Leveraging Symbolic Execution on Event-based Languages to Detect Data-Races

Distributed Systems

Group:
ds106f20

Supervisors:
René Rydhof Hansen
Danny Bøgsted Poulsen

June 3, 2020

Department of Computer Science
Aalborg University
http://cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Leveraging Symbolic Execution on Event-based Languages to Detect Data-Races

Theme:
Distributed Systems

Project Period:
Spring 2020

Group:
ds106f20

Participants:
Casper M. Grosen
Jacob A. Svenningsen

Supervisors:
René Rydhof Hansen
Danny Bøgsted Poulsen

Pages:
40

Date of Completion:
June 3, 2020

Abstract:

Concurrency can be used to make software implementations more responsive. However, concurrency also makes program analyses much more difficult and can introduce bugs which can cause two executions with the same inputs to give different outputs. Symbolic Execution has shown that it can be used to analyse a program and detect bugs deep in the code which can only be reached with specific inputs. However, this has been done mostly on sequential programs, not concurrent ones.

We explore how to leverage the strengths of symbolic execution to determine if a program contains data-races. This is accomplished through transforming an input program into a control flow graph, which contains information of all possible assignments in concurrent threads. By using this design, the symbolic execution generates some invalid models as it is an over approximation, and thus given that exact model, it is not always possible to verify if the detected data-race is a false positive or not.

The thesis thus also discusses how this can be improved, and presents an improvement for the symbolic execution, which has shown to be quite effective for concurrent programs with two threads. For concurrent portions of code, the discussion also mentions potential mitigations and extensions which can help improving the symbolic execution to a point, where a concrete execution should not be necessary to verify that the detected data-race can indeed occur.

Summary

Testing is hard and automated testing has made thorough testing more feasible. Automated testing can however be difficult in some areas especially when concerned with complex programs. Concurrency is a way to make some systems faster and more responsive, and is useful when many different inputs can happen simultaneously. But concurrency can make otherwise simple programs complex, and can introduce errors that are hard to detect, like data-races. Embedded systems such as IoT devices often utilises concurrency to process numerous different tasks a bit faster, like retrieving sensor readings from different sensors. To make the system more reactive to new input it can use techniques such as events. This can improve power consumption and can be used to control when tasks should be performed, like altering the speed of fans based on temperature readings, comparing new readings to previous read values. Being a reactive system enhances the complexity of an already complex system, since the program can have many different and unforeseen execution paths, and halting or enabling different paths based on an external environment is the cause for this added complexity. This complexity makes the program harder if not impossible to fully test manually. The level of complexity is hard to handle for automated testing as well since these types of programs can have infinitely many different program paths. In order to discover all possible data-races, all of these paths need to be covered by the analysis. Most automated testing techniques have difficulties handling infinite program states or paths.

We propose leveraging symbolic execution to handle this complexity and analyse what different program paths might lead to unwanted or erroneous program behaviour. While symbolic execution cannot handle infinite program paths, we have found some techniques to alleviate these problems. By over-approximating the possible program paths and restricting loops, we can handle this high level of complexity. Symbolic execution can handle symbolic variables and uses these to find values that yield wanted results. This is important in over-approximating program paths as a single symbolic variable represent multiple different values, which translates to multiple different program states. We leverage these symbolic variables to find data-races. Data-races are typically not easily detected, since they depend on the execution path. Since concurrent and reactive systems can have infinitely many different execution paths, analysing every path is infeasible. Leveraging symbolic variables in this way does not remove the complexity as the programs can still have infinite paths, so we limit the number of loop iterations to avoid the halting problem,reducing the number of program paths from infinite to a finite number. This makes it feasible to perform an analysis that covers the program. Since we analyse an over-approximation, it is possible to encounter infeasible program paths, so we perform conflict analysis to remove some of these infeasible program paths. For some concurrent programs, we can reduce the number of analysed program paths to only include the feasible program paths; the possible paths that can occur during a concrete execution. We perform a limited program simulation to verify some of the suspected data-races, and we propose an extended analysis to completely verify or disprove all the suspected data-races.

Glossary

EndFork The block which merges multiple threads following the execution of a fork statement

EndIf The block which merges the flow after the execution of an if statement and its body

Fork The block which contains a fork statement, spawning multiple concurrent threads

Acronyms

CCFG Concurrent Control Flow Graph

CELE Concurrent and Event-based Language for Embedded systems

CFG Control Flow Graph

CSSA Concurrent SSA

DF Dominance Frontier

SMT Satisfiability modulo theories

SSA Static single assignment

Contents

Glossary **ii**

Acronyms **iii**

1 Motivation **1**

2 Theory **2**

2.1 Concurrency . . . 2
2.1.1 Data-race . . . 2
2.1.2 Events . . . 3
2.2 Basic Blocks . . . 3
2.3 SSA . . . 5
2.3.1 Concurrent SSA . . . 8
2.4 Symbolic Execution . . . 14
2.4.1 Path . . . 15
2.4.2 Loop Unrolling . . . 16
2.5 Summary . . . 18

3 Design **19**

3.1 Constraint Building . . . 19
3.2 Finding Possible Data-races . . . 28
3.2.1 Verifying Suspected Data-races . . . 29
3.3 Summary . . . 30

4 Discussion **31**

4.1 Removing False-Positives . . . 31
4.1.1 Limiting Interleavings . . . 32
4.1.2 Interpreter . . . 37

5 Conclusion **40**

Bibliography **41**

Appendices **42**

A Syntax and Semantics **43**

A.1 Abstract Syntax . . . 43

A.2 Semantics for I/O 43
A.3 Semantics for Expressions 44
A.4 Semantics for Statements 45

Chapter 1

Motivation

Symbolic Execution has shown that it can be effective in detecting program misbehaviour such as accessing memory after it has been freed, and accessing memory that has not been allocated. While symbolic execution can find these problems in sequential programs, these problems are present in concurrent programs as well. Concurrent programs are prone to data-races, and some of these data-races can have severe consequences that leads to program misbehaviour, such as accessing an array outside its bounds. Should any program at some point introduce such a bug, debugging it or performing regression testing in order to find when this bug was introduced, can be difficult, as the problem may only ever surface with a very specific input, and only if the right interleavings are performed. With smart appliances becoming increasingly popular and a demand for systems to be responsive, it is only a matter of time before these hard-to-detect problems will become more widespread.

This thesis thus explores the concurrency problem, and makes use of symbolic execution in order to detect data-races in reactive programs. This thesis is rooted in the work performed by Grosen and Svenningsen (2019). We perform symbolic execution on a subset of Concurrent and Event-based Language for Embedded systems (CELE), a language which is intended for concurrent embedded systems. We choose to avoid arrays, which are part of CELE, as this would complicate the symbolic execution and will not affect the amount of data-races that we can discover.

The symbolic Execution proposed in this thesis is more traditional than the one proposed by Grosen and Svenningsen (2019). We choose to do static analysis of the source code and then generate constraint sets based on that. But in (Grosen and Svenningsen, 2019), the construction of constraints would have been performed dynamically, and partial constraint sets would be constructed and solved while executing statements, all during the symbolic execution. We chose the static approach to lessen the time needed to find data-races. The result is a less accurate analysis, due to over-approximation in the static analysis. We aim to find data-races, a subset of problems which Grosen and Svenningsen (2019) attempted to find, as the less accurate analysis makes it harder to detect problems that are not apparent in the data. The flow-sensitive analysis proposed by Grosen and Svenningsen (2019) can detect more problems, but the analysis has a larger time complexity, due to simulating all possible execution paths. That analysis would have a worst-case complexity of $O(n^m)$ where n is the number of threads and m is the number of concurrent instructions.

In order to perform static analysis, the program would first have to be transformed in such a way that makes static analysis easier to perform. We thus first have to delve into some theories regarding this, as well as understanding what problems concurrency and events can lead to.

Chapter 2

Theory

We explore concurrency and events, and what problems these can lead to. Moreover, in order to perform static analysis, we explain some widely utilised intermediate representations, namely a Control Flow Graph (CFG), and such a graph being on Static single assignment (SSA) form. Since SSA is not powerful enough to describe concurrent portions of code, we present the Concurrent SSA (CSSA) form, first introduced by Lee, Midkiff, and Padua (1997). Finally, we explain symbolic execution and why we choose to perform loop unrolling.

2.1 Concurrency

Concurrency is when a program has multiple threads with interleaved executions. Several issues can arise when interleaving threads. Some additional problems caused by interleavings can occur in parallel programs, but we will only look at concurrency in this study. We leverage symbolic execution in order to detect points of interests where data-races can occur due to thread interleavings.

2.1.1 Data-race

Definition: A data-race is when a variable at a given point of execution can have different values depending on the interleaving of threads.

If the order of execution between threads can influence the value being stored or used in the system there is a data-race, data-races is only affected by thread interleaving. Data-races can have severe consequences for a system, like causing a system shutdown. If we have a controller that controls the state of the system such as running certain tasks as well as shutting down the system, then a variable might get misused, causing the controller to shut down the system. This might appear random to the user and even the testing team, if this is caused by interleaving threads. This can be hard to detect as it could appear as a normal system shutdown. The results of a data-race can also be less severe. Given a system that reads a value from a potentiometer and uses this value to determine the brightness of a light bulb, the system might occasionally use old data to determine the brightness. However, if the brightness is updated frequently the user never experiences this misbehaviour.

Based on the system and its context the issues that arise from data-races can have different levels of severity. Since we cannot assess the severity of the resulting behaviour we choose to alert the developer of all detected data-races. We cannot determine the consequences of a data-race in the external environment because we choose not to model that environment. We can however determine what variables are affected and where the data-races occur, which we report to the user.

2.1.2 Events

Some embedded systems run on batteries, hence power consumption can be a big issue. In order to conserve power, developers choose to put the system to sleep for period of time so the system only request new sensor readings sparingly. This can make the system less responsive however, which is something that can be important in many systems. To get both responsiveness and conserve power, developers can utilise events, which allows the system to sleep until some criteria is fulfilled, like a new sensor reading being available or completion of a task. Events can be a powerful tool, but can be restricted without support for hardware interrupts.

Using events affects the complexity of the control flow. Concurrent systems is more complex with events, as thread interleavings can be dictated by external factors. We choose to handle events because many modern embedded systems are dependent on them.

2.2 Basic Blocks

In order to detect data-races, we require a way to represent a given program in its source-code form. Compilers usually represent the source code as a CFG, where each edge in the graph is a flow edge, and connects two basic blocks. A basic block is a block consisting of statements that can be executed in succession without breaking the flow of the program. Since CELE has concurrency, each basic block inside the concurrent `fork` statement can only have a single statement. Splitting each basic block this way allows us to model interleavings between each statement, but not interleavings between each expression evaluation. Moreover, after a branching statement, new blocks are needed for each branch, and an additional EndIf block for merging the program flow. This EndIf block is added to model the merging point prior to evaluating the next statement. The source-code shown in listing 2.1, produces the CFG and basic blocks shown in figure 2.1.

Since each edge is a flow edge, we cannot place an edge between two blocks that are not inside the same thread, despite an interleaving being possible. If we did that, then we would need to place two edges between each block as the CFG is a directional graph and interleavings being possible in both directions. However, this would be misleading, as we could then potentially evaluate block a, followed by block b, then evaluate block a again. Evaluating the same block multiple times is not possible, thus we cannot place flow edges between concurrent threads. Alternatively, we could explicitly encode all possible paths and interleavings, but this would cause the CFG to become enormous and make program analysis more complex and time consuming.

This CFG can be analysed, such as a reaching definitions analysis in order to determine what definition of a variable is used in each statement (Appel, 2004). However, this is inefficient if other analyses has to be done as well, such as detecting data-races. However, the CFG can be transformed into SSA form. A graph on this form makes each reaching definition explicit in the graph, and allows a compiler to quicker perform optimisations.

```
1  a = read(1);
2  b = a + 2;
3
4  fork {
5    if (b < a) {
6      a = a+1;
7    } else {
8      write(2, a);
9    };
10 } and {
11   a = 10;
12 };
13
14 write(2, a);
15 write(2, b);
```

Listing 2.1: Small example program

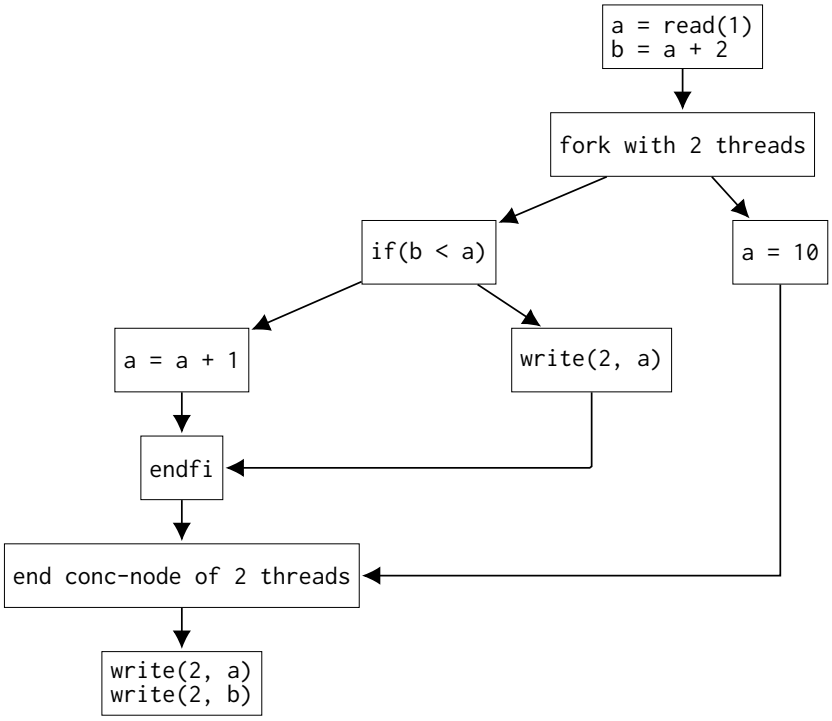

Figure 2.1: CFG of code shown in Listing 2.1

2.3 SSA

When a CFG is on SSA form, each variable is assigned exactly once statically, and any future use of a variable, references the last assignment. SSA captures data flow and was first introduced by Rosen, Wegman, and Zadeck (1988). Since then SSA form has been utilised by compilers to perform optimisations. To use this data flow information, we opt to transforming the program to SSA form.

To transform our CFG of the input program, we construct a Dominance Frontier (DF) to capture the dominance relations between each block in the graph. This frontier contains the blocks where for any block n, n stops being a dominator (Appel, 2004). The DF for a block n for instance, contains the set of blocks reachable from n, which n does not strictly dominate. So if a block n contains a variable definition for variable a, then all blocks in `DF(n)` potentially need to have a ϕ-function for variable a. To create the DF, we first compute the dominance tree using the method proposed by Lengauer and Tarjan (1979). With the dominance tree in hand, we can compute the DF using the algorithm from (Appel, 2004).

Blocks that would usually require ϕ-functions are EndIf blocks, due to these blocks being where flow merges. A ϕ-function is a function which have n arguments, where n is the amount of immediate parents of the block containing the ϕ-function. Each argument is a reaching definition. The value of a ϕ-function thus depends on what flow edge was taken to reach it. Blocks with ϕ-functions thus have to re-define variables so each future use of the variable would reference this definition.

The program we saw earlier in listing 2.1, will look like listing 2.2 once converted to its SSA form.

```
1  a1 = read(1);
2  b1 = a1 + 2;
3
4  fork {
5    if (b1 < a1) {
6      a2 = a1+1;
7    } else {
8      write(2, a1);
9    };
10   a3 = φ(a2, a1)
11 } and {
12   a5 = 10;
13 };
14
15 a4 = φ(a3, a5)
16 write(2, a4);
17 write(2, b1);
```

Listing 2.2: Code shown in Listing 2.1 on SSA form

When we have a CFG, we can use Algorithm 1 to place ϕ-functions at each EndIf block, then use Algorithm 2 to rename each variable. Both algorithms are present in (Appel, 2004), but has been included here for completeness.

Algorithm 1: Algorithm for ϕ-function placement. (Appel, 2004)

Data: A CFG (N, E, n_{start}, n_{exit})
Result: The given CFG with ϕ function at EndIf blocks

```
1  begin
2      for n ∈ N do
3          for a ∈ A_orig[n] do
4              defsites[a] ← defsites[a] ⋃ {n}
5      for a ∈ A do
6          W ← defsites[a]
7          while W is not empty do
8              n ← front(W)
9              W ← tail(W)
10             for Y ∈ DF[n] do
11                 if Y ∉ A_φ[n] then
                       /* insert statement a ← φ(a, a, ... ,a) at the top of block Y.   */
                       /* The φ-function has as many arguments as Y has predecessors */
12                     A_φ[n] ← A_φ[n] ⋃ {Y}
13                     if Y ∉ A_orig[n] then
14                         W ← W ⋃{Y}
```

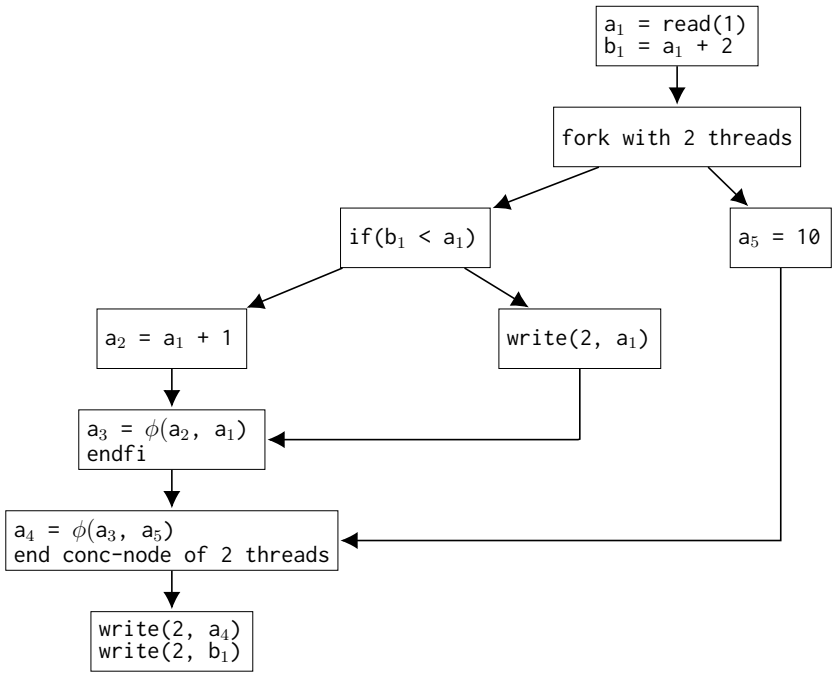

Figure 2.2: The sequential SSA form

Algorithm 2: Computing the sequential SSA-form (Appel, 2004)

```
1  initialisation:
2      for a ∈ A do
3          Count[a] ← 0
4          Stack[a] ← 0
5          push(0, Stack[a])
6  Rename(n)
7      for each statement S in n do
8          if S is not a φ-function then
9              for each use of some variable x ∈ S do
10                 i ← top(Stack[x])
11                 replace the use of x with x_i in S
12         for each definition of some variable a ∈ S do
13             Count[a] ← Count[a] +1
14             i ← Count[a]
15             push(i, Stack[a])
16             replace definition of a with definition of a_i in S
17     for each successor Y of n do
18         Suppose n is the jth predecessor of Y
19         for each φ-function ∈ Y do
20             Suppose the jth operand of the φ-function is a
21             i ← top(Stack[a])
22             replace the jth operand with a_i
23     for each child X of n do
24         Rename(X)
25     for each definition of some variable a in the original S do
26         pop(Stack[a])
```

Applying these algorithms on the CFG through traversal of the Dominance Tree, we will have computed the sequential SSA form. The output from applying these algorithms can be seen in Figure 2.2.

This transformation however, is still not enough to detect data-races. Looking at the graph, it is easy to see that the order of evaluation of the statements `a5 = 10` and `if(b1 < a1)` matters, but not when analysing the code. After all, if a program is on SSA form, then each future use of a variable referencing its definition. Each definition is thus a new variable, meaning a_1 and a_5 are two different variables, despite the inital CELE code snippet clearly showing these two to be the same. The computed form here does indeed only work sufficiently for sequential programs, not concurrent ones. Should a compiler choose to optimise this code using constant propagation, then it would fail to adhere to the semantics of the language. Assume a_1 was assigned the concrete value 5. Then, a compiler in this case could use constant propagation and replace all occurrences of a_1 with 5. This in turn allows the compiler to evaluate the statements `b1 = 5 + 2`, but also the conditional statement `if(7 < 5)`.

Thus the sequential SSA form falls apart once we introduce concurrency. Other transformations attempts to rectify this, which extends the SSA form with the introduction of a π-function. This extended form would transform the CFG into a Concurrent Control Flow Graph (CCFG).

2.3.1 Concurrent SSA

As mentioned in section 2.3, in order to detect data-races, we need to know if a variable is assigned in other threads. If we take a look at figure 2.2, then in the conditional check, we only consider the value of a_1, even though it is possible for that value to actually be a_5. To determine the correct variable for each use, we can transform the CFG into a CCFG. A CCFG is a graph that is on CSSA form (Novillo, Unrau, and Schaeffer, 2000). CSSA was first proposed by Lee, Midkiff, and Padua (1997). It requires the additional placement of π-functions prior to each statement that uses variables which can potentially be redefined in concurrent statements. In order to place these functions, we have to add conflict edges to the CFG. A conflict edge, is an edge from a block containing a definition of a variable, to a concurrent block which uses the same variable. Thus to find all conflict edges, we need to determine if two basic blocks are concurrent. Two blocks are concurrent if they share a common ancestor `fork` statement and if one of the immediate successor blocks - to this common `fork` statement - is an ancestor of one of the two blocks, but not the other.

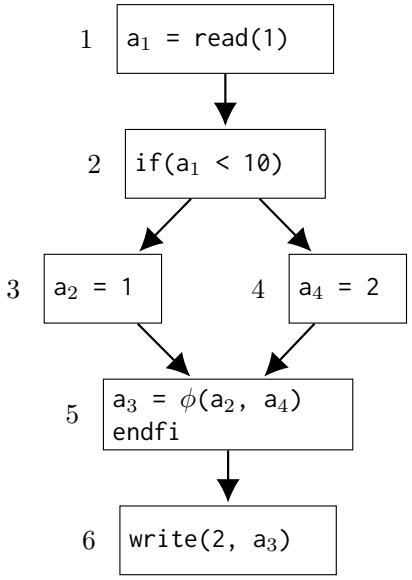

Figure 2.3: Numbered sequential program

To determine if two blocks are concurrent, we choose to number each block. Lets first consider the sequential program shown in Figure 2.3. The first block, as shown, will be numbered 1, then the following blocks will be numbered in ascending order from root to leaf. When the flow diverges, such as when an `if`-statement is encountered, then each path will be numbered from left to right. Once all blocks prior to the EndIf block has been numbered, the EndIf block itself will be numbered and and then will its children.

This means that for `if`-statements the true branch will be numbered first, then the false branch. Once both branches has been named, so will the EndIf block.

To expand this method to concurrent programs we must handle thread numbers and where the threads are spawned. To do this, we choose to model names as a hierarchy. Each time a Fork block is encountered, we reuse the blocks name when naming future blocks between the Fork block and the corresponding EndFork block. These future blocks will be named starting from 1 again, but with a prefix. This prefix is the name of the immediate parent, plus a `T` and a thread number. A thread number which is determined by the flow edge taken from the Fork block.

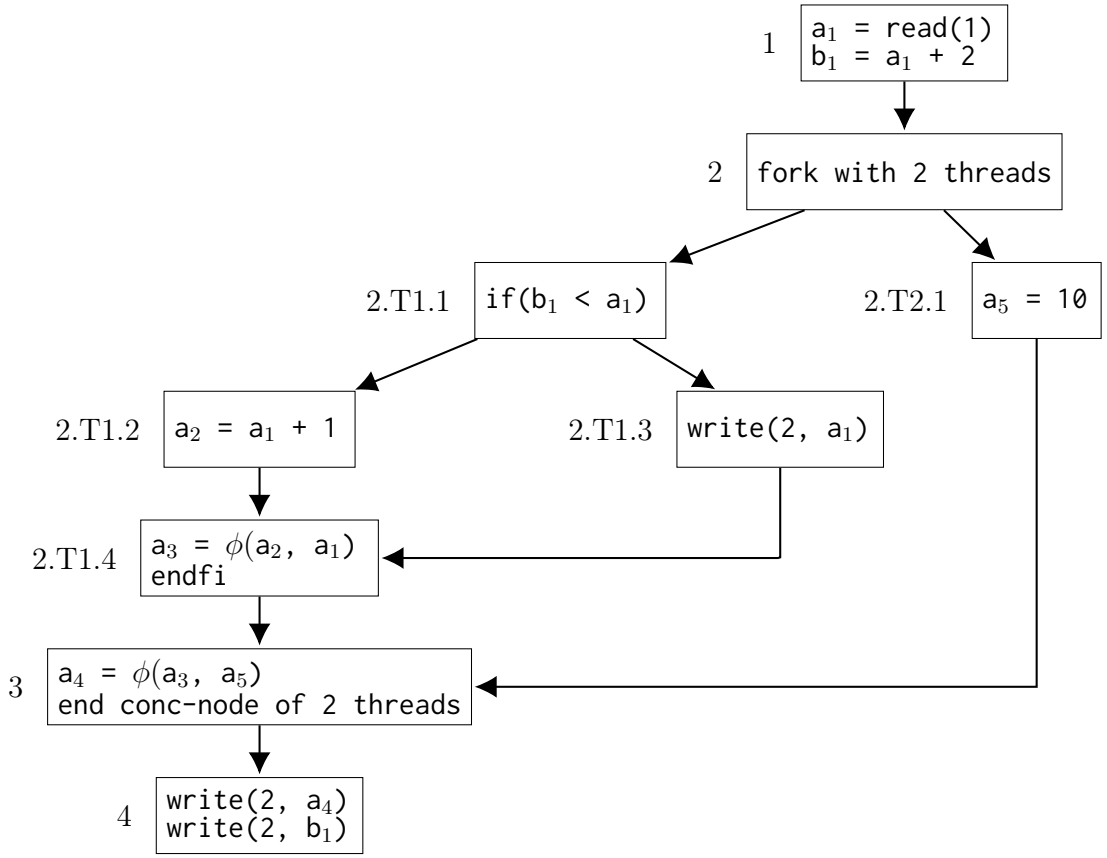

Figure 2.4: Numbered version of Figure 2.2

The T is added to differentiate this number from that of regular basic blocks. Figure 2.4 shows a numbered version of Figure 2.2. These numbers help determine the context of each basic block, which blocks are concurrent and which blocks are executed before others.

Algorithm 3 checks whether two basic blocks are concurrent, which relies on this numbering method.

The algorithm has a few checks to speed up the process when dealing with basic blocks that are obviously not concurrent, then it proceeds to check if basic block a and b are concurrent by checking whether they are part of the same `fork`-statement. For example if comparing blocks 2.T1.2 and 2.T1.3 in Figure 2.4 we check if they share the same Fork block. In both cases that is 2, meaning that the threads are spawned by the same `fork`-statement. Afterwards, the thread numbers are compared, and if the numbers are the same, in this case both are T1, then both blocks are part of the same thread. This then continues until the names no longer are identical. If the number that differs is a thread number, then the blocks are concurrent. If the number which differs are a regular thread block number, then the blocks are sequential. Algorithm 3 implements this.

Consider two blocks n and m, where n is named "3.T1.7.T1.1" and m is named "3.T1.7.T2.1". Using the algorithm, the first subname which are indifferent, is the second-to-last thread num. This shows the two blocks are concurrent.

Algorithm 3: Checks if blocks a and b are concurrent

Data: Two basic blocks a and b
Result: *true* if a and b are concurrent, $false$ otherwise

```
1  concurrent(a, b)
   /* Array of names. Each array does not contain the dot separator, and they have
      as many indices as names                                                    */
2     aNames ← names on a
3     bNames ← names on b
4     i ← 0
5     max ← aNames.length if aNames.length ≤ bNames.length else bNames.length
6     if aNames = bNames then
7        return false            // The two blocks are the same, thus not concurrent
8     if aNames.length = 1 ∨ bNames.length = 1 then
9        return false                              // Either a or b are not concurrent
10    while i < max do
11       if aNames[i] = aNames[b] then
            /* The fork ancestor is identical. We know the next name is a thread
               number, unless one block is a Fork block                           */
12          i ← i + 1
13          if i ≠ max ∧ aNames[i] ≠ aNames[b] then
               /* The thread number differs but they share a fork ancestor, meaning
                  a and b are concurrent                                          */
14             return true
15       else
            /* A fork ancestor or the basic block number differs, meaning the
               blocks are sequential                                              */
16          return false
17    return false
```

To place conflict edges, we use Algorithm 4 defined in (Novillo, Unrau, and Schaeffer, 2000), and included here for completeness.

Algorithm 4: Add Conflict Edges (Novillo, Unrau, and Schaeffer, 2000)

Data: A CFG (N, E, n_{start}, n_{exit})
Result: The CFG with added conflict edges E_c added

```
1 E_c ← ∅
2 for a ∈ N do
3 |   for b ∈ N do
4 |   |   if concurrent(a, b) ∧ conflicts(a, b) then
5 |   |   |   E_c ← E_c ⋃ {(a, b)}
6 E ← E ⋃ E_c
```

Once we have the conflict edges, we can add π-functions in each basic block which has incoming conflict edges.

A π-function in a block b has $n + 1$ number of arguments, where n is the number of incoming conflict edges to b. The first argument is the variable definition found during the sequential SSA transformation. The rest have no particular order, as the π-function is a non-deterministic choice of its arguments.

Once a π-function has been placed, the original usage is added to it, and all uses of it is redefined to match that of the π-function, similarly to what is done when placing ϕ-functions at each EndIf block during the SSA transformation.

We place π-functions using Algorithm 5, which is based on the one defined in (Novillo, Unrau, and Schaeffer, 2000). We modified it because the original algorithm only places a single π-function for each conflicting variable. In our case, this is not good enough as this would not model possible interleavings during the evaluation of an expression. A statement such as "$\mathtt{b}_1$ = $\mathtt{a}_1$ + $\mathtt{a}_1$ + 2;" would only have one π-function for the variable $\mathtt{a}_1$, should there be a conflict. This means both occurrences of $\mathtt{a}_1$ will always have identical values, thus the evaluation of this statement is done atomically. As shown in the semantics of CELE (Grosen and Svenningsen, 2019), a binary expression has its left-hand side evaluated before the right-hand side. The rules for concurrency also shows that an interleaving after applying a single semantic rule, is allowed. Thus statements are not executed atomically. The semantics for CELE has been included in Appendix A for completeness.

With our modified version, we have two π-functions, which means the first occurrence of $\mathtt{a}_1$ can have another value than the second occurrence. Moreover, we also extended it to convert the ϕ-functions at EndFork blocks into π-functions. This is because unlike the value for ϕ-functions being determined by what entry flow edge was taken, the value for π-functions is a non-deterministic choice. Since all the parent blocks for a EndFork block has to be executed before the EndFork block is reachable, only variable definitions inside the body of the `fork` statement, are allowed options. As the value depends on the interleavings taken through the execution of the `fork` statement.

Algorithm 5: Place π functions. Modified version of the algorithm from (Novillo, Unrau, and Schaeffer, 2000)

Data: A CFG (N, E, n_{start}, n_{exit})
Result: The CFG with π-functions added

```
1  count ← 0
2  for b ∈ N do
3      for conflict edge e = (a, b) do
4          v ← variable defined in a
5          if no π-function for v is present in b then
6              for each use u of v in b do
7                  Insert the statement T_v_count = π(v) immediately preceding the last
                    statement of b
8                  π(v)[0] ← u
9                  rename u to T_v_count
10                 count ← count + 1
11         for each π-function p for v in b do
12             d ← conflicting def of v in a
13             push(d, p)
14 for all EndFork blocks b do
15     for all φ-functions p in statements of b do
16         vars ← variable options for p
17         for v ∈ vars do
18             if v is not defined inside the fork statement then
                   // Remove v from vars
19         p ← π(vars)
```

Having added both conflict edges and π-functions to the CFG, we have transformed it to a CCFG, which we can use for detecting data-races, as each use can now also be a value defined in concurrent threads. The transformed CCFG can be seen in Figure 2.5.

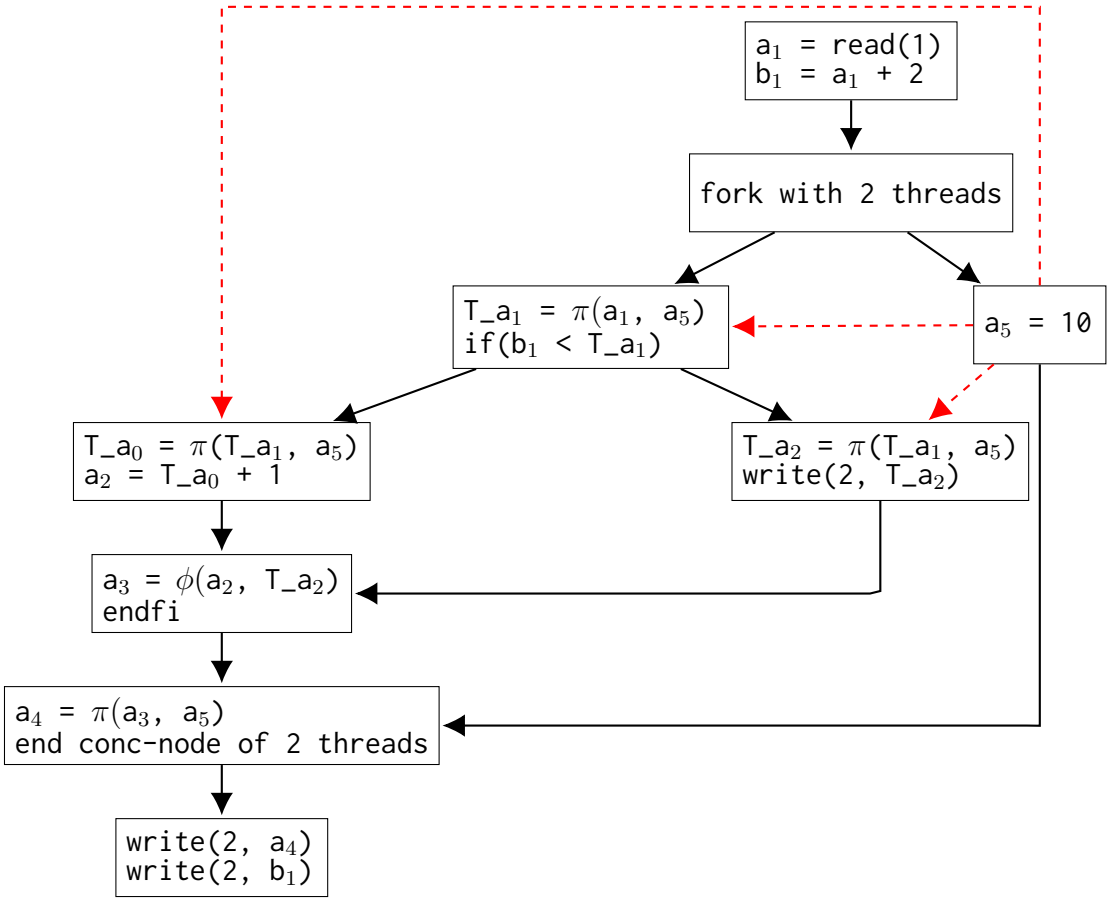

Figure 2.5: a CCFG

2.4 Symbolic Execution

Symbolic execution is a form of program analysis. Symbolic execution is very useful to determine what type of input allows a certain part of a program to execute. During a normal concrete execution, inputs from the environment will have a concrete value, whereas during symbolic execution, this value can be anything. When determining if a certain program part is reachable, the symbolic execution engine will accumulate all constraints encountered on the way to the program part in question, then use these constraints to determine what the values obtained through the environment can be.

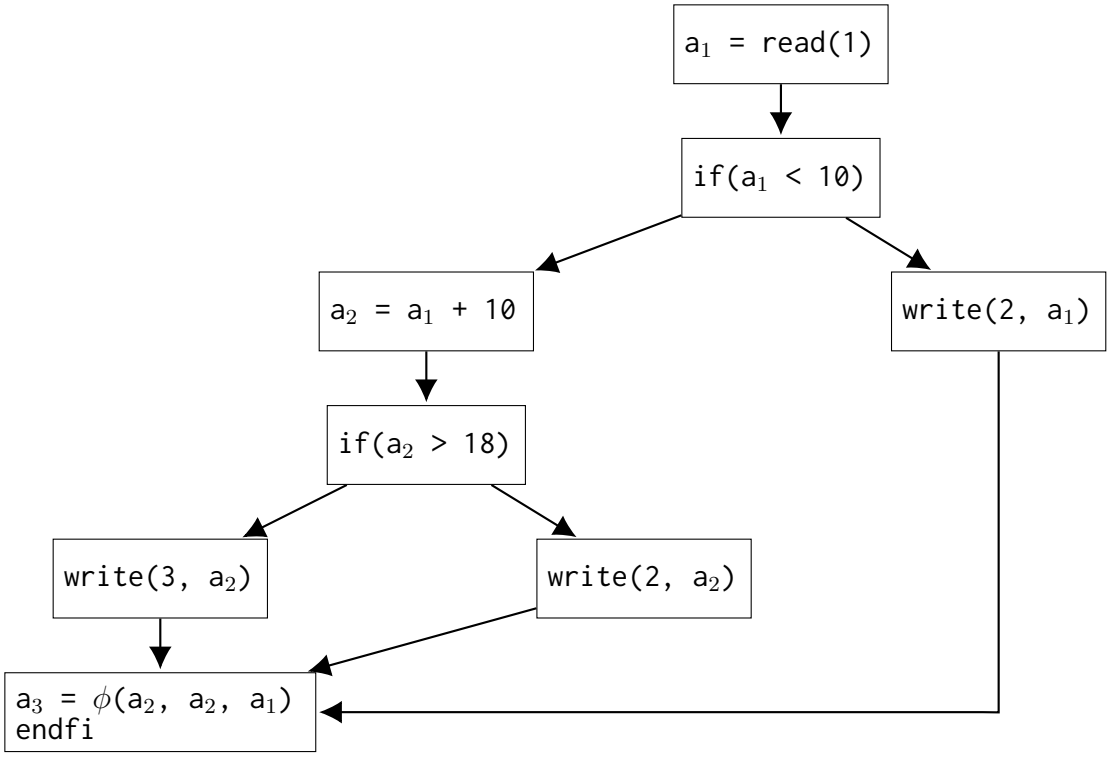

Figure 2.6: CFG containing a nested if-statement

Consider the CFG in Figure 2.6. If we wish to find out if the statement `write(3, a2);` is reachable, we can use symbolic execution. The very first statement `a1 = read(1);` is where symbolic execution and concrete execution differs. Whereas a concrete execution would get an actual value from the environment, such as the integer value `9`, the symbolic execution engine will assign a_1 a symbolic value. Lets call this symbolic value s. We continue symbolically executing this program and encounter the first `if` statement. Here, the engine has two choices for branching. To go along the first branch, a_1 `< 10` has to be true, while this expression has to be false, should the execution continue along the false branch. Continuing like this until we reach the statement we want to see is reachable, that is; statement `write(3, a2)`, we end up with the constraint set:

$$a_1 = s,\ a_1 < 10,\ a_2 = a_1 + 10,\ a_2 > 18$$

This constraint set is also the set which describes the path taken through the program. Thus this constraint set can also be referred to as the path condition. We can ask the constraint solver here for a concrete value for s given this path condition.

For this set to be satisfiable, we can see that $a_2 > 18$ needs to be true, thus the constraint solver can choose to assume that a_2 is 19. Once this is done, lets replace all occurrences of a_2 in the path condition with the concrete value 19:

$$a_1 = s,\ a_1 < 10,\ 19 = a_1 + 10,\ 19 > 18$$

We can then evaluate the second to last constraint: $19 = a_1 + 10$ more easily. For this constraint to be satisfiable, a_1 needs to have the value 9. Having solved that constraint, we can replace a_1 with the concrete value 9, similar to what we did with a_2 previously:

$$9 = s,\ 9 < 10,\ 19 = 9 + 10,\ 19 > 18$$

We see that all constraints still hold, and that s can now only have the value 9. Thus 9 is a possible concrete value that allows a concrete execution to reach the statement `write(3, a2)`. Bear in mind that this does not mean that 9 is the only solution, just that it is one possible solution.

The CFGs being on SSA form also helps constructing these constraints, as each new variable assignment is considered a new declaration. If the CFG in Figure 2.6 was not on SSA form, and each statement was transformed in the same manner as was done above, then the path condition for reaching the same statement, would be:

$$a = s,\ a < 10,\ a = a + 10,\ a > 18,$$

Looking at this constraint set, it is very clear that it is unsatisfiable. `a` cannot possibly both be less than `10` and larger than `10`, and can most definitely not have the same value as itself + 10. The path would then have to redefine each assignment to a variable dynamically. Unlike if done dynamically, the SSA form allows all possible paths through Figure 2.6 to be encoded as a single constraint set, allowing the traversal of the graph to only be done once.

This form of data analysis has shown promises. Cadar, Dunbar, Engler, et al. (2008) developed the tool Klee and showed that it is capable of detecting previously unknown bugs in real world software. Cha et al. (2012) showed that their symbolic execution tool named Mayhem, is capable of detecting vulnerabilities, such as zero-day exploits, in programs where they only have access to the binary code. Chipounov, Kuznetsov, and Candea (2011) showed that symbolic execution can also be used for analysing binary device drivers for Windows and to detect bugs that can cause kernel panics.

2.4.1 Path

As explained, a path-condition is the constraint-set which applies for the current path. Some statements can be reached through multiple paths, such as statements following EndIf and EndFork blocks. Because of this, constructing all possible constraint sets and provide these for a symbolic execution engine, then asking if a certain statement is reachable, is often more feasible, as this means the constraint solver only needs to be asked once.

To determine what path was taken through a given sequential program, each block can have a unique name assigned. The numbers assigned to each block, which was explained in Section 2.3.1, can be used as this name.

We can use this unique name to create a boolean constant, which is `true` if the statements contained in the block was used in the path condition, and `false` otherwise. For a sequential program, this makes it clear what branch of any `if` statement, was taken.

For concurrent programs, this becomes more difficult, as the paths also starts depending upon interleavings. In CELE, interleavings can potentially happen many times during the evaluation of a single statement. For example, during the evaluation of the statement `a = a + b`, it is possible to interleave four times. Once after the evaluation of `a` in the binary expression. Once after the evaluation of `b`. Once more after adding the two values together, and then once more after the assignment has taken place. So for a sequential program, the evaluation of this statement does not branch into multiple paths, but it can branch up to four times if this statement is inside the body of a `fork` statement.

Constructing a constraint set this way to represent all possible paths through a program, is unfeasible as the amount of possible paths explode almost immediately. That is another reason why we transform the CFG into a CCFG as was described in Section 2.3.1. The π-functions can capture possible values from concurrent threads, which allows us to overestimate all possible paths through the program, despite not constructing constraints for all possible interleavings explicitly. This does mean that the symbolic execution can potentially find inputs that allows us to reach locations where data-races can occur, despite this location not necessarily being reachable during a concrete execution. How this can happen, will be described in a later section.

Since we transform the entire program into a set of constraints, we have to unroll our `while` statements as well.

2.4.2 Loop Unrolling

Loops are an Achilles heel for symbolic execution. If a variable in the loop's condition is symbolic, we cannot easily determine how many times a loop will execute. Moreover, as was explained in Section 2.4, a strength SSA provides, is that the constraint building becomes more simple, since variable declarations are done statically, not dynamically. Without knowing how many times a loop executes, variable declarations here would have to be done dynamically. Moreover, since we do not know how many times a loop is executed, the constraint set for the loop can grow indefinitely, thus we are facing the halting problem.

To avoid infinite execution paths, and again to leverage the benefits of SSA we unroll each `while` statement a set amount of times, and check if the loop was left prematurely. This unrolling is done prior to transforming the program into a CFG such that the SSA and CSSA algorithms can be applied on the unrolled construction.

To keep the same semantics for the transformed program we use the CELE `[While]` rule to transform each iteration.

```
1  a = 0;
2  while(true){
3    a = a+1;
4  };
```

Listing (2.3) source program

```
1  a = 0;
2  if(true){
3    a = a+1;
4    if(true){
5      a = a+1;
6      if(true){
7        a = a+1;
8      } else { skip; };
9    } else { skip; };
10 } else { skip; };
```

Listing (2.4) transformed program with max 3 iterations

Figure 2.7: While loop transformation to nested `if` statements, three iterations

Figure 2.7 shows a small program and its loop unrolled "equivalent". In this example we unrolled the loop three times, so the highest number of possible iterations is three. This limits our ability to verify potentially found data-races in a program, because some data-races might only occur after x amount of iterations. Reducing the possible amount of discover-able data-races is severe, but never finding any is unacceptable. We assume that most data-races is discover-able after few iterations, but we acknowledge that some data-races could require a very large number of iterations. If a block contains a potential data-race, but we cannot verify it, then we can still report it as a potential location for a data-race if any of the used variables there has a definition located inside the body of a `while` statement.

We stated that we can check if our symbolic execution exits loops prematurely. We do this by introducing a boolean variable for each unrolled loop.

```
1  a = 0;
2  if(true){
3    a = a+1;
4    if(true){
5      a = a+1;
6      if(true){
7        a = a+1;
8      } else { skip; };
9    } else { skip; };
10 } else { skip; };
```

Listing (2.5) unrolled loop program, before variable introduction

```
1  a = 0;
2  early_exit = false;
3  if(true){
4    a = a+1;
5    if(true){
6      a = a+1;
7      if(true){
8        a = a+1;
9        if(true){
10         early_exit = true;
11       } else { skip; };
12     } else { skip; };
13   } else { skip; };
14 } else { skip; };
```

Listing (2.6) early_exit variable added

Figure 2.8: Add variable to help determine if the unrolled loop is exited prematurely

As shown in Figure 2.8, we introduce a boolean variable and set it to false right before the first iteration, and at the last iteration, we add an `if` statement which checks if the loop condition is still satisfiable. In case it is, we assign `true` to the `early_exit` variable. This is sufficient because if another iteration can be performed after our last iteration then there is at least one more iteration that is possible. Knowing whether we exit loops early is useful when determining whether our symbolic execution has covered all program paths, because if we cover all possible concurrent thread interleavings and we never exit a loop prematurely, we can determine that we have covered all program paths, and provide the user with this information.

2.5 Summary

Having explained the theories we utilise to perform symbolic execution in order to detect potential locations for data-races, we can present a design which is capable of generating models which are an over approximation, but can find every location where a data-race potentially can occur.

Chapter 3

Design

Having delved into theories used for performing data analysis, as well as concurrency and events, we can explain how a constraint set modelling all potential paths through a given program, can be constructed.

Instead of developing a Satisfiability modulo theories (SMT) solver ourselves, we make use of an existing one. We opt to use Microsoft's Z3(Bjørner, 2018) because it has plenty documentation and is well supported across a variety of languages, which makes the constraint solver easy to include, configure and use. Moreover, the theorem prover supports the SMTLIB2 standard(Barrett, Stump, and Tinelli, 2010), which makes it easier to replace, should an alternative theorem prover be preferred.

Having built the constraints, we present how the constraint solver is utilised to detect potential data-races, and how the resulting model can be verified using an interpreter.

3.1 Constraint Building

Given a CCFG, we can construct constraints for the entire graph, then ask the constraint solver if it is possible that a data race can occur. We only consider those data races where a variable's value in any statement depends on interleavings, except for the `when` statement. We ignore this for `when` statements, as these are blocking calls, and continued execution on this thread is only allowed when the condition evaluates to `true`.

In this section, we use the CELE syntactical operators in CCFGs, whereas we use the mathematical operators everywhere else, unless otherwise specified. For example, the mathematical symbol for equality "=" means equality by default, except in CCFGs where it indicates a CELE assignment. Moreover, the constraint sets shown here does not follow the standard notation which is SMTLIB2, and is thus not the actual output from the constraint building algorithm which is then provided to the constraint solver.

For clarity when reading the constraint sets, all block names have been appended with the letter `b`.

To build constraints for any CCFG, each possible statement needs to be transformed into constraints. Algorithm 6 accomplishes this. The algorithm traverses the CCFG from the current block *cb* and returns the conjunction of all constraints after encoding the last block *lb*. Using the algorithm on the CCFG in Figure 3.1 produces the constraint set shown in Listing 3.1.

To get this encoding, the statements `if` and `assignment` has to be implemented, as well as the ϕ-functions.

Algorithm 6: The encoding algorithm with details for cases omitted

Data: previous block *pb*, current block *cb*, last block *lb* and a CCFG
Result: The encoded blocks between *cb* and *lb*

```
1  encode(pb,cb,lb)
2      btc ← boolean tracking constant for cb
3      expression ← btc = true
4      for statement s ∈ cb do
5          switch statement type of s do
6              case Assignment do
                   /* Handle Assignment                                  */
7              case If do
                   /* Handle if-statement                                */
8              case φ do
                   /* Handle phi-function                                */
9              case Fork do
                   /* Handle fork-statement                              */
10             case π do
                   /* Handle pi-function                                 */
11             case When do
                   /* Handle event                                       */
12     if cb ≠ lb then
13         expression ← expression ∧ encode(cb, successor of cb, lb)
14     return expressions
```

```
1 {-readVal1 ≥ −32768 ∧ -readVal1 < 32768
2  a1 = -readVal1 ∧ 1b = true
3  (ite{a1 < 10}
4      {a2 = 1 ∧ 3b = true ∧ a3 = a2 ∧ 5b = true ∧ 4b = false}
5      {a4 = 2 ∧ 4b = true ∧ a3 = a4 ∧ 5b = true ∧ 3b = false}
6  ) ∧ 2b = true
7  6b = true
8 }
```

Listing 3.1: Constraint set describing all paths through the program shown in Figure 3.1

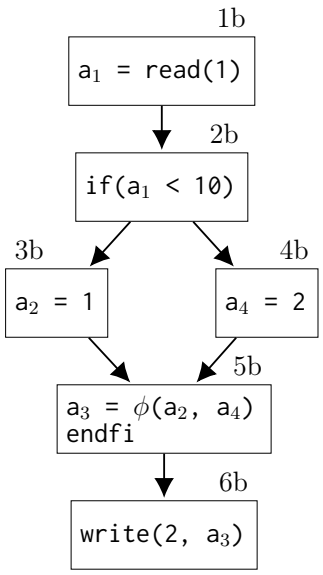

Figure 3.1: CFG with a single `if`-statement and boolean-tracking constants

`if`-statement

When encountering an `if`-statement, constraints for both the true and false branches has to be constructed. However, only the constraints on one of these branches are eligible, which is determined by the condition of the `if` statement. Thus the encoding of the `if` statement in the given example, produces the following subset of the resulting constraint set:

```
1 (ite{a1 < 10}
2     {a2 = 1 ∧ 3b = true ∧ a3 = a2 ∧ 5b = true ∧ 4b = false}
3     {a4 = 2 ∧ 4b = true ∧ a3 = a4 ∧ 5b = true ∧ 3b = false}
4 ) ∧ 2b = true
```

The encoding of an `if` statement effectively transforms the statement and each branch into an `ite` expression. This expression takes three arguments, which are all boolean expressions. The first argument determines which of the following two arguments should be used when finding a model that satisfies the complete constraint set. In short, if the first argument evaluates to true, then the second argument is used, otherwise the third argument. This `ite` expression therefore very much behaves like the CELE `if` statement. Algorithm 7 transforms a CELE `if` statement into an `ite` expression.

This algorithm calls Algorithm 6 recursively for both the true and false branches, where the last block for each recursive call, is the EndIf block for the `if` statement. The conjunction of all constraints gathered through the true branch, is then used as the second argument to `ite`, while the third argument, is the conjunction of all constraints gathered through the false branch. The `ite` expression then ensures only one of these constraint sets are used when finding a model, and which one of them is used, is determined by the first argument, which is the condition of the `if` statement.

Algorithm 7: Excerpt for encoding `if`-statements from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded `if`-statement added to expression

```
/* s is an if-statement                                                    */
 1 cond ← condition of s
 2 eb ← EndIf block for cb
 3 tb ← first successor of cb                  // first basic block of the true branch of s
 4 fb ← last successor of cb                   // first basic block of the false branch of s
 5 truebranch ← encode(cb, tb, eb)
 6 falsebranch ← encode(cb, fb, eb)
 7 boolexprT ← true
 8 boolexprF ← true
 9 for all bool tracking constants b in true branch of s do
10     boolexprT ← boolexprT ∧ {b = false}
11 for all bool tracking constants b in false branch of s do
12     boolexprF ← boolexprF ∧ {b = false}
13 expr ← ite(cond, truebranch ∧ boolexprF, falsebranch ∧ boolexprT)
14 expression ← expression ∧ expr
15 cb ← eb                          // successor of cb is the block we need to encode next
```

ϕ-functions

The encoding of ϕ-functions are always done during the encoding of an `if`-statement, as the only place we have ϕ-functions in the CCFG, is in EndIf blocks, which only ever merges branches of `if`-statements. The parameter pb to the encoding algorithm is a necessity when encoding any ϕ-function, as this parameter dictates which option of the ϕ-function is valid. Algorithm 8 transforms the ϕ-function into a constraint. It uses the pb parameter to determine where we came from, in order to build the correct constraint. This is why EndIf blocks, and thus the ϕ-function, are visited twice during the encoding of `if` statements, as the values of ϕ-functions depends on what branch was taken.

Algorithm 8: Excerpt for encoding ϕ-functions from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded ϕ-function added to expression

```
/* s is a φ-function                                                       */
1 name ← variable assigned in s
2 vars ← variable options for name
3 j ← index of pb in parents of cb
4 expr ← name = vars[j]
5 expression ← expression ∧ expr
```

`Assignment` statement and `read` expression

Having shown how the transformation of `if` statements and ϕ-functions are done, to encode the remaining constraint set, the encoding algorithm have to handle assignments.

Algorithm 9 transforms a CELE assignment into a constraint. In CELE, the left-hand side of an assignment gets the value of the right-hand side of the assignment symbol "=". When transforming this to a constraint, this can be considered as an equality because the CCFG is on CSSA form. Any future assignment creates a new variable, thus an assignment for any variable a_1, is only seen once. Stating both sides have to be equal, is therefore sufficient.

Algorithm 9: Excerpt for encoding an `assignment` statement from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded `assignment` statement added to expression
`/* s is an assignment */`
1 $name \leftarrow$ variable assigned in s
2 $rhs \leftarrow$ right-hand side of s
3 $expr \leftarrow name = rhs$
4 expression $\leftarrow$ expression $\wedge$ $expr$

When transforming the CFG to one on SSA form, each variable usage needs to be found and renamed to use the latest version of a variable. The `read` expression can be treated as it was a variable, and during the transformation, each time a `read` is encountered, increment the counter, similar to what is done when encountering an `assignment`. The `read` expression in the first basic block of the CCFG shown in Figure 3.1 is the first encountered `read` expression. This `read` thus gets the name $\texttt{-readVal}_1$. Each `read` expression has the same constraints. We need a new `read` variable for each encountered `read` expression, otherwise they would all be one variable with one value. A `read` expression can only have values between -32768 and 32767 in accordance to the semantics of CELE(Grosen and Svenningsen, 2019). When encoding any statement where a `read` expression occurs, a constraint is created stating that the corresponding read variable has these value boundaries as per the semantics in CELE.

Boolean Tracking Constants

To determine what path was taken through the program from a model which satisfies the constraint set produced for any given CCFG, we use boolean tracking constants. The names of each constant are that of each basic block, due to this name being unique. This constant is `true` if the encoded block is used in the model, or `false` otherwise. This means no matter what path is taken, they still need a constraint, as the constants would otherwise be unconstrained, which allows the SMT solver to give them any value it so desires. That is what the two **for** loops in Algorithm 7 handles. If the true branch is used, then all statements on the false branch is not used, and the boolean tracking constants for each block occurring there, needs to have the constraint `false`.

Having shown how constraints are built for a sequential program, lets consider a concurrent program such as the one shown in Figure 3.2.

Using Algorithm 6 on this CCFG produces the constraint set shown in Listing 3.2.

In order to produce this constraint set, Algorithm 6 has to be extended to handle the remaining statements: `fork` and `when`, as well as π-functions.

```
{-readVal_1 ≥ −32768 ∧ -readVal_1 < 32768 ∧
 -readVal_2 ≥ −32768 ∧ -readVal_2 < 32768 ∧
 a_1 = -readVal_1 ∧ 1b = true ∧
 2b = true ∧
 {( T_a_3 = a_1 ∨
    (ite {2.T2.3b = true} {T_a_3 = a_4} {false}) ∨
    (ite {2.T2.4b = true} {T_a_3 = a_6} {false})
  ) ∧ 2.T1.1b = true ∧
  (ite {T_a_3 = 10} {(a_2 = -readVal_2) ∧ (2.T1.2b = true)} {2.T1.2b = false})
 }
 ∧
 {(T_a_2 = a_1 ∨
   (ite {2.T1.2b = true} {T_a_2 = a_2} {false})
  ) ∧ 2.T2.1b = true ∧
  (ite {T_a_2 < 0}
       {((T_a_1 = T_a_2) ∨
         (ite {2.T1.2b = true} {T_a_1 = a_2} {false})
        ) ∧ 2.T2.2b = true ∧
        (ite {T_a_1 > 0}
             {a_4 = 10 ∧ 2.T2.3b = true ∧ a_5 = a_4 ∧ 2.T2.5b = true}
             {2.T2.3b = false ∧ 2.T2.5b = false}
        )
       }
       {a_6 = 0 ∧ 2.T2.4b = true ∧ a_5 = a_6 ∧ 2.T2.5b = true}
  )
 } ∧
 (ite {(ite {2.T1.1b = true} {T_a_3 = 10} {true}) ∧
       (ite {2.T2.2b = true} {T_a_1 > 0} {true})
      }
      {( (ite {2.T1.2b = true} {a_3 = a_2} {false} ∨
         (ite {2.T2.5b = true} {a_3 = a_5} {false}
       ) ∧ 3b = true ∧ 4b = true
      }
      {3b = false ∧ 4b = false}
 )
}
```

Listing 3.2: Constraint set for Figure 3.2

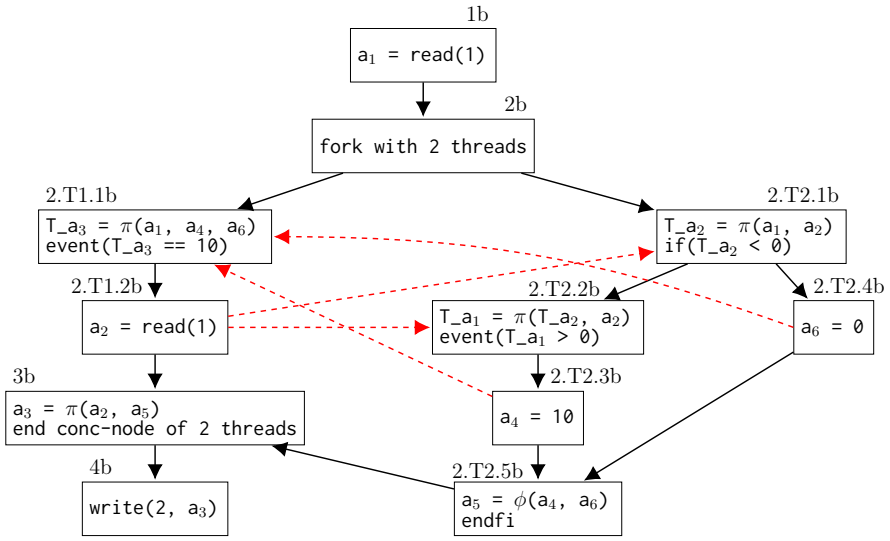

Figure 3.2: CCFG with multiple events

`fork`-statement

Extending the encoding algorithm with Algorithm 10 produces most of the constraint set shown in Listing 3.2. According to the semantics of CELE(Grosen and Svenningsen, 2019), all threads in a `fork` statement needs to finish the evaluation before the next statement can be evaluated. Due to that, the constraint of a `fork` statement is effectively the conjunction of all constraints gathered through each thread. Should any encountered `when` statement in either thread prevent a thread from finishing, this has to be handled here as well.

Unlike what is done for EndIf blocks, the EndFork block is encoded only once.

Algorithm 10: Excerpt for encoding `fork`-statements from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded `fork`-statement, excluding the EndConc block, added to expression

```
/* s is a fork-statement                                              */
1 eb ← EndFork block for cb
2 expr ← true
3 i ← 0
4 for each successor succ of cb do
5 |   p ← basic block at index i in parents of eb
6 |   expr ← expr ∧ encode(cb, succ, p)
7 |   i ← i + 1
8 expression ← expression ∧ expr
9 cb ← p    // successor of cb is now eb, which is the block we need to encode next
```

when-statement

Compared to the other statements, encoding an event is more complex, as the evaluation of the rest of the program depends on the event's condition. If the condition never evaluates to `true`, the thread with the event stalls and no future statement in that thread will be evaluated. If this happens inside a `fork`-statement, then the threads will never join either. Looking at events this way, they can be encoded similar to `if` statements. The second parameter to the resulting `ite` expression is thus the remaining program. The third parameter, is then a conjunction of all the remaining boolean tracking constants, such that they all should be equal to `false`.

However, the remaining program cannot be encoded straight away, as this could potentially lead to a situation where the rest of the program can be explored, should only one event's condition evaluate to `true`. What can be done instead, is to encode until the last block provided to the call to the `encode` function, then propagate up a boolean value which tells the caller if an event was encountered during the call to the function. If that is the case, then find all events between *cb* and *lb* which was provided to the call to `encode` where an event was encountered. All conditions for the found events, thus needs to evaluate to `true` in order to encode the rest of the program, but only if the event was actually part of the path condition. Algorithm 11 handles the encoding of a single `when` statement, by adding all the constraints between the block where this statement is located, and the block *lb*. Should the condition be `false`, then the boolean tracking constants are `false` as well, similar to what is done when encoding an `if` statement.

Algorithm 11: Excerpt for encoding `when` statements from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded `when` statement added to expression
`/* s is a when statement */`
1 $btcs \leftarrow$ boolean tracking constants between *cb* and *lb* excluding *cb*
2 $fb \leftarrow true$
3 **for** $b \in btcs$ **do**
4 $\quad fb \leftarrow fb \wedge (b = false)$
5 $expr \leftarrow$ ite(condition of s, encode(*cb*, successor of *cb*, lb), fb)
6 expression $\leftarrow$ expression $\wedge$ $expr$
7 $cb \leftarrow lb$
`/* Let caller know we have encountered an event */`

Recall the CCFG in Figure 3.2. In the body of the `fork` statement are multiple events; one in the first thread, and one in the second thread. Because we find at least one, we find each `when`-statement between the Fork and EndFork blocks. Once these statements are found, `ite` expressions are created for each of them. This is done because the boolean tracking constant has to be considered. For example, this program can reach the `write` statement, even if the `when`-statement in the second thread never evaluates to true. This is possible if the `if`-statement evaluates to `false`, as this event would then never be encountered. Since that event is never encountered, it does not matter what its condition would evaluate to. Such an `ite` expression is shown in Listing 3.3

The third parameter here is just true, because if the event is never encountered, then this condition should not be used when encoding the remaining program. Once all found events have been detected, and these `ite` expressions have been formed, a conjunction of all of them is used as the condition on a final `ite` expression. The second and third parameters on this final `ite` expression are encoded in the same way they are when encoding a single `when` statement. The constraint for the remaining program from the `fork` statement, which includes the EndFork block and the `write` statement, is shown in Listing 3.4.

```
1 { (ite {2.T2.2b = true}
2        {T_a1 > 0}
3        {true}
4   )
5 }
```

Listing 3.3: Constraint stating if an event is encountered, its condition has to hold

```
1 {(ite {(ite {2.T1.1b = true} {T_a3 = 10} {true}) ∧
2        (ite {2.T2.2b = true} {T_a1 > 0} {true})
3       }
4       {( (ite {2.T1.2b = true} {a3 = a2} {false} ∨
5          (ite {2.T2.5b = true} {a3 = a5} {false}
6        ) ∧ 3b = true ∧ 4b = true
7       }
8       {3b = false ∧ 4b = false}
9  )
10 }
```

Listing 3.4: Constraint describing the EndFork block and the remaining program of Figure 3.2

Since events need to be handled when encoding both `if` and `fork` statements, the encoding algorithm needs to be extended such that any call to the function, also provides information about whether or not a `when` statement was encountered.

Thus Algorithm 7 and Algorithm 10 are extended with Algorithm 12 in order to handle this propagation.

Algorithm 12: Excerpt for handling encountered events inside the bodies of `if` and `fork` statements

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoding of the remaining program when the conditions for all events are satisfied, added to expression

```
1  if when was encountered then
2      condition ← true
3      for all events e between cb and lb do
4          bc ← boolean tracking constant of the block of e
5          c ← condition of e
6          expr ← ite(bc = true, c, true)
7          condition ← condition ∧ expr
8      pb ← // p if s is a fork statement, or eb if s is an if statement
9      btcs ← boolean tracking constants for all blocks between pb and lb excluding pb
10     fb ← true
11     for b ∈ btcs do
12         fb ← fb ∧ (b = false)
13     expr ← ite(condition, encode(cb, successor of pb, lb), fb)
14     expression ← expression ∧ expr
15     cb ← lb
       /* Let caller know we have encountered an event                    */
```

π-functions

In practice, all π-functions are locations where a potential data-race can occur. A simple disjunction of a choice between all possible variable choices will find if a variable can have two different values. A program execution can then verify if such an occurrence is indeed possible. How the encoding of a π-function is accomplished, is shown in Algorithm 13.

We restrict which variables are available in each π-function, by forcing the solver to only consider variables that has been assigned. We utilise the boolean-tracking constants to determine what variable options are applicable. We express each variable option choice with `ite` expressions. As seen in line 6 of Algorithm 13, an `ite` expression is formulated to invalidate variables that are not assigned on the path chosen by the solver.

Algorithm 13: Excerpt for encoding π-functions from the encoding algorithm

Data: The data accessible during the call to the encoding Algorithm 6
Result: The encoded π-function added to expression
`/*` s `is a` π`-function */`
1 $name \leftarrow$ variable assigned in s
2 $vars \leftarrow$ variable options for $name$
3 $expr \leftarrow false$
4 **for** $var \in vars$ **do**
5 $\quad btcv \leftarrow$ bool tracking constant for the block where the variable assignment to var occurs
$\quad$ `/* False branch is unsatisfiable, forcing the constraint solver to pick another option in the final disjunction if` $btcv$ `is false */`
6 $\quad expr \leftarrow expr \vee \{ite(btcv, name = var, false)\}$
7 expression $\leftarrow$ expression $\wedge$ $expr$

3.2 Finding Possible Data-races

In principle all conflict-edges represent a possible data-race. If no conflict-edges are present then no data-race can occur. But reporting a data-race for each conflict-edge is in most cases an over-approximation, and could lead to wasted effort trying to remove all conflict-edges. To narrow the amount of possible data-races, we found inspiration in a paper regarding Noninterference and symbolic execution (Milushev, Beck, and Clarke, 2012). The authors perform two program executions through symbolic execution to find if some secret variables called *High* security variables can affect the value of *Low* security variables; variables are public. They state that if two program executions starting with the same values in each *Low* security variable, and *High* security variables can be different, then if at some point in the program a *Low* security variable can have different values, then the program is insecure and does not have *noninterference*. If we use a similar approach and consider all variables *Low* security variables except a variable we expect might be a subject of a data-race, then we can perform symbolic execution of two programs with similar variables that share the same values. We have then isolated that only the path can influence the suspected variable, and if two different paths lead to the variable having different values, we can conclude that a data-race is possible. While it is still an over-approximation, we can reduce the number of suspected data-races.

We perform this by encoding the same CCFG twice, then appending an extra text string to each variable name, unique to each execution. For example, the statement `a2 = T_a1 + 2` would look like the following in each execution:

execution 1: $\texttt{a}_2\texttt{run}_1$ `= T_a`$_1\texttt{run}_1$ `+ 2`

execution 2: $\texttt{a}_2\texttt{run}_2$ `= T_a`$_1\texttt{run}_2$ `+ 2`

This ensures each variable can potentially have different values in each execution, even when the only symbolic values we have comes from the `read` expression, which we make sure to be the same in each execution. To make sure `read` expressions evaluate to the same value during each execution, we encode all reads in the program at the very beginning. Since the CCFG is already on CSSA form, we just find out how many reads can occur, then give all of them constraints. During the constraint transformation, we just use the name provided to each `read` expression, but without appending the string identifying the execution. This ensures each read are identical across executions.

Having transformed both executions into constraints for the SMT solver, we constructs additional constraints that we want answered. What we want answered, is if π-functions can have two different values, as this means a data race can occur, and is also why we build constraints in this manner. One such constraint would be:

$$\texttt{T\_a}_1\texttt{run}_1 \neq \texttt{T\_a}_1\texttt{run}_2$$

Such constraints for all π-functions in the CCFG is constructed, and the constraint solver is then asked if one of these are satisfiable. If it is, then a potential data-race is found, otherwise there are none. If a data-race is discovered, then an interpreter does a concrete execution to verify if the data-race can occur in the given model.

3.2.1 Verifying Suspected Data-races

In order to verify that a detected data-race can indeed occur, we resort to concrete execution. With a model that satisfies the constraints, provided by the symbolic execution, we can find out what values are used for each variable use, and in turn, see what value are assigned to each variable. Additionally, the model also contains values for each ϕ and π functions and have concretised values for each `read` expression. Using this information allows an interpreter to execute the program with concretised values.

With a model, the interpreter does a concrete execution of the program, checking if variables get the assigned values that matches that of the model received by the SMT solver. During concurrent parts of the program, the interpreter does not know what interleavings were taken, only what the values each π-function should have, thus the interpreter saves multiple states and explores the state space of the concurrent part in a depth-first manner. When a π-function is encountered, the value for the variable this π-function is for, is checked against the model, and should the values not match, the interpreter stops executing more statements from this thread, as an interleaving either should have, or should not have been taken previously.

When the π-function, which is being checked for a data-race, is encountered, the interpreter checks the π-function to see if the value for it matches that of the model. If it does, then the interpreter repeats all the steps using the model matching that of the second execution. If both executions are valid, the interpreter reports the data-race location and its values, thus having verified the data-race using the model from the symbolic execution component.

Once a data-race has been verified, the interpreter resets and awaits another potential location for a data-race to verify.

3.3 Summary

The design proposed in this chapter can find possible data-races in any program written in the CELE language, and it can verify some of the data-races. But this design is prone to over-approximation and can report multiple data-races that are impossible during concrete execution, and it will not verify all the possible data-races. While this solution is sufficient we will propose a design that can more accurately[1] report data-races.

[1] Accuracy: Here accuracy refers to how accurate the prediction is. High accuracy means no false-positives and low accuracy means many false-positives.

Chapter 4

Discussion

The design proposed in Chapter 3 leads to an over-approximation, causing the constraint solver to generate many invalid models. Recall that a π-function contains all the possible variable assignments that can occur in concurrent threads, including the same value propagated down along the thread it occurs in. A variable in the block with a π-function is then replaced by the value of the π-function. This value can either be that of a previous definition in the thread, or an assignment taking place in a concurrent thread.

With the current encoding of π-functions, we get a constraint-set which cover states that are not encounterable during a concrete execution. To show this, consider the CCFG in Figure 4.1.

By taking this approach, the SMT solver can choose to do the following:

$$T\_a_1 = a_2,\ T\_a_2 = a_4,\ T\_a_3 = a_5,\ T\_a_4 = a_4,\ T\_a_5 = a_2,\ T\_a_6 = a_2$$

We know that this is impossible during an actual program execution, because in order for this to ever happen, the assignment $\texttt{a}_4$ `= 20` needs to happen twice. In this case, the boolean tracking constants does not help us, because these only explain if statements are part of the path condition or not. However, all variable assignments would indeed be part of the path condition, thus all options in the π-functions are available. Nothing therefore prevents the SMT solver from re-using the same variable in any order. With this encoding, the statement $\texttt{reachable}_2$ `= true` would be reachable, despite this being impossible during any concrete program execution.

All possible paths could in theory be encoded in each π-function, but then each variable usage in each π-function would be disjunctions containing disjunctions, and the encoding itself would be $\Theta(n^m)$, where n is the amount of π-functions present in the graph, and m is the maximum amount of variable choices in any π-function. We therefore choose not to opt for this as a program with even a few π-functions would take an infeasible amount of time to encode. Instead, a more feasible approach can be used.

4.1 Removing False-Positives

The solution presented in Chapter 3 can find all data-races, but it can also report many data-races that are impossible during concrete execution. Reporting too many false positives in a program will affect how powerful the solution is, as the users might disregard some reported data-races if presented with too many false-positives.

We have not encountered any solutions, with a reasonable run time, that can completely eliminate the false-positives reported by the symbolic execution, but have discovered techniques that can mitigate the number of false-positives. Additionally, we have also found a way that may help verify each data-race reported by the symbolic execution.

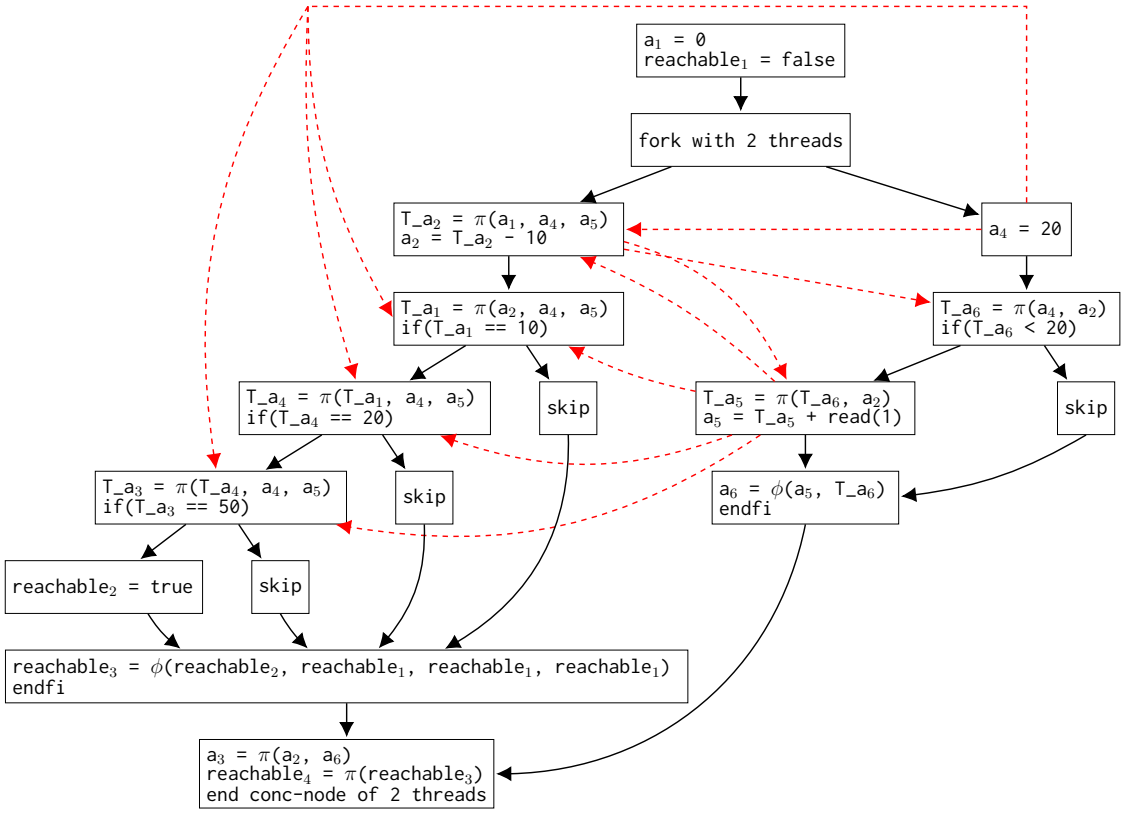

Figure 4.1: CCFG of a program where the value of $\texttt{reachable}_4$ will never be `true`

4.1.1 Limiting Interleavings

π-functions capture thread interleavings, and they over-approximate in order to capture all possible interleavings. If the constraints for π-functions can be extended to invalidate impossible interleavings, then the amount of false-positives reported by the symbolic execution, can be reduced. The conflict edges can be utilised to restrict the amount of valid options a π-function can have. A simple yet efficient restriction is that only a maximum of one outgoing conflict edge is possible for each block. So if an assignment occurs in one thread that has multiple outgoing edges to another thread, then only one of these conflict edges can be chosen. As shown in Figure 4.1, the assignment $\texttt{a}_4$ can influence multiple π-functions in the other thread, but since the assignment can only occur once, it can at most affect one of them. Lets explore this approach in more detail.

Leveraging Conflict Edges

The cause of many invalid models being generated from the over-approximation is caused by interleavings. The graph already have conflict edges so instead of each π-function being encoded as a simple disjunction, each option could also encode an invalidation of incoming and outgoing conflict edges, should the block also contain an assignment.

Consider Figure 4.1. For the π-function $\texttt{T\_a}_1$, it is clear that the option $\texttt{a}_4$ is not a valid option should $\texttt{a}_4$ have been used in the previous block. Thus when choosing an option, only if the corresponding edge is valid, should the π-function have the value of the chosen option, and in doing so, invalidate the other options. Moreover, should the option $\texttt{a}_4$ be chosen for $\texttt{T\_a}_1$, then all other outgoing edges from $\texttt{a}_4$ should be invalidated, if the edge leads to any other block in the same thread containing $\texttt{T\_a}_1$. This works because the value of the π-function propagates down in the thread.

If a π-function gets its value from another statement, then this would mean that all statements in that block have finished execution. This is because all concurrent blocks only have a single statement, and π-functions. The π-functions are always placed above the CELE statement, and thus, the assignment would always be the very last thing to occur in a block with outgoing conflict edges. Due to this, a value provided from a block cannot have two valid outgoing conflict edges to the same thread.

Each conflict edge are assigned names which matches that of the two blocks. They follow the form `ed-DEF-to-USE`, whereas each `USE` is a π-function. For instance, the conflict edge from the block containing the assignment $\texttt{a}_4$ `= 20;` to the block containing the π-function $\texttt{T\_a}_1$ would be called $\texttt{ed-a}_4\texttt{-to-T\_a}_1$. The constraint for $\texttt{T\_a}_1$ when using this approach, can be seen in Listing 4.1.

```
1  T_a1 = a2 ∨
2  (ite {ed-a4-to-T_a1 ∧ 2.T2.1b}
3       {ed-a4-to-T_a2 = false ∧
4        ed-a4-to-T_a3 = false ∧
5        ed-a4-to-T_a4 = false ∧
6        ed-a5-to-T_a1 = false ∧
7        ed-a5-to-T_a2 = false ∧
8        T_a1 = a4
9       }
10      {false}
11 ) ∨
12 (ite {ed-a5-to-T_a1 ∧ 2.T2.3b}
13      {ed-a4-to-T_a1 = false ∧
14       ed-a4-to-T_a3 = false ∧
15       ed-a4-to-T_a4 = false ∧
16       ed-a5-to-T_a2 = false ∧
17       ed-a5-to-T_a3 = false ∧
18       ed-a5-to-T_a4 = false ∧
19       T_a1 = a5
20      }
21      {false}
22 )
```

Listing 4.1: constraint for $\texttt{T\_a}_1$ `=` $\pi(\texttt{a}_2, \texttt{a}_4, \texttt{a}_5)$

This approach limits each assignment's influence to a maximum of one π-function in every thread. However, this does not prevent an assignment to influence multiple π-functions in the entire program, as this would only limit it to a maximum of $n - 1$ π-functions, where n is the amount of concurrent threads. With multiple threads, this prevention of constructing invalid models is no longer as effective.

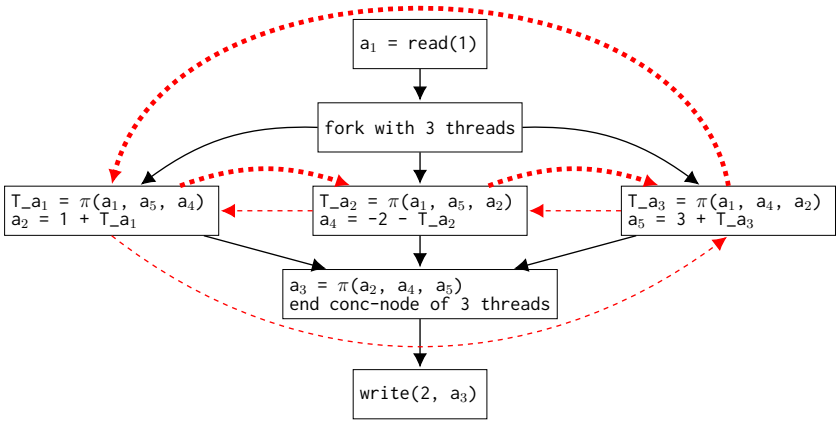

Figure 4.2: Figure with three threads showing interleaving issue

Consider Figure 4.2. Using this approach, the constraint solver can still choose interleavings which results in an invalid model. One such example is highlighted by the thicker conflict edges. It is possible to prevent this in theory during constraint construction since the edge invalidation is done based on the picked incoming edge. Having picked $\texttt{a}_2$ for $\texttt{T\_a}_2$, then the reversed edge is invalidated. Because each incoming conflict edge in this example leads to an assignment, it is easy enough to ensure that the value from this assignment does not end up being used for $\texttt{T\_a}_1$. When having picked $\texttt{a}_2$ for $\texttt{T\_a}_2$, then the value from this block's assignment, is only usable for $\texttt{T\_a}_3$, and since that block has an assignment, then we can in turn follow its outgoing edges and invalidate the conflict to $\texttt{T\_a}_1$.

As can be seen, it is clear that just adding one more thread leads to constraint building becoming significantly more complex, as it is no longer necessary to just consider the immediate conflict edges, but also the conflict edges from the blocks containing the conflicting values. It is fair to say that the complexity grows at least exponentially for each additional thread added.

Results

The approach to the encoding shown in Chapter 3 will be able to find data races that are otherwise unreachable. Consider the example shown in Figure 4.3. With the current encoding, asking the constraint solver if it is possible for $\texttt{reachable}_4$ to have the value `true`, will return a model that satisfies that constraint. Although for this to be the case, the statement $\texttt{reachable}_2$ needs to be reachable, which it is not.

To reach this statement, the assignment $\texttt{a}_4$ `= 20` need to have been executed, otherwise $\texttt{T\_a}_1$ will not get a value that is needed for the condition of the `if` statement to evaluate to `true`. But since this assignment has now taken place $\texttt{T\_a}_4$ cannot get the value `20` from $\texttt{T\_a}_4$ as this assignment has already taken place. The only way to obtain the required value, is through the assignment to $\texttt{a}_5$.

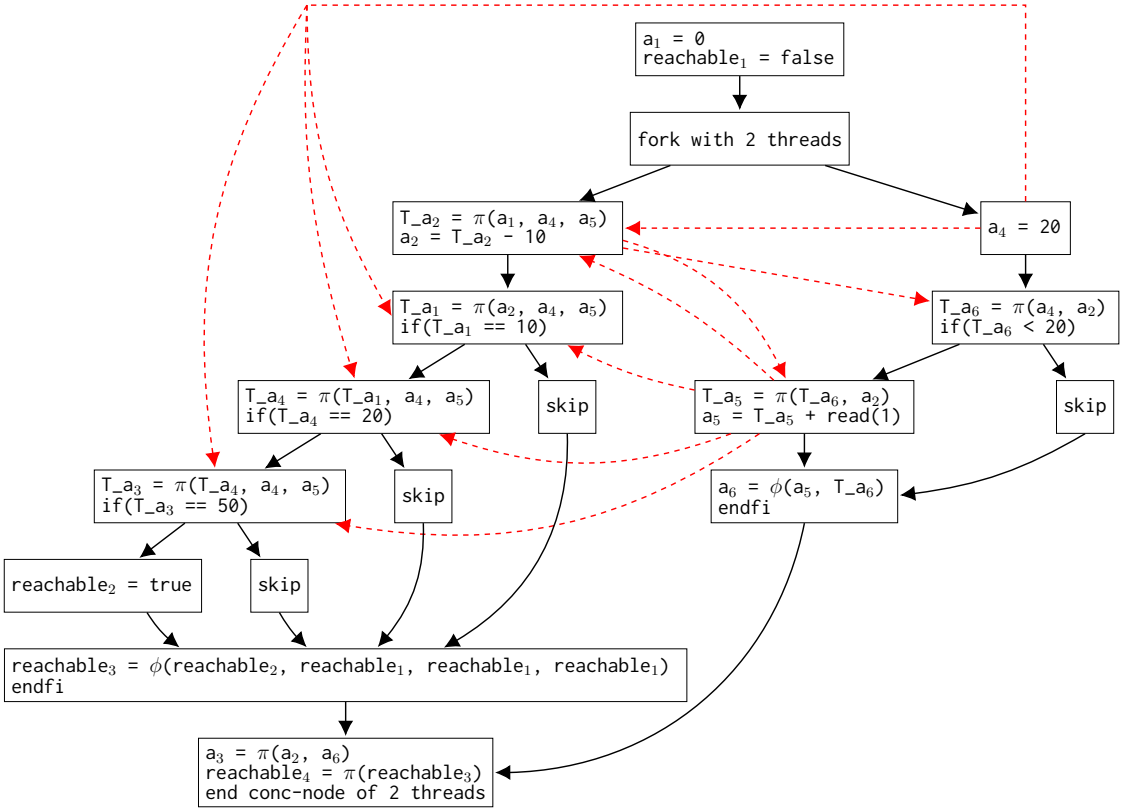

Figure 4.3: CCFG of a program where the value of $\texttt{reachable}_4$ will never be `true`

If however $\texttt{a}_5$ gets the value `20`, then there are no remaining assignments left inside this `fork` statement, thus the condition $\texttt{(T\_a}_3\texttt{ == 50)}$ will never evaluate to `true`. Moreover, the only value $\texttt{T\_a}_3$ can have, is `20`, which is propagated down from the block above.

The first encoding, the one explained in Chapter 3, ignores previous interleavings when determining the value for a π-function. That encoding will cause the constraint solver to produce a model where the assignment $\texttt{reachable}_2\texttt{ = true;}$ takes place, despite that from not being the case. The following values for all π-functions allows this to be the case:

$$T\_a_1 = a_2,\ T\_a_2 = a_4,\ T\_a_3 = a_5,\ T\_a_4 = a_4,\ T\_a_5 = T\_a_6,\ T\_a_6 = a_2$$

This over-approximation lead to the proposed changes shown in Section 4.1.1. With that proposed change, each encoding of a π-function invalidates the other edges to it, while also ensuring that the conflicting statement is not a valid option for any other block in the same thread where the π-function is located. Moreover, any other conflict is not a valid option either for blocks earlier than this π-function, if that conflict is from a statement that happens after the chosen one.

Having implemented that approach, the constraint solver can no longer return a satisfiable model where the statement $\texttt{reachable}_2\texttt{ = true}$ takes place. Moreover, it also only detects five locations for possible data-races, despite there being six π-functions. This is correct, as the only value that $\texttt{T\_a}_3$ can have, is `20`, as just discussed.

$$[\text{BIN\_1}]\quad \frac{\text{env} \vdash e_1 \Rightarrow e_1'}{\text{env} \vdash e_1 \diamond_b e_2 \Rightarrow e_1' \diamond_b e_2} \qquad [\text{VAR}]\quad \text{env} \vdash a \Rightarrow v \text{ where } v = \text{env}(a)$$

$$[\text{BIN\_2}]\quad \frac{\text{env} \vdash e_2 \Rightarrow e_2'}{\text{env} \vdash v_1 \diamond_b e_2 \Rightarrow v_1 \diamond_b e_2'} \qquad [\text{ASS\_1}]\quad \frac{\text{env} \vdash e \Rightarrow e'}{a\texttt{=}e,\ \text{env} \Rightarrow a\texttt{=}e',\ \text{env}}$$

$$[\text{BIN\_3}]\quad \text{env} \vdash v_1 \diamond_b v_2 \Rightarrow v \text{ where } v = v_1 \bar{\diamond}_b v_2 \qquad [\text{ASS\_2}]\quad a\texttt{=}v,\ \text{env} \Rightarrow \text{env}[a \mapsto v]$$

Table 4.1: CELE semantics for assignments and binary expressions. The complete semantics is first presented by (Grosen and Svenningsen, 2019) and has been included in Appendix A for completeness

However, the conflict edge invalidation technique is still a gross over approximation for programs with more than two threads, as it does not invalidate conflict edges between other threads than the thread with the π-function being encoded, and the thread containing the conflicting definition. Extending it leads to an algorithm that grows exponentially and is thus not sufficient. Limiting the maximum amount of allowed outgoing conflict edges from each block to one, would make the encoding more feasible, but would lead to an under approximation. For instance, consider Figure 4.2. It is clear that both $\texttt{T\_a}_1$ and $\texttt{T\_a}_3$ can get the value $\texttt{a}_4$, if the assignments in those two blocks have yet to take place. In the semantics of CELE an assignment is not atomic. It first evaluates an expression until that becomes a value, and only then can the actual assignment take place. Thus, for the assignment to take place, the binary expressions have to become a single value first. For reference, the valid rules is shown in Table 4.1.

The rule `[ASS_1]` needs to apply both rules `[BIN-2]` and `[BIN-3]` before the variable mapping is applicable, which is done through the rule `[ASS_2]`. To show that both $\texttt{T\_a}_1$ and $\texttt{T\_a}_3$ can have the same value, lets apply the semantics until $\texttt{a}_4$ `= 2 +` $\texttt{T\_a}_2$ is reached, without visiting either thread 1 and 3.

Once $\texttt{a}_4$ is assigned, then an interleaving occurs, and the rule `[BIN-2]` is applied to the statement $\texttt{a}_2$ `= 1 +` $\texttt{T\_a}_1$. That rule states the right-hand side of the binary expression needs to be evaluated, and as $\texttt{T\_a}_1$ is the value from a π-function for `a` in this statement, the only rule applicable, is `[VAR]`. `[VAR]` looks up the variable `a` and substitutes it in the statement with the value that `a` currently has; which is the same value as $\texttt{a}_4$. Another interleaving can then be taken here, and the same rules can be applied for the statement $\texttt{a}_5$ `= 3 +` $\texttt{T\_a}_3$. The result is that $\texttt{T\_a}_1$ and $\texttt{T\_a}_3$ have the same value.

Thus only allowing one outgoing conflict edge from each block means that the constraint solver will not be able to find some models which can actually occur during a concrete execution, thus potentially limiting the amount of detectable data-races.

However, as loop unrolling is already being done, the amount of correct models is already limited, so this under approximation may still prove to be good enough for detecting data-races. We choose to keep an over approximation, and instead choose to use an interpreter to validate if the data-race can occur in the found model. The interpreter can do with some extensions to improve its usefulness.

4.1.2 Interpreter

The interpreter currently receives a satisfiable model from the symbolic execution and executes the input program such that the execution does not break the received model. It knows what statement the symbolic execution found a data-race for, as well as the blocks which was encountered. The interpreter executes the program to see if it can encounter the statement where the data-race occurs using this model in order to verify it. If this statement is reachable, and the current value matches that given by the model, then the data-race is possible.

But if this is not the case then the data-race might still be possible, because the model provided by the symbolic execution component may be invalid. The model can be analysed to find where this invalid interleaving was taken. This would often occur in π-functions where a chosen variable option is allowed by the constraints but is not a valid option due to previous interleavings.

With the design proposed in Section 3.1, the model does not contain information about interleavings, but it does contain information about what blocks were executed, and the value of each variable. This information can be used to construct additional constraints and then provided for the constraint solver in order to obtain a new model, should the current one be invalid.

Consider the CCFG shown in Figure 4.1. As it stands, and as was described in Section 4.1.1, the proposed design means that the constraint solver will be able to find a model that shows a data-race can occur at $\texttt{T\_a}_3$. The constraints in the model for the π-functions in a single execution could look like:

$$T\_a_2 = 20,\ T\_a_1 = 10,\ T\_a_4 = 20,\ T\_a_6 = 10,\ T\_a_5 = 10$$

During a concrete execution, the interpreter will discover that $\texttt{T\_a}_4$ cannot have the value which this model states it has, unless $\texttt{a}_5$ gets the value `20`. However in that case, the model would be unsatisfiable, as $\texttt{T\_a}_3$ would have the value `20` in all cases. We therefore know, for the symbolic execution to report a data-race, that $\texttt{a}_5 \neq \texttt{20}$ holds. Moreover, since the block in this execution is unreachable, the interpreter could construct a constraint using the values it found during the execution. The constraint for this example is shown in Listing 4.2.

```
1 {(ite {T_a2 = 20 ∧ T_a1 = 10 ∧
2         T_a6 = 10 ∧ T_a5 = 10 ∧
3         a1 = 0 ∧ a2 = 10 ∧
4         a4 = 20 ∧ a5 = 50
5        }
6        {2.1T.4b = false}
7        {true}
8  )
9 }
```

Listing 4.2: Constraint stating that given these values, the block containing $\texttt{T\_a}_3$ is unreachable

With this constraint, the SMT solver will choose not to generate a new model where this condition is true, as that would lead to an invalid model. After all, a part of the constraint asking if a π-function can have two different values, also explicitly states that the block containing this π-function needs to be present. Were this not part of the constraint, the π-function could have an unconstrained value if the SMT solver found a means to avoid using the constraints for the block containing the π-function. This would thus allow the SMT solver to pick any value it wants.

The constraint does not include the value for $\texttt{T\_a}_4$ because the model states this value should be something that the interpreter never finds to be true. Stating that $\texttt{T\_a}_4$ should not be the value found during this execution, would be an additional condition that does assist much in finding a valid model. If this constraint was added, then the SMT solver could return the very same model, because in that model, $\texttt{T\_a}_4$ was `20`, not whatever the interpreter discovered. Moreover, adding $\texttt{T\_a}_4 \neq \texttt{20}$ to the constraint would provide the same information as the `if` statement where the value is used.

A problem with this approach is immediately apparent. Recall that the only data-races considered, are those where a π-function can have different values. With this extra constraint, there is no stopping the SMT solver from just reading another value from the environmment, as this is symbolic. Doing so would change the value of $\texttt{a}_5$ to something else, again yielding a satisfiable problem. The interpreter would thus keep building these constraints, only modifying $\texttt{a}_5$ with each execution until killed or all possible values from the environment has been tested, finally making the constraint solver return an unsatisfiable model.

Utilising Conflict Edges

Having discussed the utilisation of conflict edges in Section 4.1.1 and discovered that invalidating conflict edges upon variable choices in π-functions can significantly help producing correct models for programs with two threads, a similar approach could be done with the interpreter.

An interpreter that are aware of conflicts can utilise this information to know where exactly the values for each π-function came from. By still validating the model as previously, it can use this information to help assist it in choosing what thread to execute. For instance, consider Figure 4.4 which also shows the problem that the approach in Section 4.1.1 cannot model accurately.

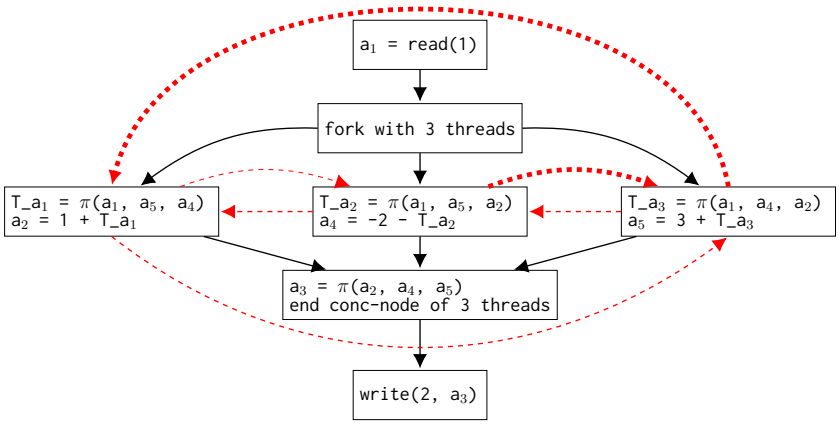

Figure 4.4: Program with three threads. These interleavings **are** valid

When the interpreter encounters the `fork` statement, it then adds all children blocks to a set which holds all potential blocks it can execute next. It does not know which one is the right block to visit first however. Lets assume it visits its first child. Here, it sees that the value for the given π-function does not match the value in the model, but the model does state that this block's incoming conflict edge containing the statement $\texttt{a}_5 \texttt{ = 3 + T\_a}_3$ determined this value. The interpreter sees that this block is a potential candidate for execution, so it attempts executing this block instead.

This continues until it ends up at the middle child, where the value for $\texttt{T\_a}_2$ matches that of the model, thus it can execute this statement. It then follows conflict edges, as these are allowed and executes the statements accordingly, matching it against the model.

If this model had an invalid interleaving, such as the ones shown in Figure 4.2, the interpreter would attempt the execution and realise it encounters a loop, thus breaking the execution. Having followed these interleavings, the interpreter will thus be able to build the constraint shown in Listing 4.3.

```
1 {(ite {ed-a2-to-T_a2 ∧ ed-a4-to-T_a3 ∧ ed-a5-to-T_a1}
2       {false}
3       {true}
4  )
5 }
```

Listing 4.3: Constraint stating that given these conflict edges, the model generated will be invalid

With this constraint, the interpreter tells the symbolic execution not to consider this set of conflict edges for a new model, as these leads to one that is invalid. By not considering values of variables, we avoid the problem of repeatedly generating similar models where only the value of a symbolic input has been changed.

Chapter 5

Conclusion

This concludes our research in making use of symbolic execution in order to detect the locations where data-races can occur. The constraint generation presented in Chapter 3 results in constraints that over approximates the location of data-races and leads to the generation of many invalid models. Moreover, without information of the chosen interleavings, only values for each variable can be used for constructing additional constraints which can prevent the same model from being considered next time.

We therefore opted to encode conflict edges and the invalidation of others, when building constraints for π-function, as described in Section 4.1.1. Using that method, the symbolic execution component can much more precisely find data-races in concurrent portions of code which have two threads. It does have the problem of not invalidating all invalid conflict considerations, if there are more than two threads, as doing so would lead to an exponential computation time depending on threads and the number of options for each π-function. An under approximation such as one which only allows a maximum of one outgoing conflict edge per block may be good enough in practice, but this was not experimented with during this thesis.

Instead, we considered expanding the interpreter to build constraints using the conflict edges that are now part of the problem, then updating the final constraint set, such that the symbolic execution component will not consider the same set of conflict edges when asked for a new model. The implementation for what is presented and discussed in Section 3.1 and Section 4.1.1 can be found on Github (Grosen and Svenningsen, 2020). We have leveraged Symbolic execution to find errors specific to concurrent and reactive programs, our solution over-approximates the possible data-races. We have found ways to mitigate this over-approximation, eliminating many of the impossible interleavings. We have shown that we can find data-races within the programs tested in this thesis.

Bibliography

Grosen, Casper M. and Jacob A. Svenningsen (2019). *Symbolic Execution for Embedded Systems*. Student Project Report. AAU.

Lee, Jaejin, Samuel P. Midkiff, and David A. Padua (1997). "Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs". In: *Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota, USA, August 7-9, 1997, Proceedings*. Ed. by Zhiyuan Li et al. Vol. 1366. Lecture Notes in Computer Science. Springer, pp. 114–130. URL: https://link.springer.com/chapter/10.1007/BFb0032687.

Appel, Andrew W. (2004). *Modern compiler implementation in C*. Cambridge University Press.

Rosen, Barry K., Mark N. Wegman, and F. Kenneth Zadeck (1988). "Global Value Numbers and Redundant Computations". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by Jeanne Ferrante and P. Mager. ACM Press, pp. 12–27. URL: https://dl.acm.org/doi/10.1145/73560.73562.

Lengauer, Thomas and Robert Endre Tarjan (1979). "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1, pp. 121–141. URL: https://dl.acm.org/doi/10.1145/357062.357071.

Novillo, Diego, Ronald C Unrau, and Jonathan Schaeffer (2000). *Analysis and optimization of explicitly parallel programs*. University of Alberta.

Cadar, Cristian, Daniel Dunbar, Dawson R Engler, et al. (2008). "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8, pp. 209–224. URL: http://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf.

Cha, Sang Kil et al. (2012). "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE, pp. 380–394.

Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea (2011). "S2E: A platform for in-vivo multi-path analysis of software systems". In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM, pp. 265–278.

Bjørner, Nikolaj (2018). "Z3 and SMT in Industrial R&D". In: *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. Ed. by Klaus Havelund et al. Vol. 10951. Lecture Notes in Computer Science. Springer, pp. 675–678. URL: https://link.springer.com/chapter/10.1007/978-3-319-95582-7_44.

Barrett, Clark, Aaron Stump, and Cesare Tinelli (2010). *The SMT-LIB Standard: Version 2.0*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa.

Milushev, Dimiter, Wim Beck, and Dave Clarke (2012). "Noninterference via Symbolic Execution". In: *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. Ed. by Holger Giese and Grigore Rosu. Vol. 7273. Lecture Notes in Computer Science. Springer, pp. 152–168. URL: https://link.springer.com/chapter/10.1007/978-3-642-30793-5_10.

Grosen, Casper M. and Jacob A. Svenningsen (May 2020). *MasterThesis*. URL: https://github.com/CMGrosen/MasterThesis.

Appendices

Appendix A

Syntax and Semantics

This appendix contains the semantics referenced in this thesis and originally presented by Grosen and Svenningsen (2019). Language construct not considered in this thesis has not been included as part of this appendix. Moreover, an oversight was discovered. In the abstract syntax, the read expression expected a variable, as did the write statement as its first argument. However, the semantics for IO had no rules for such constructs. The abstract syntax for the read expression and the write statement has thus been modified such that the semantics apply. The syntax and semantics as they were originally presented, can be seen in (Grosen and Svenningsen, 2019).

A.1 Abstract Syntax

S	$\in$	STMT	statements
a	$\in$	VAR	variable names
e	$\in$	EXPR	expressions
v	$\in$	VAL	values

$$e := (e) \mid e\texttt{+}e \mid e\texttt{-}e \mid e\texttt{*}e \mid e\texttt{/}e \mid e\texttt{\%}e \mid e\texttt{>}e \mid e\texttt{>=}e \mid e\texttt{<}e \mid e\texttt{<=}e \mid e\texttt{==}e \mid e\texttt{!=}e \mid e\texttt{\&\&}e \mid e \mid\mid e \mid \texttt{!}e \mid \texttt{-}e \mid \texttt{read(}v\texttt{)} \mid v \mid a$$

$$S := a \texttt{ = } e \mid \texttt{write(}v\texttt{, }e\texttt{)} \mid \texttt{if (}e\texttt{) \{}S\texttt{\} else \{}S\texttt{\}} \mid \texttt{while(}e\texttt{) \{}S\texttt{\}} \mid \texttt{fork \{}S_1\texttt{\} and \{}S_2\texttt{\} ... and \{}S_n\texttt{\}} \mid \texttt{when (}e\texttt{)} \mid \texttt{skip} \mid S\texttt{;}S$$

A.2 Semantics for I/O

$$\text{[READ\_1]} \quad \text{env} \vdash \texttt{read(}v_1\texttt{)} \Rightarrow v \quad \text{where } v < 2^{15} \text{ and } v \geq -2^{15}$$

$$\text{[WRITE\_1]} \quad \frac{\text{env} \vdash e \Rightarrow e'}{\texttt{write(}v\texttt{, }e\texttt{)}, \text{ env} \Rightarrow \texttt{write(}v\texttt{, }e'\texttt{)}, \text{ env}}$$

$$\text{[WRITE\_2]} \quad \texttt{write(}v_1\texttt{, }v_2\texttt{)}, \text{env} \Rightarrow \texttt{skip}, \text{ env}$$

A.3 Semantics for Expressions

Literals

[INT_LIT] $v \in \mathbb{Z}$

[BOOL_LIT] $v \in \{$ `true, false` $\}$

Binary Expressions

$\diamond_b \in \{$`+, -, *, /, %, >, >=, <, <=, ==, !=, &&, ||`$\}$

$$[\text{BIN\_1}] \quad \frac{\text{env} \vdash e_1 \Rightarrow e_1'}{\text{env} \vdash e_1 \diamond_b e_2 \Rightarrow e_1' \diamond_b e_2}$$

$$[\text{BIN\_2}] \quad \frac{\text{env} \vdash e_2 \Rightarrow e_2'}{\text{env} \vdash v_1 \diamond_b e_2 \Rightarrow v_1 \diamond_b e_2'}$$

[BIN_3] $\text{env} \vdash v_1 \diamond_b v_2 \Rightarrow v$

where $v = v_1 \bar{\diamond}_b v_2$

$\diamond_b$	$\bar{\diamond}_b$
`+`	$+$
`-`	$-$
`*`	$\cdot$
`/`	$\div$
`%`	modulo
`<`	$<$
`<=`	$\leq$
`==`	$=$
`>=`	$\geq$
`>`	$>$
`!=`	$\neq$
`&&`	$\wedge$
`||`	$\vee$

Access Expression

[VAR] $\text{env} \vdash a \Rightarrow v$

where $v = \text{env}(a)$

Unary Expressions

$\diamond_u \in \{$`!, -`$\}$

$$[\text{UNARY}] \quad \frac{\text{env} \vdash e \Rightarrow e'}{\text{env} \vdash \diamond_u e \Rightarrow \diamond_u e'}$$

[NOT_1] $\text{env} \vdash$ `!true` $\Rightarrow$ `false`

[NOT_2] $\text{env} \vdash$ `!false` $\Rightarrow$ `true`

[NEG] $\text{env} \vdash$ `-`$v_1 \Rightarrow v$

where $v = -1 \cdot v_1$

A.4 Semantics for Statements

Assignments

$$[\text{ASS\_1}] \quad \frac{\text{env} \vdash e \Rightarrow e'}{a\texttt{=}e,\ \text{env} \Rightarrow a\texttt{=}e',\ \text{env}}$$

$$[\text{ASS\_2}] \quad a\texttt{=}v,\ \text{env} \Rightarrow \text{env}[a \mapsto v]$$

Conditionals

$$[\text{If-Else\_1}] \quad \frac{\text{env} \vdash e \Rightarrow e'}{\texttt{if(}e\texttt{) \{}S_1\texttt{\} else \{}S_2\texttt{\}},\ \text{env} \Rightarrow \texttt{if(}e'\texttt{) \{}S_1\texttt{\} else \{}S_2\texttt{\}},\ \text{env}}$$

$$[\text{If-Else\_2}] \quad \texttt{if(true) \{}S_1\texttt{\} else \{}S_2\texttt{\}},\ \text{env} \Rightarrow S_1,\ \text{env}$$

$$[\text{If-Else\_3}] \quad \texttt{if(false) \{}S_1\texttt{\} else \{}S_2\texttt{\}},\ \text{env} \Rightarrow S_2,\ \text{env}$$

$$[\text{While}] \quad \texttt{while(}e\texttt{) \{}S\texttt{\}}, \text{env} \Rightarrow \texttt{if(}e\texttt{) \{}S\texttt{; while(}e\texttt{) \{}S\texttt{\}\} else \{skip\}},\ \text{env}$$

$$[\text{EVENT}] \quad \texttt{when(}e\texttt{)},\ \texttt{env} \Rightarrow \texttt{BLOCKED(}e\texttt{)},\ \texttt{env}$$

$$[\text{BLOCK}] \quad \frac{\text{env} \vdash e \Rightarrow^* \texttt{true}}{\texttt{BLOCKED(}e\texttt{)},\ \text{env} \Rightarrow \texttt{skip},\ \text{env}}$$

Control Structures

[SKIP] $\texttt{skip},\ \text{env} \Rightarrow \text{env}$

[CONC-1]
$$\frac{S_k,\ \text{env} \Rightarrow S_k',\ \text{env}'}{\begin{array}{l}\texttt{fork } \{S_1\} \texttt{ and } ..., \texttt{ and } \{S_{k-1}\} \texttt{ and } \{S_k\} \texttt{ and } ..., \texttt{ and } \{S_n\},\ \text{env} \Rightarrow \\ \texttt{fork } \{S_1\} \texttt{ and } ..., \texttt{ and } \{S_{k-1}\} \texttt{ and } \{S_k'\} \texttt{ and } ..., \texttt{ and } \{S_n\},\ \text{env}'\end{array}}$$
where $1 \leq k \leq n$

[CONC-2]
$$\frac{S_k,\ \text{env} \Rightarrow \text{env}'}{\begin{array}{l}\texttt{fork } \{S_1\} \texttt{ and } ..., \texttt{ and } \{S_{k-1}\} \texttt{ and } \{S_k\} \texttt{ and } ..., \texttt{ and } \{S_n\},\ \text{env} \Rightarrow \\ \texttt{fork } \{S_1\} \texttt{ and } ..., \texttt{ and } \{S_{k-1}\} \texttt{ and } ..., \texttt{ and } \{S_n\},\ \text{env}'\end{array}}$$
where $1 \leq k \leq n$

[CONC-3] $\texttt{fork } \{S\},\ \text{env} \Rightarrow S,\ \text{env}$

[SEQ-1]
$$\frac{S_1,\ \text{env} \Rightarrow S_1',\ \text{env}'}{S_1\,;S_2,\ \text{env} \Rightarrow S_1';S_2,\ \text{env}'}$$

[SEQ-2]
$$\frac{S_1,\ \text{env} \Rightarrow \text{env}'}{S_1\,;S_2,\ \text{env} \Rightarrow S_2,\ \text{env}'}$$