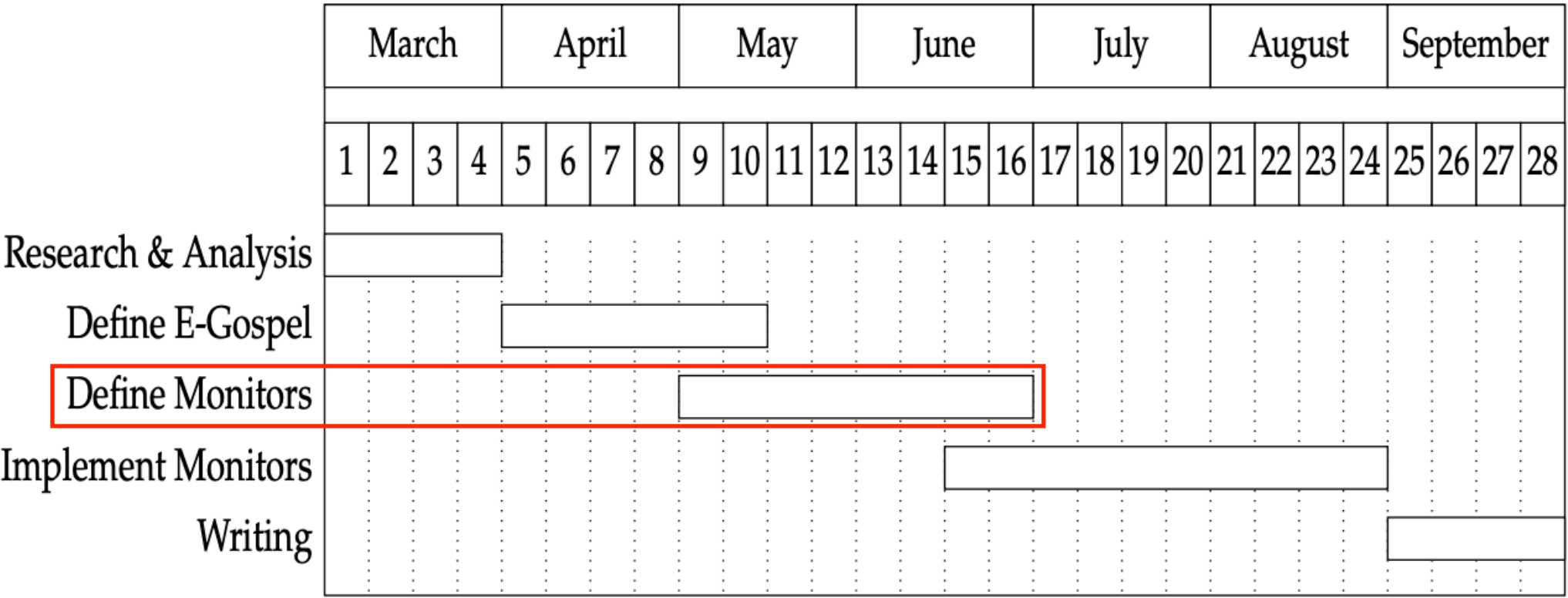
- Define [executable](#) portion of [GOSPEL](#) contracts
- Use data and results from [Research & Analyse](#)
- Duration:
 - From [first week of April](#) until [the second week of May](#)
 - Approximately [6 weeks](#)

[illegible]

Define Monitors

Work Plan

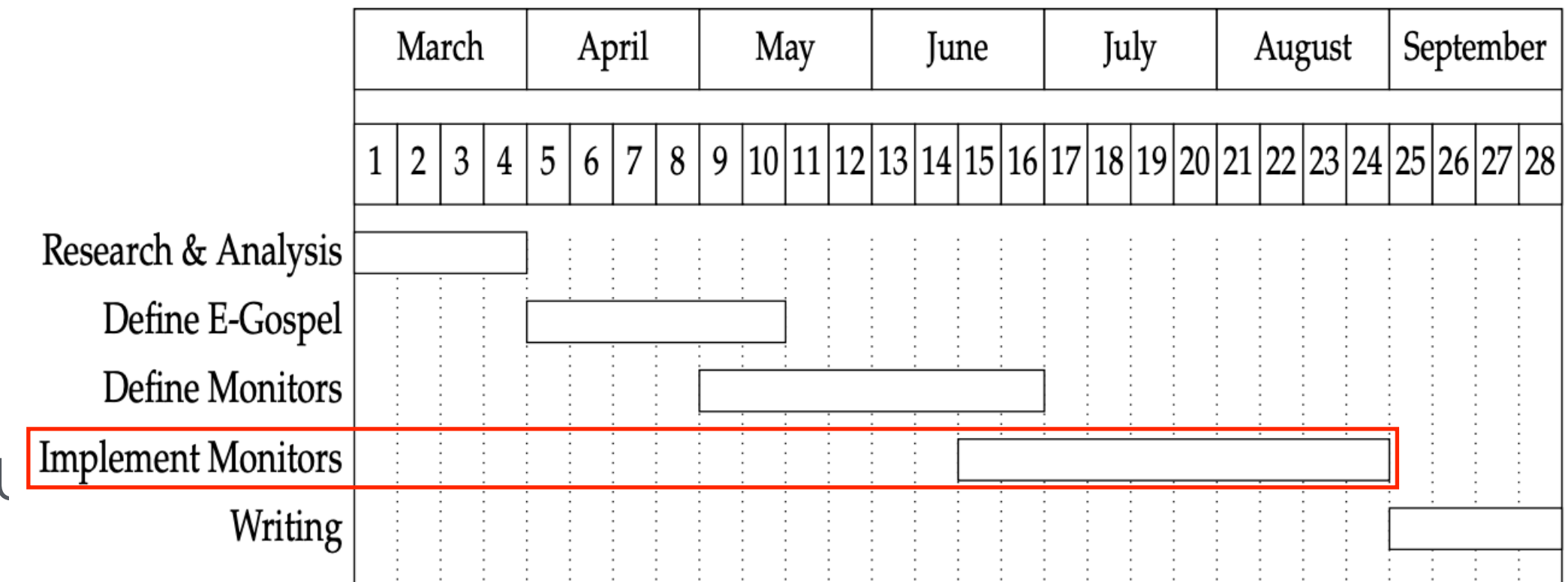
- Using E-GOSPEL, define the [specification of monitors](#)
- Analyse some cases and examples for refinement
- Duration:
 - From [first week of May](#) until the [end of June](#)
 - Approximately [8 weeks](#)
 - Coincides with *Define E-GOSPEL* as it is concurrent work



Implement Monitors

Work Plan

- [Implementation of Monitors](#) using results from previous
- Most demanding step - also the [most crucial!](#)
- Duration:
 - From [second week of June](#) until the [end of August](#)
 - Approximately [10 weeks](#)
 - Coincides with *Define Monitors* as it is concurrent work



Writing

Work Plan

- [Writing](#) of the dissertation
- Involves [all the previous steps' results](#)
- Last work to be done
- Duration:
 - [All of September](#)
 - Approximately [4 weeks](#)

