Providing Concurrency Guarantees Using An Event-Driven Language

Alexander Stekelenburg
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
a.v.stekelenburg@student.utwente.nl

ABSTRACT

The need for programs to become concurrent is ever-growing. This is a problem because concurrent programs are very complex which makes it very difficult to write error-free programs. In this paper, I design a programming language that attempts to solve this problem by separating critical and non-critical code sections and employing an event-driven concurrency model. Additionally, I implement a compiler for the language to ensure that providing the language's concurrency guarantees is feasible. The designed language alleviates some of the problems which cause concurrency issues at the potential cost of performance. The event-driven concurrency model makes it easier to effectively utilize a multiprocessor architecture and interacts well with the separation of critical and non-critical code sections.

Keywords

Concurrency, event-driven, compilers, immutability, critical sections

1. INTRODUCTION

Programming error-free concurrent programs continues to be a challenging task. The cause of this is the complex interactions between different threads that are executing code at the same time. To ensure these programs behave as intended the programmer must use mutual exclusion where necessary or use carefully designed lock-free data structures. Additionally, computer hardware is becoming increasingly parallel, so concurrent programs are needed to effectively use the capabilities of this hardware. Hence we have a problem, concurrent programs are the way of the future but programming them is challenging and error-prone. There are attempts at solving this problem. One such attempt is Transactional Memory (TM) which can be implemented both in hardware and in software.[5] Additionally, some languages like Rust provide stronger concurrency guarantees through extensive compile-time checking. These checks help the programmer by detecting when they have made a mistake, but they still have to fix the error themselves. The goals of this research are defined as such:

- Design a language that provides strong guarantees about the behaviour of concurrent programs
- Implement a compiler for said language to discover whether it is feasible to provide these guarantees
- Analyse whether the language's concurrency guarantees and event-driven concurrency model help to solve or prevent the issues that are found in concurrent programs

To achieve these goals, I will answer the following research question: *To what degree can the use of a language which separates critical and non-critical code sections and has an event-driven concurrency model provide concurrency guarantees and lead to intuitive parallelization for the programmer?* To answer this research question, I will first answer these sub-questions:

RQ1 *What concurrency issues are most common and what causes these to be introduced into programs?*

RQ2 *What concurrency guarantees can be provided using the information that the designed language has access to at compile-time?*

RQ3 *To what degree do the compile-time concurrency guarantees from **RQ2** and the event-driven concurrency model provided by the designed language prevent the concurrency issues identified in **RQ1**?*

This paper is divided up into several sections. Firstly, in section 2 I will discuss some existing solutions to the problem. Secondly, in section 3 the design of the language is explained along with the reasoning behind the choices that have been made. Thirdly, the specifics of the compiler implementation can be found in section 4. Fourthly, I will present my results in section 5. Finally, I will conclude in section 6 and discuss possible future topics of research in section 7.

2. EXISTING SOLUTIONS

To implement our language's concurrency model lock-based synchronisation is used. An alternative to using locks is using (Software) Transactional Memory (TM). TM is a multiprocessor architecture designed to support lock-free synchronisation without the performance drawback that the conventional lock-free data structures have compared to structures that require mutual exclusion.[5] TM works by providing instructions to the programmer which allow them to define sets of shared memory locations that should be guarded against concurrent reads and writes. The programmer can then use these instructions to read from and update the data at these locations, but the changes do not become permanent until the transaction is committed. If no other transaction has changed the data in the set of

35^{th} Twente Student Conference on IT July 2^{nd}, 2021, Enschede, The Netherlands.

memory locations then the transaction succeeds, otherwise, it is reverted. TM does not suffer from the conventional problems that mutual exclusion has. Namely, deadlocks and higher priority processes having to wait on lower priority processes. TM can be implemented in hardware (HTM), software (STM), or using a hybrid (HyTM) approach.[13] Meier and Gross (2019)[9] reflect on the results of the Python VM being parallelised using STM and fine-grained locking. STM and fine-grained locking both have a significant performance overhead. However, in practice the STM based implementation seemed to suffer less from this depending on the use case. The Python VM implementations based on fine-grained locking have to employ quite complicated locking strategies due to the need to adhere to Python's semantics. However, the language that is proposed in this paper has semantics that are less restricting than Python's semantics. For this reason, I believe that, despite Meier and Gross's results, an implementation of its concurrency model using fine-grained locking is a good choice.

Another approach to prevent concurrency errors from ending up in programs is using extensive compile-time checking. A prime example of a language that has implemented this is Rust. Rust's safety mechanisms are based on its borrow checker which keeps track of the *ownership* of variables.[10] Using the borrow checker Rust effectively eliminates errors due to mishandling of pointers and references. Ownership of variables cannot be transferred to a different thread which effectively prevents low-level data races. To actually share variables one has to explicitly use data structures that provide access using code that is not checked by the borrow checker. These synchronisation constructs can be marked with the *Send* trait which allows them to be shared across multiple threads. The Rust standard library provides several synchronisation constructs which implement this trait. While Rust's borrow checker makes sure that variables cannot be shared across thread boundaries without using these synchronisation constructs, it does not check whether these constructs are used appropriately. Especially Rust's mutual exclusion construct *Mutex* can cause high-level data races because it is automatically unlocked when the borrow checker determines the guarded variable is no longer used.[10]

3. LANGUAGE DESIGN

Because the language will compile to run on the Java Virtual Machine (JVM) it shares many of Java's semantics. This includes the type system, heap allocation and implicit boxing and unboxing of primitive types and their heap-allocated *Object* counterparts. Furthermore, code blocks follow the same scoping rules as Java. The choice to follow Java's semantics was made because the JVM is built to optimise bytecode generated by the Java compiler. The language provides a clear syntactic divide between contexts where fields can be mutated and where they can not. This is done using the concept of *Read-Write Flows* which apply to any number of *Components* (data structures). These *Components* are the only data the *Read-Write Flow* is able to mutate. An example of the syntax of a *Read-Write Flow* and a *Component* can be found in Listing 1. The *Read-Write Flow* in this example declares the targets *A* and *B* as mutable, has one (immutable) parameter called *amount* and returns nothing (*void*). Before a *Read-Write Flow* can be invoked it must be instantiated using the *new* keyword. Whenever the *Read-Write Flow* is invoked the language ensures that it has exclusive (mutable) access to its fields and targets.

```
component BankAccount {
  String owner;
  float balance;
}

read_write_flow Transaction(float amount
) for BankAccount A, BankAccount B ->
void {
  {
    A.balance += amount;
    B.balance -= amount;
  }
}
```

Listing 1. A *Component* representing a bank account and a *Flow* that performs a bank transaction

Furthermore, the language is built around an event-driven concurrency model. Event-driven program structures have advantages when it comes to the effective use of system resources, but tend to make programs more complex.[3] To implement this, the language has structures called *Events* which are defined based on the data types that make them up. An example of an *Event's* syntax and how to fire it can be found in Listing 2. The *Event* in the example has 3 fields which are passed to every *Handler* that is bound to it when it's fired.

```
event TransactionEvent -> BankAccount,
BankAccount, float;

fire TransactionEvent(accountA, accountB
, 20.0f);
```

Listing 2. An *Event* that is fired when a bank transaction takes place

Events can be bound to *Handlers*. An example of a *Handler's* syntax and how to bind it to an *Event* can be found in Listing 3. The handler in Listing 3 also shows an example of a field. Whenever a *Handler* is invoked through a bound event being fired (in the example that would be the *TransactionEvent*) it ensures that the fields and parameters it uses cannot be mutated while the *Handler* is running. However, when the *Handler* invokes a *Read-Write Flow* or a *Read Flow* these constraints are let go to prevent deadlock issues.

```
handler TransactionHandler for
BankAccount A, BankAccount B, float
amount {
  String prefix;
  {
    println(prefix);
    println(A.owner);
    println(B.owner);
    println(amount);
  }
}

TransactionEvent => new
TransactionHandler("Transaction:");
```

Listing 3. A *Handler* that is bound to a *TransactionEvent*

Whenever an *Event* is fired all of the *Handlers* that are bound to it are executed by submitting them to the runtime thread pool. Additionally, the language has two more constructs the *Function* and the *Read Flow*. *Functions* are pure procedures that only depend on their parameters and do not have any side-effects. *Read Flows* are similar to *Read-Write Flows* but they only get immutable access to their targets. Examples of a *Function* and a *Read Flow*

can be found in Listing 4. When the *Read Flow* in the example is invoked the language will ensure that the target *A* and field *splitter* remain unchanged until the *Read Flow* has finished executing.

```
function fibonacci(int n) -> int {
  if (n < 2) return 1;
  return fib(n-2) + fib(n-1);
}

read_flow AccountBuilder() for String A
-> BankAccount {
  StringSplitter splitter;
  {
    String[] split = splitter(A);
    String owner = split[0];
    Option<float> balance = parseFloat(
    split[1]);
    return new BankAccount(split[0],
    unwrapFloat(balance));
  }
}
```

Listing 4. An example of a *Function* that calculates the n-th fibonacci number and a *Read Flow* that builds a *BankAccount* from a *String*

Listing 4 also shows the language's type system in action. The local variable balance is of the generic type *Option<float>*. The *Option<T>* type is defined in the language's standard library and can be instantiated for any type *T*. The type system is based on Java's types because the compiler targets the JVM. *Components*, *Read(-Write) Flows*, *Events*, and *Handlers* all map to Java classes whereas *Functions* are mapped to static methods. Whenever a compiled program is executed a thread pool is created and the initialisation *Event* is fired from the main event loop. The program's main behaviour, whose syntax is simply a code block without surrounding keywords, is a *Handler* for this *Event*. The event loop will keep running until the exit *Event* is fired using the *quit()* standard library *Function*.

4. COMPILER IMPLEMENTATION

The compiler is implemented using ANTLRv4[1] and the ObjectWeb ASM library[2] in Java. ANTLR provides a way to quickly generate the parser for the language. The resulting parse tree is then turned into an Abstract Syntax Tree (AST). This AST is passed to the code generator which uses the ASM library to generate JVM bytecode. The choice to target the JVM was made because it allows the use of its garbage collector and the Java standard library. This simplified the code generation a lot because the JVM handles memory management. Additionally, the Java standard library contains many synchronisation constructs which the generated code can use. In particular the *java.util.concurrent.locks.ReentrantReadWriteLock* and the *java.util.concurrent.LinkedBlockingQueue* were used for mutual exclusion and the event queue respectively. To allow event handling to be done concurrently the execution of these *Handlers* is done by submitting them to the thread pool. The thread pool of choice was the work-stealing pool, created using *java.util.concurrent.Executors#newWorkStealingPool*. This pool was chosen because it is particularly well suited for the efficient use of threads when the program schedules many short-lived tasks which is a common occurrence in event-driven programs. After compilation, the resulting JVM bytecode is packaged in an executable JAR file along with the language's standard library. The standard library contains classes that are used internally by the generated code to perform automatic locking and run the main event loop. Additionally, the standard library contains a few *Functions* and *Flows* that act as a wrapper for the Java standard library to perform tasks like printing to the standard output and conversions between Strings and primitive types. The locking strategies that are generated by the compiler to ensure mutual exclusion for *Flows* and *Handlers* are generated as separate classes to allow them to be more easily replaced in the future. A more detailed description of the way these locking strategies work can be found in subsection 5.2.

[1] `https://www.antlr.org/`
[2] `https://asm.ow2.io/`

5. RESULTS

5.1 RQ1

Strömbäck et al. (2019)[11] and Strömbäck et al. (2020)[12] analysed common problems that students have while making concurrency exercises. Students seemed to have the most trouble with the proper identification of critical sections and deciding upon the best granularity for locks. The improper identifications of critical sections can be split up in two groups. The first group of students saw critical sections as bits of code that should not be executed at the same time and thus required synchronization. This is an incomplete view because it is not just the code that needs to be "protected" but also the data that the code uses. As such these students did not apply synchronization in other locations where this same data was used or written. The second group of students saw critical sections as variables that should be accessible without data races. This is an incomplete view because often a piece of code requires that the variables it uses do not change while the calculation is being performed. These students often used locks that were too granular (one lock per variable and locking and unlocking only for the direct usage of the variable). In situations where a variable was read a calculation was performed and the variable was written again, synchronization would be applied around each read and write separately instead of around the entire code fragment.

Kolikant (2004)[7] studied the evolution of high school student's understanding of synchronisation between two tests. As was the case for [11, 12] student's main problem was the identification of the ways in which code required synchronisation. However, even when these synchronisation goals were identified the students still had problems understanding the concurrency constructs (e.g. semaphores) that enabled synchronisation. Instead, students relied on applying usage patterns for these constructs that they had seen previously. This caused students to come to the wrong conclusions about the thread-safety of the programs discussed in the test. Apparently, these concurrency constructs were so complex that students did not have sufficient understanding of them even after being refreshed on the definitions of these constructs. This shows that a lack of understanding can be hidden for a long time which could cause concurrency issues if a programmer fails to (find and) apply the correct usage pattern to their program.

Lu et al. (2008)[8] examined 105 concurrency bugs in four large mature concurrent applications. Of these bugs, 31 were classified as deadlock bugs. Additionally, 72 of the non-deadlock bugs could be classified as either atomicity or order violations. Of the non-deadlock bugs 34% involves more than one variable that is semantically connected. Only 20 out of the 74 non-deadlock bugs were fixed by adding locks. They give three reasons for this: locks cannot guarantee some intent like ordering of ac-

cesses, locks often come at a performance cost and they may introduce new deadlock bugs. 44 out of 105 concurrency bugs could be avoided using Transactional Memory (TM). TM is especially well suited for preventing data race and deadlock bugs when they apply to relatively small critical sections. However, if critical sections get too big or if they involve I/O operations the use of TM gets more complicated because it is harder to reliably revert these operations. Additionally, 20 out of the 105 bugs could not be solved using TM because TM cannot guarantee that the programmer's order intentions are observed. Unfortunately, mutual exclusion also cannot guarantee to solve this problem.

5.2 RQ2

The language design allows us to provide some guarantees about the behaviour of concurrent programs written using the language. By only allowing side-effects to occur in *Read-Write Flows* a compiler can easily generate locking strategies that prevent low-level data races. This is achieved by creating a *ReentrantReadWriteLock* for each *Component*. Whenever a *Handler* is invoked its locking strategy is applied. The locking strategy will acquire read-locks for each *Component* field of the *Handler* unless this *Component* is marked with the *concurrent* keyword. Similarly, the compiler also generates locking strategies for *Flows* these strategies involve acquiring read-locks for the *Flow's Component* fields and the *Read Flows's* targets and acquiring write-locks for the *Read-Write Flow's* targets. If the locks are acquired (the order in which they are released does not matter) in a fixed order by every locking strategy, it becomes impossible for a deadlock scenario to occur.[4] The compiler generates the code for these locking strategies. The strategy that was implemented orders all components that need to be locked based on a Universally Unique Identifier (UUID) whenever the locking strategy's *startSection* method is invoked. This UUID is generated for each instance of a component whenever it is instantiated using the *new* keyword. Whenever a locking strategy's *endSection* method is invoked it will release all the read and write locks that were acquired by the *startSection* method. To do this the *startSection* method writes the list of components that were locked to a cache which is re-used by the *endSection* method. The language also attempts to prevent livelock scenarios by employing an *Event* queue which prevents repeated *Events* from being handled before the previous instance was handled. This is not a watertight solution because there is no way to pre-empt a long-running *Handler* which means such a *Handler* can hold up the execution of any subsequent *Handlers* that need access to the same data. *Flows* can only be invoked from *Flows* and *Handlers* because *Functions* are prevented from having any side-effects. This has the upshot that the compiler does not need to generate locking strategies for these procedures while still guaranteeing that low-level data races cannot occur. Similarly, *Events* can only be fired from *Flows* and *Handlers* because *Events* trigger *Handlers* which may in turn cause side-effects. Additionally, *Flows* can only invoke other *Flows* on local variables or on targets that they already have a read or write lock on. This is needed to avoid the locking order changing arbitrarily because nesting *Flows* would allow locking to be delayed until the nested *Flow* was invoked.

Low-level data races are not the only data races that appear in concurrent programs. Artho et al. (2003)[1] define high-level data races as: "sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out atomically can be interleaved with conflicting operations". High-level data races can occur even if there are no low-level data races[1]. This form of data race occurs when the scope of variables that are "guarded" by a lock is inconsistent. Artho et al. (2003)[1] provide an algorithm that attempts to find these inconsistencies but this algorithm still results in false positives and false negatives. This is because it is hard to infer the intent of the programmer. The language design discussed in this paper attempts to make it easier for the programmer to express the program's intended behaviour. It does this by providing *Read-Write Flows* and *Read Flows* which are syntactic structures that represent (read-only) critical sections. By providing a clear syntactic separation between code that is a part of a critical section and code that is not, it becomes intuitive to put all variable accesses that belong together in the same code block. Another way to express this is that this syntactic separation encourages the programmer to divide their code up into *linearizable* operation. If an operation on an object is *linearizable* it is essentially atomic with respect to other operations on that object[6]. This means it is impossible for operations to be executed on a *linearizable* object that lead to unpredictable behaviour.

5.3 RQ3

After answering **RQ1** we find that most concurrency issues have on of three causes.

1. It is hard to identify which code sections and which memory locations are part of a critical section
2. The use of specific synchronisation constructs is hard and students often rely on the combination of previously seen patterns instead of fully grasping the semantics of the data structures.
3. A programming language's execution model is often thought of as simpler than it actually is. This leads programmers to make unconscious assumptions about the execution of the program.

The language attempts to solve these issues in the following ways. Firstly, the language provides the *Read Flow* and *Read-Write Flow* constructs that correspond to the program's critical sections. Because these constructs have their own scope it is impossible to forget to include a data access in the section. Because the mutation of non-local variables is exclusively possible in *Read-Write Flows* it is assured that every write operation whose result might be accessed concurrently is automatically guarded with a lock. Secondly, the language intentionally obscures the underlying synchronisation constructs that are used to provide its concurrency guarantees. This comes with the downside of reduced performance compared to the best-case scenario of a programmer's direct usage of synchronisation constructs. However, abstracting away these constructs allows the programmer to focus on the intended behaviour of the program and precludes the possibility of writing code that results in deadlocks, data races or other concurrency issues. Care should still be taken to use data structures that require a minimal amount of mutual exclusion. However, because of the concurrency guarantees the language provides, this can only lead to degraded performance and no longer to incorrect behaviour. Thirdly, most programming languages are sequential in nature. However, when threading is added to a language this breaks a lot of the guarantees that these languages normally provide in a single-threaded context. To visualise the program execution of programs that use

a thread-based concurrency model one has to think about multiple sequential programs executing in parallel. This is different for programs that use an event-driven concurrency model. Event-driven programs can easily be visualised as a graph of events and the event handlers that are triggered by these events and the events that are in turn fired by the event handlers. This is easy to understand because it is analogous to many real-world processes which can also be understood as sequences of actions (events) and responses(event handlers). Additionally, Dabek et al. (2002)[2] have shown that event-driven programs can be efficient, easy to use and make good use of multiprocessor hardware.

5.4 Example Application

To provide a comparison between Java and our language a simple banking application was implemented in both languages. This application consists of six producers which will execute transactions between the same two bank accounts. If no race conditions occur the resulting balances in each account will be the same as the starting balances. Listing 5 shows the Java implementation of the application using *Threads* and *synchronized* blocks. I decided to use these constructs to provide an example of a program written using a thread-based concurrency model and which uses synchronisation constructs in an explicit manner. Listing 6 shows the implementation of the application in our language using *Read-Write Flows* and *Handlers*. As opposed to the Java implementation this implementation uses the language's event-driven concurrency model. Additionally, the choices that determine which variables and code sections need to be guarded with specific synchronisation constructs are implicitly defined through the language's syntax which requires the programmer to separate the mutable from the immutable context. These listings can be found in Appendix A. To compare the performance of these implementations some benchmarks were performed. These benchmarks were performed on a machine with an i7-8750H (6 cores/12 threads at a 2.2ghz base clock) processor. Because the thread pool used by our language automatically utilises a number of threads matching the number of threads the processor has, the implementation uses at most 12 threads. However, because the application only has 7 event handlers (including the main behaviour) not all of these threads are in use. Therefore, the amount of threads used by the Java implementation and the implementation in our language is the same. subsection 5.4 shows the average execution times over 20 runs of each implementation.

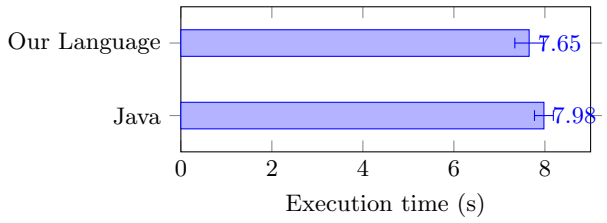

Figure 1. The average execution times of the banking application in Java and in our language

While these execution times seem to show our language being slightly faster than Java in this use case, this is most likely caused by the complexity of run time lock-ordering giving other threads extra time to execute which leads to less failed transactions due to insufficient balances. To test this the implementations were updated to include a *Thread#Yield* call at bottom of the while loop body. Figure 5.4 shows the execution times for the updated implementations. As expected, the implementation in our language got slower because it spent more time waiting between each loop iteration. However, the Java implementation became more than ten times faster because threads were no longer able to repeatedly re-acquire the locks before the other threads got a chance to perform a transaction. The run-time lock ordering performed by our language evidently introduces a significant performance overhead. Notably, there are several ways in which this issue can be minimised which are discussed in section 7. While this is a trivial example it does show a way in which our language simplifies some of the choices a programmer must make. For the *synchronized* blocks in lines 38-45 the programmer had to not only decide which variables needed to be guarded but also the order in which these synchronisation measures were used. This order has to be preserved throughout a program to avoid deadlocks. By contrast, while in our language the programmer still has to identify the bank accounts as the variables which are mutable this automatically follows from the fact that their balance must be mutated in a transaction. Additionally, the lock ordering is handled by the generated program runtime. The thread-based concurrency model is arguably more intuitive for this program because it is more intuitive to wait for (join) all the threads to finish executing instead of keeping track of this yourself using the *FinishCounter* and the *Incrementer*. This problem could be resolved by implementing *Futures* and *Callbacks* which are described in more detail in section 7.

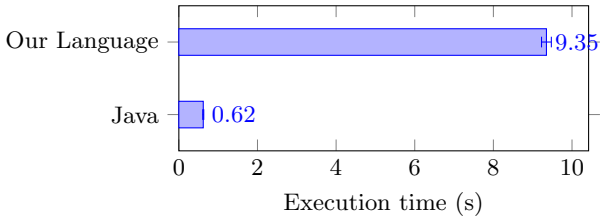

Figure 2. The average execution times of the updated banking application in Java and in our language

6. CONCLUSION

In this research, an attempt was made to discover whether a language that makes the programmer separate their code's critical sections from the non-critical sections while employing an event-driven concurrency model has benefits when creating concurrent programs. Firstly, most concurrency issues are caused because it is hard to identify which code segments and variables need to be in a critical section. Especially for complex projects, it is easy to forget to include something. The language solves this by separating non-critical and critical sections into different lexical scopes. This helps to avoid both low-level and high-level data races. Secondly, it can be hard to grasp the semantics of specific synchronisation constructs. This can lead to deadlocks, livelocks and data races. By automatically generating locking strategies the compiler assures that the resulting program is free of these problems. However, this comes at a performance cost compared to manually using synchronisation constructs. Finally, the language's use of an event-driven concurrency model helps to make writing concurrent programs more intuitive by making concurrency a central part of the program's execution model, instead of an addition like with thread-based models.

7. FUTURE WORK

There is still a large part of this topic that this research does not cover. Firstly, a language that clearly separates not only critical and non-critical section but also mutable and immutable contexts is a great target for many optimization techniques. The compiler implemented for this research makes no attempt to optimize the code that it puts out. Particularly the locking strategies that are generated by the compiler have some obvious ways in which they could be optimized. For *Handlers* read locks could be released once a variable stops being used instead of at the end of the handler and for *Read-Write Flows* that invoke other *Flows* on the same targets re-entering the locks could be skipped. There are many more of these potential optimizations to explore. Secondly, in section 2 I discussed (S)TM as an alternative to fine-grained locking. A re-implementation of the language's compiler using TM could yield further optimization. Finally, many event-driven languages make extensive use of so-called *Futures* and *Callbacks* which describe values computed asynchronously and code that is executed when another task is complete. While the programmer can implement this themselves using a multitude of *Events* and *Handlers* the integration of these constructs in the language itself could provide additional information to the compiler about the way certain variables are used which might allow for more efficient locking strategies to be generated. Finally, more work is needed to test whether identifying mutability constraints, like our language requires the programmer to do, is indeed easier than identifying critical sections.

APPENDIX

A. LISTINGS

```
class Banking {
    static class BankAccount {
        float balance;

        public BankAccount(float balance
        ) {
            this.balance = balance;
        }
    }

    static class TransactionProducer
    implements Runnable{
        final int count;
        final float amount;
        final BankAccount A;
        final BankAccount B;

        public TransactionProducer(int
        count, float amount, BankAccount
         A, BankAccount B) {
            this.count = count;
            this.amount = amount;
            this.A = A;
            this.B = B;
        }

        boolean transaction() {
            if (A.balance - amount < 0
            || B.balance + amount < 0) {
                // Failed
                return false;
            }
            A.balance -= amount;
            B.balance += amount;
            // Success
            return true;
        }

        @Override
        public void run() {
            int i = 0;
            while(i++ < count) {
                synchronized (A) {
                    synchronized (B) {
                        if (!transaction
                        ()) {
                            // Failed so
                             repeat
                            i--;
                        }
                    }
                }
            }
        }
    }

    public static void main(String[]
    args) {
        long start = System.
        currentTimeMillis();
        BankAccount A = new BankAccount
        (250f);
        BankAccount B = new BankAccount
        (250f);
        Thread a = new Thread(new
        TransactionProducer(250000, 50,
        A, B));
        Thread b = new Thread(new
        TransactionProducer(250000, -50,
         A, B));
        Thread c = new Thread(new
        TransactionProducer(250000, 50,
        A, B));
        Thread d = new Thread(new
        TransactionProducer(250000, -50,
         A, B));
        Thread e = new Thread(new
        TransactionProducer(250000, 50,
        A, B));
        Thread f = new Thread(new
        TransactionProducer(250000, -50,
         A, B));
        a.start();
        b.start();
        c.start();
        d.start();
        e.start();
        f.start();
        try {
            a.join();
            b.join();
            c.join();
            d.join();
            e.join();
            f.join();
        }catch (InterruptedException
        ignored){}
        System.out.println("Amount after
         I'm done");
        System.out.println(A.balance);
        System.out.println(B.balance);
        System.out.println("Took:");
        System.out.println(System.
        currentTimeMillis() - start);
    }
}
```

Listing 5. An implementation of a simple banking application in Java

```
component FinishCounter {
    int count;
}

read_write_flow Incrementer() for
FinishCounter counter -> int {
    long start;
    {
        return ++counter.count;
```

```
    }
}

component BankAccount {
    float balance;
}

read_write_flow Transaction(float amount) for BankAccount A, BankAccount B -> boolean {
    {
        if (A.balance - amount < 0 || B.balance + amount < 0) {
            // Failed
            return false;
        }
        A.balance -= amount;
        B.balance += amount;
        // Success
        return true;
    }
}

handler TransactionProducer for BankAccount A, BankAccount B {
    int count;
    float amount;
    Incrementer incrementer;
    {
        Transaction t = new Transaction(A, B);
        int i = 0;
        while(i++ < count) {
            if(!t(amount)) {
                // Failed so repeat
                i--;
            }
        }
        if(incrementer() == 6) {
            println("Amount after I'm done");
            println(floatToString(A.balance));
            println(floatToString(B.balance));
            println("Took:");
            println(longToString(millis() - incrementer.start));
            exit();
        }
    }
}

event ProduceEvent -> BankAccount, BankAccount;

{
    long start = millis();
    FinishCounter counter = new FinishCounter(0);
    Incrementer incrementer = new Incrementer(counter, start);
    ProduceEvent => new TransactionProducer(250000, 50f, incrementer);
    ProduceEvent => new TransactionProducer(250000, -50f, incrementer);
    ProduceEvent => new TransactionProducer(250000, 50f, incrementer);
    ProduceEvent => new TransactionProducer(250000, -50f, incrementer);
    ProduceEvent => new TransactionProducer(250000, 50f, incrementer);
    ProduceEvent => new TransactionProducer(250000, -50f, incrementer);
    fire ProduceEvent(new BankAccount(250f), new BankAccount(250f));
}
```

Listing 6. An implementation of a simple banking application in our language

References

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. volume 13, pages 207–227. John Wiley & Sons, Ltd, 12 2003. doi: 10.1002/stvr.281. URL https://doi.org/10.1002/stvr.281.

[2] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. pages 186–189. ACM Press, 2002. doi: 10.1145/1133373.1133410. URL https://doi.org/10.1145/1133373.1133410.

[3] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. pages 134–143. ACM Press, 1 2007. ISBN 9781595936202. doi: 10.1145/1244381.1244403. URL https://doi.org/10.1145/1244381.1244403.

[4] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7:74–84, 4 1968. ISSN 0018-8670. doi: 10.1147/sj.72.0074. URL https://doi.org/10.1147/sj.72.0074.

[5] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. pages 289–300. Association for Computing Machinery (ACM), 1993. doi: 10.1145/165123.165164. URL https://doi.org/10.1145/165123.165164.

[6] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12, 7 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.

[7] Y. B.-D. Kolikant. Learning concurrency: Evolution of students' understanding of synchronization. *International Journal of Human Computer Studies*, 60:243–268, 2004. ISSN 10715819. doi: 10.1016/j.ijhcs.2003.10.005. URL https://doi.org/10.1016/j.ijhcs.2003.10.005.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. volume 42, pages 329–339, 2008. ISBN 9781595939586. doi: 10.1145/1346281.1346323. URL https://doi.org/10.1145/1346281.1346323.

[9] R. Meier and T. R. Gross. Reflections on the compatibility, performance, and scalability of parallel python. pages 91–103. Association for Computing Machinery, Inc, 10 2019. ISBN 9781450369961. doi: 10.1145/3359619.3359747. URL https://doi.org/10.1145/3359619.3359747.

[10] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. pages 763–779. Association for Computing Machinery, 6 2020. ISBN 9781450376136. doi: 10.1145/3385412.3386036. URL https://doi.org/10.1145/3385412.3386036.

[11] F. Strömbäck, L. Mannila, M. Asplund, and M. Kamkar. A student's view of concurrency-a study of common mistakes in introductory courses on concurrency. pages 229–237. ACM, 8 2019. ISBN 9781450361859. doi: 10.1145/3291279.3339415. URL https://doi.org/10.1145/3291279.3339415.

[12] F. Strömbäck, L. Mannila, and M. Kamkar. Exploring students' understanding of concurrency – a phenomenographic study. pages 940–946. ACM, 3 2020. ISBN 9781450367936. doi: 10.1145/3328778.3366856. URL https://doi.org/10.1145/3328778.3366856.

[13] Tabassum and Meenu. Transactional memory: A review. pages 370–375. Institute of Electrical and Electronics Engineers Inc., 3 2020. ISBN 9781728151977. doi: 10.1109/ICACCS48705.2020.9074423. URL https://doi.org/10.1109/ICACCS48705.2020.9074423.