Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *February 11, 2025*

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *February 11, 2025*

Abstract

Software development has had a significant impact on the world, but sometimes it has not been for the better. There are many examples of software failures that have caused terrible outcomes. To mitigate this, software testers have developed many techniques that require significant human effort and time. Although these efforts have been successful in many cases, there are still many software failures.

The work we propose in this dissertation aims to improve the quality of software by resorting to formal verification techniques, not as a replacement for testing, but as an improved tool to help build more reliable software. Our work is based on the use of `Monitors`, a technique that uses formal verification to check the correctness of software. This is employed by the use of static verification and, for the parts of the code that were unable to be checked, RAC or *Runtime Assertion Checking*. Tools like GOSPEL, ORTAC, and Cameleer are used to implement this technique. Why3 is also essential to provide the necessary support for this work.

Here, we provide some background for OCaml- the language we will focus on, formal verification itself, GOSPEL, ORTAC, Cameleer, and Why3. We also present the work we have done in a `Queue` example, showing promising results in the implementation of the `Monitors`.

Finally, we present some conclusions and future work.

Keywords: OCaml, Formal Verification, Dynamic Verification, GOSPEL, ORTAC, Cameleer, Why3

Resumo

O desenvolvimento de software tem tido um impacto significativo no mundo, mas, por vezes, esse impacto não foi positivo. Há muitos exemplos de falhas de software que causaram consequências desastrosas. Para mitigar esse problema, *testers* de software desenvolveram várias técnicas que exigem um esforço humano considerável e tempo. Embora esses esforços tenham sido bem-sucedidos em muitos casos, ainda ocorrem muitas falhas de software.

O trabalho que propomos nesta dissertação visa melhorar a qualidade do software recorrendo a técnicas de verificação formal, não como um substituto de testes, mas como uma ferramenta aprimorada para ajudar a construir software mais confiável. O nosso trabalho baseia-se no uso de `Monitors`, uma técnica que utiliza verificação formal para verificar a correção do software. Isto é realizado através da verificação estática e, para as partes do código que não puderam ser verificadas, com RAC ou *Runtime Assertion Checking*. Ferramentas como GOSPEL, ORTAC e Cameleer são utilizadas para implementar essa técnica. O Why3 também é essencial para fornecer o suporte necessário a este trabalho.

Aqui, fornecemos algumas informações sobre OCaml – a linguagem na qual nos focaremos –, sobre a própria verificação formal, bem como sobre GOSPEL, ORTAC, Cameleer e Why3. Também apresentamos o trabalho que realizamos em um exemplo de fila (Queue), mostrando resultados promissores na implementação dos `Monitors`.

Por fim, apresentamos algumas conclusões e perspetivas para trabalhos futuros.

Palavras-chave: OCaml, Verificação Formal, Verificação Dinâmica, GOSPEL, ORTAC, Cameleer, Why3

Contents

Glossary

Alt-Ergo An open-source SMT solver developed at LRI, INRIA. *9)*

Cameleer A deductive verification tool for OCaml programs. *i, ii, 2, 4, 5, 8, 13, 19)*

Coq A formal proof management system. *9)*

GOSPEL Generic OCaml SPEcification Language - a behavioral specification language for OCaml. *i, ii, 2, 4, 5, 8, 9, 11–13, 15–17, 19)*

ORTAC OCaml Runtime Assertion Checker - a tool to integrate runtime assertion checking into OCaml programs. *i, ii, 2, 4, 5, 11–13, 19)*

RAC Runtime Assertion Checking - a specific technique that resorts to assertions to check the correctness of a program during its execution. *i, ii, 2, 3, 5, 10–14)*

Why3 A platform for deductive program verification. *i, ii, 5, 8, 9, 13, 14, 16)*

Z3 An efficient SMT solver developed at Microsoft Research. *9)*

1

Introduction

1.1 Motivation/Context

Errors are closely connected with human performance, and they are a part of everyday life. They can be found in all areas of human activity, and they can have different consequences, from minor inconveniences to catastrophic events. In the context of software development, errors can have a significant impact on the quality of the software, leading to financial losses, loss of reputation, and even loss of life. The complexity of software systems has been increasing over the years, and it is becoming more and more challenging to develop error-free software. This complexity is due to the increasing size of software systems, the increasing number of features, and the increasing number of interactions between different components. As a result, software developers are facing new challenges in developing software that is reliable, secure, and efficient.

Software verification consists of checking whether a software system meets its requirements and specifications. In the particular case of OCaml programs, static and dynamic verification are two powerful techniques that can be used to verify the correctness of OCaml programs. Static verification consists of analyzing the program's code without executing it, using techniques such as type checking, abstract interpretation, and model checking. These techniques can detect potential errors and prove properties about the program, such as the absence of certain types of runtime errors.

Dynamic verification, on the other hand, involves executing the program and observing its behavior to ensure it meets its specifications. Techniques such as testing, runtime assertion checking, and formal methods like model-based testing can be used to dynamically verify OCaml programs. These techniques can help identify errors that may not be detectable through static analysis alone.

The two can be combined to provide a more exhaustive verification of OCaml programs, collaborating for a more structured and correct code. This can even be more useful in the context of systems that have a direct impact in everyday life.

For example, `Monitors` follow a set of steps to ensure that the system is correctly implemented. First, static verification is performed to ensure that the code may not have any errors during compilation. Then, for the parts of the code that the previous step could not verify, dynamic verification is performed to verify the code's behavior during execution. This process is repeated until the system is fully verified.

1.2 Problem Definition

This work has the goal of assert the stability and validity of combining static and dynamic verification techniques to verify OCaml programs extensively, to the point whether the system is fully verified, both statically and dynamically. This goal may arise questions such as:

> *Is it possible to combine static and dynamic verification for OCaml programs?*

The question above introduces a large scope, bigger than what can be covered in this work. As such, we can split it into smaller questions:

1. Identify an executable subset of GOSPEL;
2. Should we use RAC when Deductive Verification (Static Verification) is not enough?

Work in this dissertation will focus on the second question, as it emcompasses both static and dynamic verification techniques in a single process/technique. It is important to note that the described process in the second question will appear along the work, as it basically describes the process of `Monitors`, one of the main focus here.

For a more visual example, consider the following code snippet:

GOSPEL + OCaml

```
while B do S done
(* @invariant I *)
```

We can note that the invariant `I` can be defined as a `Monitor` `M[[I]]` - it will be verified statically, and if it is not possible, a dynamic verification will be performed.

Our work will focus on turning the above code into a more practical format, as we now present:

GOSPEL + OCaml

```
while B do
    assert M[[I]];
    S
    assert M[[I]];
done
```

This change will allow us to verify the invariant in a more practical way by translating contracts into built-in code assertions. This will produce both code and its verification process in the same file and in a way that it can be verified semi-automatically.

For this to be done, we will have to understand tools like Cameleer and ORTAC and their capacity to generate `Monitors` for dynamic verification when static verification fails.

1.2.1 Static Verification

Source code can be verified without the machine running it. This process allows to programmer to find error in type-checking, abstract interpretation, model checking, theorem proving and symbolic execution. This is a form of static verification - verifying the correctness of code without the need to run it. For programmers and testers, it is helpful to verify if the code being written/analysed is correct without wasting resources (time, CPU, RAM) by running the program, and static verification techniques provide just the solution.

Static verification can enhance early error detection during the development process, when the code is still being written, aided by the exhaustive analysis of the code, that covers all logical paths and scenarios for a given domain. It is a resource-efficient approach that gives the garantee of correctness of critical pieces of code. This leads to an overall efficiency in verifying code.

1.2.2 Runtime Assertion Checking

Runtime Assertion Checking (RAC) is a powerful technique that can be used to verify preconditions, postconditions, and invariants, proving to be of utmost importance in the verification of programs, specially in the context of the OCaml language [0], where the type system, although powerful, may not be enough to ensure the correctness of the code.

RAC is used during runtime, which by itself means that there needs to be a precondition required by the programmer: the code must be executable. In other words, RAC cannot be used if the code has errors that prevent it from being executed. This technique is better suited for verifying dynamic structures and algorithms, as these are mutable and may change during execution. This is a limitation of static verification, which is unable to verify mutable pieces of code.

Although very powerful, RAC is not enough to ensure the correctness of the code. One of its limitations is errors that can arise from the code that is static. For example, it may be unable to type-check code, check declarations or definitions, or verifying that the code is not going to compile.

This is where static verification comes in. It is a technique that can be used to verify the correctness of the code before it is executed. As explained in **??**, this is done by analyzing the code without executing it, using techniques such as type checking, abstract interpretation, and model checking. This technique can detect potential errors during compilation, thus ensuring that the code is able to run. Both dynamic and static verification can be combined to provide a more exhaustive and complete verification of the code.

One example of cooperation between the two techniques is applied in `Monitors`- a technique that, in first instance, uses static verification to verify some code, and then uses dynamic verification to verify the parts of the code that previously were unable to be verified. This process is repeated until the code is fully verified.

1.3 Expected Contributions

This work aims to contribute to the field of software verification of OCaml programs. We will focus first in developing a set of cases that will be abled to be verified both statically and dynamically. The second step would be to implement `Monitors` that can help in the verification process. Thus, the main tasks are:

- Research and identify an executable set of GOSPEL- E-GOSPEL, as in Executable GOSPEL;
- Implement `Monitors` that can help in the verification process;
- Evaluate the effectiveness of the verification process with the aforementioned sets.

1.4 Report Structure

- In Chapter **??** we present the necessary information to understand the context of the work, such as the tools, techniques and concepts involved. All the information presented in this chapter is essential to understand the work.
- Chapter **??** presents the existing work in the area of both static and dynamic verification of OCaml programs and how they are currently combined to verify software systems.
- Chapter **??** will focus on presenting preliminary results of some exercises that were performed. These exercises consist of verifying the effectiveness of verification in a `Queue` implementation in OCaml. Resorting to GOSPEL, Cameleer and ORTAC, this is an essential chapter to understand what can be done to combine both static and dynamic verification.
- Chapter **??** will present the scheduled plan of work ahead and explain birefly how we will manage it.

2

Background

In this chapter, we will present the concepts and tools that are essential for the understanding of the work that we propose. We will start by presenting the concept of Formal Verification (**??**) to present one the pillars of software verification, as it is one of the fundamental concepts that we work with.

Then, we will proceed to present OCaml (**??**), the language that we will be the focus of this dissertation. We will present its features and how it can be useful for the tools that we will be using. It provides an overview of the language, as well as some of the most relevant features, such as modules, functors and higher order functions. All of these will be presented with examples to better illustrate the concepts.

After that, we will present GOSPEL and Cameleer (**??**), two tools used for both static (also referenced as deductive) and dynamic (also referenced as RAC) verification. They are tools that are used to provide OCaml code with formal contracts, such as preconditions, postconditions, invariants and other specifications. These will be explained more in depth in the section dedicated to them, as well as some examples for better understanding.

The section **??** will be dedicated to Why3 and its usages in software verification. In that section we also provide some insight to the solvers that are used in the platform.

Lastly, we will present RAC (**??**), a technique that is used to verify code during runtime. We provide details on how it works (**??**), how it relates to static verification (**??**), and how it is used in ORTAC (**??**), a tool of utmost importance for our work.

2.1 Formal Verification

Formal Verification is the process of using mathematical methods to rigorously prove the correctness of a software system. Its goal is to ensure that a system is compliant with a given set of specifications, leaving no room for ambiguities or unforeseen errors. Unlike traditional testing, which can only cover a finite set of scenarios, formal verification provides a guarantee of correctness for all possible inputs and states within the specified domain [0].

The use of this process requires a certain level of expertise in formal methods, such as logical reasoning, model checking, and theorem proving. It is a complex and time-consuming process, but the results are proven to be worth the effort, so much so that Formal Verification is used in critical systems, such as avionics [0], automotive [0], and medical devices [0]. It also helps reduce workload and costs of software maintenance, as it can detect and prevent errors early in the development process.

2.2 OCaml

OCaml is a functional programming language with a strong type system and a powerful module system. It supports imperative, functional, and object-oriented programming paradigms, making it a versatile choice for various types of software development. OCaml's type inference system helps catch errors at compile time, reducing the likelihood of runtime errors and improving code reliability.

These aspects of this functional language make it a useful and resourceful tool for developing software that requires high standards of verification and thus, correctness. It is language becoming more widely used for a plethora of applications.

Some of the key features of OCaml are: its expressive type system; the pattern matching cases; the higher-order functions; and the module system. All these features make OCaml an expressive and concise language, which can be used to develop both simple and complex systems.

2.2.1 Language Overview

Being a functional language, OCaml encourages the use of higher-order functions, which can accomodate for more generic solutions for a variety of problems. However, OCaml's expressive type system and powerful abstractions can lead to more robust and maintainable code, especially in large and complex systems. The language's modules allow for encapsulation and organization, making code readability and organization easier to achieve.

As an example, consider the following code snippet, which defines a simple stack

OCaml

```ocaml
type α stack = α list

let empty : α stack = []

let push x s = x :: s

let pop a =
    match a with
    | [] → failwith "Empty stack"
    | x :: s → x
```

This code defines a stack data structure and its basic operations. It then can be used in a reusable manner throughout the codebase. As an example:

OCaml

```
let s = push 1 (push 2 empty)
let t = pop s
```

For programmers, it is a familiar and easy-to-understand way of defining and using data structures and algorithms, as they can be previously declared and used throughout the code.

2.2.2 Modules and Functors

Modules and functors are core features of OCaml 's type and abstraction system. They allow for organizing and structuring code effectively, enabling encapsulation, reuse, and flexibility.

A module is a collection of definitions, such as types, functions, and values, grouped under a single name. Modules provide namespaces to prevent naming conflicts and enable better code organization.

OCaml

```
module type Stack = sig

    type α t

    val create : unit → α t

    val push : α → α t → unit

    val pop : α t → α

end
```

2.2.3 Higher Order Functions

Higher-order functions are functions that can take other functions as arguments or return them as results. They are a cornerstone of functional programming in OCaml. They allow for concise and expressive code, enabling powerful abstractions and code reuse. Here is an example of a higher-order function in OCaml:

OCaml

```
let apply_twice f x = f (f x)
```

In this example, `apply_twice` is a higher-order function that takes a function `f` and an argument `x`, and applies `f` to `x` twice.

We can also take as an example the code in the previous section, adding the `filter` function, taking a function as an argument:

OCaml

```
module type Stack = sig
    ...
    val filter : (α → bool) → α list → α list
```

Here, the `filter` function takes a predicate function and a list, and returns the elements of the list that satisfy the predicate. This is an example of an Higher Order Function, as it takes a function as an argument.

2.3 GOSPEL & Cameleer

GOSPEL is a specification language built with the goal of providing OCaml code with formal contracts, such as preconditions, postconditions, invariants and other specifications. These contracts can be used to specify the intended behavior of specific parts of the code [0], helping to ensure correctness and reliability.

Cameleer is a tool that translates the beforementioned contracts into WhyML [0] - the specification language of Why3, a platform that we will talk about in section **??**.

These tools are strongly coupled to provide a more reliable way of verifying OCaml programs [0].

2.3.1 Example

Let's take the `Stack` module from the previous section and add some GOSPEL. The example should look something like this:

GOSPEL + OCaml

```
module type Stack = sig

    type α t
    (*@ mutable model contents: α list *)

    val create : unit → α t
    (*@ s = create ()
        ensures s.contents = [] *)

    val push : α → α t → unit
    (*@ push x s
        modifies s.contents
        ensures s.contents = x :: (old s.contents) *)

    val pop : α t → α
    (*@ r = pop s
        requires s.contents ≠ []
        modifies s.contents
        ensures s.contents = (List.tl (old s.contents))
        ensures r = List.hd (old s.contents) *)

end
```

The mutable, ensures, modifies, and requires are all specifications, or contracts, defined in GOSPEL. They specificy what each function should do.

The mutable keyword is used to define the state of the data structure, in this case, that the contents of the stack are allowed to be changed during runtime.

The keyword ensures is used to define the expected outcome of the function, or what the function should return. In the case of the push function, it should return the stack with the new element added to the top of the old values, defined by old s.contents.

For the pop function, the requires keyword is used to define a precondition, a state that must be true when the execution enters the function. Here, it is defined that the stack must not be empty.

Lastly, the modifies keyword is used to define that a certain data structure is being altered during the execution of the function. During pop, the contents of the stack are being modified, by removing the first element.

2.4 Why3

OCaml in itself is a powerful language, but it lacks the tools to provide clear and concise specifications for the code. Why3 is a platform that provides a framework for deductive program verification. It allows for the specification of programs, such as the ones described previously in section **??**, to be verified using logical verification conditions. It works with SMT solvers, such as Z3, Alt-Ergo, and Coq.

Why3 has its own specification language, called WhyML, which is used to define the contracts and the logical verification conditions. It is a language that is based on functional languages, as it contains constructs that are similar to those found in functional languages, such as OCaml. These aspects are pattern-matching, algebraic types and polymorphism [0]. Being a specification language means that its importance is in the verification of code statically, before it is exectuted.

When a program is verified using Why3, it generates verification conditions that are sent to the solvers. These solvers then try to prove the conditions, and if they are proven, the program is considered correct. If not, the solvers provide counterexamples that can be used to debug the code. It allows for full verification of the code in terms of type-checking, logical verification, the use of polymorphism, and other features that are essencial for the proper functioning of the program.

As an example, consider the following code snippet:

WhyML

```
let rec factorial (n: int) : int
```

```
    requires { n ≥ 0 }
    ensures { result ≥ 1 }
= if n = 0 then 1 else n * factorial (n - 1)
```

To verify the correctness of the `factorial` function, we can insert assertions to check for preconditions, postconditions, and invariants. For example:

WhyML

```
let rec factorial (n: int) : int
    assume { n ≥ 0 }
    = if n = 0 then 1
    else n * factorial (n - 1)
    assert { result ≥ 1 }
```

See "assume" with professor.

This example provides a simple but really important illustration of how RAC works, as well as what our work will focus on. The assertions are inserted into the code, and when the function is executed, the assertions are checked. If they fail, an error is raised, indicating that the code is not behaving as expected. This is a useful technique for RAC, although here it is used in a static verification context, resorting to SMT solvers.

2.5 Runtime Assertion Checking

There are several approaches to formal verification, such as static verification, which is based on the analysis of static code (with an example being type checking), and dynamic verification, which is based on the execution of the code, with the use of techniques resorting to assertions and logical contracts. Runtime Assertion Checking (RAC) is a dynamic verification technique that consists of inserting assertions into the code to check for correctness during runtime.

These assertions can be used to verify preconditions, postconditions, and invariants, providing a way to ensure the correctness of the code during execution. For this to happen, the programmer (or tester), need to insert assertions to define the intended behaviour of functions or modules. At runtime, these assertions are checked, and if any of them fail, an error is raised, indicating that the code is not behaving as expected.

2.5.1 Technique

As explained before, RAC is a technique that consists of inserting assertions into parts of the code that are the target of verification, and then executing the program focusing in those same parts. Therefore, the technique consists of two parts:

- Defining assertions with the intended behaviour of the code;

- Executing the code and verifying if the aforementioned assertions are met.

2.5.1.1 Assertions

Assertions are logical expressions that define the intended behaviour of the function or module they are inserted in. They consist of preconditions (`requires`), postconditions (`ensures`), and invariants (`invariant`).

The first are conditions that must be met before the execution enters the portion of code that is being verified. A use of this is to check if the input parameters are within the expected range, or if the data structures are in a valid state. Validating the input can prevent the code from ending abrutly, raising an error.

The second represent the expected results of the execution of the code, i.e. the expected outcome of the function or module. This can be used to check if the output is correct and if the data structures and modules accessed remained valid and/or in a predefined state.

Lastly, invariants delimit the state of the variables and data structures that are being used in that portion of code. Here, the programmer can define the limits of the data structures, the expected values of the variables, and other conditions that must remain true. Invariants are used when `loops` (such as `for` and `while`) are present.

2.5.1.2 Execution

When the code is executed, tools focus in verifying that the products of the execution are in accordance with the assertions, and if they are not, an error is raised, alerting the programmer or tester that the code is not behaving as expected. The errors raised can also be used to debug code if they provide information about the difference between the expected and the actual results.

2.5.2 Relation with Static Verification

Dynamic verification has the goal of verifying code, checking for problems and errors that may be present. Almost as a form of testing. Static verification in not much different, as the goal is precisely the same. However, the approach is different. While dynamic ficuses on execution, static verification focuses on the code itself, analyzing it to find possible errors.

Both techniques are complementary, with ones strengths being the others weaknesses. Dynamic verification cannot run if code compilation fails - this can be enabled by static verification as it can catch errors like type mismatches and syntax errors. On the other hand, static verification on mutable pieces of code, like data structures, is much less effective than dynamic verification, as the states of these structures can change during execution.

2.5.3 ORTAC

ORTAC is a tool for RAC in OCaml programs. It translates OCaml module interface with GOSPEL specifications into code to be verified [0].

It is an essential tool for dynamic verification in OCaml, more specifically for RAC. It is a bridge between the GOSPEL specifications, OCaml code, and the other tools used for dynamic verification. Although it uses some plugins for the testing of the code it generates, ORTAC presents an ingenious model for the verification process.

First, the programmer writes the OCaml code and the GOSPEL specifications. Here the specifications are inserted in the code, allowing for a more clear and straightforward way of defining the intended behaviour of the code. Then, ORTAC translates the code and the specifications into a format that can be verified by the plugins [0]. On their part, plugins "bombard" the code with tests, checking multiple possibilities in a search for errors.

3

State of the Art

This chapter is reserved for the current state of software verification, focusing on the tools that are used for this purpose. We present some of the tools and how they are used, as well as a brief history of the development of software verification, from *Eiffel* to the present day. The objective of the State of the Art is to provide a better understanding of the current state of the theme of this dissertation, giving practical examples and brief explanations of uses of both tools and techniques in the everyday life of software verification.

We start by presenting the current use of both static and dynamic verification tools (**??**), and how they are used in combination, if that is being employed. It is an important section for us to understand the current position of this vision of software verification, as it directly impacts the work that we are proposing.

We then present the current tools and their respective uses in section (**??**). Here we focus in more detail on the tools that are being described and the languages that they emcompass. This provides a more "vivid" picture of the current state in the field of software verification.

Finally, we present the concept of RAC, or Runtime Assertion Checking, and how it is used in the context of executable specifications (**??**). It is important to note de difference between RAC and its derivatives of executable specifications, as they will be very much present in our work. We provide the reader with a brief example of ACSL and E-ACSL for a better understanding of the concept.

3.1 Combine Static and Dynamic Verification

This work that we propose in similar to that of Soares, Chirica and Pereira's [0]. This dissertation circles around GOSPEL, one of the tools used in dynamic verification, and its relation to its static counterparts. What we propose to do with this work diverges from this, as we focus in combining the two in a unique technique, called `Monitors`, that verifies the program initially with static verification, and then, when that process concludes, resorts to dynamic verification (more specifically RAC [0]). It resorts not only to GOSPEL, but also ORTAC, Why3 and Cameleer, as well as static verification, thus being more robust and complete.

This cannot be done without including Formal Verification, as demonstrated by Brian and Polgreen's work [0]. It is one of the pillars of software verification, as it permits the search for correctness to be as precise, robust and complete as it can be.

Currently, the combination of these two techniques is either theoretical or it exists losely coupled together. What we propose is directed to a single and efficient way of fully verifying an OCaml program resorting to tools that focus in either static or dynamic strands of software verification. Both can be used in a way that one strengthens the disadvantages of the other, and vice-versa.

3.2 Current Tools

In the real world, there are many tools that can be used to verify software. Examples such as Why3, Frama-C, SPARK [0] present types of tools that are used for verification of software. Why3 focuses on the verification of OCaml programs, while Frama-C focuses on the verification of C programs. SPARK is a technology that focuses on the verification of Ada and also C/C++ programs. They provide coverage for different languages, but they all focus on static verification and their goal is to prove the correctness of the software.

The tools that focus on dynamic verification - more specifically, runtime assertion checking, or RAC for short - are also used for the same purpose, with a different approach [0].

- JML, or Java Modeling Language, is a language that is used to express specifications of Java [0].

- ACSL, or ANSI/ISO C Specification Language, is a language used to express specifications of C [0].

- SPARK focuses on the verification of Ada programs [0].

These are some of the examples of tools that are used for dynamic verification. Their objective is to verify software during runtime, checking for behaviour that is not expected.

Eiffel is a programming language that has a built-in assertion language [0]. At the time of its creation, it was one of the first languages to have this feature, and it presented a new way of thinking about software verification. Designed in the 1980s, it was a language that focused on the development of software that was correct by construction. This means that the software was developed in a way that it was correct from the beginning. It had a built-in assertion language that allowed the programmer to express the specifications of the software. Contracts, static type checking, garbage collection are just some of the features that *Eiffel* has.

Due to the fact that this language focused in the development of software that was correct from the start, it influenced a lot of the tools that are used today. The tools above mentioned are just some examples of tools that were influenced by *Eiffel*.

3.3 Runtime Assertion Checking - Executable Specifications

Although dynamic verification is in itself a powerful technique for the verification of systems, developers found that it was necessary to have a way to express specifications directly in the code. This is where executable portions of specification languages come in.

ACSL, as mentioned before, expresses specifications, but it is not executable. Thus, E-ACSL was created. It is a tool that is used to express specifications amidst the code, making those specifications executable. This tool translates C code with specifications into C code that is instrumented with assertions. It is a conversion tool that makes verification simpler and more efficient [0]. This is one of the principles that we will use in our work, as E-ACSL is only designed for C programs.

This is what we hope to accomplish with our work. We want to create a tool that is able to express specifications into OCaml programs, making them executable. We hope to produce a GOSPEL executable specification language that is able to express the specifications produced by the programmer, and converting them into assertions, inserted in the right places of the code. Thus, E-GOSPEL will provide a step forward in the verification of OCaml, allowing programmers specialized in this language to use the tool to verify their software in a more efficient and robust way. It is also a step towards `Monitor` implementation, as it is a technique that has some promising results in an area where `Monitors` are inserted in.

Executable Specification Languages bridge the gap between what the programmer wants the program to do and what the system will try to verify in the code. It is a way to express the desire of the programmer directly in the code, making it easier to verify the software as verification and code are being executed in the same environment.

4 Preliminary Results

For our research and introduction on the topic of static and dynamic verification, we had to prepare and test an example that would be used to demonstrate the execution model of these techniques of verification.

The chosen example was a `Queue` implementation in OCaml, verified in Why3 and GOSPEL. The `Queue` implementation consists of two lists, one for the front of the queue and another for the back. Its functioning is quite simple: it has two operations, `push` and `pop`, that add and remove elements from the `Queue`. This approach uses the first List as the elements that are being removed through the `pop` operation, and the second List as the elements that are being added through the `push` operation.

4.1 Queue Example

The implementation of a `Queue` we opted for a double list approach, where the first list is the front of the queue and the second list is the back. The principle behind this implementation is described above, where the `push` operation adds elements to the back of the queue and the `pop` operation removes elements from the front of the queue.

4.1.1 Implementation

Below is our implementation of the `Queue` in OCaml:

GOSPEL + OCaml

```
open List

type α queue = {
    mutable front : α list;
    mutable back : α 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
```

As one can observe, we opted for a definition of `Queue` as a record type, with three fields: `front`, `back` and `size`. The `front` and `back` fields are lists of elements of type α - meaning a certain type not yet defined -, and the `size` field is an integer that represents the number of elements in the `Queue`.

The `is_empty` function is a logical function that returns `true` if the size field of the `Queue q` is equal to zero, and `false` otherwise.

The `make` function is a constructor for the `Queue` type, that returns a new `Queue` with the `front` and `back` fields as empty lists.

The `pop` function removes the first element of the `Queue` and returns it. If the `Queue` is empty, it raises a `Not_found` exception. If the `front` only has one element, it reverses the `back` list and assigns it to the `front`, clearing the `back` list in the process and returning the first value. If the `front` has more than one element, it removes the first element of the `front` list and returns it. Finally, it decrements the `size` field by one.

The `push` function adds an element to the `Queue`. If the `Queue` is empty, it assigns the element to the `front` list. Otherwise, it adds the element to the `back` list. In both cases, it increments the `size` field by one.

4.1.2 Specification

With the implementation of the `Queue`, we can now specify the intended behavior of its operations. We used GOSPEL contracts to introduce the necessary specifications for further testing. Below is our specification:

GOSPEL + OCaml

```
type α t
(*@ mutable model view: α list *)

val is_empty : α t → bool
(*@ b = is_empty a
    ensures b ↔ t.view = []*)

val make : int → α → α t
(*@ t = make ()
    ensures t.view = []*)

val pop : α t → α
(*@ a = pop t
    modifies t.view
    requires t.view ≠ []*
    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 : α t → α → α
(*@ push t a
    modifies t.view
    ensures t.view = append_last a (old t.view)*)
```

This interface has all the specifications that present the behavior of our `Queue` implementation. We found it necessary to add a `model` `view` field to the `Queue` type, to allow for the visualization of its elements as a simple OCaml list.

The `is_empty` function returns `true` if the `Queue` is empty, and `false` otherwise. The ↔ operator is a logical equivalence operator, meaning that the value of `b` is equivalent to the expression of `t.view = []`.

The `make` function creates a new `Queue` with an empty `view` field. It ensures that the creation of a new `Queue` results in an empty one, as defined by `ensures` clause.

The `pop` function removes the first element of the `Queue` and returns it. This is the function with more complex specifications. It modifies the `view` field with the `modifies` clause, and requires that the `view` field is not empty, with `requires`. For the postconditions, the specifications have to ensure that, first, the `t.view` field remains the same if there are no elements in the `Queue` (`ensures t.view = if old t.view = [] then [] else List.tl (old t.view)`), and, secondly, that the returned element is the first element of the `Queue` (`ensures if old t.view=` [] then false else a = List.hd (old t.view)).

Finally, `push` adds an element to the `Queue`. It modifies the `view` field and ensures that the `t.view` field is the result of appending the new element to the last position of the `Queue`, resorting to the `append_last` function.

5 Work Plan

5.1 Gantt Chart

- *Task 1: Research* For the month of March, we will be researching study cases and analysing them in ORTAC + Cameleer. This will provide data for the next tasks throughout the work.
- *Task 2: Define E-Gospel* The next step will be to define E-Gospel - an executable portion of both OCaml and specification/contract tools that can be used in GOSPEL. The defining of E-Gospel will use the data from the research and analysis. This will be done throughout April until the middle of May.
- *Task 3: Define Monitors* Using the previous steps as a base, we will define the specification of monitors that are best suited according the conclusions of the previous tasks. This will be done throughout May until the end of June.
- *Task 4: Implement Monitors* From mid June throughout July and August, we will implement the monitors previously defined and integrate the first set of data to test them. It is expected that the first implementation will be ready by the end of August.
- *Task 5: Writing* The last month of the work will be dedicated to writing the final document. This will occupy the month of September.

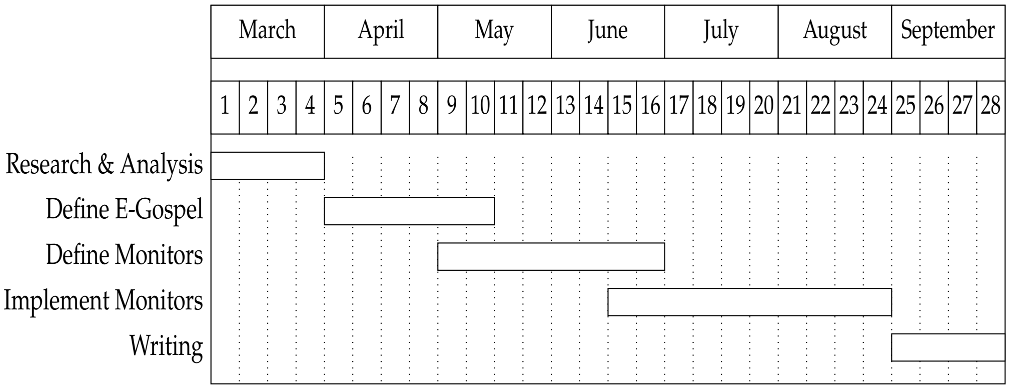

Figure 5.1: Planned Schedule

Bibliography

[0] J.-C. Filliâtre and C. Pascutto. *Ortac: Runtime Assertion Checking for OCaml (tool paper)*. URL: https://github.com/ocaml-gospel/ortac..

[0] M. Brain and E. Polgreen. "A Pyramid Of (Formal) Software Verification". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 14934 LNCS. Springer Science and Business Media Deutschland GmbH, 2025, pp. 393–419. ISBN: 9783031711763. DOI: 10.1007/978-3-031-71177-0_24.

[0] Y. Yin, B. Liu, and D. Su. "Research on formal verification technique for aircraft safety-critical software". In: *Journal of Computers* 5 (8 2010), pp. 1152–1159. ISSN: 1796203X. DOI: 10.4304/jcp.5.8.1152-1159.

[0] N. Rajabli et al. "Software Verification and Validation of Safe Autonomous Cars: A Systematic Literature Review". In: *IEEE Access* 9 (2021), pp. 4797–4819. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3048047.

[0] J. S. Bezerra et al. "Formal Verification of Health Assessment Tools: a Case Study". In: *Electronic Notes in Theoretical Computer Science* 324 (2016-09), pp. 31–50. ISSN: 15710661. DOI: 10.1016/j.entcs.2016.09.005.

[0] T. L. Soares, I. Chirica, and M. Pereira. "Static and Dynamic Verification of OCaml Programs: The Gospel Ecosystem (Extended Version)". In: (2024-07). URL: http://arxiv.org/abs/2407.17289.

[0] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: ed. by A. Silva and K. R. M. Leino. Vol. 12760. Springer International Publishing, 2021-07, pp. 677–689. ISBN: 978-3-030-81688-9. DOI: 10.1007/978-3-030-81688-9_31.

[0] M. Pereira. *Practical Deductive Verification of OCaml Programs*. Ed. by A. Platzer et al. 2024-09. DOI: 10.54499/UIDB/04516/2020. URL: https://sciproj.ptcris.pt/157572UID.

[0] M. José and P. Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. URL: https://theses.hal.science/tel-01980343v1.

[0] N. Kosmatov et al. *Static versus Dynamic Verification in Why3, Frama-C and SPARK*. 2014. URL: http://frama-c.com.

[0] F. Maurica, D. R. Cok, and J. Signoles. “Runtime assertion checking and static verification: Collaborative partners”. In: *Computer Science* (2018), pp. 75–91. DOI: 10.1007/978-3-030-03421-4_6Ãŕ. URL: https://cea.hal.science/cea-04477117v1.

[0] G. T. Leavens et al. *JML Reference Manual*. 2008.

[0] P. Baudin et al. *ACSL: ANSI C Specification Language*.

[0] *The Work of Proof in SPARK*. 2022. URL: https://github.com/AdaCore/spark2014.