rev. 2

UFR de mathématique et d'informatique
Université de Strasbourg

MASTER'S DEGREE IN COMPUTER SCIENCE
SCIENCE AND ENGINEERING OF NETWORKS, INTERNET, AND SYSTEMS

Internship Report

Lucian MOCAN
lucian.mocan@etu.unistra.fr

Improving the Modularity and Robustness of the Althread Language

August 17, 2025

Supervisors
Anissa LAMANI — alamani@unistra.fr
Quentin BRAMAS — bramas@unistra.fr

Host Institution
Department of Mathematics and Computer Science
University of Strasbourg

Internship Period
June 10 – September 5, 2025

Contents

Chapter 1

Introduction

In this report, I'll walk you through the work carried out during my first-year Master's internship in the Cursus Master en Ingénierie[1] (CMI) in Internet, Systems, and Networks at the University of Strasbourg. Over three months (June 10 – September 5, 2025), I contributed to the development of Althread, an educational programming language for modeling and verifying concurrent and distributed systems.

My main objectives were to improve Althread's modularity and usability by implementing a complete import and package system, finalizing user-defined functions with robust control-flow analysis, and enhancing the web-based platform with a virtual file system, package manager, and improved visualization tools. The work combines language design, compiler engineering, and UI/UX development, drawing on comparative studies of existing languages (Promela, TLA+, Python, Go, JavaScript) to inform architectural decisions. The resulting system supports modular code organization, remote dependencies, and a modern browser-based development environment, which positions Althread as a viable teaching tool for distributed algorithms.

This opportunity arose under unexpected circumstances. After securing an internship in January 2025 following an early search that began in September 2024, I had to cancel it due to personal obligations requiring travel to the US. Fortunately, I was able to pursue a remote internship building on my prior TER (Research Project) work on user-defined functions in the Althread language [2]. I am grateful to my supervisors, especially Mr. Quentin Bramas, for embracing this challenge with me, and to Mr. Pierre David for his flexibility in accommodating my situation.

Other tasks during the internship included integrating contributions from my colleagues' TER projects [3, 4] and writing extensive technical documentation to ensure the Althread website provides clear, accessible guidance for users.

This experience reaffirmed my interest in contributing to open-source projects that mix theoretical research with practical applications. The true value of tools like Althread lies in their long-term educational impact, empowering future generations of computer scientists and researchers. In my opinion, this is one of the most meaningful ways to invest in the continued relevance and impact of our field.

[1] The CMI is a five-year engineering program accredited by the Réseau Figure, emphasizing research-driven education in engineering [1].

Chapter 2

Context

2.1 Presentation of the Host Institution

2.1.1 University of Strasbourg

Founded in 1538, the University of Strasbourg is one of the oldest and most prestigious universities in France and in the world [5]. It is widely recognized for its excellence in teaching and research, as well as its significant contributions to culture and science.

The university serves over 55,000 students and employs more than 3,400 faculty members and researchers, along with a comparable number of professionals across libraries, engineering, administration, and healthcare services. It comprises 35 faculties, schools, institutes, and research units, distributed across six campuses [6].

2.1.2 UFR de Mathématique et d'Informatique

The *UFR de Mathématique et d'Informatique* (Department of Mathematics and Computer Science) is the University's academic unit dedicated to mathematics and computer science. It hosts a large academic community involved in advanced research and education. The department includes around 150 professors and researchers and serves a diverse student body of approximately 2,000 students.

The department is organized into two main divisions: Mathematics and Computer Science, each linked to research units. The administrative structure oversees areas such as finance, logistics and technical services, academic affairs, and other support services.

Closely affiliated with ICube [7], a prominent interdisciplinary research laboratory, the department benefits from strong collaboration between research and teaching. Many faculty members are also active researchers within this lab.

2.1.3 My Role

My internship took place within the Computer Science division of the Department of Mathematics and Computer Science, led by Mr. Stéphane Cateloin. My role focused on advancing Althread, an educational programming language, to support the division's mission of developing innovative tools for teaching concurrent and distributed systems.

In the first year of the master's program at the UFR, the "Distributed Algorithms" class uses Spin and Promela to teach model checking and verification of distributed systems. My internship aimed to enhance Althread to become a viable alternative for this class in the near future. Through my work on user-defined functions and the import and package system, I significantly improved Althread's modularity, making it more suitable for educational use.

I believe my work fully aligns with the division's core mission: advancing research and supporting the education of future generations. I kept this goal in mind throughout the internship and considered it a privilege to contribute within such a respected academic environment.

2.2 The Althread Language

To properly situate the context of Althread's ongoing development, we have to go back to the basics.

2.2.1 Althread's Raison d'Être

Distributed systems are at the forefront of technological and research advancements in today's world, and they will likely remain so for years to come. This paradigm is built on several key pillars that aim to minimize costs, improve integration, ensure availability, reliability, and security, all while maintaining an acceptable performance loss: resource sharing, transparency, openness, dependability, and scalability [8].

At the heart of distributed systems lies software: algorithms that define how these systems behave. Detecting flaws in the design of distributed systems as early as possible is critical, as these flaws can lead to significant issues in production. In formal language, such an algorithm is called a model. A model is a mathematical representation of a system's behavior, expressed through states, events, or communication patterns [9]. It provides a foundation for verifying correctness before development begins [2].

Over the years, various tools and their accompanying languages have been developed to model and verify distributed systems, many of which are widely used in industry. Examples include CSP (Communicating Sequential Processes)[10], SPIN (with its language Promela)[11], and TLA+[9]. TLA+ has gained significant traction, with notable applications at Amazon AWS[12], Microsoft [13], MongoDB [14], and even Google, which used it to specify BGP [15]. It has also attracted considerable academic interest [16].

While these tools are powerful and have proven their value in industry, they are not particularly education-focused. For example, after lots of efforts Promela offers a somewhat extensive visualization support [17, 18, 19, 20], but its interface and tooling are outdated. TLA+ has seen more recent attention, with new tools like Spectacle [21], an interactive, web-based platform developed by engineers at MongoDB for exploring, visualizing, and sharing formal specifications in TLA+. However, even with tools like Spectacle, the complexity of using a formal mathematical language to specify an algorithm remains a significant barrier, especially in educational contexts. This friction can make it harder to teach distributed algorithms effectively.

Althread addresses this gap by providing an accessible alternative designed specifically for education. It simplifies the process of exploring distributed systems and model checking, removing potential stumbling blocks for students and educators alike. By focusing on usability and accessibility, Althread makes it possible to teach and learn distributed algorithms without the steep learning curve associated with more formal tools.

2.2.2 Althread's Architecture

Before diving into the language itself, a quick overview of the underlying architecture is useful. Althread consists of a compiler and a virtual machine (VM), both written in Rust. The compiler parses source code into an abstract syntax tree (AST) and then translates that AST into bytecode. The VM consumes the bytecode, executes it, and provides the runtime environment for processes, channels, and shared variables.

Syntax overview

Rather than enumerate every syntactic rule, a few distinguishing constructs are highlighted. Figure 2.1 shows a short program that demonstrates the most salient features: a `shared` block, an `always` block, a process template, and the `main` block that instantiates the template.

```
shared { // variables available anywhere in the code
    let N: int = 0;
    let Start = false;
}

always { // check if conditions are always true
    N == 0;
}

program A() { // program template A
    await Start; // waits until Start == true

    // waits on the process' input channel
    await receive in (x,y) => {
        print("received ");
        N = N + 1; // update the value to trigger a violation
    };
}

main {
    // start a process with program template A
    let pa = run A();

    // create and link a channel to A's input
    channel self.out (int, bool)> pa.in;

    Start = true;
    send out (125, true); // send in the channel
}
```

Figure 2.1: The `shared` block declares variables that are visible everywhere (note the convention: shared variables start with an uppercase letter). The `always` block enforces a safety condition that must hold in every possible execution. `program A()` defines a process template that waits for the boolean `Start` to become true, then receives a tuple on its input channel, prints a message, and increments `N`, triggering a violation of the `always` condition. Finally, the `main` block creates an instance of `A`, links a channel to its input, sets `Start` to true, and sends a value through the channel.

Grammar

Althread's grammar is expressed as a parsing-expression grammar (PEG), which guarantees a single, unambiguous parse tree. The implementation relies on the Rust-based open-source Pratt parser `pest.rs` [22].

A practical pitfall of PEGs is that the first rule that matches wins; therefore, more specific rules must precede more general ones. Figure 2.2 illustrates a case where an identifier is mistakenly matched before a function call, leading to a parse error.

```
fn_call = { identifier ~ args_list ~ ; }
expression = { identifier | fn_call | ... }
```

Figure 2.2: If the rule `identifier` appears before `fn_call`, the parser will treat `max(1,2)` as an identifier followed by an unexpected `'('`. Swapping the order resolves the issue.

Compiler

The compiler uses `pest.rs` to parse the source code according to the PEG grammar, producing a parse tree. From this tree, it recursively constructs an AST, which captures the hierarchical structure of the program: top-level blocks (`main`, `shared`, `program`, `always`, `fn` for function blocks), variable declarations, channel definitions, and so on.

The compilation pipeline then walks the AST, emitting bytecode for the VM and building a global context that records all the information required at runtime. This includes runtime data (stack, scope, global variables, and channels), program metadata (function signatures, program templates, the current program name, and standard library bindings), and verification flags (e.g., `shared`, `atomic`). The final artifact of the compilation phase consists of the bytecode, a global memory layout, verification conditions, compiled user-defined functions, and the Althread standard library [23].

Virtual machine

The VM receives the compiled artifacts, initializes its global state, and begins execution with the instructions in the `main` block. It then schedules instructions from any active program in a nondeterministic fashion, thereby modeling the interleaving of concurrent processes.

For function calls, each program maintains its own call stack, recording the caller's bytecode address, the return address, and the expected return type [2]. A concise description of the VM can be found in the source directory `src/interpreter/vm/` and the entry point `cli/main.rs` [24].

With the core architecture established, the next step was to ensure that Althread's features are accessible through a modern, browser-based interface.

2.2.3 The Althread Web-Based Platform

Althread's promise of ease of use is realized through a modern, browser-based platform. The compiler and virtual machine are written in Rust, compiled to WebAssembly, and exposed to the JavaScript front-end via a small shim.

WebAssembly is a low-level bytecode format designed to run efficiently in web browsers, providing a portable and secure execution environment [25]. It offers significantly better performance than JavaScript for computationally intensive tasks, though it remains slightly slower than native code due to browser sandboxing [26]. Because all compilation, execution, and model-checking happen locally in the browser, the platform works offline after the initial page load: no network connection is required for any of its features.

Key editor features

- **Write, compile, run.** The editor sends the source to a WebAssembly module, which compiles it to bytecode and runs it on the embedded VM. The compiled project (bytecode, memory layout, verification conditions) is shown to the user.
- **Static safety check.** A "Check" button triggers a check that analyses the program for `always/never` violations; it reports a violation or declares the program safe.
- **State-graph visualisation.** When the check succeeds a graph of all reachable states is rendered. Each node displays shared-variable values, the program stack, running processes, and the status of `always/never` conditions. Nodes are green when no violation is present and red otherwise. Rendering time grows with the number of states, as expected for exhaustive model-checking.

2.3 Missions and Objectives

I entered the internship with a single, clear-cut goal: make Althread an easy-to-use educational tool as quickly as possible. The work broke down into several concrete tasks, which I describe in the subsections that follow; the detailed technical contributions are presented in the later chapters.

2.3.1 Finishing the Implementation of User-Defined Functions

My TER work [2] left a solid foundation for user-defined functions, but a few important gaps remained. The aim of this sub-project was to make the function implementation more complete, while still acknowledging that the work is ongoing. The two main items on my to-do list were:

- Enable the compiler and the VM to handle nested calls such as `max(max(1, 3), 5)`.
- Build a control-flow graph so the compiler can verify that every function returns on all possible execution paths.

These two tasks are essential for a robust language, though their complexities vary. When I worked on nested function calls, I found that tasks can often bring unexpected subtasks, making the process more complex and time-consuming. For example, in this case, I had to deal with managing temporary stack values and creating new bytecode instructions to handle cleanup.

2.3.2 Integrating Other TER Contributions

Last semester, together with three fellow students, I worked on Althread, each of us focusing on a different aspect of the language.

The `eventually` block. Julien Clavel added support for an `eventually` block [3]. The idea is that if a condition is false (or true) at the start of a time path, it must become true (or false) before the last state of that path in **all** possible executions. I analyzed Julien's implementation and merged it into the main development branch.

Communication-graph visualization. Violette Lesouef created a visual communication graph, a trace diagram that connects `send` and `receive` events with directed arrows between processes [4]. While testing the feature, I uncovered several issues in the Althread channel implementation as well as in the way the communication graph tracks those messages. Those bugs were fixed and the visualizer was integrated into the core code base.

Benchmarking. Théo Weber performed a set of performance and benchmarking tests. Because the project is still in an early stage, I did not focus on optimization in this report; the benchmarking results will be useful later when the language is ready for larger case studies.

2.3.3 Designing and Implementing an Import and Module System

This was the most complex and extensive task of the internship. At first I imagined the import system would be straightforward, but the design quickly revealed many hidden challenges. The motivation is simple: modularity. An import system lets users split a program into reusable pieces, share code across projects, and eventually build libraries or packages.

To decide how to structure Althread's import mechanism I performed a comparative study of several existing languages, starting with those that were created specifically for model checking. First I examined Promela (the language behind SPIN) and TLA+, because both expose the kinds of safety[1] and liveness[2] properties that Althread supports. After those two model- checking languages, I surveyed a handful of other widely used programming languages like C/C++, Python, JavaScript, Go.

2.3.4 Improving and Adapting the Web-Based Platform

Recent modifications highlighted a fundamental problem: the current UI cannot accommodate the new import system without a full redesign. An editor that looks polished and works intuitively is essential, otherwise the UI itself becomes a barrier that keeps users from trying Althread.

Design guidelines

- **Modern look and Consistency.** A clean, contemporary aesthetic with uniform hover effects, colour palettes, themes, tab styles, button shapes, icons, fonts, and text sizes.
- **Accessibility.** Clear labels or tool-tips, WCAG-compliant[3] colour contrast, and platform standard interaction patterns.
- **CLI-replacement.** Graphical wrappers for operations that would otherwise require the command line (compile, load example, run a check).

The core UI/UX work will focus on a series of improvements, the most notable of which are a full overhaul of the interface, the addition of a left-hand pane that combines a file explorer with tabbed-file support, a search bar, a package manager, and a help window, the implementation of syntax-highlighting for Althread keywords and built-in constructs, and the provision of inline error reporting that includes line/column diagnostics and clickable links.

[1] safety = things that should never happen (always/never);

[2] liveness = things that should eventually happen (eventually).

[3] WCAG-compliant means meeting accessibility guidelines for readable contrast between text and background, typically a ratio of at least 4.5:1 [27].

Chapter 3

Contributions

This chapter is going to follow the same chronology as in Section 2.3. The following sections are going to tackle each of the missions by covering design considerations, implementation details, challenges, and weak points.

3.1 Finishing the Implementation of User-Defined Functions

3.1.1 Cross-Function Calling

Previously, the compiler handled recursive function calls by creating a temporary function definition that stored only the name of the function. This allowed the function to compile, and the definition was later updated with the compiled body. However, this approach worked only for single functions and failed when one function called another that had not yet been added to the list of existing user-defined functions. As a result, such cases would fail to compile. The code snippet in Figure 3.1 illustrates this issue. Compiling the function `yes` would fail because the function `no` is not yet defined, and vice versa.

The solution was to introduce a loop that processes all user-defined functions in two stages. In the first stage, the compiler creates function definitions for all functions, storing everything except the compiled body. In the second stage, as each function body is compiled, the compiler updates the corresponding function definition with the generated instructions. This ensures that cross-function calls are resolved correctly, even when the called function is defined later in the code.

```
fn yes(n: int) -> void {
    if n > 0 {
        no(n - 1);
        return; // explicit
    }
    print("yes");
    return;     // explicit
}

fn no(n: int) -> void {
    if n > 0 {
        yes(n - 1);
    }
    print("no");
    // implicit return
}

// yes(5); outputs:
// no
// no
// no
```

Figure 3.1: The function `yes` calls `no`, and vice versa. The function `no` relies on an implicit return.

3.1.2 Implicit Returns

In Althread, functions with a return type of `void` do not require an explicit `return` statement [28]. Previously, the compiler applied a patch that added a `return` instruction at the end of every function body, regardless of whether the return type was `void` or if a `return` statement was already present. This behavior was incorrect and could lead to redundant or unnecessary instructions.

The implemented fix assumes that if the user wants to exit the function early, they will

explicitly provide a `return` statement. For functions with a return type of `void`, if no explicit `return` statement is present at the end of the function body, the compiler automatically adds one. This ensures that the function behaves as expected while avoiding redundant instructions.

The code snippet in Figure 3.1 also illustrates this behavior. The function `yes` always exits early and never prints `yes`, while the function `no` prints `no` every time, relying on an implicit return. This example demonstrates how explicit and implicit returns are handled correctly by the compiler after the fix.

3.1.3 Control-Flow Analysis for Return Path Safety

This was an improvement I suggested at the end of my TER [2]. Ensuring that all possible execution paths in a function have a return statement is tricky when the bytecode produced by the Althread compiler has no representation of the hierarchy between nodes. Control-flow analysis [29] solves this by mapping out all possible execution paths, which in this case is used to check that functions with non-`void` return types always return a value. Back then, I knew why it was needed and what it should achieve, but I wasn't sure how to actually implement it. Normally, control-flow analysis is used for optimizations, but here the goal is purely to guarantee return path safety.

```
1 fn max(a: int, b: int) ->
    int {
2     if a > b {
3         return a;
4     } else if a == b {
5         // return here ?
6     }
7     // return here ?
8 }
```

Figure 3.2: The function `max` has execution paths where no value is returned, violating its declared return type (`int`).

In the example in Figure 3.2, the function `max` is supposed to return the larger of two integers, `a` and `b`. But as written, there are code paths where it doesn't return anything, even though the return type is `int`. There are two obvious fixes: either add a `return` in the second `if` branch and another one after the `if` block, or just have a single `return` after the whole conditional. If the user doesn't do one of these, the compiler should throw an error saying the function is unsafe because it doesn't guarantee a return value.

The main concept behind control-flow analysis, no matter the approach, is the *basic block* [29]. A basic block is a straight-line sequence of instructions (bytecode) that you can only enter at the first instruction and only leave at the last, no jumps or branches in the middle. To build the control-flow graph (CFG), the algorithm starts by creating two special nodes: *entry* and *exit*. Then it walks through the function body, turning each statement or group of statements into CFG nodes, linking them according to the possible flow of execution. Conditionals like `if` create branches: the `then` and `else` parts each become their own sequences of nodes, and their open ends are rejoined later. A `return` marks a block as terminal and links it directly to the exit node. At the end, any block that doesn't end with a return is also linked to the exit node, so every path is accounted for.

The result is a rooted, directed graph where each node represents a basic block, and edges represent possible control transfers [29]. The entry node connects to the first block(s) of the function, and any block with no successors connects to the exit. With this graph, it's easy to detect if there's a path from entry to exit that never hits a return which is exactly what is needed to enforce return path safety in Althread.

Once the CFG is built, as shown in Figure 3.3, the compiler runs a depth-first search (DFS) starting from the entry node to check for missing returns. The DFS keeps track of whether a return has been encountered along each path, and if it reaches the exit node without one, it

reports the first such case it finds. When that happens, the compiler produces an error like:

```
Function 'fn_name' does not return a value on all code paths.
Problem detected in construct starting at line <line #>.
```

This points the user directly to the construct where the problem was detected, making it clear which part of the function needs to be fixed. The full DFS algorithm and its implementation are described in Appendix B.

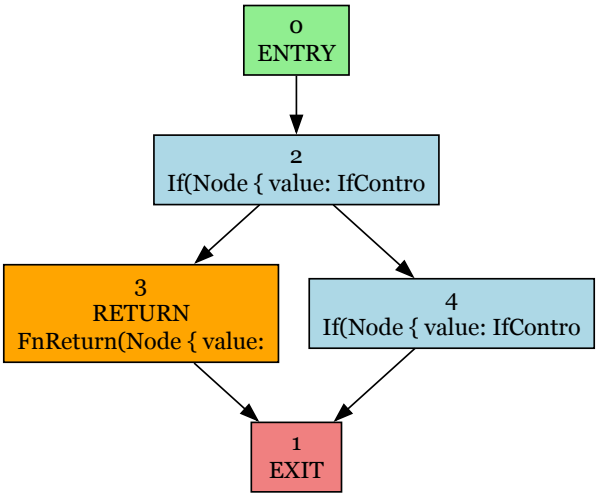

Figure 3.3: **Control-flow graph for the `max` function from Figure 3.2** Node 0 is the entry, Node 1 is the exit. Node 2 is the first `if` branch, Node 3 is the explicit `return a;` path, and Node 4 is the `else if` branch that currently has no return. The edge from Node 4 to EXIT is added automatically because it's an open end without a return.

3.1.4 Handle Nested Function Calls

Why nested calls are a compile-time / run-time problem

The Althread compiler operates on the AST produced by the parser. At this stage, the compiler knows the *shape* of an expression, for example, that the term

```
max(max(1, 2), l.at(0))
```

contains two function calls. However, it does not know the concrete values that will be produced when the program is executed. By default, the compiler attempts to evaluate the arguments of a function call at compile time. In this case, the values cannot be determined statically, so the compiler correctly refuses to fold the expression.

To execute such nested calls correctly, the compiler must ensure that the virtual machine (VM) evaluates the *inner* calls first, stores their results, and then uses those results as arguments to the *outer* call. This requires generating a sequence of instructions that:

1. evaluate the inner call(s),
2. store the intermediate results on the VM stack,
3. evaluate the outer call using those results, and
4. discard any temporary values that belong only to the inner calls.

In the original instruction set, there was no mechanism to discard temporary values immediately after they were consumed by the outer call. As a result, the stack could end up containing both the original temporary values, and the new tuple or result that also contains those values. This is incorrect for two reasons: unnecessary duplication of values on the stack and the VM's stack cleanup logic would not remove enough elements, leaving stale values that could corrupt later operations.

The fix is to perform the evaluation and the cleanup in a single instruction, so that the stack is always in a consistent state. Two new byte-code instructions were introduced to solve this. `MakeTupleAndCleanup` builds a tuple from a list of sub-expressions, then discards the same number of temporary values from the stack, while `ExpressionAndCleanup` evaluates an expression, then removes a configurable number of temporary values that belong to inner calls.

Both instructions are emitted only after the compiler has fully transformed the AST into a linear sequence of byte-code instructions. At this stage, the compiler has complete information about:

- how many temporary values each sub-expression will produce, and
- the exact points at which these temporaries can be safely discarded without affecting subsequent computations.

This guarantees that nested calls are executed in the correct order, with minimal stack usage and no risk of leaving stale values behind.

Working example – Fibonacci

A good example showcasing the use of function calls as an expression is the recursive version of the Fibonacci function in Figure 3.4. It returns the sum of two function calls. On top of that, it allows testing directly by using the result of the function call in the `print` function. Executing the code displays: `55`, which is the expected output. More complex edge cases were tested but they will not be detailed in this report.

```
fn fib(n: int) -> int {
    if n <= 1 {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

main {
  print(fib(10));
}
```

Figure 3.4: **Recursive Fibonacci program.** The function `fib` contains a nested-call expression `fib(n-1) + fib(n-2)`; the call `print(fib(2))` in `main` triggers the evaluation of that expression. The execution trace for this code is available in Appendix D, where the behavior of the two new instructions, `ExpressionAndCleanup` and `MakeTupleAndCleanup`, can be observed in detail.

3.2 Communication-Graph Visualization

The communication-graph visualization reconstructs the flow of messages between concurrently running programs in Althread by analyzing the VM execution trace. Each program state becomes a node, and each send or receive operation becomes a directed edge. This allows inspection of the sequence and direction of communications during execution.

3.2.1 Problems and Solutions

I identified several critical issues in earlier versions, but by applying fixes I managed to improve program discovery, event handling, and visualization.

The original algorithm for program discovery only inspected the `locals` field of the first VM state, which caused it to miss processes started without assignment (e.g., `run B();`) and those created inside loops. Additionally, only programs started in `main` were captured, while those spawned later inside other programs were ignored. This meant I got incomplete graphs and duplicate name errors when variable names were reused in loops. The solution is to scan all VM states. This way, dynamically spawned processes are detected, uniquely named, and included in the visualization.

Event matching and clock handling also required significant improvements. Previously, receive events were matched to sends by strict clock equality, which fails under Lamport's model where the receiver's clock must be at least the sender's clock $+1$. Furthermore, while the sender's clock was incremented for each event, the receiver's clock was left at zero, causing duplicate event identifiers. I corrected this by updating the receiver's clock during each receive event as:

$$\text{receiver\_clock} \leftarrow \max(\text{received\_clock}, \text{receiver\_clock}) + 1$$

This results in consistent ordering across processes. Additionally, I updated the JavaScript algorithm for matching nodes and events to filter send events by matching sender and receiver IDs, ignore already matched send nodes, and select the send event with the smallest Lamport number less than the receiver's timestamp. These changes avoid false matches and support multi-process communication and broadcasts.

Another issue was the timing of edge creation. Edges were previously created in the same pass as nodes, which could fail if the sender node had not yet been created. By separating edge creation from node creation, we ensure all nodes exist before edges are added to the graph.

Finally, the visualization was enhanced to provide a more comprehensive view of the system's execution. The updated display now includes program name, PID, clock, stack contents, and active channels for each program, offering users a clearer understanding of the runtime state.

3.2.2 Implementation Overview

On the Rust side, the VM trace is processed to detect `SEND` and `RECV` instructions. For receives, the previous VM state is inspected to extract the message tuple, the receiver's clock is updated using Lamport's rule, and a `MessageFlowEvent` is recorded with the sender, receiver, message, updated clock, and the post-receive VM state. For sends, the sender's current clock, channel name, and VM state are recorded in a similar event structure.

With these changes, the communication-graph visualization now reliably identifies all processes, maintains correct event ordering, and accurately represents message flow. To showcase

this, the example in Appendix C implements a simple leader election in a ring topology. Figure 3.5 shows the resulting communication graph. The visualization now produces the correct result without the previous duplicate-ID error.

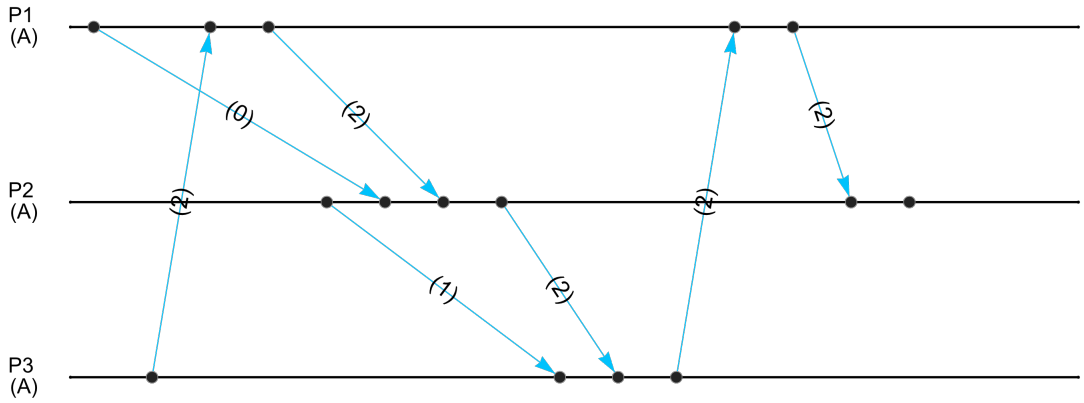

Figure 3.5: **Communication graph for the leader election example in Appendix C.** Each horizontal line represents a process, with events ordered from left to right. Blue arrows show message sends and their corresponding receives, matched using Lamport timestamps.

3.3 Import Support

Import functionality is a key feature of any mature programming language. While Althread is still in its early stages, implementing imports is essential for its growth. Imports enhance code usability, promote modularity, and enable the creation of reusable modules and packages. They also foster collaboration within the Althread community by allowing users to share and organize code effectively.

To design an import system for Althread, I analyzed how imports are implemented in several existing languages. Starting with Promela (and C/C++), the language currently used to teach Distributed Algorithms at the University of Strasbourg, I then examined TLA+, Python, JavaScript, and Go.

3.3.1 Promela (and C/C++)

Promela uses the `#include "file.pml"` directive to import files, identical to the preprocessor directive in C/C++. Promela relies on a `gcc` call with the `-E` flag [30], which stops compilation after preprocessing, and the `-x c` flag, which treats `.pml` files as C code (`man gcc`). The `#include` directive inserts the contents of the specified file at the location of the directive [31], effectively combining all included files into a single large file for compilation.

Key characteristics of Promela's import system include:

- Use of C preprocessor directives (`#ifndef`, `#define`, `#endif`) to prevent multiple and circular includes.
- Lack of modules, packages, or imports; all included files share a single `init` block as the entry point.
- No namespaces, requiring all inline functions, variables, and process names to have unique identifiers.

C follows the same approach as Promela, relying on the preprocessor for imports. C++, however, introduces namespaces, which provide a way to organize and encapsulate code. For example, a namespace `math` containing a function `add` can be accessed using `math::add()`. To make a method private, C++ provides a `private` declaration inside a class. While C++ namespaces offer additional features, their ability to avoid name conflicts is particularly relevant for modularity and imports.

3.3.2 TLA+

TLA+ modules provide a way to organize and reuse specifications. While modules may not always be critical for small specifications (which are often only a few hundred lines long, as noted by Hillel Wayne, a prominent advocate for formal methods [32, 33]), they are invaluable for structuring larger projects and improving code readability [34].

By default, all TLA+ files are modules. The two main concepts for working with modules are `EXTENDS` and `INSTANCE`, which allow functionality from other modules to be included and reused.

EXTENDS

The `EXTENDS` keyword merges the contents of another module into the current module's namespace. This is similar to the `#include` directive in Promela, where all definitions from the included file become part of the current file. In TLA+, the `LOCAL` keyword can be used to make certain operators private, preventing them from being accessed by other modules like in Figure 3.6.

```
---- MODULE math ----
EXTENDS Integers
LOCAL Helper(x) == x + 1  \* This is private
Square(x) == x * x        \* This is public
Increment(x) == Helper(x) \* This is public, uses the private helper
```

Figure 3.6: `EXTENDS` merges the contents of the `Integers` module into the current module. The `Helper` operator is private due to the `LOCAL` keyword, while `Increment` is public and uses `Helper` internally.

INSTANCE

The `INSTANCE` keyword allows the inclusion of another module while keeping its contents in a separate namespace. This approach avoids name conflicts and provides better organization for specifications. By naming the `INSTANCE`, as shown in Figure 3.7, its operators can be accessed using the `!` operator. This behavior differs from `EXTENDS`, which merges everything into the same namespace.

```
---- MODULE math ----
I == INSTANCE Integers
Square(x) == I!*(x,x)
```

Figure 3.7: `INSTANCE` creates a namespace for the `Integers` module, named `I`. The operator `Square(x)` uses `I` to access the `*` operator and calculate the square of `x`.

Unlike `EXTENDS`, multiple `INSTANCE`s can be declared within a single module, each with its own namespace. Additionally, marking an `INSTANCE` as `LOCAL` prevents it from being transitively included in other modules [34].

3.3.3 JavaScript

The JavaScript import/module system is a powerful and widely used feature, as described in the MDN documentation. However, I find that it has some limitations that can make code organization and debugging less intuitive. For example, when importing specific functions or objects from a module, they can be used directly by name, but there's no clear indication of their origin unless the file explicitly imports the entire module or uses aliases.

For instance, the following import: `import { name, draw } from "square.js";` allows the use of `draw()` directly, but it's not immediately clear where it came from. A more explicit approach, such as using a namespace, would make the origin of the function clearer. For example: `square.draw();`

This approach improves traceability and debugging but introduces its own challenges. For example, if the file name (`square.js`) changes, all imports referencing it will break. A potential solution could involve exporting a namespace directly from the module, like: `export square;` However, this raises additional questions, such as what happens if `square` is already a function or variable name within the module itself. Lastly, JavaScript uses `#` before a method name, for example, to declare it as private.

Ultimately, while the JavaScript module system is flexible, it has trade-offs. Using namespaces, file names as namespaces, or aliases each comes with its own drawbacks.

3.3.4 Python

In Python, every source file is a module, and imports provide access to its functionality [35]. For example, Figure 3.8:

```python
# spam.py
def grok(x):
    ...

import spam
a = spam.grok('hello')

from spam import grok
a = grok('hello')
```

Figure 3.8: Example of importing and using functions from a Python module. In both cases, the entirety of the files gets executed.

```python
# foo.py
import bar
def grok():
    pass

# bar.py
import foo
x = foo.grok()
# Fails: grok() is not yet defined
```

Figure 3.9: Example of cyclic imports causing failure due to uninitialized functions.

Each module operates in its own isolated namespace, with global variables bound within the module where they are defined. Importing a module executes its code to initialize the module object, which is cached in `sys.modules` for reuse [36]. Python searches for modules in the directories listed in `sys.path`, including:

- The directory of the script being run (or the current working directory).

- Directories specified in the `PYTHONPATH` environment variable.
- Standard library directories and `site-packages`.

If the module is not found, Python raises an `ImportError` [36].

Packages are directories containing a special `__init__.py` file, which initializes the package. Submodules are accessed using dot notation, such as `package.submodule`. Code inside an `__name__ == '__main__'` block runs only when the file is executed directly, not when imported as a module. Private methods, in Python, are prefixed with a single or double underscore.

Python handles cyclic imports by caching partially initialized modules in `sys.modules`. However, accessing uninitialized functions or variables during a cyclic import will fail.

David Beazley [37], a leading expert in Python, emphasizes keeping imports simple and understanding Python's mechanisms to debug issues effectively [35].

3.3.5 Go

Go organizes code into packages, and every Go program starts execution in the `main` package [38]. This means even the simplest program, such as one that only prints a message, must be part of a package. By convention, the package name matches the last element of the import path. For example, the `math/rand` package contains files that begin with `package rand`.

Go uses the package name to access its exported functionality, but the full import path must be declared. For instance:

```go
package main

import (
    "fmt"
    "math/rand"
)
func main() {
 fmt.Println(
 "My favorite number is",
 rand.Intn(10))
}
```

Figure 3.10: Example of importing multiple packages using parentheses [38]

```go
package main

import "fmt"
import "math"

func main() {
 fmt.Println(
 "Now you have %g problems.",
 math.Sqrt(7)
 )
}
```

Figure 3.11: Example of importing packages without parentheses [38].

Go supports two styles of imports: using parentheses to group multiple packages (Figure 3.10) or importing each package individually (Figure 3.11). While the parentheses style is considered good practice, the individual import style may be preferred for single imports, as it avoids unnecessary syntax. This design decision is interesting but somewhat inconsistent, as having only the parentheses style would provide a single, uniform way to write imports.

In Go, to decide what is exported is based on if they begin with a capital letter or not. For example, `Pizza` and `Pi` are exported names, while `pizza` and `pi` are not. This convention simplifies visibility rules, as exported names are automatically accessible across packages [38].

To use external imports, Go requires initializing a module with `go mod init`. This module system tracks release versions and dependencies, allowing external packages to be imported directly using their repository URL. For example, running `go get <url>` fetches the

package without requiring additional tools like `pip` in Python. This approach is elegant and integrates seamlessly with version control systems like GitHub.

3.3.6 Althread

I designed this version of Althread's import system to be quite straightforward. It draws influence from all the languages we've previously discussed, but was particularly influenced by Go, which I really liked. Table F.1 summarises the import mechanisms of the surveyed languages and highlights the gaps that motivated the design of Althread's import system.

A single `import` block can be declared anywhere in the file. It contains a list of relative paths from the importing file to the target file, without the `.alt` extension. Duplicate import names aren't allowed, that's why Althread provides aliasing. Althread also checks for circular imports and returns an error if detected. Figure 3.12 demonstrates the basic import syntax and how imported elements are accessed using the dot operator:

```
import [
    math,
    cool/fib,
    display as d
]
main {
    print(math.max(7,3));
    print(fib.N);
    print(fib.fibonacci_iterative_N());
    fib.N = 10;
    print(fib.fibonacci_iterative_N());
    // print(fib.fibonacci_iterative(fib.N, 0, 1));
    run d.Hello();
}
```

Figure 3.12: Import syntax demonstration showing namespace access with the dot operator. The commented line attempts to call a `@private` function, which would result in a compilation error. The code for the imported modules can be found in Appendix E.

Like most languages, Althread provides a way to prevent external access to local helpers through the `@private` directive. This can be applied to function blocks or `program` blocks, and even allows multiple `main` blocks to coexist: adding `@private` before a `main` block makes it invisible to importing files. Shared variables are always imported and modifiable, while conditions (always/never/eventually) are imported but read-only. It is possible to add additional conditions to imported shared variables. It would be neat to notify users when constraints contradict each other, or better yet, when there's a constraint error, show all constraints affecting those shared variables for easier debugging.

Althread also supports remote dependencies, as shown in Appendix F. This requires a dependency manager and version storage, so I implemented a package manager in the Althread CLI that lets users add remote dependencies with a few commands. It's a neat feature that makes code sharing easy. For versioning, I took inspiration from Rust (the language Althread's compiler is written in) and chose TOML with a file called `alt.toml`. Since Rust has plenty of TOML parsers available, this simplified implementation significantly. There's one caveat though, I haven't fully implemented packages yet. Unlike Go, Althread won't declare packages inside files. Instead, the package name will be the enclosing folder's name. Currently, Althread just imports files, but the goal is to add a `mod.alt` file that regroups folder contents

and makes them accessible under the folder name: `import [path/module_name]`. I think it's smart to keep the current behavior while adding this new package system.

Implementing this required extensive changes to both the Althread compiler and the grammar. The grammar for imports is available in Appendix F, and I also had to update the definition of an `identifier`, to support multiple names joined with `'.'`.

To achieve an important part of these tasks, I coded a module resolver that handles the heavy lifting: validating paths, checking for circular imports, managing remote dependencies, finding project roots, building ASTs for imported files, and returning both the AST and resolved path to the compiler.

Import resolution would be straightforward without namespaces, but that's where I hit some challenges. The process seems simple: start with the main file, check for import blocks, call the module resolver, get the AST, compile it, and merge everything into the global compiler context. But functions, shared variables, programs, and conditions need proper module prefixes. I found a working solution where at each import depth, the compiler only qualifies its own imports with the current module name, leaving parent qualification to the parent level, see Figures 3.133.14. This worked beautifully, though I had to delay private access checking to a second phase after qualification.

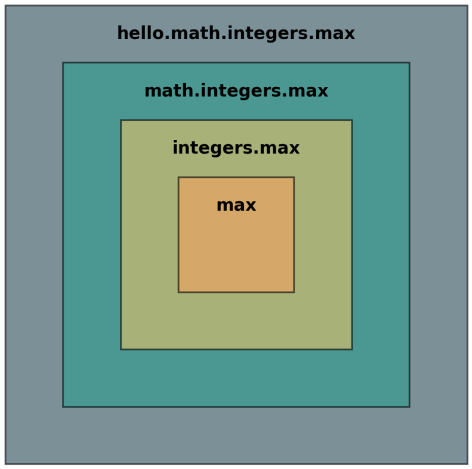

Figure 3.13: Each layer represents an import depth, with the outermost layer showing the fully qualified name as seen from the root file. The compiler qualifies names at each level, building the complete namespace path from inside out.

```
# root file
import [hello]

# hello.alt
import [math]

# math.alt
import [integers]

# integers.alt
fn max(...)
```

Figure 3.14: Each file imports the next level, and the compiler builds the qualified names by traversing this chain from the deepest level upward.

The trickiest part was implementing imported channels. Previously, channels could only be safely declared in the `main` block [39]. Declaring channels elsewhere would likely cause type inference errors when receives are set up on program input channels without corresponding output declarations. This made importing modules impossible since they couldn't infer channel types, and forced all channel declarations into the main file's `main` block. I solved this with a precompiling phase that scans the AST for all channel declarations (including imports) and adds their definitions to the global compiler context. This approach works well and I'm quite happy with it, but there are still a few issues to address before the end of the internship. One edge case is declaring a `list` of `list`s of processes, which is trickier because the analysis needs to follow the access into the outer list and then into the inner list to determine the actual process type.

Finally, I updated the error reporting system to properly show the filepath of the file where the error happens, and build an error stack for improved logging and debugging.

3.4 The Web-based Platform

The modifications described earlier, together with other updates not included here due to space constraints, required substantial changes to the Althread web editor. This section presents two of the most significant improvements; the rest can be best appreciated by exploring the editor directly. At the time of writing, the previous stable version is available at: https://althread.github.io/editor and the current development version, which also contains additional updates, at: https://althread.github.io/dev/editor.

3.4.1 Virtual File System

One interesting challenge I had to tackle for the web platform was implementing a complete virtual file system that runs entirely in the browser. This was necessary to make imports work properly, because single-file programs aren't sufficient when building a real development environment.

Each file and directory gets a unique ID, and I store the file tree structure separately from the actual file contents. The tree structure lives in `localStorage` under one key (see Appendix F), while each file's content gets its own storage key. This approach works well because it enables lazy loading of file contents.

This feature integrates with the Rust compiler. When compilation is triggered, the system builds a flat key-value map of all files (path → content) and serializes it to pass to the Rust side. The Rust compiler then deserializes this into a `HashMap<String, String>` and creates its own `VirtualFileSystem` for compilation. The compiler has no idea it's not working with real files.

The file explorer supports all the expected operations: drag-and-drop to move files around, context menus for operations, multi-selection for batch operations, and even ZIP export to download entire projects. I also implemented a search feature that makes navigating larger projects much easier.

3.4.2 Package Manager

Building a package manager for a web-based IDE was another engaging challenge. I wanted Althread developers to be able to share code easily, so I implemented a complete dependency management system that works entirely in the browser.

The core architecture is built around a `WebPackageManager` that integrates with the virtual file system. When installing packages, it fetches content from GitHub using the available API, recursively discovers all `.alt` files in the repository, and caches everything in `localStorage`.

Packages integrate with the virtual file system. All dependencies get installed under a `deps/` directory, with each package getting its own subdirectory. The system preserves the original directory structure from GitHub, so imports work exactly as expected. The installation process is simple: validate the package name using WebAssembly-based validation, fetch the content from GitHub, cache it locally, and load it into the virtual file system.

The UI has a full-featured dialog for complete package management. Both support adding dependencies, installing packages, viewing the cache, and managing project configuration. The WebAssembly integration handles TOML parsing and package name validation, ensuring everything follows Althread conventions.

Chapter 4

Conclusion

4.1 Summary of Contributions

During my three-month internship, I made significant progress on Althread's capabilities as an educational programming language for concurrent and distributed systems. My main contributions include completing the implementation of user-defined functions with cross-function calling support, control-flow analysis for return path safety, and nested function call handling through new bytecode instructions. I also integrated and improved the communication graph visualization, fixing critical issues in program discovery and event matching.

The most significant contribution was designing and implementing a complete import and package system. Drawing inspiration from modern languages like Go and Python, I created a namespace-based system supporting both local file imports and remote dependencies, with circular import detection and proper namespace qualification. I also enhanced the web platform with a virtual file system, package manager, and redesigned user interface featuring syntax highlighting and comprehensive project management tools.

4.2 Personal Reflection

I really enjoyed working during this internship. I can say that I've faced the pros and cons of doing a remote internship, but in both cases it requires lots of self-discipline and continued effort even when it gets hard and feels like giving up. The technical challenges I encountered, from implementing control-flow analysis to designing a browser-based package manager, pushed me to explore areas of computer science I hadn't previously encountered in depth.

This internship reinforced my interest in educational technology and open-source development. Knowing that my work will potentially impact future students learning about distributed systems provides a sense of purpose that extends beyond the immediate technical achievements.

4.3 Future Perspectives

My internship hasn't finished yet, I have less than four weeks remaining. One area I didn't explore was improving Althread's verification capabilities. If time permits, I would like to add an LTL (Linear Temporal Logic) block, providing more powerful notation than the current `always/never/eventually` blocks, and provide access to the VM state directly within Althread code for sophisticated verification scenarios.

While I have tested numerous edge cases, there are still issues that need fixing and comprehensive test coverage to establish. I also need to complete the technical documentation for all implemented features. Looking beyond my internship, the import system and package manager create the foundation for a thriving ecosystem of shared algorithms, positioning Althread as a significant educational tool for the distributed systems community.

Chapter 5

Bibliography

[1] European Association for Quality Assurance in Higher Education. *FIGURE – Engineering Education by Research Universities*. URL: https://www.enqa.eu/membership-database/figure-engineering-education-by-research-universities/.

[2] Lucian Mocan. *(TER) Compilation and Interpretation of Functions in the Althread Language*. Research Project (TER), University of Strasbourg, supervised by Quentin Bramas. URL: https://lucianmocan.com/projects/althread_functions/.

[3] Julien Clavel (JulClav). *add "eventually" support to althread: A lightweight threading library*. 2025. URL: https://github.com/JulClav/althread.

[4] Violette Lesouef (lqpi0). *add communication graph to althread: A lightweight threading library*. 2025. URL: https://github.com/lqpi0/althread.

[5] Center for World University Rankings. *University of Strasbourg Ranking (2024)*. URL: https://cwur.org/2024/university-of-strasbourg.php.

[6] Université de Strasbourg. *Portail des statistiques – Université de Strasbourg*. URL: https://statistiques.unistra.fr/.

[7] University of Strasbourg ICube Laboratory. *ICube – Laboratory of Engineering, Computer Science and Imagery (UMR 7357)*. Institution homepage. Engineering-science, computer-science & imagery research laboratory. URL: https://en.unistra.fr/research/sciences-and-technologies/icube-laboratory-of-engineering-computer-science-and-imagery-umr-7357.

[8] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems 4th edition*. 2025. URL: https://www.distributed-systems.net/index.php/books/ds4/.

[9] Leslie Lamport. *A High-Level View of TLA+*. https://lamport.azurewebsites.net/tla/high-level-view.html.

[10] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[11] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997). URL: https://spinroot.com/spin/Doc/ieee97.pdf.

[12] Chris Newcombe et al. *Formal Methods at Amazon Web Services*. Tech. rep. Amazon Web Services, 2015. URL: https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf.

[13] Markus A. Kuppe et al. "Validating System Executions with the TLA+ Tools". In: *TLA+ Conference 2024*. Presentation, slides/paper available online. Seattle, USA, Apr. 2024. URL: https://conf.tlapl.us/2024/MarkusAKuppe-ValidatingSystemExecutionsWithTheTLAPlusTools.pdf.

[14] Murat Demirbas. *TLA+ Modeling of MongoDB Transactions*. Slides, TLA+ Community Event @ ETAPS 2025. Hamilton, Ontario, Canada; slides available online. May 2025. URL: https://conf.tlapl.us/2025-etaps/demirbas-slides.pdf.

[15] Aman Shaikh. *Specifying BGP using TLA+*. Slides, TLA+ Community Event, satellite to FM 2024. Presentation slides available online. Sept. 2024. URL: https://conf.tlapl.us/2024-fm/slides-shaikh.pdf.

[16] TLA+ Community Event & Conference. *TLA+ Community Event & Conference (Homepage)*. General homepage for the TLA+ Community Event & Conference series. 2025. URL: https://conf.tlapl.us/home/.

[17] ResearchGate. *SPIN windows showing PROMELA code for B2 Scheme and message sequence for a simulation run*. URL: https://www.researchgate.net/figure/SPIN-windows-showing-PROMELA-code-for-B2-Scheme-and-message-sequence-for-a-simulation-run_fig1_2531660.

[18] ResearchGate. *A part of Promela code obtained in Experiment 2*. URL: https://www.researchgate.net/figure/A-part-of-Promela-code-obtained-in-experiment-2_fig2_220772534.

[19] MDPI. *Figure G003: [Description from article, e.g., "Diagrammatic plot from Appl. Sci. 2022, 12(6)"]*. URL: https://www.mdpi.com/applsci/applsci-12-02990/article_deploy/html/images/applsci-12-02990-g003-550.jpg.

[20] Matthias Papesch and Cora Burger. *Constructivist Approach to Learning Security Protocols (Technical Report TR-2002-09)*. See especially page 12 for visualization prototype. URL: https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/ncstrl.ustuttgart_fi/TR-2002-09/TR-2002-09.pdf.

[21] William Schultz. *Spectacle: Interactive web-based tool for exploring and sharing TLA+ specifications*. GitHub repository; MIT license; last accessed: July 2025. 2025. URL: https://github.com/will62794/spectacle.

[22] pest Project Contributors. *pest. The Elegant Parser*. pest v2.8.0. URL: https://docs.rs/pest/latest/pest/.

[23] Althread Project Contributors. *Internal Architecture Guide*. URL: https://althread.github.io/docs/guide/internal/architecture.

[24] Althread Contributors. *Virtual Machine Module – Althread Project*. URL: https://github.com/althread/althread/tree/53d5816baed79d6a648a8f3ad28f1f03202cfc79/interpreter/src/vm.

[25] W3C Community Group. *Introduction*. June 2025. URL: https://webassembly.github.io/spec/core/intro/introduction.html.

[26] Yutian Yan et al. "Understanding the performance of webassembly applications". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. Virtual Event: Association for Computing Machinery, 2021, pp. 533–549. ISBN: 9781450391290. DOI: 10.1145/3487552.3487827. URL: https://doi.org/10.1145/3487552.3487827.

[27] W3C. *Web Content Accessibility Guidelines (WCAG) 2.1 - Use of Color*. May 2025. URL: https://www.w3.org/TR/WCAG21/#use-of-color.

[28] Althread Documentation. *Functions — User-Defined Functions (Althread Guide)*. URL: https://althread.github.io/dev/en/docs/guide/modularity/user-defined-functions/.

[29] Steven S. Muchnick. *Control-Flow Analysis (Chapter 7 in Advanced Compiler Design and Implementation)*. Specifically chapter 7 (page 169 onwards) of the 1997 book. URL: https://www.scribd.com/document/323133878/Steven-S-Muchnick-Advanced-Compiler-Design-And.

[30] Nimble Code. *Promela PreProcessing — main.c (lines 578–614)*. URL: https://github.com/nimble-code/Spin/blob/master/Src/main.c#L578-L614.

[31] Microsoft Docs. *#include Directive (C/C++)*. URL: https://learn.microsoft.com/en-us/cpp/preprocessor/hash-include-directive-c-cpp?view=msvc-170.

[32] Hillel Wayne. *LinkedIn Profile*. URL: https://www.linkedin.com/in/hillel-wayne.

[33] Hillel Wayne. *Consulting Services — Hillel Wayne*. URL: https://hillelwayne.com/consulting/.

[34] Learn TLA+ Guide. *Modules (Core section — Learn TLA+)*. URL: https://learntla.com/core/modules.html.

[35] David Beazley. *Modules and Packages: Live and Let Die! — PyCon 2015*. URL: https://www.youtube.com/watch?v=0oTh1CXRaQ0.

[36] Python Documentation. *Standard Modules — Python 3 Tutorial*. URL: https://docs.python.org/3/tutorial/modules.html#standard-modules.

[37] David M. Beazley. *About — David Beazley*. URL: http://dabeaz.com/about.html.

[38] The Go Authors. *A Tour of Go – Basics*. URL: https://go.dev/tour/basics.

[39] Althread Documentation. *Sending Messages — Creating a Channel (Althread Guide)*. URL: https://althread.github.io/en/docs/guide/channels/create/#sending-messages.

Appendix A

AST

This is the resulting AST for the code snippet presented in Subsection 2.2.2. It can be reproduced by running the code in a local environment using the `althread-cli`:

```
althread-cli compile demo.alt
```

where `demo.alt` is the name you would give the file in which you would store the code showcased in Figure 2.1.

```
shared
|-- decl
|   |-- keyword: let
|   |-- ident: N
|   |-- datatype: int
|   `-- value
|       `-- int: 0
`-- decl
    |-- keyword: let
    |-- ident: Start
    `-- value
        `-- bool: false

always
`-- binary_expr
    |-- left
    |   `-- ident: N
    |-- op: ==
    `-- right
        `-- int: 0

main
|-- decl
|   |-- keyword: let
|   |-- ident: pa
|   `-- value
|       `-- run: A
|-- channel decl
|-- binary_assign
|   |-- ident: Start
|   |-- op: =
|   `-- value:
|       `-- bool: true
`-- send
    |-- out
    `-- tuple
        |-- int: 125
        `-- bool: true
```

```
A
|-- wait_control
|   `-- await case
|       |-- ident: Start
`-- wait_control
    `-- await case
        |-- receive
        |   |-- channel 'in'
        |   |-- patterns (x,y)
        |   `-- print
        |       `-- tuple
        |           `-- string: "received "
        |   |-- binary_assign
        |   |   |-- ident: N
        |   |   |-- op: =
        |   |   `-- value:
        |   |       `-- binary_expr
        |   |           |-- left
        |   |           |   `-- ident: N
        |   |           |-- op: +
        |   |           `-- right
        |   |               `-- int: 1
```

Figure A.1: AST for the code snippet in Figure 2.1

Appendix B

DFS Algorithm for Detecting Missing Returns

The compiler uses a depth-first search (DFS) on the control-flow graph (CFG) to detect the first path in a function that does not contain a `return` statement. The algorithm works as follows:

1. Start from the entry node with a flag indicating no return has been seen yet.
2. Maintain two tracking sets:
   - `visited_on_current_path`: to avoid cycles in the current DFS path.
   - `globally_visited_tuples`: to avoid reprocessing `(node, return_state)` pairs.
3. For each node visited:
   (a) Skip if this `(node, return_state)` combination has already been processed.
   (b) Skip if this node is already in the current path (*cycle detection*).
   (c) Add the node to the current path tracking set.
   (d) Update the flag to `true` if the current node is a `return` statement.
   (e) If this is the exit node **or** a dead-end node (no successors):
      - If the flag is `false`, report this path as missing a return and **stop**.
      - Otherwise, continue to the next path.
   (f) Otherwise, add all successors to the stack with the updated flag.
   (g) Remove the node from the current path tracking set (*backtrack*).
4. Stop as soon as the first missing return is found.

The Rust implementation of this algorithm can be found in the file at:
https://github.com/althread/althread/blob/dev/interpreter/src/analysis/control_flow_graph.rs

Appendix C

Leader Election with Eventually Condition

```
shared {
  let Done = false;
  let Leader = 0;
}

program A(my_id: int) {
  let leader_id = my_id;
  send out(my_id);

  loop atomic await receive in (x) => {
    print("receive", x);
      if x > leader_id {
        leader_id = x;
        send out(x);
      } else {
        if x == leader_id {
          print("finished");
          send out(x);
          break;
        }
      }
  };

  if my_id == leader_id {
    print("I AM THE LEADER!!!");
    ! {
        Done = true;
        Leader += 1;
    }
  }
}

eventually {
    Leader == 1;
}

main {
  let n = 3;
  let a:list(proc(A));
  for i in 0..n {
    let p = run A(i);
    a.push(p);
  }
  for i in 0..n-1 {
    let p1 = a.at(i);
    let p2 = a.at(i+1);
    channel p1.out (int)> p2.in;
```

```
48    }
49    let p1 = a.at(n-1);
50    let p2 = a.at(0);
51
52    channel p1.out (int)> p2.in;
53
54    print("DONE");
55  }
```

Figure C.1: **Leader election with an `eventually` condition.** This program launches three instances of `A`, each sending its ID around a ring of channels. A process updates its `leader_id` if it receives a higher ID, and when the leader's ID comes back to it, it declares itself leader, sets the shared `Leader` variable, and terminates. The `eventually` block asserts that `Leader == 1` in all possible executions. This is the program used to generate the communication graph in Figure 3.5.

Appendix D

Execution Trace for Fibonacci Example

The following is the execution trace for the recursive Fibonacci program shown in Figure 3.4. It demonstrates how the virtual machine evaluates the sum of two function calls `fib(n-1) + fib(n-2)` and evaluates the result of `fib(10)` and sets it as an argument to `print` by using the two new instructions, `ExpressionAndCleanup` and `MakeTupleAndCleanup`, to manage the stack and intermediate results.

```
#0: 11: eval (2)
#0: 11: fib()  (unstack 1)
#0: 3: eval [0] <= 1
#0: 3: jumpIf 5 (unstack 1)
#0: 6: eval ([0] - 1)
#0: 6: fib()  (unstack 1)
#0: 3: eval [0] <= 1
#0: 3: jumpIf 5 (unstack 1)
#0: 4: eval [0]
#0: 4: return "value"
#0: 6: eval ([1] - 2)
#0: 6: fib()  (unstack 1)
#0: 3: eval [0] <= 1
#0: 3: jumpIf 5 (unstack 1)
#0: 4: eval [0]
#0: 4: return "value"
#0: 6: eval [1] + [0] and cleanup (unstack 2)
#0: 6: return "value"
#0: 11: make tuple (1) and cleanup (unstack 1)
#0: 11: print()  (unstack 1)
#0: 11: unstack 1
#0: 10: end program
```

Figure D.1: **Execution trace for the Fibonacci example.** Each line shows the instruction index, the operation performed, and the number of stack elements removed (`unstack`).

Appendix E

Import Example Modules

This appendix contains the source code for the three modules imported in the example shown in Figure 3.12.

```
fn max(a: int, b: int) -> int {
  if a > b {
    return a;
  }
  return b;
}
```

Figure E.1: **math** Simple math utility function that returns the maximum of two integers.

```
shared {
  let N: int = 8;
}

@private
fn fibonacci_iterative(n: int, a: int, b: int) -> int {
  for i in 1..n {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

fn fibonacci_iterative_N() -> int {
  return fibonacci_iterative(N, 0, 1);
}
```

Figure E.2: **cool/fib** Fibonacci module demonstrating shared variables and private functions. The shared variable `N` can be accessed and modified by importing modules, while the `@private` function `fibonacci_iterative` is only accessible within this module.

```
program Hello() {
  print("Hi there!");
}
```

Figure E.3: **display** Simple display module containing a program block that prints a greeting message.

Appendix F

Imports

Import grammar in Figure F.1, URL imports in Figure F.2 and Virtual Filesystem in Figure F.3.

```
// Import block definition
import_block = { IMPORT_KW ~ "[" ~ import_list? ~ "]" }
import_list = { import_item ~ ("," ~ import_item)* ~ ","? }
import_item = { import_path ~ (AS_KW ~ identifier)? }
import_path = { import_segment ~ ("/" ~ import_segment)* }
import_segment = { domain_identifier | identifier }
domain_identifier = @{ ASCII_ALPHA ~ domain_char* }
domain_char = _{ ASCII_ALPHANUMERIC | "_" | "." | "-" }
```

Figure F.1: Import grammar definition showing support for both local paths and GitHub URLs. The `domain_identifier` rule handles GitHub domain parsing, while `import_path` supports nested directory structures.

```
import [
    github.com/lucianmocan/math-alt/algebra/integers
]

main {
    print(integers.add(1,2));
}
```

Figure F.2: Remote dependency import from GitHub. The package is accessed using the last segment of the path (`integers`) as the namespace identifier, following Go's convention.

```
althread-file-system: [
    {
        "id":"e0b971c7-bba1-42f2-9830-b6a876ccc101",
        "name":"utils",
        "type":"directory",
        "children":[]
    },
    {
        "id":"ab4c6b89-26f4-45ec-97d8-c2c9e33387e7",
        "name":"main.alt",
        "type":"file"
    }
]
```

Figure F.3: The virtual file system tree structure stored in `localStorage`.

Table F.1: A side-by-side comparison of import mechanisms across various programming languages, highlighting the design choices made for Althread.

Language	Syntax	Namespace / Scoping	Circular-import detection	Granularity	Remote / URL imports	Package manager	Visibility modifiers
Promela / C	`#include "file.pml"`	No (single global namespace)	Guard macros (`#ifndef`) prevent multiple includes; no built-in cycle check	Whole file only	No	None (manual copy)	None
TLA+	`EXTENDS Mod / INSTANCE Mod`	`EXTENDS` merges namespace; `INSTANCE` creates a separate one (access via `Mod!`)	No explicit cycle detection (duplicate definitions cause errors)	Whole module (all operators)	No	None (manual)	`LOCAL` keyword makes an operator private
JavaScript	`import {foo} from "./mod.js"`	Module-level namespace; imported symbols become local bindings	Runtime module cache detects cycles and throws an error at load time	Named exports, default export, or `* as ns`	No (only local files)	npm / yarn (external)	`export` / `export default` (no language-level private keyword)
Python	`import pkg.mod / from pkg import name`	Modules act as namespaces; accessed as `pkg.mod`	Detects cycles at import time; partially-initialised modules may raise `AttributeError`	Whole module; can import specific attributes	No (only local files)	pip / poetry (external)	`_private` naming convention, `__all__` for export control
Go	`import "math"`	Package name is a namespace; accessed as `pkg.Func`	Compile-time error for import cycles	Whole package (only exported identifiers are visible)	Yes – import path can be a remote repository URL	`go mod` (built-in)	Exported identifiers start with a capital letter; others are private
Althread (this work)	`import [ math, ... ]`	Each import becomes a module; accessed via `module.name`	Resolver phase checks for circular imports and aborts with an error	File-level import; future work may allow symbol-level imports	Yes – GitHub URL syntax (e.g. `import [ git... ]`)	`alt.toml` + WebPackageManager (browser-side)	`@private` on functions/programs; shared vars are always public (read-only for conditions)