Pontifícia Universidade Católica
do Rio de Janeiro

Renan Almeida de Miranda Santos

Revisiting Monitors

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor : Prof. Noemi de La Rocque Rodriguez
Co-advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
February 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Renan Almeida de Miranda Santos

Revisiting Monitors

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Noemi de La Rocque Rodriguez
Advisor
Departamento de Informática – PUC-Rio

Prof. Roberto Ierusalimschy
Co-advisor
Departamento de Informática – PUC-Rio

Prof. Ana Lucia de Moura
Departamento de Informática – PUC-Rio

Prof. Luiz Fernando Bessa Seibel
Departamento de Informática – PUC-Rio

Rio de Janeiro, February the 1st, 2020

Renan Almeida de Miranda Santos

Majored in Computer Science by Pontifícia Universidade Católica do Rio de Janeiro. Currently working as a researcher in Programming Languages and Distributed Systems for PUC-Rio and Instituto Tecgraf.

Bibliographic data

Almeida de Miranda Santos, Renan

Revisiting Monitors / Renan Almeida de Miranda Santos; advisor: Noemi de La Rocque Rodriguez; co-advisor: Roberto Ierusalimschy. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2020.

v., 56 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Concorrência;. 3. Monitores;. 4. Memória Compartilhada;. 5. Imutabilidade;. 6. Problemas de Concorrência Clássicos;. 7. Linguagens de Programação Concorrentes.. I. Rodriguez, Noemi. II. Ierusalimschy, Roberto. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

I thank my parents and grandparents for prioritizing and encouraging my education.

I also thank my girlfriend for being by my side in times of uncertainty.

I thank PUC-Rio and LabLua for providing the perfect environment for the development of this dissertation.

Finally, I thank my advisors for their dedication to this project among countless meetings, discussions and reviews.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Abstract

Almeida de Miranda Santos, Renan; Rodriguez, Noemi (Advisor); Ierusalimschy, Roberto (Co-Advisor). **Revisiting Monitors**. Rio de Janeiro, 2020. 56p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Most current programming languages do not restrict the use of the concurrency primitives they provide, leaving it to the programmer to detect data races. In this dissertation, we revisit the monitor model, which guards against data races by guaranteeing that accesses to shared variables occur only inside monitors, and show that this concept can be implemented in a programming language with referential semantics, given appropriate typing rules. We describe the Aria programming language, designed with native monitors according to these rules. Through the discussion of classic concurrency problems, we evaluate the use of Aria monitors for synchronization at different levels of granularity and extend the language with new features to address the limitations of monitors regarding performance and expressiveness.

Keywords

Concurrency; Monitors; Shared Memory; Immutability; Classic Concurrency Problems; Concurrent Programming Languages.

Resumo

Almeida de Miranda Santos, Renan; Rodriguez, Noemi; Ierusalimschy, Roberto. **Revisitando Monitores**. Rio de Janeiro, 2020. 56p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A maioria das linguagens de programação modernas fornece ferramentas para programação concorrente sem restringir seu uso. Assim, fica a cargo do programador evitar a ocorrência de condições de corrida. Nessa dissertação, revisitamos o modelo de monitores, projetados para prevenir condições de corrida ao limitar o acesso à variáveis compartilhadas, e mostramos que monitores podem ser implementados em linguagens de programação com semântica referencial, dadas as regras de tipagem apropriadas. Nós descrevemos a linguagem de programação Aria, projetada com monitores nativos seguindo a proposta original do modelo. Através da resolução de problemas clássicos de concorrência, nós avaliamos o uso de monitores em Aria para sincronização em diferentes níveis de granularidade, e extendemos a linguagem com novos recursos a fim de contemplar as limitações do modelo envolvendo desempenho e expressividade.

Palavras-chave

Concorrência; Monitores; Memória Compartilhada; Imutabilidade; Problemas de Concorrência Clássicos; Linguagens de Programação Concorrentes.

Table of contents

1
Introduction

During the last years, CPU core count has been steadily increasing in mainstream computers. Along this time, many authors have discussed the complexity of programming multicore applications, specifically as regards avoiding race conditions in shared memory (1). However, although most popular programming languages offer tools for shared memory programming, be it through libraries or native language constructs, these tools largely leave it up to the programmer to deal with race conditions.

We chose monitors as the main concept for our work because they were designed precisely to avoid race conditions. Monitors allow concurrent processes to share data in an orderly manner, with monitor procedures as the only way to access shared variables (2). However, monitors were not envisioned to coexist features of modern programming languages such as referential semantics. In our work, we revisit the concept to allow it to maintain its original guarantees in a programming language with references. Much of this adaptation consisted of adding constraints to the type system that ended up being as important as the monitors themselves.

In this dissertation, we discuss how a programming language can be designed to guarantee the absence of race conditions in shared memory applications. We showcase a programming language, Aria, that was designed from scratch to use monitors and provide strong guarantees regarding data shared among threads, so that it is possible for the compiler to enforce the absence of data races. In order to showcase Aria's capabilities, we provide an implementation for two classic concurrency problems. The first of these problems is a data structure example, while the second is more interesting in that it illustrates how Aria's monitors can be used to control shared access at larger levels of granularity than that of a single function call.

We also discuss some of the limitations of the monitor model regarding performance and expressiveness, and address those by extending the monitor concept with two new features. We demonstrate the benefits of the added features by revisiting one of the previous classic problems and providing an example that uses a more complex data structure. We explain the compiler implementation for the new features and monitors in detail, while also provid-

ing benchmark results that measure the language's performance.

Lastly, because we had safety as our main concern when designing Aria, we provide a rigorous argument for the absence of data races in the language. Ours is not a formal proof for safety in Aria, nonetheless, it serves as a solid argument for the guarantees the language aims to provide.

Regarding how this dissertation in organized, Chapter 2 presents the ideas behind the original monitor proposal and discusses references to the concept in modern programming languages. Chapter 3 gives and overview of Aria focused on its concurrency constructs alongside some examples. Chapter 4 presents the new features added to monitors and discusses the rationale behind their design. Chapter 5 contains an in-depth explanation of the implementation of Aria, and Chapter 6 provides performance benchmarks for monitors and the added features. Finally, Chapter 7 argues for the absence of data races in Aria.

2
Monitors

In this chapter, we recall the main ideas behind the original proposals for monitors and discuss some of the references to the concept in modern programming languages.

Notations for monitors were proposed in independent publications by Brinch Hansen (3) and Hoare (4), with exchanges between the two authors leading to the development of the concept (2). Monitors are a combination of data structures and operations used to access them that can be called by different execution flows, always under mutual exclusion. Monitors only have access to variables within their own scope (4). Execution flows calling monitor operations may block in condition queues under conditions specified by the programmer.

In his personal history on Monitors and Concurrent Pascal, Brinch Hansen states explicitly:

> Race conditions are prevented by a simple scope rule that permits a process, monitor, or class to access its own variables only. In a suitably restricted language this rule can easily be checked by a compiler. However, in a language with pointers and address arithmetic, no such guarantee can be offered.

Both the scope rule and the observation about the possibility of having the compiler check it are fundamental to our discussion.

Brinch Hansen's proposal for monitors was closely tied to his implementation of the *Concurrent Pascal* (2) programming language, which was intended for writing operating systems. The design of Concurrent Pascal did not include references, and the compiler was able to check the scoping rule, enforcing the absence of data races. Reading and writing shared variables occurred only inside monitor procedures.

Nowadays, some programming languages and libraries are said to provide monitor-like constructs because they offer the possibility of protecting functions with mutual exclusion and some construct similar to a condition queue. A rough idea of how monitors are widely perceived today can be gained from the definition from Wikipedia (5). Monitors are described as synchronization

constructs that *"enable threads to have both mutual exclusion and the ability to block depending on a given condition"*. The same entry offers a variant definition of a monitor as a *"thread-safe class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread"*. In these definitions, a monitor is defined in terms of the things it can do, with disregard to the restrictions it should enforce. This reflects a lack of understanding of the original proposal. If it were up to the programmer to encapsulate access to shared data inside such functions, with no support from the compiler, monitors would not ensure the absence of data races.

In Java, for example, every *Object* has a lock and a condition variable (a *wait set*) that can be manipulated, respectively, through the *synchronized* keyword and *wait*, *notify*, and *notifyAll* methods. An object with that set of capabilities by itself has often been characterized as a monitor (6). Evidently, such "monitors" are missing a crucial feature from the original definition: Java's constructs do not prohibit unprotected data sharing, simply providing programmers with a monitor-like syntax with which to code their own rules.

Besides, most mainstream languages today include references, which, as pointed out by Brinch Hansen, make it impossible for the compiler to provide the desired guarantees. Consider what would happen if a language tried to provide the previously described monitor semantics while including pointers. A given monitor could store a pointer to an integer value and provide a function that returned this pointer. After that function is called, the caller would have a reference to the integer value and would be able to access it without going through the monitor. Symmetrically, a monitor could retain pointers passed to its procedures as arguments, allowing future calls from different processes to share variables which were not declared in the monitor's scope.

In this dissertation, we revisit the monitor concept addressing the aforementioned issues relating to references. We showcase a language designed to enforce the absence of time-dependent programming errors, combining the original monitor properties with modern programming language concepts.

3
Aria

In this chapter, we give an overview of Aria's features and discuss how the design of the language addresses the goals we established of combining the original monitor guarantees with referential semantics. We also use two examples of classic concurrency problems to analyze the language's capabilities.

3.1
Language Design

Aria has many features common to ordinary procedural languages. It provides basic primitive types, arrays, global variables, functions, loops, if-else statements and the like. We will discuss such familiar components only in view of the changes we made to them to accommodate monitors and the guarantees we aim to provide.

The aspects of interest in the language are those related to memory sharing between threads. Aria offers two constructs for this goal, immutable values and monitors, both allowing safe sharing. Furthermore, the language provides a native `spawn` statement for creating new threads, and an unique type to represent condition variables used by monitors for synchronization. The next subsections discuss these constructs.

3.1.1
Immutable Values

Type systems and immutability are often used by programming languages to enforce safety and avoid race conditions (14, 15). Aria adopts these concepts to allow for monitors to deal with references. Immutability in Aria is built upon immutable types and immutable variables.

Essentially, a value with an immutable type cannot be altered. Primitive types are inherently immutable. An aggregate type can be declared immutable by adding the `Immutable` qualifier to its declaration. For instance, `Immutable [Integer]` declares an immutable array of integers. The `Immutable` qualifier ensures two things: First, members in the aggregate cannot be assigned; second, members in the aggregate must be themselves immutable. For example, `Immutable [Immutable [Integer]]` is a valid type, but `Immutable`

`[[Integer]]` is not (because it is an immutable array of mutable arrays of integers).

Together with monitor types (to be explained next), immutable types are called *safe types*. These objects can be shared among concurrent threads with no possibility of race conditions.

Moreover, in Aria, variables can be declared either as `value` or `variable`. A `value` must be defined at the same time it is declared and cannot be reassigned. A `variable` can be declared without an initial value and can be reassigned throughout the program.

Global variables are naturally shared among threads, both by direct access by the thread code and through global functions that the threads share. Therefore, to ensure safety, global variables must be immutable (values) and must have safe types.

Listing 3.1 illustrates the usage of the `Immutable` type qualifier and the `value` and `variable` keywords.

```
1  // globals must be safe values; the Integer type
2  // is a primitive (inherently immutable) type.
3  value x: Integer = 10;
4
5  function main {
6    // 'a' contains a (mutable) array of strings; 'a'
7    // cannot be reassigned, because it was declared
8    // as a 'value'.
9    value a: [String] = ["hello", "there"];
10
11   // however, the contents of 'a' can be reassigned.
12   a[1] = "world";
13
14   // 'b' has type 'Immutable [Integer]', which does
15   // not need to be explicitly stated because the
16   // language infers it.
17   variable b = Immutable [1, 2, 3];
18
19   // variable 'b' contains an immutable array, but
20   // 'b' itself can be reassigned, because it was
21   // declared as a variable.
22   b = Immutable [3, 1, 4, 1, 5];
23
24   // these do not compile:
25   // a = ["not", "valid", "!"];
26   // b[0] = 1;
27 }
```

Listing 3.1: A basic overview of Aria

3.1.2 Monitors

A monitor definition is a template for the creation of monitor instances. Each monitor definition has a name and a block containing declarations and/or definitions of variables and functions. We instantiate a monitor by calling its constructor—a special function defined within the monitor with the `initializer` keyword. A function inside a monitor can be declared as `private`, in which case it cannot be called from outside the monitor.

A monitor ensures the following key properties:

1. All monitor objects have their functions accessed in mutual exclusion.
2. All variables declared or defined within a monitor are private to that monitor's code, so that it is impossible to access them from outside the monitor.
3. All values received or returned by monitor non-private functions must have a safe type.
4. Variables with the `ConditionPool` type used to represent condition variables can only be declared inside monitors.
5. Only monitor functions are allowed to use the synchronization statements `signal`, `broadcast` and `wait-for-in` that operate over condition variables.
6. A monitor function cannot create new threads.

Properties 1 and 2 came straight from the original definition of monitors. Variables inside monitors store the shared data we are trying to protect from race conditions. Hence, they must be accessed through functions that ensure mutual exclusion.

Property 3 deals with the "reference problem". If data shared with and by a monitor is always immutable, then references consumed and produced by it are always harmless for concurrency purposes. This property is similar to the rule about global variables.

Properties 4 and 5 will be explained in the next subsection. The last property exists only to avoid over-complicating monitors.

3.1.3 Condition Pools

Monitors can wait on special conditions specified by the programmer. For this goal, Aria has condition variables, like so many other monitor implementations. A condition variable is a pool of blocked threads waiting on a certain condition. In Aria, a condition variable is called a `ConditionPool`.

Threads inside condition pools can be awoken using either the `signal` or `broadcast` statements. A `signal` awakes one thread from the pool, while a `broadcast` wakes all of them. However, unlike traditional conditional variables, threads cannot be put inside a condition pool without an associated condition being given. Aria combines the condition and the `wait` in a single `wait-for-in` statement, as illustrated in Listing 3.2. The `wait-for-in` statement is built using a simple loop enclosing a call to `wait`. The loop guarantees the condition is always valid after the wait, even in face of spurious wakeups (7).

```
1 wait for numElems > 0 in emptyPool;
```

Listing 3.2: The wait-for-in statement

As a traditional `wait`, the `wait-for-in` statement releases the mutual exclusion during the wait. Thus, the programmer should pay attention to the monitor's state when using more than one `wait-for-in` statement inside a function. After a second wait, the condition associated to the first one may not be valid anymore.

3.1.4 Creating new threads

To create a new thread, Aria uses a `spawn` statement. This statement specifies a block of code to run in the new thread. This block of code can access not only global variables, but also variables declared in the blocks enclosing the `spawn` statement. Again, to ensure the absence of race conditions, the variables that a `spawn` accesses must be safe, that is, they must be values with a safe type (either an immutable type or a monitor).

3.2 Using Monitors in Aria

In this section, we provide solutions for the classic bounded-buffer and readers-writers problems. We solve the bounded-buffer problem using a straightforward concurrent stack and the readers-writers problem with two different approaches in which regards to safety and simplicity. Finally, we

discuss Aria's strengths and shortcomings when dealing with different kinds of concurrency problems.

3.2.1 Stack

```
1  monitor Stack {
2    variable top = 0;
3    variable capacity: Integer;
4    variable items: [Integer];
5
6    value fullPool = ConditionPool();
7    value emptyPool = ConditionPool();
8
9    initializer(n: Integer) {
10     capacity = n;
11     items = [Integer](n);
12   }
13
14   private function full: Boolean {
15     return top == capacity;
16   }
17
18   private function empty: Boolean {
19     return top == 0;
20   }
21
22   function push(number: Integer) {
23     wait for not full() in fullPool;
24     items[top] = number;
25     top += 1;
26     signal emptyPool;
27   }
28
29   function pop: Integer {
30     wait for not empty() in emptyPool;
31     top -= 1;
32     signal fullPool;
33     return items[top];
34   }
35 }
```

Listing 3.3: Concurrent stack

A thread-safe stack can be implemented in Aria by a monitor with an internal buffer, two condition pools and push/pop operations. The internal buffer is an array of integers, with a size defined by the constructor. The condition pools keep threads that are waiting for the not-empty and not-full conditions. The push and pop operations are conventional, except for the `wait-for-in` and `signal` statements.

Listing 3.3 illustrates the basic facilities of Aria monitors: private functions (`full`, `empty`), condition pools (`fullPool`, `emptyPool`), plus the

`wait-for-in` and `signal` statements. It is worth noting how the wait statements read quite like preconditions to the corresponding functions.

Signaling in Aria is explicit. Moreover, Aria implements the *Signal and Continue* policy (8). In this policy, a thread continues to execute normally after signaling, not releasing the monitor to the signaled thread. The signaled thread(s) is moved from the condition pool to the monitor's mutual-exclusion queue, possibly competing with other threads to enter the monitor after the signaling thread exits. Without this policy, the function `pop` in the stack example would be incorrect.

Listing 3.4 illustrates the use of the `Stack` monitor. Line 2 creates the monitor by calling its initializer. Lines 3 and 8 create two new threads. Lines 5 and 10 contain calls to `stack.push` and `stack.pop`. The programmer must be aware that these calls may block if other threads are accessing the monitor or if the `wait-for-in` conditions are not met. Apart from that, the use of the concurrent safe stack is similar to that of a plain stack data structure.

```
1  function main {
2    value stack = Stack(10);
3    spawn {
4      while true {
5        stack.push(randomInteger());
6      }
7    }
8    spawn {
9      while true {
10       print(stack.pop());
11     }
12   }
13 }
```

Listing 3.4: Using the concurrent stack

For this kind of problem, Aria's monitors provide a simple and direct approach for programmers. Basically, since the problem can be modeled as a single monitor, no other elaborate abstractions are necessary. However, this is only possible because the stack's operations are always mutually exclusive. In some scenarios, we may want to synchronize access without necessarily imposing mutual exclusion. Clearly, we cannot solve these scenarios with a single monitor. This kind of situation is covered in the next example, the readers-writer lock.

3.2.2 RWLock

The basic readers-writers problem refers to the situation in which two kinds of threads need to access a shared resource. *Readers* can read (access the shared resource) simultaneously with other readers, as long as no one is writing. *Writers* must write in mutual exclusion with any other thread, either reader or writer. The classic solution to this problem (4) is built upon four basic functions: `startReading`, `stopReading`, `startWriting`, and `stopWriting`. A `RWLock` monitor provides this set of functions to coordinate actions of readers and writers.

Listing 3.5 shows the `RWLock` monitor implemented in Aria. This monitor uses two internal variables to account for readers/writers state: variable `readers` holds the number of current readers and variable `writing` indicates whether any thread is currently writing. The condition pools `readersPool` and `writersPool` maintain respectively waiting readers and writers. The `startReading` and `startWriting` functions wait on their respective conditions and change the monitor's inner state. Functions `stopReading` and `stopWriting` undo whatever their counterparts did and signal on one or both condition pools when appropriate. This implementation gives priority to readers and can lead to writers starvation. A version giving priority to writers can easily be implemented by creating a counter `waitingWriters` for waiting writers and adding the condition `(waitingWriters == 0)` to `startReading`'s `wait-for-in` statement, in line 15.

In this solution for the readers-writers problem, the monitor acts purely as a synchronization construct. Ideally, we would like a single object to encapsulate functions `read` and `write` with appropriate synchronization, but we cannot put these functions inside a monitor because we do not want readers to be mutually exclusive. This limitation of the monitor construct requires the programmer to explicitly manage two separate objects—the synchronization monitor and the shared resource—at each read and write request. If the synchronization functions are called in the wrong order, or not called at all, we lose the reader/writer policy over the shared resource.

To show how Aria allows us to compose monitors in order to build higher-level synchronization policies, we will present another implementation for the readers-writers problem that strongly couples a RW lock with a shared resource.

3.2.3 ReadersWriters

In this solution for the readers-writers problem, we will use a thread-safe integer array as the resource to be safely shared among readers and writers. Although each individual operation on such an array is already thread-safe, we can interpret a writer as a thread that needs to make a set of changes atomically. We want to make sure that a thread can only access the array after the `startReading`/`startWriting` functions are called, and that the thread won't be able to access it after it calls `stopReading`/`stopWriting`. To guarantee this, the `ReadersWriters` monitor will handle access permissions over the integer array, and the integer array will require valid permissions to be accessed.

```
1  monitor RWLock {
2    variable readers = 0;
3    variable writing = false;
4
5    value readersPool = ConditionPool();
6    value writersPool = ConditionPool();
7
8    initializer { /* empty */ }
9
10   private function canWrite: Boolean {
11     return not writing and readers == 0;
12   }
13
14   function startReading {
15     wait for not writing in readersPool;
16     readers += 1;
17   }
18
19   function stopReading {
20     readers -= 1;
21     if readers == 0 {
22       signal writersPool;
23     }
24   }
25
26   function startWriting {
27     wait for canWrite() in writersPool;
28     writing = true;
29   }
30
31   function stopWriting {
32     writing = false;
33     signal writersPool;
34     broadcast readersPool;
35   }
36 }
```

Listing 3.5: RWLock

Our code will represent permissions by *tokens*, which are unique objects. The semantics of Aria dictates that each new created object is different from all other objects, so a thread cannot forge a token.[1]

The `Resource` monitor in Listing 3.6 implements our thread-safe contents. This monitor has two arrays, declared in lines 5 and 6, that hold the tokens authorized for reading and writing. (Because only one writer can write at a time, we could have provided a single boolean variable to hold the writing token. However, this piece of synchronization logic belongs to the `ReadersWriters` monitor and, therefore, the `Resource` monitor should be neutral to this logic.)

The `get` and `set` operations are always called with a token as their last arguments. If the provided token is not authorized, the call fails. To ensure that only the `ReadersWriters` monitor can manage the authorized tokens, our code uses a master token. This master token is given to the `Resource` monitor upon its creation (see line 11), and it is required in all calls to functions `allow` and `disallow`, which change the lists of valid tokens (see lines 22 and 33).

Listing 3.7 shows the `ReadersWriters` monitor, which encapsulates the shared resource. Function `newToken`, in line 4, creates a token as an unique value. The monitor initializer, in line 11, creates the shared resource, defining the master token. From then on, a call to `getResource` returns the integer array, but the `start` and `stop` functions must also be called by any thread wishing to access this array. The `start` functions provide valid tokens without which access to the shared resource is not possible.

The function `spawnWriter` (Listing 3.8) illustrates how the `Resource` and `ReadersWriters` monitors are used together.

The `start` and `stop` functions lock the `Resource` monitor when managing token permissions. Because the `get` function operates over a single element of the array, access to the `Resource` monitor is highly disputed when reading. Therefore, the `startReading` and `stopReading` functions can be delayed while trying to call the `allow`/`disallow` functions. This delay can be minimized by making the `get` function return more than one element of the array at a time.

This example has shown how monitors under Aria's language restrictions can be composed to create synchronization at different granularities. However, this implementation is not completely safe. Tokens could be shared between threads, since they are immutable, leading to the same problems that the `RWLock` presented. Although this requires a more active role of the programmer in bypassing safety restrictions, it is still not in line with the guarantees we

[1]The implementation uses an array of one boolean for tokens (see line 1 in Listing 3.6); they are the smallest objects that can be created in Aria.

desire to provide. Besides, the code required for a relatively simple task is rather complex. In Chapter 4 we address the safety, performance, and complexity issues of this problem again.

3.2.4 Analysis

In the previous examples, we provided three implementations for classic concurrency problems. Analyzing these and other examples (9), we arrived at some usage patterns, which we discuss here.

The `Stack` monitor falls into the *Data Structure (DS)* category. Basically, any concurrency problem that requires mutual exclusion in the access to a single data structure works well with Aria's monitors. Monitor operations fit perfectly as concurrency-safe wrappers for data structures. This category includes countless variations of the producer-consumer problems (10) (from which our stack example derives).

The `RWLock` represents a *Synchronization Mechanism (SM)*. This category does not provide functions that externalize a shared resource, providing instead operations that can be used to coordinate actions over shared resources such as DS monitors or system files. Lower-level concurrency mechanisms, such as semaphores and barriers, can also be implemented as SM monitors.

The `ReadersWriters` monitor is a complete implementation for the readers-writers problem. However, it is also a more complex engineering solution. We call such monitors *Hybrid Solutions (HS)*, because they are a problem-oriented amalgamation of DS and SM monitors. As pointed out in subsections 3.2.2 and 3.2.3, hybrid monitors guarantee correct access to shared resources, instead of leaving up to programmers the coordination of calls to DS and SM monitors. However, HS monitors are not straightforward to develop, and they may introduce permission complexity in the DS monitor code.

This is only a loose categorization. Not all problems fall into these categories perfectly, and sometimes a problem can be implemented within more than one category (as seen with the readers-writers problem). Nevertheless, this classification is useful for reflecting upon different types of concurrency problems and monitor properties.

After looking at concurrency problems using this classification, we gained some insights regarding the drawbacks of designing a language with monitors strictly as they were originally proposed. The next chapter explores the addition of new constructs to the monitor concept inspired by the shortcomings of the model discussed so far.

```
alias Token = Immutable [Boolean];

monitor Resource {
  variable master: Token;
  variable readingTokens = [Token](100);
  variable writingTokens = [Token](100);

  variable items: [Integer];
  variable size: Integer;

  initializer(token: Token, n: Integer) {
    master = token;
    items = [Integer](n);
    size = n;
    for i in [0 -> n] {
      items[i] = 0;
    }
  }

  function getSize: Integer { return size; }

  function allow(isWrite: Boolean, mTk, tk: Token) {
    if master != mTk {
      error("invalid master token");
    }
    if isWrite {
      writingTokens.append(tk);
    } else {
      readingTokens.append(tk);
    }
  }

  function disallow(isWrite: Boolean, mTk, tk: Token) {
    <similar to "allow", but uses "remove" instead of "append">
  }

  function get(index: Integer, token: Token): Integer {
    if not readingTokens.contains(token) {
      error("invalid reading token");
    }
    return items[index];
  }

  function set(index, item: Integer, token: Token) {
    if not writingTokens.contains(token) {
      error("invalid writing token");
    }
    items[index] = item;
  }
}
```

Listing 3.6: Resource

```
monitor ReadersWriters {
  <variables and values from RWLock>

  private function newToken: Token {
    return Token(1);
  }

  value master: Token = newToken();
  variable resource: Resource;

  initializer(arraySize: Integer) {
    resource = Resource(master, arraySize);
  }

  function getResource: Resource {
    return resource;
  }

  function startReading: Token {
    <code from RWLock startReading>
    value token = newToken();
    resource.allow(false, master, token);
    return token
  }

  function stopReading(token: Token) {
    resource.disallow(false, master, token);
    <code from RWLock stopReading>
  }

  function startWriting: Token {
    <code from RWLock startWriting>
    value token = newToken();
    resource.allow(true, master, token);
    return token;
  }

  function stopWriting(token: Token) {
    resource.disallow(true, master, token);
    <code from RWLock stopWriting>
  }
}
```

Listing 3.7: ReadersWriters

```
1  function spawnWriter(rw: ReadersWriters) {
2    value array = rw.getResource();
3    value size = array.getSize();
4    spawn {
5      value token = rw.startWriting();
6      for i in [0 -> size] {
7        value number = i * 5;
8        array.set(i, number, token);
9      }
10     rw.stopWriting(token);
11   }
12 }
```

Listing 3.8: A function that spawns a new writer

4
Extending Monitors

In this chapter, we present additions that were made to Aria in order to deal with issues from the original monitor proposal regarding performance and expressiveness. These extensions do not break any of the previously stated rules from monitors, and their usage does not alter the core features presented so far. The issues are:

Issue 1 Constantly acquiring and releasing locks from monitors dramatically hinders performance for a lot of basic use cases. For example, consider a monitor encapsulating an array of integers, with basic `size`, `get` and `set` functions. Iterating over such an array will be slow, because the monitor will be locked and unlocked with each call to `get`/`set`.

Issue 2 HS monitors are complex in nature. Furthermore, they force high coupling and high cohesion solutions. We would like the language to allow for low (or lower) coupling and high cohesion solutions. This issue encompasses the composability problems mentioned in subsection 3.2.3 when using monitors to synchronize access to other monitors.

We added two language features to deal with these issues: unlocked monitors and scoped values.

4.1
Unlocked Monitors

In Aria, every monitor has a mutual exclusion lock that synchronizes access to its methods. Usually, the lock is acquired then released each time a method is called, which causes the problem described in Issue 1. An unlocked monitor, however, does not lock and unlock when its methods are called.

Aria distinguishes regular from unlocked monitors through the `Unlocked` type qualifier. Listing 4.1 illustrates how unlocked monitors can be used to solve precisely the problem of Issue 1. In the example, the `Array` monitor encapsulates an array of integers with basic `size`, `get` and `set` methods. The `printArray` function receives an `Unlocked` monitor and prints its elements.

Since the call in line 4 won't demand a lock, the function will be able to iterate through the array without the overhead of constant locking.

```
1 function printArray(array: Unlocked Array) {
2   value size = array.size();
3   for i in [0 -> size] {
4     print(array.get(i));
5   }
6 }
```

Listing 4.1: Using unlocked monitors

As we will explain next, unlocked monitors do not break Property 1 from monitors regarding mutual exclusion. However, unlike regular monitors, unlocked monitors are not *safe*. Hence, the language statically enforces they cannot be shared between threads.

4.2 Scoped values

Unlocked monitors are created using the `acquire-value` statement, that simultaneously defines an unlocked monitor and links it to a block of code. Unlocked monitors are valid only while within the block of code with which they were created. Attempts to use unlocked monitors outside their designated blocks cause an error. For this reason, we also refer to unlocked monitors as *scoped values*.

An `acquire-value` statement (explicitly) calls an `acquire` method and (implicitly) calls a `release` method. Two monitor methods form an acquire-release pair of methods if they obey the following rules:

1. An acquire-release pair of methods must always be declared together using the same name and the `acquire`/`release` keywords. One cannot declare an `acquire` function without declaring its `release` counterpart, and vice-versa, as seen in Listing 4.2 (lines 5 and 6).

2. An `acquire` method must always return a monitor value.

3. An `acquire` method can only be called through the `acquire-value` statement (line 10).

4. A `release` method cannot be called directly by the programmer, instead being implicitly invoked at the end of the `acquire-value` statement

that called its `acquire` counterpart[1]. In the example from Listing 4.2, `release-foo` is called in line 12.

```
1  monitor N { <...> }
2
3  monitor M {
4    <other variables and functions from monitor M>
5    function acquire foo: N { <...> }
6    function release foo    { <...> }
7  }
8
9  function main {
10   acquire value n: Unlocked N = m.foo() {
11     <...>
12   }
13 }
```

Listing 4.2: Using scoped values

Note that, despite `acquire-foo` returning a monitor of type `N` (line 5), the scoped value `n` has type `Unlocked N` (line 10). The `acquire-value` statement converts the returned value into an unlocked monitor. It locks the monitor at the end of line 5, and unlocks it in line 12 before `release-foo` is called.

Listing 4.3 exemplifies the usage of an invalid scoped value. As previously stated, scoped values are only valid while inside their designated block. Attempts to escape the scoped value from its block (line 4) are not allowed by the language. Therefore, using an escaped value (line 6) causes a runtime error.

```
1 variable cheater: Unlocked N;
2 acquire value n = m.foo() {
3   n.doSomething();
4   cheater = n;
5 }
6 cheater.doSomething(); // error
```

Listing 4.3: Invalid usage of scoped values

Monitors in Aria natively posses a predefined acquire-release pair of methods called `unlocked`, as illustrated in Listing 4.4. These methods serve as a way to acquire the unlocked version of a monitor and nothing else. Despite its name and function, the `acquire-unlocked` method called in line 2 returns

[1]As a special rule, a syntax block delimited by an `acquire-value` statement may not contain a return statement. This rule avoids complicating the implementation of the `acquire-value` statement when ensuring a `release` function will always be called after its `acquire`.

the monitor `m` of type `Array`, not `Unlocked Array`. As previously discussed, it is the `acquire-value` statement that converts `m` to `Unlocked` when defining the `array` value.

```
1 value m = Array(10);
2 acquire value array: Unlocked Array = m.unlocked() {
3   printArray(array);
4 }
```

Listing 4.4: The `unlocked` acquire-release pair of methods

Finally, the ability to unlock monitors is a powerful feature, as it allows a function from outside the monitor to extend its functionalities. It also provides a way to freely acquire a monitor's mutual exclusion, without necessarily doing anything with it. Therefore, this feature must be used with caution. Ideally, the least necessary amount of code will be executed inside a monitor's critical region, and that must be kept in mind when unlocking a monitor.

4.3 Examples

In this section, we provide a safer and cleaner implementation for the readers-writers problem using the concept of scoped permissions. Lastly, we implement a parallel hash table data structure to demonstrate the flexibility of the new features.

4.3.1 Scoped Permissions

Scoped permissions are used to model permissions valid only while within the block of code for which they were created. We can create scoped permissions by combining scoped values and interfaces. This concept allows us to naturally model permissions in Aria, instead of using tokens as in subsection 3.2.3.

An interface in Aria defines a set of functions. Similar to interfaces from object-oriented languages, a monitor must implement these functions if it wishes to comply with the interface. Listing 4.5 illustrates the declaration of interfaces (lines 1 and 5). As seen in line 9, a monitor must explicitly state its compliance with an interface.

The `RW` monitor revisits the previously discussed readers-writers problem and implements the *read* and *write* permissions using acquire-release pairs of functions. The `Resource` monitor simply encapsulates an array, providing basic `get` and `set` operations while being free of any kind of permission logic. The resource held by the `RW` monitor is cast as a `ReadResource` or `WriteResource` before being returned (lines 21 and 30), ensuring only the right operations are

visible to the caller. This is in line with the safety guarantees that the original monitor proposals seek to provide.

This solution for the readers-writer problem ensures safety and is also cleaner when compared to the solutions provided in subsections 3.2.2 and 3.2.3. Cleaner because permission logic is more elegantly expressed, and safe because the language ensures the correct operations are going to be accessible strictly during the period for which the correspondent permissions are held. In summary, the new feature allows for low coupling and high cohesion constructions without sacrificing safety, addressing the problem described in Issue 2 regarding composability.

4.3.2 Hash Tables

In this subsection, we present a scalable parallel implementation of hash tables using Aria's monitors. For a basic hash table, we are going to use the `HashTable` and `Bucket` monitors shown in Listing 4.6 and Listing 4.7, respectively. The `HashTable` monitor has an array of buckets, and each bucket is associated with a hashed value. The `Bucket` monitor has a list of key-value pairs. In order to get or set the value associated with a key, one must first get the associated bucket as a scoped value. The hash table must be accessed through the `get` and `set` functions in lines 43 and 51, respectively.

This implementation is simple in that the hash table is just an array of monitors. Furthermore, functions from `HashTable` and `Bucket` are intentionally not executed under the same lock. The operations in the `bucket` acquire-release pair of function are elementary, therefore, the `HashTable`'s lock is held briefly (unless rehashing is taking place). High parallelism can be achieved because many threads can be working with different buckets simultaneously.

A "stop-the-world" rehashing function can be easily implemented using condition queues and `wait-for-in/signal/broadcast` statements. Since buckets are scoped values, we can guarantee no previously acquired bucket will be used after rehashing has started.

The fact that each bucket is a monitor leads to too many mutexes being created. An alternative two-layer approach could be devised by having one intermediate monitor supervise `N` buckets. For example, if the hash table has 1000 buckets and `N` is 10, then 100 intermediate monitors would be created.

4.3.3 Analysis

The `RW` and `HashTable` monitors are HS monitors, as defined in subsection 3.2.4. Both monitors coordinate access to a DS monitor (`resource` and `bucket`), imposing some kind of access logic (controlling, respectively, permissioning and rehashing). With scoped values, it becomes impossible to use the controlled resource in an unsafe manner, and the code for providing this access logic becomes simpler.

The combination of scoped values and unlocked monitors allows Aria to address issues of performance and expressiveness that are sometimes ascribed to the monitor concurrency model. These mechanisms allow the programmer to compose operations on monitors, controlling the scope of locking and creating abstractions that maintain the compile-time guarantees against data races.

Scoped values are useful in cases where the programmer needs the language to provide an extra level of guarantees. HS monitors, because of their nature of mixing synchronization and data access, are always going to need extra guarantees the monitor concept alone does not provide. Hence, scoped values are useful to cover common use cases monitors didn't previously address or addressed poorly.

```
1  interface ReadResource {
2    function get(index: Integer): Integer;
3  }
4
5  interface WriteResource {
6    function set(index, item: Integer);
7  }
8
9  monitor Resource: ReadResource, WriteResource { <...> }
10
11 monitor RW {
12   <variables from RWLock>
13
14   variable resource: Resource;
15   initializer(r: Resource) {
16     resource = r;
17   }
18
19   function acquire read: ReadResource {
20     <code from RWLock startReading>
21     return resource as ReadResource;
22   }
23
24   function release read {
25     <code from RWLock stopReading>
26   }
27
28   function acquire write: WriteResource {
29     <code from RWLock startWriting>
30     return resource as WriteResource;
31   }
32
33   function release write {
34     <code from RWLock stopWriting>
35   }
36 }
37
38 function usage(rw: RW) {
39   acquire value resource = rw.write() {
40     resource.set(0, 10);
41   }
42 }
```

Listing 4.5: Revisiting the readers-writer problem

```
alias K = String;
alias V = Integer;

monitor HashTable {
  value n = 100;
  variable buckets = [Bucket](n);

  value rehashingPool = ConditionPool();
  value bucketsPool = ConditionPool();
  variable counter = 0; // acquired buckets
  variable rehashing = false;

  initializer {
    for i in [0 -> n] {
      buckets[i] = Bucket();
    }
  }

  function acquire bucket(key: K): Bucket {
    wait for not rehashing in rehashingPool;
    counter += 1;
    return buckets[hash(key)];
  }

  function release bucket {
    counter -= 1;
    if rehashing and counter == 0 {
      signal bucketsPool;
    }
  }

  function rehash {
    rehashing = true;
    wait for counter == 0 in bucketsPool;
    <rehash logic>
    rehashing = false;
    broadcast rehashingPool;
  }
}

// usage functions

function get(ht: HashTable, k: K): V {
  variable v: V;
  acquire value bucket = ht.bucket(k) {
    v = bucket.get(k);
  }
  return v;
}

function set(ht: HashTable, k: K, v: V) {
  acquire value bucket = ht.bucket(k) {
    bucket.set(k, v);
  }
}
```

Listing 4.6: HashTable

```
record KV {
  value k: K;
  variable v: V;
  variable next: KV;
}

monitor Bucket {
  variable list: KV;

  initializer { /* empty */ }

  function get(k: K): V {
    variable node = list;
    while node != nil {
      if k == node.k {
        return node.v;
      }
      node = node.next;
    }
    return nil;
  }

  function set(k: K, v: V) {
    if list == nil {
      list = KV(k, v, nil);
      return;
    }

    variable previous: KV;
    variable node = list;
    while node != nil {
      if k == node.k {
        node.v = v;
        return;
      }
      previous = node;
      node = node.next;
    }
    previous.next = KV(k, v, nil);
  }
}
```

Listing 4.7: Bucket

5
Compiler Implementation

In this chapter, we discuss the implementation of both regular and unlocked monitors. We will not focus on Aria's type system and other semantic rules because they are trivially checked and enforced during compilation.

A compiler for Aria is available at `github.com/renan061/aria`, version `0-2`. The compiler uses standard compiler techniques and tools, such as Lex, Yacc, and LLVM.

We use *pthreads* to implement the language's concurrency features, but any other API or library that provides thread creation, mutexes, and condition variables could easily be used instead.

5.1
Implementing regular monitors

The Aria compiler declares a monitor structure in LLVM for each monitor definition in Aria. A monitor structure contains a mutex and a virtual method table (vtable), plus all monitor variables defined by the programmer.

```
1 monitor Counter {
2   variable a: Integer;
3   initializer(n: Integer) { a  = n; }
4   function increment      { a += 1; }
5 }
```

```
1 %Counter = type { i8*, [3 x i8*]*, i32 }
2
3 @Counter-L-vtable = global [3 x i8*] undef
4
5 define i8*  @Counter-initializer(i32) { <...> }
6 define void @Counter-vtable()         { <...> }
```

Listing 5.1: LLVM representation of a monitor

Listing 5.1 shows the `Counter` monitor in Aria and the `%Counter` structure in LLVM. The first field of the structure is a reference to the monitor's mutex. This field has type `i8*`, which is a general pointer type in LLVM. The second field is a reference to the global variable `@Counter-L-vtable` (line 3), a vtable with three pointers `[3 x i8*]` that contains the natively provided `acquire-unlocked` and `release-unlocked` functions, and the user provided `increment` function. `@Counter-L-vtable` is allocated and initialized once by

the `@Counter-vtable` function (line 6) invoked at the beginning of program execution. The last field in the structure represents the variable `a`; the `i32` type represents a standard integer.

The global function `@Counter-initializer` (line 5) implements the monitor's constructor. It uses `pthread_mutex_init` and `@Counter-L-vtable` to create new `%Counter` objects.

The Aria compiler also creates two functions in LLVM for each non-private method in a monitor, the `R` (regular) and `L` (locking) versions of the method.

`R`-functions are straightforward translations from methods in Aria to their LLVM representations. They are invoked by their associated `L`-functions. Listing 5.2 shows the function `@increment-R`, compiled from the `increment` method from the `Counter` monitor. We reify methods as functions with an additional first parameter representing the *self* reference, as conventional. (Also, in this and in all LLVM code examples where applicable, we will write the LLVM instruction `getelementptr` as `gep`, in order to avoid breaking lines and improve code readability.)

```
1  define void @increment-R(i8*) {
2  entry:
3    ; casts the `self` parameter to the proper type
4    %self = bitcast i8* %0 to %Counter*
5    ; gets the address of `self.a`
6    %aptr = gep %Counter, %Counter* %self, i32 0, i32 2
7    ; loads the contents of `self.a`
8    %a1 = load i32, i32* %aptr
9    ; increments the value of `self.a`
10   %a2 = add i32 %a1, 1
11   ; stores the incremented value back in memory
12   store i32 %a2, i32* %aptr
13   ret void
14 }
```

Listing 5.2: Regular version of function `increment`

Calls to methods from regular (not unlocked) monitors are translated to invocations of `L`-functions. As we can see in Listing 5.3, this type of function loads the monitor's mutex from the *self* reference (line 5), locks the mutex (line 6), invokes the associated `R`-function (line 7), unlocks the mutex (line 8), and returns the value originally returned by the `R`-function, if any (line 9). Note that lines 6 and 8 enforce that the `R`-function is invoked while the mutual exclusion lock is being held.

```
1  define void @increment-L(i8*) {
2  entry:
3    %self = bitcast i8* %0 to %Counter*
4    %mtxptr = gep %Counter, %Counter* %self, i32 0, i32 0
5    %mtx = load i8*, i8** %mtxptr
6    %t1 = call i32 @pthread_mutex_lock(i8* %mtx)
7    call void @increment-R(i8* %0)
8    %t2 = call i32 @pthread_mutex_unlock(i8* %mtx)
9    ret void
10 }
```

Listing 5.3: Locking version of function `increment`

To wrap up this section, Listing 5.4 illustrates a call to the `increment` method, provided by `Counter`, from the main function. First, we initialize the monitor's vtable (line 4) and call a constructor to create the monitor object (lines 6 and 7). We then retrieve the reference to `increment` from the vtable (lines 9, 10, and 12) and load the function while casting it to the correct type (lines 13 and 14). Finally, the function is called (line 16).

```
1  define void @main() {
2  entry:
3    ; allocates and initializes Counter's vtables
4    call void @Counter-vtable()
5    ; calls the constructor
6    %c1 = call i8* @Counter-initializer(i32 5)
7    %c2 = bitcast i8* %c1 to %Counter*
8    ; retrieves Counter's L-vtable
9    %vtableptr = gep %Counter, %Counter* %c2, i32 0, i32 1
10   %vtable = load [3 x i8*]*, [3 x i8*]** %vtableptr
11   ; retrieves the `increment` function
12   %fptr = gep [3 x i8*], [3 x i8*]* %vtable, i32 0, i32 2
13   %f1 = load i8*, i8** %fptr
14   %f2 = bitcast i8* %f1 to void (i8*)*
15   ; calls the function
16   call void %f2(i8* %c1)
17   ret void
18 }
```

Listing 5.4: Calling the `increment` method

5.2 Implementing unlocked monitors

Unlocked monitors are scoped values acquired through the `acquire-value` statement. Methods from unlocked monitors do not lock or unlock the mutex. Also, unlocked monitors are not safe values and cannot be shared among threads.

The implementation of unlocked monitors must enforce that `acquire-value` statements properly lock and unlock monitors, and that scoped values won't escape their scope. This last property cannot be verified

at compile time, as scoped values can be assigned to other memory locations and become untraceable in our current type system. An extended type system that detects such cases could be developed, at the cost of increasing code complexity to the programmer, which we want to avoid. For this reason, we implement runtime scoped value validation. We address the cost of runtime checking and possible optimizations later in this section.

We will use *proxy* structures to represent unlocked monitors. Each proxy structure wraps a monitor structure. A proxy object is valid while the program is inside its `acquire-value` statement, and invalid otherwise. Proxies invoke special functions that do not lock or unlock the monitor. Instead, they check whether the proxy is valid, that is, the wrapped monitor is locked. If so, the associated `R`-function is invoked; otherwise, the program raises a runtime error. We will explain these constructions in detail next.

```
1  %Counter-P = type { i8*, [3 x i8*]*, i1  }
2
3  @Counter-P-vtable = global [3 x i8*] undef
```

Listing 5.5: Implementation of the proxy structure

Listing 5.5 shows the proxy structure `%Counter-P` for the `Counter` monitor in LLVM. The first field of the structure is a reference to the associated monitor object. The second field is a reference to the global variable `@Counter-P-vtable` (line 3), a vtable that contains the `P` (proxy) versions of the monitor's methods. `@Counter-P-vtable` is allocated and initialized once, alongside `@Counter-L-vtable`, at the beginning of program execution. The last field in the structure represents a boolean variable `ok` (with type `i1`) that indicates whether the proxy is valid.

```
1  define void @increment-P(i8*) {
2  entry:
3    %proxy = bitcast i8* %0 to %Counter-P*
4    %okptr = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 2
5    %ok = load i1, i1* %okptr
6    %bool = icmp eq i1 %ok, true
7    br i1 %bool, label %if-ok, label %if-error
8  if-ok:
9    %selfp = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 0
10   %self = load i8*, i8** %selfp
11   call void @increment-R(i8* %self)
12   ret void
13 if-error:
14   call void @raise-error()
15   unreachable
16 }
```

Listing 5.6: Proxy version of function `increment`

Calls to methods are translated into invocations of `P`-functions, which do not lock or unlock the mutex. A `P`-function receives as an argument a

proxy structure (`%Counter-P`), instead of a monitor structure (`%Counter`). As Listing 5.6 illustrates, this function checks whether the `ok` field of the proxy object is `true` (lines 4 to 7). If so, it calls the associated `R`-function (lines 9 to 11), otherwise, it raises an error (line 14). Thus, we guarantee the `R`-function will only be called if the proxy is valid (`ok` is `true`), else the program contains an error detected during runtime and its execution is (rightfully) interrupted.

Listing 5.7 illustrates the implementation of the `acquire-value` statement using the `Counter` monitor and its `unlocked` acquire-release methods. The implementation of the `acquire-unlocked` method returns a regular monitor, while the `acquire-value` statement locks/unlocks the monitor and manages the proxy object. The code in Listing 5.7 contains the following steps:

1. The program calls the `acquire` method, which returns a scoped value (a monitor structure).
2. The proxy is created.
   (a) The monitor structure returned by the `acquire` function is stored in the proxy.
   (b) The proxy is initialized with its `P-vtable`.
   (c) The `ok` variable from the proxy is set to `true`.
3. The scoped value's mutex is acquired.
4. Instructions from the `acquire-value` block are executed.
5. The `ok` variable from the proxy is set to `false`.
6. The scoped value's mutex is released.
7. The `release` method is called.

Remember that, despite its name, the `acquire-unlocked` method does not return an unlocked scoped value, neither do any user provided acquire-release methods. Mutual exclusion in unlocked monitors is enforced by `acquire-value` statements through steps 3 and 6. In summary, `acquire` functions produce scoped values to be unlocked by `acquire-value` statements.

Note also that the monitor providing the `acquire-unlocked` function and the monitor returned by it are the same, which is not always the case for `acquire` functions. In Listing 4.5, for example, the `RW` monitor returns a `Resource` monitor when `acquire-read` is called. Thus, the `acquire-value` statement must necessarily unlock the acquired monitor. If `acquire-unlocked`

```
value c = Counter(5);
acquire value uc = c.unlocked() { uc.increment(); }
```

```
define void @main() {
entry:
  call void @Counter-vtable()
  ; %c1 => type i8*
  ; %c2 => type %Counter*
  %c1 = call i8* @Counter-initializer(i32 5)
  %c2 = bitcast i8* %c1 to %Counter*
  ; (1) calls %c's `acquire unlocked` function
  %Lptr = gep %Counter, %Counter* %c2, i32 0, i32 1
  %Lvtable = load [3 x i8*]*, [3 x i8*]** %Lptr
  %acqptr = gep [3 x i8*], [3 x i8*]* %Lvtable, i32 0, i32 0
  %acq1 = load i8*, i8** %acqptr
  %acq2 = bitcast i8* %acq1 to i8* (i8*)*
  ; scoped value
  %sv1 = call i8* %acq2(i8* %c1)       ; type i8*
  %sv2 = bitcast i8* %sv1 to %Counter* ; type %Counter*
  ; (2) creates the proxy structure object
  %malloc = tail call i8* @malloc(<...>)
  %proxy = bitcast i8* %malloc to %Counter-P*
  ; (2a) stores the scoped value in the proxy
  %p1 = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 0
  store i8* %sv1, i8** %p1
  ; (2b) stores P-vtable in the proxy
  %p2 = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 1
  store [3 x i8*]* @Counter-P-vtable, [3 x i8*]** %p2
  ; (2c) proxy.ok = true
  %p3 = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 2
  store i1 true, i1* %p3
  ; (3) calls `pthread_mutex_lock`
  %mtxptr = gep %Counter, %Counter* %sv2, i32 0, i32 0
  %mtx = load i8*, i8** %mtxptr
  %t1 = call i32 @pthread_mutex_lock(i8* %mtx)
  ; (4) executes instructions from the acquire-value block
  %Pptr = gep %Counter-P, %Counter-P* %proxy, i32 0, i32 1
  %Pvtable = load [3 x i8*]*, [3 x i8*]** %Pptr
  %fptr = gep [3 x i8*], [3 x i8*]* %Pvtable, i32 0, i32 2
  %f1 = load i8*, i8** %fptr
  %f2 = bitcast i8* %f1 to void (i8*)*
  call void %f2(i8* %proxy)
  ; (5) proxy.ok = false
  store i1 false, i1* %p3
  ; (6) calls `pthread_mutex_unlock`
  %t2 = call i32 @pthread_mutex_unlock(i8* %mtx)
  ; (7) calls %c's `release unlocked` function
  %relptr = gep [3 x i8*], [3 x i8*]* %Lvtable, i32 0, i32 1
  %rel1 = load i8*, i8** %relptr
  %rel2 = bitcast i8* %rel1 to void (i8*)*
  call void %rel2(i8* %c1)
  ret void
}
```

Listing 5.7: Implementation of the `acquire-value` statement

also unlocked the monitor, calling `unlocked` through the `acquire-value` statement would repeatedly lock and unlock the same mutex unnecessarily: first in step 1 when calling the `acquire` function, again in steps 3 and 6, and lastly in step 7. Our implementation guarantees the mutex is only handled once. Exceptionally, `L` versions of `acquire-unlocked` and `release-unlocked` methods do not lock/unlock the mutex. The former simply returns the *self* reference and the latter does nothing.

This implementation guarantees that using scoped values after their intended lifespan always causes a runtime error. Also, when combined with `L`-functions, it enforces monitor methods will abide with the original proposal and always be called in mutual exclusion.

5.3 Optimizations & Performance

As an important optimization, the compiler can evaluate the possibility of a scoped value escaping its block. It can look at the usage of the value, checking whether it is assigned to a variable outside its scope or passed to a function as an argument. If neither of these actions occur, the scoped value cannot escape and, therefore, the compiler does not need to insert code for runtime checking.

Regarding performance, the benefits of using scoped values and unlocked monitors are clear: monitors can be accessed without constant locking. However, runtime checks for scoped values impose penalties over the originally statically checked monitor rules. We believe in most cases it will be possible for the compiler to use the previously discussed optimization and avoid inserting runtime checks.

6
Performance

In this chapter, we measure the performance of Aria programs in different scenarios in order to evaluate our compiler implementation, the overhead of using monitors, and the possible gains obtained from using unlocked monitors.

We ran our benchmarks in a computer with an Intel Core I7-6700K 4.0 GHz CPU and 16 GB of DDR4 2133MHz RAM. In total, this machine has 4 CPU cores and 8 threads.

The full code used for the benchmarks (in Aria and in C) is available at `github.com/renan061/aria`, in the `benchmarks` folder.

6.1
The cost of monitors

In the next subsections, we compare benchmark results between Aria and C for two embarrassingly parallel problems: matrix multiplication and numerical integration. It is our goal with these benchmarks to assess how taxing on performance are the data sharing restrictions imposed by the language, particularly in cases where Aria's monitors need to be constantly accessed to manipulate data.

Our solutions for both problems follow a fork-join logic where independent executions do not write to the same memory location concurrently and, therefore, cannot cause data races. In this case, C has an advantage over Aria, since it allows for data-race-free programs to be developed without the need to employ synchronization mechanisms such as locks. Hence, the C benchmarks serve as a base for comparison to measure the cost of using monitors.

In our benchmarks, we created new threads using the `pthreads` library in C and the `spawn` statement in Aria. Moreover, in order to wait for thread completion, C programs used the `pthread_join` function while Aria used a custom `Barrier` monitor.

Regarding methodology, we measured the running times of each problem 10 times, with each accounting for 100 iterations of the program. This way, we were able to analyze consistent times with little to no variance. Furthermore, we measured single core performance using non-parallel algorithms, without

Language	Threads	Time	Speedup	Efficiency
C	1	9.76	-	-
	2	4.99	1.96	98%
	4	2.94	3.32	83%
	8	2.85	3.43	43%
Aria	1	9.79	-	-
	2	5.02	1.95	98%
	4	2.96	3.30	83%
	8	2.89	3.39	42%

Table 6.1: Matrix multiplication benchmark results

creating new threads, while our multithreaded benchmarks were tested with 2, 4, and 8 threads.

6.1.1 Matrix Multiplication

In this benchmark, we computed the multiplication between a square matrix $A_{(n\times n)}$ and a column vector $B_{(n\times 1)}$, with $n = 10^4$. Our measurements did not take into account the time taken allocating memory and initializing A and B with random rational numbers.

Our implementation is based on the $O(n^3)$ algorithm that results from the definition of matrix multiplication, where we calculate $c_{ij} = \sum_{n=1}^{k} a_{ik} \times b_{kj}$ for each element of the resulting matrix $C_{(n\times 1)}$. This algorithm is not optimal; the Coppersmith–Winograd algorithm, for example, achieves $O(n^{2.376})$ complexity, however, we opted for the naive algorithm for its ease of implementation. Besides, matrix multiplication computations are heavily dependent on cache optimizations, which we also chose to ignore for simplicity.

The multithreaded code divided the resulting vector $C_{(n\times 1)}$ in sections, each assigned to a different worker thread. In the C program, the resulting vector was stored in a global array and, in Aria, since we cannot have global mutable arrays, we used an `Array` monitor. In this benchmark, using an `Array` monitor meant acquiring the monitor's mutual exclusion n times, one for each time a worker finished calculating a value of the C vector and called the `set` method.

Table 6.1 shows performance results for matrix multiplication in Aria and C. Despite C's slight superiority, the results are similar enough for the difference to be statistically irrelevant. We believe that efficiency decreases in the 8 threads scenario because the program's threads start competing with other processes in the computer for CPU time. In the future, we aim to evaluate

Language	Threads	Time	Speedup	Efficiency
C	1	12.60	-	-
	2	6.30	2.00	100%
	4	3.35	3.76	94%
	8	2.72	4.63	58%
Aria	1	13.76	-	-
	2	6.85	2.00	100%
	4	3.63	3.79	95%
	8	2.93	4.70	59%

Table 6.2: Numerical integration benchmark results

this claim by running the benchmark in a computer with more than 8 threads.

6.1.2 Numerical Integration

For this benchmark, we computed the area under the curve of a quadratic function using the middle Riemann sum. Specifically, we approximated the value of $\int_0^{16} 5x^2 - 10x + 10dx$ using 8×10^7 rectangles.

Both of our Aria and C programs followed the same logic: different worker threads were created and tasked with calculating the area of a subrange of the curve, which constitutes the *fork* part of the algorithm; then, when all workers were done, the main thread calculated the total area by *joining* the partial results. As regards to implementation, the main thread received the workers' partial results by accessing shared *struct* references in C and a monitor in Aria. We were required to a use a monitor in Aria because, unlike in C, we could not share mutable records between threads.

Table 6.2 shows the benchmark results for C and Aria. Despite Aria being approximately 8% slower than C, both languages achieve near equal levels of parallelism. In particular, the speedup is almost linear with 2 and 4 threads, and, again, efficiency decreases in the 8 threads scenario.

6.2 The benefits of unlocked monitors

In Chapter 4, we stated calling monitor methods repeatedly is cumbersome because locks need to be constantly acquired then released, and that unlocked monitors solve this issue. In this section, we developed a benchmark to analyze these claims. Additionally, we tested whether the optimization mentioned in Chapter 5 could indeed enhance performance for unlocked monitors. We hand optimized LLVM code generated by the Aria compiler, removing overhead checking code for proxy objects when the unlocked monitor was not

	Time	**Slower**
Integer array (base)	0.32	-
Regular monitor – `get`	14.50	44 times
Regular monitor – `sum`	0.32	-
Unlocked monitor – `get`	1.43	4.3 times
Unlocked monitor – `get` (optimized)	1.20	3.6 times

Table 6.3: Unlocked monitors benchmark results

used as a function argument nor assigned to a variable. This way, we were able to compare results and check if this was an optimization worth implementing.

The problem we chose to benchmark computed the sum of all elements in an integer array of 10^8 elements, with each measurement accounting for 100 iterations of the program. We developed five different solutions for this problem so we could better analyze the results of using unlocked monitors.

The first solution used a regular `[Integer]` array. It served as a base for comparison with other solutions (as it did not employ any synchronization mechanisms). Next, we used an array encapsulated by a monitor and iterated over it through a `get` method. We expected this solution to be the slowest, because the monitor's mutual exclusion would be acquired then released a number of times equal to the array's size.

In our third solution, we implemented a `sum` method inside the encapsulating monitor, expecting this approach to be as fast as the base case. However, adding methods to a monitor is not always desirable from a software engineering perspective. Ideally, we want to define monitors to be generic, without creating multiple specialized methods that are each used sparsely throughout the program.

Lastly, we tested unlocked monitors using the `get` method, with and without the aforementioned hand optimization. We expected the optimized program to be as fast as the base case, and the non-optimized version to be slower than base, but faster than regular monitors.

As seen in Table 6.3, we were mostly correct in our predictions. Accessing a monitor multiple times is costly, but using unlocked monitors can speed execution considerably. In our benchmarks, unlocked monitors were 10 times faster than regular monitors, but still roughly 4 times slower than the alternative with no monitors. Unexpectedly, optimized unlocked monitors were only marginally faster than their unoptimized implementation. After investigating, we discovered that the usage of virtual tables for non-interface monitor types burdens execution with unnecessary memory loads and precludes the compiler from inlining some function calls. In the future, we believe changing the way

we use virtual tables in the compiler will make the hand optimization worth implementing.

7
Safeness

In this chapter, we provide a rigorous argument for the absence of data races in Aria programs. Our definition of data races (Definition 7.2) is essentially equivalent to other common definitions, such as C++'s (11) and Java's (12).

Definition 7.1 (Happens-Before Relation) *Two operations A and B are ordered by a happens-before relation if both execute in the same thread or if they synchronize with each other.*

Definition 7.2 (Data Races) *Two or more data operations on the same memory location form a data race if at least one of them is a write operation, and they do not follow a happens-before order.*

In our argument, we will use `var` to refer to variables declared as `variable` in Aria, and `val` for variables declared as `value`. Similarly, we will use `Immut` to refer to the type qualifier `Immutable`.

7.1
Important concepts and definitions

In Aria, variables (global or local, from monitors or records) and array elements are *memory locations*. Basically, memory locations correspond to constructions that can appear in the left-hand side of an assignment statement.

A memory location can be read from and written to. We will refer to both operations as *accesses* to memory locations. Variables are accessed through their names, according to scope visibility rules. Record variables, exceptionally, require field selection (e.g., `rec.a`), and array elements are accessed through indexing (e.g., `arr[i]`).

A *value* is the result of evaluating an expression. It may or may not be assigned to a memory location. Booleans, numbers, and strings are primitive values. Arrays, records, monitors, and condition queues are reference values.

The *invalid* value is the value of every non-initialized memory location. Reading from a memory location that contains an invalid value raises an error.

A *constant* memory location cannot be rewritten with other values. In Aria, variables declared as `val` are constant because they cannot be reassigned.

In the same way, elements from `Immut` arrays are constant since they cannot be rewritten with other values.

An *immutable* memory location must be constant and indexing or field selection operations on it must result in immutable memory locations. Essentially, immutable memory locations transitively only expose access to constant memory locations. In Aria, a `val` with an `Immut` type, the elements of an `Immut` array, and the fields of an `Immut` record are immutable memory locations.[1]

A *safe* memory location is either immutable or a constant that references a (not unlocked) monitor.

A *local* memory location can only be accessed by a single thread or monitor. Moreover, if a local memory location exposes access to other memory locations, those must be safe or local to the same monitor/thread.

We use these concepts to define three auxiliary lemmas.

Lemma 1 *Accesses to memory locations that are local to a single thread do not cause data races.*

Proof 1 *According to Definition 7.1, operations executing in the same thread follow a happens-before order. By Definition 7.2, data races won't occur between operations that follow a happens-before order.*

Lemma 2 *Accesses to constant memory locations do not cause data races.*

Proof 2 *Because there are no write operations on constant memory locations, this lemma derives from Definition 7.2.*

Lemma 3 *Accesses to memory locations do not cause data races if they are always executed in mutual exclusion.*

Proof 3 *Mutual exclusion guarantees synchronization between operations and, therefore, a happens-before order of execution according to Definition 7.1.*

[1]Remember that: primitive types are naturally `Immut`; an array is `Immut` if it has `Immut` elements; a record is `Immut` if all its variables are declared as `val` with `Immut` types; condition queues cannot qualify as `Immut`.

7.2 Arguing safeness

We base our argument for the absence of data races in Aria on one invariant.

Definition 7.3 (Invariant) *At any time during program execution, each thread or monitor can only access memory locations that are either local or safe.*

According to Lemma 1, accessing memory locations local to a thread cannot cause data races. Memory locations local to monitors can only be accessed inside monitor methods. Because such methods can only be called in mutual exclusion, Lemma 3 assures the absence of races for these locations. Moreover, immutable memory locations are constant and only expose access to constant memory locations, therefore Lemma 2 guarantees accessing them does not cause data races. Finally, accesses to constants with monitor values are covered by Lemmas 2 and 3, as monitors only expose their variables indirectly through mutually exclusive method calls.

7.3 Arguing the invariant

We next argue that the invariant holds by demonstrating that it is true when threads and monitors are created, and that it is preserved by any operational step the language can take.

Thread Creation. When a thread is created (either the main thread or one of its descendants), the language enforces the thread can only access variables from other scopes that were defined as `val` with (not `Unlocked`) monitor or `Immut` types. Hence, newly created threads can only access safe memory locations, thus complying with the invariant.

Monitor Creation. Monitors are created through constructor calls. Before executing the first line of code of its constructor, a monitor has access to its own variables, globals, and parameters. A non-initialized monitor variable is trivially local. Otherwise, if the monitor variable was initialized with a value resulting from expression evaluation, the variable is local by definition. Additionally, a global or a parameter must be a `val` with a (not `Unlocked`) monitor or `Immut` type, which constitutes a safe memory location. In summary, monitor variables are local memory locations, and parameters and globals are safe, which complies with the invariant.

Preservation. We now prove that each step a thread or monitor can take preserves the invariant. A step can be a variable declaration/definition,

(V1) and (V2), a statement, (S1) to (S8), or expression evaluation, (E1) to (E9).

(V1) Variable declaration **`var x: T`**

Declaring a variable creates a new local memory location containing the invalid value.

(V2) Variable definition **`val x: T = y`**

`var x: T = y`

Defining a variable creates a new memory location that can only be accessed by the current thread/monitor. Other memory locations accessible through the evaluated expression `"y"` must be local or safe, because they comply with the invariant. Therefore, the new memory location is local by definition.

(S1) Assignment **`x = y`**

`rec.x = y`

`array[i] = y`

Assigning the result of an evaluated expression to a variable or an array position is a rewrite operation. Assignments are only permitted if `"x"` is a `var` (not a `val`) and `"array"` is not `Immut`. Therefore, we guarantee constant and, consequently, safe memory locations cannot be reassigned, only locals. A local memory location remains local after reassigned, which leaves the invariant unchanged.

(S2) Function, method, and constructor calls

See (E5), (E6), (E7), (E8), and (E9).

(S3) Wait-For-In **`wait for x in y`**

A `wait-for-in` statement can only appear inside monitor methods. It does not create or alter memory locations, neither changes memory visibility. Moreover, despite temporarily releasing the monitor's lock, the statement does not break the mutual exclusion rule. Therefore, `wait-for-in` statements are inconsequential to the invariant.

(S4) Signal & Broadcast **`signal x`**

`broadcast x`

These statements are local to monitors.

(S5) Return **`return`**

`return x`

Returning from a function changes memory location visibility. Local

variables cease from being accessible, and memory locations that were visible before the function being called become accessible once again. Since the invariant was preserved during the function call, it remains so in the scope the program returns to. Furthermore, if the program returned from a monitor method, `"x"` must be `Immut` or a monitor, which only exposes memory locations that are safe.

(S6) While, If & If-Else **`while x {...}`**
`if x {...}`
`if x {...} else {...}`

A `while` statement does not create or alter data, neither changes data visibility. The statement evaluates `"x"`, which preserves the invariant, and executes the loop's block following the operational steps. Therefore, this statement is inconsequential to the invariant. Conditional statements (`if` and `if-else`) follow the same reasoning.

(S7) Spawn **`spawn {...}`**

The spawn statement creates a new thread.

(S8) Acquire-Value **`acquire value x = m.f( exp-list ) {...}`**

This statement creates a new memory location, the unlocked monitor, that is local to the running thread/monitor, thus preserving the invariant. The `Unlocked` type qualifier prevents it from being shared with monitors or threads. Execution of the statement's block follow the operational steps.

(E1) Logical operations **`not or and`**
Arithmetic operations **`+ - * /`**
Equality operations[2] **`== !=`**
Primitive values **`true false 10 3.14 "string"`**

Evaluating these expressions does not create memory locations, only values. Therefore, they are inconsequential to the invariant.

(E2) Variables

A variable represents a memory location that can be accessed by the program. Evaluating a memory location with an invalid value raises an error, and evaluating a local or safe variable memory location is inconsequential to the invariant.

[2]Equality on reference values checks if the two values reference the same memory location. Primitive values are trivially comparable.

(E3) Indexing & Field Selection **`array[i]`**
`rec.x`

According to the invariant, the evaluated expression `"array"` is either local or safe. If `"array"` is local, indexing it results in a local memory location by the definition of *local*. If `"array"` is safe, indexing it results in a safe memory location by the definition of *safe*. Field selection follows the same reasoning.

(E4) Type conversions **`m as T`**

Type conversions do not convert values, and can only cast monitors to interfaces, which does not break the `Immut` and `Unlocked` qualifiers. Hence, evaluating this expression is inconsequential to the invariant.

(E5) Array and record constructors **`Immut [ exp-list ]`**
`[ exp-list ]`
`Immut T(n)`
`T(n)`
`...`

Constructing an array creates new memory locations filled with values from `exp-list`, which comply with the invariant. Alternatively, an array of size `n` can be created with invalid values. In both cases, if the array is `Immut`, its elements must also be `Immut`. The invariant is preserved because the new array can only be accessed by the current thread or monitor and, thus, is local. Record constructors follow the same reasoning.

(E6) Function calls **`f( exp-list )`**

Calling functions changes memory location visibility. At the start of a function, only globals variables and parameters are visible. The invariant is preserved because global variables are a subset of previously accessible memory locations, and the function's parameters are local memory locations containing also a subset of previously accessible memory locations. The execution of the function follows the inductive steps, and the `return` statement was discussed in (S5).

(E7) Method calls **`m.f( exp-list )`**

Calling methods change memory location visibility from threads to monitors. At the start of a method, only globals, parameters, and the monitor's variables are visible to the monitor. The invariant is preserved because parameters from methods and constructors are constants (`val`) that must be immutable (`Immut`) or (not unlocked) monitors. Therefore,

only safe (globals and parameters) or local (monitor variables) memory locations are accessible to the monitor. Moreover, local memory locations shared through return values could potentially break the invariant when visible to other threads, since they would not be local anymore. However, the language guarantees methods can only return safe memory locations, which preserves the invariant.

(E8) Monitor constructors **`M( exp-list )`**
A monitor constructor creates a new monitor.

(E9) Queue constructors **`ConditionQueue()`**
Condition queue constructors can only be called inside monitors, and condition queue values cannot escape their monitors because they are not *safe*. Therefore, the result of evaluating this expression is always local to a monitor.

8
Conclusion

In this dissertation, we have shown how it is possible to combine monitors with referential semantics given an appropriate set of typing rules. Using type qualifiers and immutability, Aria provides monitors as originally proposed, while also enabling the addition of new features. Mutable shared data can only be accessed in Aria inside monitor operations. Thus, the absence of data races is guaranteed by the compiler.

To demonstrate that Aria's rules allow enough flexibility to solve classic concurrency problems, we discussed the implementation of three different examples of monitors. Monitors encapsulating data structures that require mutually-exclusive access are trivial to implement. For the readers-writers problem, we discussed two different solutions, with trade-offs regarding complexity, coupling, and the avoidance of programming errors. The first solution, with a `RWLock`, is similar to the one found in classic texts. The hybrid `ReadersWriters` monitor shows that is possible to compose basic monitors in order to guarantee synchronization at different levels.

In order to deal with some of the drawbacks of the monitor concept, we extended Aria with unlocked monitors and scoped values. The `RW` monitor introduces the concept of scoped permissions, effectively eliminating the complexity seem in the previous `ReadersWriters` implementation. Meanwhile, the `HashTable` monitor showcases how to implement scalable parallel solutions of higher complexity using monitors. These examples show that unlocked monitors and scoped values improve the language's flexibility and, as argued in Chapter 7, maintain Aria's guarantees regarding data races. At all times memory locations are either local to monitors and threads or safe for access.

In Chapter 5, we described our current implementation of Aria, highlighting monitors and the added features. We plan to investigate possible optimizations and alternatives to deal with the runtime checks present in our solution. In Chapter 6, we tested the language's overall performance and observed it is comparable with C in the examples we benchmarked. We also measured the performance benefits of unlocked monitors and concluded using this new feature indeed enhances performance when accessing multiple monitor methods sequentially.

8.1 Acknowledgments

Chapters 2 and 3 of this dissertation were published as a paper in the Proceedings of the XXII Brazilian Symposium on Programming Languages (SBLP '18) (13). We thank the anonymous reviewers for their insights and contributions to this work.

Bibliography

[1] BOEHM, H.-J.; ADVE, S. V.. **You don't know jack about shared variables or memory models**. ACM Queue, 9(12):40–49, December 2011.

[2] HANSEN, P. B.. **Monitors and concurrent pascal: A personal history**. ACM SIGPLAN Notices, 28(3):1–35, March 1993.

[3] HANSEN, P. B.. **Operating System Principles**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[4] HOARE, C. A. R.. **Monitors: An operating system structuring concept**. Communications of the ACM, 17(10):549–557, October 1974.

[5] WIKIPEDIA CONTRIBUTORS. **Monitor (synchronization) – Wikipedia, the free encyclopedia**, 2018. [Online; accessed 27-March-2018].

[6] LEA, D.. **Concurrent Programming in Java. Second Edition: Design Principles and Patterns**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] BUTENHOF, D. R.. **Programming with POSIX Threads**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] HOWARD, J. H.. **Signaling in monitors**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '76, p. 47–52, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[9] SANTOS, R.. **The Aria programming language: a study on monitors in programming languages**, December 2017. [Bachelor's Dissertation].

[10] DOWNEY, A. B.. **The Little Book of Semaphores**. Green Tea Press, 2005. [Online; accessed 21-May-2018].

[11] C++ REFERENCE CONTRIBUTORS. **Memory model (threads and data races)**, 2019. [Online; accessed 11-October-2019].

[12] MANSON, J.; PUGH, W. ; ADVE, S. V.. **The java memory model**. SIGPLAN Not., 40(1), 2005.

[13] SANTOS, R.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Revisiting monitors**. In: PROCEEDINGS OF THE XXII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, SBLP '18, p. 75–82, New York, NY, USA, 2018. ACM.

[14] SKYRME, A.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **A survey of support for structured communication in concurrency control models**. J. Parallel Distrib. Comput., 2014.

[15] ALEXANDRESCU, A.. **The D Programming Language**. Addison-Wesley, 2010.