Verification of Concurrent Programs: History of approaches illustrated

Jeremy Joel Harisch
Technical University of Munich

Abstract

TODO: Write abtract ...

1 Introduction and Motivation

The first computer program, written by Ada Lovelace in the year 1843 [6], was an algorithm to calculate a sequence of Bernoulli numbers. Since then the machine architectures, or in general the computer hardware, as well as the way algorithms/programs are developed changed dramatically. The performance and the capability of such hardware scaled up, thus in the early 20ths century, the first concurrent programs have been introduced. Then at the beginning of the 1960s, the academic study on concurrent programs began [4], and the start point of trying to verify such programs was set. In this paper, we will introduce you to the history of verifying concurrent programs starting with *State-Transformer*-approaches [3.1], *Temporal Logic* [3.2], and approaches which mostly are based on those two approaches. All those approaches are theoretically based - later on, we will also give an overview of programs that can be used nowadays to verify a concurrent program automatically, such as Chalice [4.1].

But why do we need to verify concurrent programs and why is it that hard? - Well, in general, we need to verify concurrent programs for the same reasons as for sequential programs, including some additional requirements. The main reason would be to check whether a program produces the correct output or not and that this output does not change if we rerun the program with the same input (also called *reproducability*). In concurrent programs, we also need to check for possible dead-locks, data-races, and potential interference among threads. If and only if all those requirements are satisfied by a program then this program is always able to produce the wanted output.

In Figure 1 we can see a small example for a simple program which runs totally fine if it is executed as a sequential program. But if we would split this program up and run certain calls on two threads, then we will not have the same output on multiple runs.

2 Examples for Verification

To get a better understanding of the approaches which will be introduced, most of them will be shown using two one or two examples. To give a better understanding of what the differences between those algorithms are, the same two examples will be used for each of the algorithms. Those two examples will be introduced in the next two subsections [2.1, 2.2].

2.1 Lamport's Bakery Algorithm

The first example which is going to be introduced is the *Lamports's Bakery Algorithm* by Leslie Lamport. This algorithm is one solution for the mutual exclusion problem in the case of N concurrent processes. Each process has non-critical sections as well as one critical section. It needs to be assumed that each process can be able to enter its critical section and that that processes can be halt in their non-critical sections. Lamport makes up a scenario in a bakery example to introduce its idea behind the problems he wants to solve. The scenario takes place in a bakery where customers are getting number-tickets at the entrance from a ticket machine - Each customer represents one process and the ticket machine represents the global variable needed by all the processes. The numbers written on the tickets are counting up on each customer. Additionally, a scoreboard shows the number which customer is being served at the moment. When the customer finished his/her's purchasing the number on the scoreboard increments by one. The non-critical sections are where the processes/customers are waiting for the process/customer, who is in its critical section, to finish its trade. The critical section for each customer is when the customer trades with the baker, thus requiring exclusive access to a resource needed by all the customers. [9]

This scenario can be translated into scenarios of concurrent systems, where threads need to access or change one or more global variables needed by more than one thread. Lamport solves this problem using a *FCFS*-principle(First Come First

Serve) approach. The Algorithm goes as follows: We assume we have N processes, named $\{p_0, p_1, ...p_{N-1}\}$ and each process chooses a number from 0 to $N-1$. The process with the lowest number goes first. It is possible that more than one process chooses the same number, then the process with the lower name goes first. When the first process enters its critical section, it locks the critical sections of the other processes. When the process is done in its critical section, it unlocks the critical section for the process with the lowest ticket number - Or in the case, more than one process have the same lowest ticket number, then the process with the lowest ticket number and lowest process name will be unlocked. [9]

On some of the verification approaches, we need a more precise example to verify, so not just the algorithm itself. Thus we will use *Lamport's Bakery Algorithm* to count up using two threads simultaneously, and each thread will increase the counter 500 times. In this paper we will use two threads, for simplicity. In Listing 1 you can see a small extraction of this algorithm implemented in Java [1].

Listing 1: Extraction of the implemented Lamport's Bakery Algorithm in Java. Full code can be found at: https://github.com/JeremyHarisch/seminar_software_security_analysis/blob/main/Code/lamports_bakery.java

```
public class Bakery extends Thread {
   public int id;
   public static final int countGoal = 500;
   public static final int numberOfThreads = 2;
    [...]
   public Bakery(int id) { this.id = id; }

   public void run() {
      int scale = 2;

      for (int i = 0; i < countGoal; i++) {

         lock(id);
         // Critical-Section-Start
         count = count + 1;
         System.out.println("Thread-ID: " + id + "
              Count: " + count);
         // To create a realstic time buffer, since
              trade section is empty
         try {
            sleep((int) (Math.random() * scale));
         } catch (InterruptedException e) {}
         // Critical-Section-End
         unlock(id);
      }
   }

   // Main algorithm of Bakery algorithm
   public void lock(int id) {
      choosing[id] = true;

      ticket[id] =
          Arrays.stream(ticket).max().getAsInt() +
          1;
      choosing[id] = false;

      for (int j = 0; j < numberOfThreads; j++) {
         if (j == id)
            continue;
         while (choosing[j]) { } // Waiting until
              other thread stops fethcing new ticket
         while (ticket[j] != 0 && (ticket[id] >
              ticket[j] || (ticket[id] == ticket[j]
              && id > j))) { } // Waiting for other
              thread to choose a new ticket
      }
   }

   private void unlock(int id) { ticket[id] = 0; }

   public static void main(String[] args) {
      Bakery[] threads = new Bakery[numberOfThreads];
      for (int i = 0; i < threads.length; i++) {
         threads[i] = new Bakery(i);
         threads[i].start();
      }
        [...]
   }
}
```

[1] www.oracle.com/java/

2.2 Ticket Algorithm

For the second example, we are going to have a look at a parametrized version of the ticket algorithm by Gregory R. Andrews [1]. The algorithm itself is short and easy to read, but it very hard to prove as we will see in the next sections. In algorithm 1 is a pseudocode given, which represents the analogy behind the algorithm.

Algorithm 1: Ticket Algorithm

```
   // Initially set s and t to 0
1  s = t = 0
   // Assigning Ticket Number to Process N
2  m_N := t++;
3  if m_N ≤ s then
   |  // Critical Section
4  |  <...>;
5  s = s + 1;
```

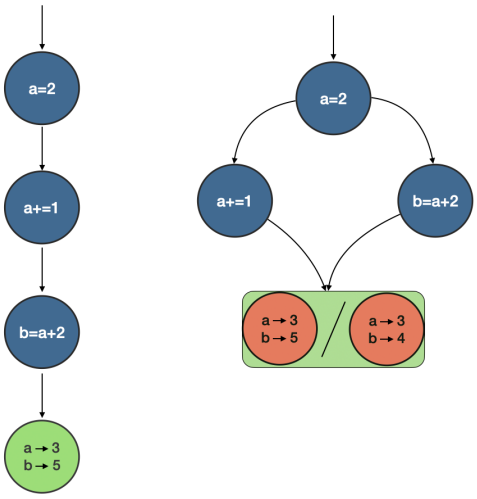

Figure 1: Simple program executed as a sequential program(left) and as a concurrent program(right) - This program does not handle data-races, thus in the sequential execution we can have multiple outputs.

3 Theoretical Approaches to Verify Concurrent Programs

The following sections will be about approaches that are used to verify concurrent programs through a more theoretical approach. To get a better understanding of verifying concurrent programs, we will first introduce and discuss some methods which are used for verifying sequential programs, which then will be used as a basis for other concurrent verification methods. Furthermore, we will notice that the step from verifying sequential programs to verifying concurrent programs, is not a challenge because more things are happing at the same time. We will notice that is more about switching from functional to reactive programs, as Lamport states in a paper [10]. Functional systems specifies programs which are mapping input to output in a sequence, as the state-transformer approaches of Floyd and Hoare 3.1. Reactive systems are programs that interact in more complex ways with its environment, so the mapping (input/output) gets replaced with a set of all possible behaviors of a program. This also means, even when the programs only calculate one single value, the interaction between the processes need to be considered - If there is no communication between the processes than the verification methods of sequential programs can be used [10].

3.1 State Transformer

In this section, we will have a look at verification-methods which are based on seeing the program as a State-Transformer. This means that the sequential or concurrent statements of a program can be translated into different program states, and then while execution time the program transfers from state to state. All these approaches have been successful at proofing simple sequential and concurrent, depends on the approach chosen, programs, but none of them have been successful at verifying "real" programs [10]. Although, we still want to present those approaches, because they changed the way we are able to verify programs in general, and thus they build-up the base for a lot of modern concepts for verifying concurrent systems.

3.1.1 Floyd-Hoare State Transformer

Robert W. Floyd introduced a concept for verifying sequential programs which is still in use nowadays: Partial correctness of programs [5]. His main concept is to pre-define pre-and postconditions for the program which needs to be verified. Besides, certain control-points/states in the program need to be annotated with assertions. Every time the program reaches one of those control-points the assertions need to be checked and evaluated to be true, otherwise the program is false and needs to be rechecked. By using the pre-/postconditions, as well as the control assertions, the termination of the program is also considered automatically. Furthermore, since the proof of a program is made up of several smaller proofs ($\rightarrow$control-points), it also considers programs with arbitrary control structure [10]. The complexity of verifying a program using this technique lies in $O(n)$ when the program has n statements.

Tony Hoare later took Floyd's approach and adapted it into a logical framework - This method was then later called *Floyd-Hoare-Method* [10]. Hoare's new approach consists of one main formula $P\{S\}Q$, which means that if the assertion P is true before the initiation of section S then the assertion Q will be true after S's termination. Thus, P is called the precondition and Q is called post-condition. Hoare here does not consider termination, and thereby we do not have proof for total correctness. For total correctness, we need partial correctness including termination [12]. But at this stage, the main focus was on partial correctness to ensure safety.

3.1.2 State-Based Verification by Ashcroft

Following the Floyd-Hoare-Method was an approach introduced by Edward Ashcroft - A generalization of Floyd's method of proving partial correctness in sequential programs. Ashcroft maintained the idea of setting up control points and assigning certain assertions to them, which should always resolve to true if the program is at this certain point. The big difference from Floyd's approach is that the control of the program can be at multiple control points at the same time - thus he introduced two new operators called the *fork*- and *join*-operator. Additionally, a new general assertion needs to be made: The annotations of all control points now need to be seen as a one global invariant. Furthermore, executing each statement of the program needs to leave this invariant

true [10]. Including those new operators and the new assertion, Floyd's method now can handle proofing partial correctness for concurrent programs.

Include Graph of comparing floyd and ashcroft

3.1.3 Generalization of Hoare by Owicki and Gries

Owicki and Gries adapted the approach given by Hoare to adapt it to concurrent programs. To deal with several program-parts executed at the same time, he introduced *cobegin*-statements. Besides, the proof rule Equation 1 needed to added to Hoare's logic [8].

$$\frac{\{P_1\}S_1\{Q_1\},\ldots,\{P_n\}S_n\{Q_n\}}{\{P_1\wedge\ldots\wedge P_n \text{ cobegin } S_1\}\|\ldots\|S_n \text{ coend } \{Q_1\wedge\ldots\wedge Q_n\}} \quad \text{provided } \{P_1\}S_1\{Q_1\},\ldots,\{P_n\}S_n\{Q_n\} \text{ are interference-free [8]} \tag{1}$$

"The Owicki-Gries method is really a generalization of Floyd's method in Hoare's clothing"(Leslie Lamport, [10]) Owicki and Gries took the syntax of Hoare, used the basic idea of Floyd, and introduced their *cobegin*-statements for proofing concurrent programs. One drawback of this method is that they refused to use a global invariant, by just simply adding auxiliary variables which store the control information - Thus, this method only involves local assertions. Furthermore, by assuming that a program has n statements than the Owicki-Gries method has $O(n^2)$ verification conditions which need to be held. At Floyd's method there would only be $O(n)$ [10].

3.2 Temporal Logic by Pnueli

Maybe not write this section; not neccessary

3.3 Unity by Chandy and Misra

Jayadev Misra and K. Mani Chandy took the approach of Owicki and Gries on step further, they tried to unconfuse the program's control structure of the Owicki-Gries algorithm by introducing a new pseudo-code programming language called *Unity*. Unity is expressed in terms of Dijkstra's *do*-structure [10], an example of this structure can be seen in Listing 2. A Unity program consists of three main sections: declare-section, initially-section, and assign-section. The declare-section is for declaring all variables needed in the program. The initially-section is for prescribing initial values for these variables. And the assign-section is for listing all statements of the program. The special point about this language is that the execution of a Unity program consists of an infinite amount of steps, which means it starts at any point which satisfies the described initial conditions, mentioned in the initially-section, and runs indefinitely [7]. With this approach, Misra and Chandy wanted to create a language that focuses on what happens inside a program and that the execution is correct, and not where and when it is executed [7].

Furthermore, they introduced the *Unity-logic* to prove Unity programs. It is a restricted form of temporal logic which does not allow nested temporal operators and is based on proving an invariance and leads-to properties [10].

Listing 2: Dijkstra's **do** Structure

```
do P1 → S1 | P2 → S2 | ... | Pn → Sn od
```

3.4 Verification by Specification

A different approach, then the *State-Transformer*-approaches, are approaches which are based on constructing specifications for the concurrent algorithms. The reason for doing so is that proving invariance and leds-to properties are not enough, as mentioned by Lamport [10]. Therefore, we can only ensure correctness for proved properties by express them more abstractly, in addition to the concrete proofs. But we will also see that verification by specification cannot stand on its own to prove concurrent programs. In this paper the *Axiomatic Specification*(subsubsection 3.4.1) and the *Operational Specification*(subsubsection 3.4.2) will be shown.

3.4.1 Axiomatic Specification

The verification of concurrent programs by axiomatic specification means constructing a written list of properties that should be held when executing the program. This list should include pre- and postconditions, as well as assertions that should be describing the main algorithm. In particular, it should look the following: If specification A is held and the assertions of A imply the properties of specification B, then B is implemented by A. This "rule" should be held for all property specifications. This is also called *logical implication*. The big advantage but also disadvantage of this specification type is that it only says what the program should do, but not how. Thus, the implementation of an algorithm which is proofed by an axiomatic specification is still a barrier to cross. Another open question of this verification method is how to write down these properties. One way, for example, would be using the Unity toy language described in subsection 3.3. But there we have the problem that it is not expressive enough and that some new auxiliary variables need to be added - but those should not be added in the final implementation of the program and according to this the verification by axiomatic specification can only be seen as a semi-formal method. In general, we will see at the examples that it is hard to figure out if the specification covers 100% of what the program should and should not, and is thus nearly impossible - or as Lamport states: It is just for convincing yourself and hopeless for a program with a significant degree of complexity [1].

Now we will have a closer look at axiomatic specification using Lamport's bakery Algorithm. An axiomatic specification can be found in Listing 3.

Listing 3: Axiomatic Specification for Lamport's Bakery Algorithm. Can also be found at https://github.com/JeremyHarisch/seminar_software_security_analysis/blob/main/AxiomaticSpecification/lamports_bakery.spec

```
Pre-Conditions:
{id >= 0}
{count = 0}
{countGoal > 0}
{numberOfThreads >= 1}
{len(choosing) = numberOfThreads}
{len(ticket) = numberOfThreads}

Conditions while execution:
// id_{x} gets new ID for Thread X
{0 <= x < numberOfThreads} => {id_{x} < id_{x+1};
    id_{x} >=0}
// Getting new Ticket for Thread X with ID
{choosing[id_{x}] = true} => {ticket[id_{x}] =
    max(ticket)+1 && choosing[id_{x}] = false;
    max(ticket) > 0} = A_{1}
// Make sure no other Thread is getting new Ticket
    at this time
A_{1} => {choosing[id_{x}] = false; j = 0; j <
    numberOfThreads; j != id; choosing[j] ==
    false;} => A_{2}
// Condition for continuing run on Thread X with ID
A_{2} => {ticket[j] != 0 && !(ticket[id_{x}] >
    ticket[j] || (ticket[id_{x}] == ticket[j] &&
    id_{x} > j))} => A_{3}
// Trade section
A_{3} =>{count = count + 1; count <= countGoal;
    ticket[id_{x}] = 0}

Post-Conditions:
{count == numberOfThreads * countGoal}
```

3.4.2 Operational Specification

The specification of concurrent programs using operational specification switches some things up. Instead of describing what the program should do, it shows how the program should do the execution. This is done by creating a specification in form of an abstract program using an abstract programming language. An operational specification consists of three main parts: A set of possible states, next-state relations, and fairness-requirements [1]. The next-state relations consist of relations describing which state can be reached from which in one step and those are partitioned into separate program actions. The fairness requirements are then described in the actions from the next-state relations.

Operational specifications have the advantage that it can be easily understood by programmers because normally they are familiar with a kind of pseudo-code [1]. But this also shows us a downside of this verification method, because some programmers tend to take it word-for-word which means they just force the program to do what the abstract code does. Furthermore, it can happen that programmers use language-specific operations that cannot be verified by an abstract program easily because every abstract program needs to be proven as well. And because of this reason, using operational specifications can lead to double the work. But an advantage of operational specification is that prooving an abstract program π_2, which is based on an abstract program π_1, is rather straight forward, by doing the following [1]: Every possible initial state of π_1 is a possible initial state of π_2. In addition, π_2 needs the same next-state relations as π_1 for the same initial states. And finally the fairness requirements of π_1 need to implement the fairness requirements of π_2, how this is done depends on how the fairness requirements are defined.

Listing 4: Operational Specification for Lamport's Bakery Algorithm. Can also be found at https://github.com/JeremyHarisch/seminar_software_security_analysis/blob/main/OperationalSpecification/lamports_bakery.spec

```
Possible States:
{Start, Lock, Waiting, Trade, Unlock, End}

Next-State-Relations:
- {Start|init()} => {checkGoalReached()|Check}
- {Check|} => {lock()|Lock}
- {Check|} => {|End}
- {Lock|} => {waiting()|Waiting}
- {Waiting|} => {enterTrade()|Trade}
- {Trade|leaveTrade()} => {|Unlock}
- {Unlock|} => {|Check}

Actions:
@Start-Main
def init():
    count = 0;
    countGoal = 500;
    for i in range(len(threads)):
        threads[i].startAndRunThread();

@Thread-Start
def checkGoalReached():
    if count == countGoal:
        trigger(end)
    else:
    count ++
        trigger(lock)

def lock():
    this.choosingTicket = true;
    this.ticket.id = getNewId();
```

```
    this.choosingTicket = false;

def unlock():
    this.ticket.id = 0

def waiting():
    for thread in threads and thread != this.thread:
        // Waiting for thread to choose new Ticket
            Number
        while thread.choosingTicket: { }
        // Waiting until it is the turn of this
            threads ticket
        while thread.ticket.id != 0 and
            (this.ticket.id > thread.ticketid.
        or (this.ticket.id == thread.ticket.id &&
            this.id > thread.id)) { }

def enterTrade():
    /* Include actions when entering Trade-State*/
    pass

def leaveTrade():
    /* Include actions when leaving Trade-State*/
    pass
```

3.5 Verification using Flow Graphs

Now, we are going to have a look at approaches which are using completely different technologies, in particular *Flow-Graphs*. In the following, we will see a state-based approach using branching-time temporal logic and Inductive Data Flow Graphs which try to verify concurrent programs using the programs data flow.

3.5.1 State-Based Approach using Branching-Time Temporal Logic

The following approach is based on the main idea of state-transformers, but with the main difference that we now have a program tree of possibilities instead of a set of sequences. Talking about the tree of possibilities: Meant by this is that the program will be drawn as a graphical tree, which mostly relies on branching-time when possibilities are set while the execution of the program. Because this is a very basic description and not easy to understand, we will have to look at a short example before we will go to the bigger examples. In this Figure 2 we can see three different graph which have the same execution set: $\{(a,b,c),(a,b,d)\}$. But there are differences when the decision is made which of the execution sets is being chosen. The graph on the left shows us the most 'natural' way of deciding which set will be chosen. There only at the point where the execution path differs from another execution path, the decision needs to be made. This means if this relies on i.e. a user input, the program always runs a and b, and then the user chooses if he/she wants to trigger c or d. When

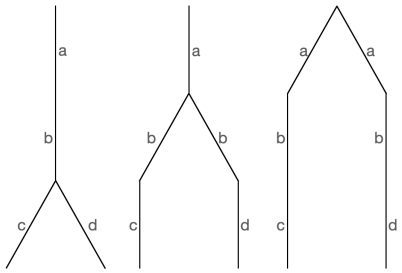

Figure 2: State-Based Verification using Branching-Time

using the graph in the middle of the figure, a will always be executed then the decision needs to be made if $\{b,c\}$ or $\{b,d\}$ will be executed. While on the right graph, the decision needs to be made before the main program actually starts. If sequence-based approaches would be used, then this can not be considered because they cannot distinguish between the three graphs [10].

Since this verification approach suits better to more interactive algorithms, we will not demonstrate the two examples using the branching-time temporal logic. Furthermore, we started to implement verification for the two examples, but thus it an approach for smaller interactive programs, we did not manage to come up with a valid solution.

3.5.2 Inductive Data Flow Graphs

The next approach we are going to have a look are the *Inductive Data Flow Graphs* (short: *iDFGs*). An iDFG is a set of concurrent program traces and shows their dependencies between each other. Since in other approaches space required for proofing correctness of concurrent programs grows exponentially in the number of used threads, this approach tries to prove them in a more space-efficient way [2] - Which means it tries to solve the problem of verifying concurrent programs in a polynomial-size in comparison to the number of data dependencies, by only representing the data flow of the program and abstracting away everything else using efficient space exploration techniques. Thus, the proof only grows when more data dependencies occur, and not when the number of threads used in the program grows.

This approach also uses the Floyd-Hoare annotation of control-flow-graphs and the Hoare-Triples for each node in the iDFG. To prove the validity of such annotations in an iDFG different techniques will be used, but mainly it will be based on a fixpoint-iteration, while the construction of an iDFG. To construct an iDFG is straight forward by following certain steps: First, you need to pick a program trace τ, from the program P, and then check if this program trace is valid. If this trace is not correct, then we can directly tell that the program P is incorrect and can not be validated. But if the trace τ is correct then we construct an iDFG G_τ from it, we will have a look later on how to do that. Then you take G_τ

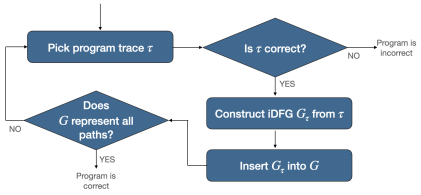

Figure 3: Construction-Algorithm of an inductive Data-Flow-Graph

and insert it into the main iDFG G. If τ was the first trace you picked from the program, then $G = G_\tau$ in this step. In the next step, you check if G represents all paths from the program, if it does then the iDFG is complete and the program is proved correct. But if it does not contain all the traces of P then we make a new iteration of the constructing algorithm and start at the point where we pick a program track again. This constructing algorithm can also be seen in Figure 3.

To construct an iDFG from a program trace τ we need to look at all data dependencies in τ, those will be expressed by the edges of the iDFG. All the data dependencies which do not modify data displayed in the end state or do not affect them in general, they will be abstracted away. Furthermore, by constructing an iDFG from τ we directly need to prove it before we are going to insert it into G. Let us have a look at a small example program-trace τ_{ex}:

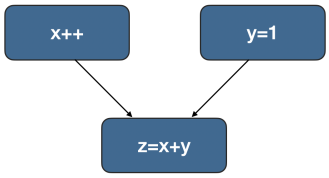

Figure 4: Example program trace τ_{ex}

The corresponding Hoare triple as a proof for this trace τ_{ex} would be:

$$\{x \geq 0 \wedge y = 1\} x{+}{+}; y = 1; z := x * y \{z > 1\}$$

This triple was created from the following three Hoare triples:

$$\begin{array}{ccc} \{x > 0 \wedge y = 1\} & z := x + y & \{z > 1\} \\ \{\} & y = 1 & \{y = 1\} \\ \{x \geq 0\} & x{+}{+} & \{x > 0\} \end{array}$$

To conclude the proof we need the following stability triples:

$$\begin{array}{ccc} \{x \geq 0\} & y = 1 & \{x \geq 0\} \\ \{\} & x{+}{+} & \{y = 1\} \end{array}$$

And if we then compose the last four triples together we get the following, which then leads us to the main Hoare Triple at the beginning.

$$\begin{array}{ccc} \{\} & y = 1; x{+}{+} & \{y = 1\} \\ \{x \geq 0\} & x{+}{+}; y = 1 & \{x > 0\} \end{array}$$

And now using the constructed Hoare-triples we will construct the final iDFG for trace τ_{ex}:

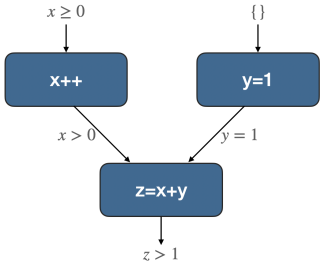

Figure 5: inductive Data-Flow-Graph for τ_{ex}

If τ_{ex} would be a program trace taken of a bigger program, then we now would need to merge this iDFG into the other already constructed ones. For this, we can use different technologies, but this would go out of the scope of this paper.

4 Practical Approaches to Verify Concurrent Programs

In this section, we want to have a closer look at already implemented programs/algorithms to verify concurrent programs.

4.1 Chalice

Chalice is an implemented experimental language to define specifications and verify concurrent programs [3]. This language is mainly based on a permission model [3], which means giving, transferring, or deleting permissions to the processes of the program. These specific permission define which thread is allowed to do which operations at which data. Thus, it deduces the upper bounds of modifiable data of each thread and guarantees the absence of data races [11].

In a Chalice program, developers need to make certain annotations which the Chalice verifier then checks that those are never violated during execution. Those annotations can be pre-and postconditions for specific program parts, like functions, constructors, or destructures, as well as they can be permission checks or transformations. How these annotations look will, will be shown in the two examples at the end of this section.

Permissions can be held by *activation records*, which are calls of methods inside the program. In particular, at the beginning of a program, there is only one activation record existing

for the *Main* function of the program. When a new object gets created, then the activation record which created it gains the permission for this object. So, if some other activation record wants to get access to this new object, it needs to borrow the permission of it and later on, if needed, give it back. Another method to gain the permission of an object is by calling a method that returns the permission of the object to the caller. And only if the program runs through completely without having a permission error, then the program is successfully verified by Chalice.

Listing 5: Chalice Verification Code for Lamport's Bakery Algorithm. Can also be found at https://github.com/JeremyHarisch/seminar_software_security_analysis/blob/main/Chalice/lamports_bakery.chalice

```
// Very simplistic implementation of Lamports
    Bakery algorithm
// In here two threads are counting up to 1000
    together (500*2)
// Which menas they share the counter and lock each
    other up, using a lock and Chalice specific
    permissions

class Bakery {
   var thread_id: int;
   var choosing:int;
   var countToThis: int;
   var count: int;
   invariant acc(countToThis) && acc(count) &&
       acc(thread_id)

  method Bakery(id: int) {
     thread_id := id
  }

  method Main() {
    call Init();
    fork t1 = bk1.run();
    fork t1 = bk1.run();
    join tk0; join tk1;
  }

  method Init()
  ensures acc(bk1) && acc(bk2)
  {
    countToThis := 500;
    count := 0;
    var bk1:= new Bakery(0);
    var bk2:= new Bakery(1);
    bk1.choosing := 0;
    bk2.choosing := 0;
  }

  method run()
  requires acc(thread_id)
  ensures acc(thread_id)
  {
      while (count <= countToThis) {
          call countUp(thread_id);
      }
  }

  method countUp()
  requires acc(count)
  ensoures acc(count){
      lock(thread_id);
        count := count+1;
      lock(thread_id);
  }

  method lock(id: int)
  {
      if (id == 0) {
          bk1.choosing := 1;
          while (bk2.choosing == 1) { }
      }
      else {
          bk2.choosing := 1;
          while (bk1.choosing == 1) { }
      }
  }

  method unlock(id: int)
  {
      if (id == 0) {
          bk1.choosing := 0;
      }
      else {
          bk2.choosing := 0;
      }
  }
}
```

5 Conclusion

References

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Co., Inc., USA, 1991.

[2] Andreas Podelski Azadeh Farzan, Zachary Kincaid. Inductive data flow graphs. 2013.

[3] Microsoft Corporation. Chalice.

[4] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. 1974. https://doi.org/10.1145/361179.361202.

[5] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[6] J. Fuegi and J. Francis. Lovelace babbage and the creation of the 1843 'notes'. *IEEE Annals of the History of Computing*, 25(4):16–26, 2003.

[7] Misra Jayadev. A foundation of parallel programming. 1988.

[8] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 311–323, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. 1974.

[10] Leslie Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, pages 347–374, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[11] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. pages 195–222, 01 2009.

[12] Amir Pnueli Zohar Manna. Axiomatic approach to total correctness of programs. *Acta Informatica*, pages 243–263, 1974.