Reachability analysis for continuous one-counter automata

Lars Van Roy

dept. of Mathematics and Computer Science
University of Antwerp

Supervised by
Guillermo Alberto Perez
Tim Leys

August, 2021

ABSTRACT

Reachability is an important metric and lack thereof is a known problem that can heavily affect the efficiency of code. It has been proven that reachability is decidable for continuous one counter automata which we will use in our approach. The proposed approach will provide a means to convert C code to continuous one counter automata followed by an algorithm which is capable of analysing the reachability of each of the nodes within the generated automata. This is an extension to existing research in which we already provided a means to perform the conversion from code to automata.

The newly added approach to compute reachability will make use of reachability intervals, which will be used to represent the set of values which a conditional variable can be equal to at any point throughout the evaluation of the automaton. There will be two variations, one for parametric counter automata and for non-parametric counter automata.

Finally, we will apply the combination of approaches on an existing project called xrdp, as to determine the usefulness of the approach. This application resulted in a total of 33 functions being identified as analysable of which all were correctly identified as fully reachable. To ensure that non reachable cases can also be handled further manual evaluation was performed which resulted in correct identification for all considered cases.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

NEDERLANDSTALIGE SAMENVATTING

Een gekend probleem bij het schrijven van code is het vinden van lijnen code die niet bereikbaar zijn [12, 6]. Onbereikbare code kan veel oorzaken hebben maar één van de moeilijker te analyseren situaties zijn condities die nooit voldaan worden. In deze gevallen is de code omringd door de conditie niet bereikbaar en zou deze dus eigenlijk weg moeten.

Niet bereikbare code is een probleem en kan zelfs kritiek worden in energie gelimiteerde omgevingen zoals ingebedde systemen [10]. In deze situaties is het vitaal dat onze code zo efficiënt mogelijk is en dus is er absolute nood aan een methode die niet bereikbare code kan detecteren.

Continue één counter automaten zijn hiervoor een zeer interesante oplossing, aangezien het bewezen is dat de bereikbaarheid aantoonbaar is voor deze automaton [5, 8, 1]. In dit onderzoek zullen we een praktische implementatie geven voor de theoretische werkwijze gegeven door Michael Blondin et al. [1]. Deze theoretische implementatie was specifiek opgezet om gebrek aan bereikbaarheid te kunnen aantonen.

De beperking om maar één parametrische klok (of counter) toe te staan heeft een groot effect op het bereik aan code dat we kunnen analyseren. Dit is echter onvermijdelijk aangezien er geen bewijs is voor de bereikbaarheid van continue automaten met twee counters. Voor drie counters en meer is er bewezen dat dit niet beslisbaar is [4].

In dit onderzoek zulen we een werkwijze geven om C code om te zetten naar continue één counter automaten. Gebruik makend van deze gegenereerde automaten zullen we analyseren of er in deze code lijnen zijn die niet kunnen bereikt worden. Dit zal gebeuren aan de hand van de volgende stappen:

1. Voer pre analyse uit en analyseer alle compiler specifieke code
2. Converteer de C code naar een parse tree met behulp van ANTLR
3. Voer context vrije reducties uit op de parse tree

4. Voer context sensitieve reducties uit op de parse tree
5. Verifieer of de parse tree converteerbaar is naar een automaat
6. Genereer de automaat
7. Analyseer de bereikbaarheid
8. Raporteer onbereikbare code indien deze gevonden is

De code zal, buiten het feit dat deze maar één conditionele variabel kan hebben, ook beperkt worden in de operaties die hierop kunnen uitgevoerd worden. Aan een conditionele variabel kunnen enkel constante of parametrische waarden toegewezen worden of combinaties van sommen en aftrekkingen met constante of parametrische waarden. De conditionele waarde mag verder enkel met parameters of constante waarden vergeleken worden met de condities is gelijk aan, is niet gelijk aan, is strikt kleiner dan, is kleiner dan, is groter dan en is strikt groter dan. Ten slotte mogen de parameters zelf nooit aangepast worden tijdens de evaluatie van de functie.

Niet alleen de functie inhoud maar ook de definitie zal een beperking krijgen, zo accepteren we enkel parameters van het type integer.

1. INTRODUCTION

A well known problem in writing code is determining whether or not every line of code is reachable. It is often impossible to properly determine whether or not a line of code guarded by a conditional statement is going to be reachable in any possible execution of the code [12, 6]. Unreachable code is never desired and is an ever more pressing issue for energy restrained environments such as embedded computer systems [10]. In these systems, we want our execution to be as efficient as possible, and therefore avoid conditions that can never be satisfied or lines that can never be reached. To remedy this, a proper unreachable code detection approach is needed.

In order to analyse the reachability, we will make use of the fact that code is easily convertible to automata. All statements within the code that are related to counter updates can directly be converted to operations within an automaton. Each line of code can be represented by one or more nodes and a traversal of an automaton can be directly linked to the traversal of the code itself.

Continuous one counter automata are especially interesting as it has been proven that reachability is decidable for this type of automata [5, 8, 1]. In this paper we will provide a practical implementation for the theoretical approach that was provided by Michael Blondin et al.[1] which can be used to prove reachability of nodes.

Adding the restriction of only allowing a single parametric clock has a big effect on the types of code we can analyse. However, this is unavoidable, since there is no proof for continuous two counter machines. Three counter machines and up have been proven to be undecidable [4].

In this paper we will provide the overview of an approach which will convert C code to a continuous one counter automaton. Using the result of this first step, we will then provide a means to analyse the reachability of any of the nodes of said automaton. The following steps are taken without our approach to prove the existence of dead code.

1. Perform pre analysis to resolve all C macro's
2. Convert C code into a parse tree, using ANTLR
3. Perform context free reductions on this parse tree
4. Perform context sensitive reductions on this parse tree
5. Validate the resulting parse tree, to be conforming to the code constraints
6. Generate the counter automaton
7. Analyse which nodes are reachable
8. Link potential unreachable statements back to the lines within the original code base

The code can at most contain one counter and for the continuation of this paper we will only consider examples where this holds. Cases with no counter will not be considered as these are assumed to be trivial. Furthermore, the operations on the counter will be restricted, the counter can only be updated via assignments with parameters or constants, and the counter can only be compared using equality, inequality, strict less than, strict greater than, less than or equal, greater than or equal, and only to parameters or constants. As a final code constraint, the function parameters are not allowed to be altered throughout the duration of the code.

Other than the described code constraints, functions must have integer parameter types (if any) to be converted into a counter automaton.

1.1 Research questions

The objective of this research can be summarised in two core research questions:

Question 1:

Can we practically determine the reachability of continuous one counter automata?

Question 2:

Can we identify dead code in software written in the C language by analysing the reachability of the corresponding counter automata?

To solve these questions we will give two different approaches that are capable of determining reachability of nodes in both parametric and non-parametric one counter automata. Both approaches will additionally be applied to xrdp, a project used to login to remote machines that is entirely written in C.

The solution to the questions above can be divided into three different steps, where the final step will be further divided in two different solutions.

Convert the C code to a one counter automaton
This approach was previously done in a separate research but the general approach will be repeated here. It will furthermore be modified and perfected as an improvement of the existing work. This approach will be discussed in Section *4.6 Counter automaton generation* in which we will discuss the different steps part of converting C code.

Specify a textual format for automata to allow reading and writing of automata
The result of the previous step will be stored using the dot format which is part of Graphviz [3]. This input should be readable to perform the reachability on it. We furthermore want to be able to provide this approach as a stand-alone concept, thus allowing the reachability analysis to be performed on arbitrary automata without the coupling with the compiler. In the Section *4.7 Automaton specification format* we will discuss the steps needed to read an automaton from a dot file.

Perform reachability analysis on non-parametric continuous one counter automata
The first reachability analysis approach will be used to analyse the reachability of non-parametric continuous one counter automata. This implies that there can never be any operation applied to a counter with parameters involved. This approach will iteratively go through the automaton by making use of reachability intervals. The different steps and operations will all be discussed in detail in Section *4.8 Reachability of non-parametric COCA*.

Perform reachability analysis on parametric continuous one counter automata

A secondary approach for reachability analysis will be given in Section *4.9 Reachability of parametric COCA*. This approach will estimate the reachability intervals which were also used for the non-parametric variant. We will define conditions on the intervals and employ a SAT solver to guess a valid reachability interval configuration. The SAT solver will also support parametric automata and this approach will therefore be used as an alternative to the first approach in case the function uses parameters.

2. MOTIVATION

Code efficiency is one of the core aspects on which the quality of code is measured. Inefficient code can incrementally lead to a slower product, even though it only represents a small part of a much larger whole. This is because code reuse is a critical aspect in Object Oriented Programming (OOP) [9]. In OOP we tend to write code by encapsulating it within classes (objects) based on the functionality they provide. This allows us to simply reuse that object whenever we need that specific functionality. As a direct result of this, every time we introduce code inefficiency, this will get replicated whenever we need use of the specific functionality in which the inefficiency was introduced.

Code inefficiency can also indirectly affect other projects. Many languages such as C, C++, Java and Python support the option to introduce libraries. These libraries are other projects which will entirely be introduced within the current project as to prevent the need to add a new implementation for a functionality which has already been implemented elsewhere. The same effect can occur here where an inefficiency within the library will get propagated to the other projects which make use of said library. As a means to speed up development there are even tools such as Hunter [11] which are intended to query large code bases to find code which already implements the feature that is needed, thus increasing the likelihood of using existing code.

One of the contributors to the efficiency of code is the reachability of code. Dead code implies that there is a piece of code that can never be reached and is thus useless. This might be due to a certain function simply never being invoked and in most cases this will not affect the actual efficiency of the code as this is trivially filtered out by compilers. However, guarded (conditional) pieces of code can be an issue in case they are not satisfiable. This can not simply be resolved by a compiler nor will it always get caught by all IDE's (Integrated Development Environment). A dead conditional statement will get evaluated every time said piece of code is used and can therefore contribute to a major efficiency reduction considering the code reuse that was mentioned above.

Popular IDE’s do already implement reachability analysis up to a certain degree. However, the dead code elimination they perform is shallow and limited. The code example given below was executed using Clion and Eclipse, two of the more commonly used C IDE’s. While it is known that both can find some dead code, it was not able to resolve this case, where the condition on line 4 can never evaluate to true. This is because the for statement on line 3 will always prevent line 4 from evaluating to true and yet they did not detect this.

```c
 1 bool is_mul(int a, int b) {
 2   int temp = b;
 3   for(; temp > 0;){
 4     if(temp == 0){
 5       break;
 6     }
 7     temp -= a;
 8   }
 9   return temp == 0;
10 }
11
12 int main() {
13   printf("5 is a multiple of 2: %d\n", is_mul(2, 5));
14   printf("6 is a multiple of 2: %d\n", is_mul(2, 6));
15 }
```

Listing 2.1: Example C code where Eclipse and Clion would not identify dead code.

The approach provided within this paper will be able to detect this, as it can simply evaluate the possible values with which line 5 can be reached, it will result that there is no sequence in which line 5 can be reached and we can therefore conclude that the condition surrounding this statement can never be satisfied.

In the past research has been done regarding dead code analysis. In 1994 there was a research done by Jens Knoop et al. [7] where they attempted to analyse partial dead code. Partial dead code is code that is only dead in some of its execution paths. It will aim to resolve this by changing the branching structure or the semantics of the program which will result in the bit of code no longer being dead in any program path. This resolves one of the cases that is not touched by the approach prescribed within this paper and they can therefore both provide benefits to projects without any redundancy in evaluation.

A different research has been done on the analysis of the data model behind software in an attempt to identify dead code at the declaration level. This was done in 1997 by Yih-Farn R. Chen et al. [2] and was applied on C++ projects. It concluded by showing that it was in fact capable of identifying a significant amount of dead functions and variables. This is however on a different level than the current research will work on. The research in question did analysis on the data model and might have therefore missed a significant amount of dead code that was introduced within the code itself.

As can be concluded from the examples given above, there has been research on dead code analysis. This research is however dated and it did not specifically cover the environment which we will cover within this research. It is furthermore shown that within the majorly used IDE's such as Eclipse and Clion the cases which we will attempt to detect with this new approach are not yet covered. We can conclude that this research brings additional value to what is known about dead code analysis and can further enhance the optimization of software.

3. PRELIMINARIES

This paper will make use of counter automata and more specifically continuous one counter automata. These automata are derived from the general definition of automata and signify an extension by adding the notion of counters.

3.1 General automata

The automata used in this research are a specific subset of the generally defined automata. The general definition for an automaton A is given by the tuple (Q, Σ, δ, q_0, F) where

- Q is the set of states
- Σ is a finite set of symbols, called the alphabet of the automaton
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of states of Q defining the final states,

3.2 Continuous one-counter automata

A continuous one-counter automaton (COCA) is defined as a tuple $\mathcal{A} = (Q, q_0, T, \tau)$ where

- Q is a finite set of states
- $q_0 \in Q$ is the initial state
- $T \subseteq Q \times op \times Q$ is the transition relation with op the set of operations

- I is a representation of all intervals over $\mathbb{Q}$
- $\tau : Q \to I$ is the mapping of the reachability intervals

At all times the automaton will be in a configuration (q, c) where $q \in Q$ and c the counter valuation where $(q_0, 0)$ is the initial configuration. The counter valuation $c \in \mathbb{N}$ represent one single value for which the counter is evaluated. We are interested in the reachability sets for each state q where the reachability set for q is $\{v | (q, v) \text{ is a reachable configuration}\}$.

The τ parameter represents a function that maps all nodes to their corresponding bounds. For a counter value to be part of the reachability interval of a node, it must respect the bounds implied by the τ function. The τ entry for a node q is represented by the an interval $[\underline{\tau(q)}, \overline{\tau(q)}]$ where $\underline{\tau(q)}$ and $\overline{\tau(q)}$ represent the lower and upper bounds of a node respectively.

The set of supported constraints within our evaluation are visualised by the *cond* set below with ϵ representing the empty condition. Note that this is not binding for the code that we can analyse, several conditions might be convertible to constructs using only supported statements.

$$\mathit{cond} = \{\leq c, = c, \geq c, \leq p, = p, \leq p : c \in \mathbb{N}, p \in P\} \cup \{\epsilon\}$$

Parametric counter automata, as compared to regular counter automata, allow counters to be modified by parameters as well as constants. As the counters are assumed to be integers, we will only allow integer type parameters. We will consider the following set of allowed operations where P is the set of parameters, the ϵ symbol will be used to represent an empty label.

$$\mathit{op} = \{+c, -c, +p, -p : c \in \mathbb{N}, p \in P\} \cup \{\epsilon\}$$

The final state is omitted from this definition as we are not interested in this. In general the reachability of every line of code will be under scope and not just a single one. Furthermore, making the assumption that the final state is the only state of which we want to analyse reachability creates a significant reduction in the use cases of this approach, and was therefore discarded.

3.3 COCA reachability

In this section we give an overview of how we approach the concept of reachability. The provided concept was proven by Michael Blondin et al. [1] and will be the basis for the remainder of the approach.

Throughout the evaluation we will be tracking the reachability interval(s) of all nodes denoted by $reach(q)$. $Reach(q)$ will represent the set of possible counter values c that can exist in a configuration (q, c).

A *run* ρ is a sequence of configurations $(q_0, v_0)(q_1, v_1)...(q_n, v_n)$ such that the following holds with T the set of all transitions:

- $\forall i \in [0, n) \; \exists \alpha \in (0, 1] \; \exists t \in T : (q_i, v_i) \xrightarrow{\alpha\, t} (q_{i+1}, v_{i+1})$
- $\forall i \in [0, n] : v_i \in reach(q_i)$

A state q_i, and thereby represented line l, will be considered reachable if there exists a *run* ρ with the sequence $(q_0, v_0)(q_1, v_1)...(q_i, v_i)$. We can therefore convert the reachability of lines of code to the the following problem.

COCA reachability:
Given a COCA $\mathcal{A}$, an initial configuration (q_0, v_0) and a target configuration (q_n, v_n) does there exist a run $\rho = (q_0, v_0)...(q_n, v_n)$?

In this paper we will give two different approaches that will approximate the $reach(q_i)$ intervals for each node $q_i \in Q$, thus attempting to prove whether or not such a run exists. This is equivalent as the following results from the definition given above:

- $reach(q_i) \neq \emptyset \rightarrow \exists \rho = (q_0, v_0)...(q_i, v_i)$

4. ANALYSIS OF DEAD CODE IN C

The approach given within this paper consists out of two major phases. The first phase will consist of the conversion from C code to one counter automata. The second phase will use the result from the first phase, and perform a reachability analysis on these generated automata. The first phase was already covered in preceding research and the majority of the notes on the approach in the first phase were taken from that research. They were copied here as this research will conclude the unfinished work that was done there, as well as make continuous references to the flaws which arose from it.

The first step of the conversion will be the preprocessing step in which all c compiler macro's will get resolved.

Next, the second step starts with an initialization, which consists of the generation of a parse tree. This will be done using the default C grammar, with the extension of the boolean type as this is not within the C language by default.

Using the resulting parse tree we will perform a simple context-free optimization where we will remove all needless nodes. This is done as an intermediate step as it will allow us to make assumptions in the second part of the optimization, where we will do a more in depth optimization cycle. This second part will perform optimizations based upon the context in which the nodes appear, and will try to simplify the syntax tree as much as possible. The resulting tree will contain no more potentially ambiguous nodes and all

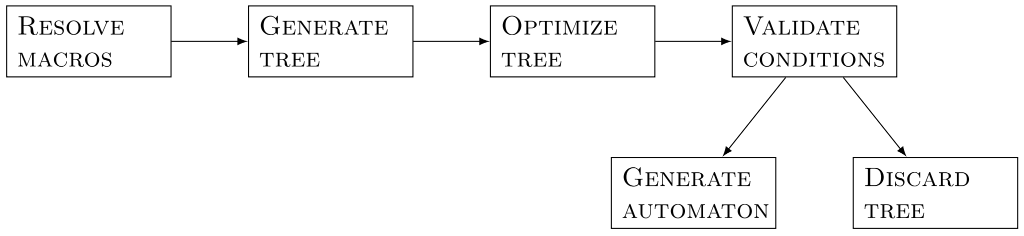

Fig. 4.1: Overview of the automaton conversion process

remaining nodes will be those relevant in the remainder of the evaluation.

The fourth step will be an analysis on whether or not the resulting abstract syntax tree conforms to all earlier specified requirements and, if this is the case, we will continue with the fifth step, which will be the generation of the counter automaton.

For a function to conform the arguments (if any) must be of type integer and must never be mutated within the function body, there can only be one counter and the counter can only be mutated using the operations part of the *op* set displayed below.

$$op = \{+c, -c, +p, -p, \leq c, = c, \geq c, \leq p, = p, \geq p : c \in \mathbb{N}, p \in P\} \cup \{\epsilon\}$$

The generation of the automaton within the fifth step is done by traversing the tree and converting every statement that is directly relevant to the counter modifications to their corresponding automaton labels.

4.1 Preprocessing

The program will start by resolving all the compiler macros that are part of C. These are intended to be resolved by the compiler at compile time, and are often dependent on which compiler was chosen/the environment on which the code is executed. For our use case I decided to select the global variables in such a way that these represent a windows machine that is using 64 bits. This is however not at all restricted to that and could further be expanded to support any environment and any combination of global variables possible. This selection is in essence not relevant as it should not affect the presence of dead code.

Something else that is part of these macros are the include statements. These statements were partially resolved in the sense that only the imports from the current project were handled. Includes from global C libraries are not relevant for the type of dead code elimination we attempt to do within this project and we therefore decided that the addition of said libraries would only result in a much larger code base to be analysed without any additional benefits for the dead code analysis of the actual project.

The if/elif/else macros were simply resolved by converting them to boolean expressions which can then be resolved by any boolean solver. These macro's are intended to identify whether or not global variables are defined and/or are of a current value and are thus relatively simple to resolve.

Finally, the define and undefine macros were resolved by simply tracking the values they defined and/or undefined throughout the evaluation of the code.

4.2 Initialization

Using the flattened result of the preprocessor, the next step will start by running the generator ANTLR compilation unit on the given C code. In case there are any issues regarding simple compilation rules, the program will exit with an error, indicating the issues in the code. For the remainder of the execution, the code is assumed to be correct, and there will be no regard for potential issues within the code itself.

4.3 Context-free abstract syntax tree generation

Using the parse tree resulting from the ANTLR compilation, we will try to generate an abstract syntax tree that is as simple as possible without regarding any context.

First of all, the code uses the notion of a node stack. This node stack will be used to store the nodes that are currently being evaluated.

The first loop will operate as a visitor, which will traverse the parse tree in a depth first manner. The first discovery of a node is done by calling the enter function corresponding to the type of node. As soon as all children of said node have been evaluated, the corresponding exit function will be called.

However, the creation of nodes is not the same in all situations, in some cases, a specific kind of node will be created and in other cases it will not, depending on the rule used for the current node (which we can determine from the context of the current node). In order to know whether or not we need to pop a node from the node stack, we will allow the code to look at the top node in the stack, and evaluate the type of set node. No other context related evaluations will be allowed.

Context-free analysis assignmentExpression example
To look at a concrete example, consider the following rule in the grammar:

```
assignmentExpression
:   conditionalExpression
|   unaryExpression assignmentOperator assignmentExpression
|   Digitsequence;
```

The rule for conditionalExpression needs to be there, as there is a chain of expressions, and some expressions need to be higher in the order than others, therefore all expressions with a higher priority need to be evaluated before the lower priority expressions. This is done by adding a rule to a lower priority expression in each expression class. It is however undesirable for these nodes to be added. Without regard for context in other nodes, we can simply look at the current node to see whether or not the conditionalExpression line is used (which ANTLR generated context allows us to do).

Since there is a chance no new node was added within the enter function, we will also be required to evaluate the top of the stack, to see whether or not the top of the stack is a node we need to pop, the code will do the same check that was done in the enter function of the traversal.

4.4 Context-sensitive abstract syntax tree reduction

There is still the need for reductions. We will attempt to implement constant propagation and substitution, in which we compute constant expressions, and replace variables with known value, by their actual value. Furthermore we will try to rephrase expressions by expressions with the same meaning, so that all expressions, if possible, are in an acceptable format for the counter automaton.

The context-sensitive reduction will also keep track of a lot of data about the variables in a symbol table. It will track whether or not variables are current initialized with a known value, values at certain points within the execution, it will track struct, union and enum definitions used for folding and so on. It will keep track of the kind of variables we have (eg. counters and parameters). All of this info will be used to enable the operations this cycle will perform, but will also be used in the validation loop.

The program will do this by traversing the abstract syntax tree resulting from the previous step, but will now specifically go through the children of relevant expressions (eg. assignment expressions, additive expressions, ...).

Important to note is that no folding/substitution will occur within loops, as these operations occur a variable number of times, and since there is no way of determining the exact number of times without evaluating the expression, this is considered too complex and often impossible, as counters may depend on parameters, which need to be chosen at evaluation time.

Context-sensitive reduction example: constant propagation

Another example of a needed reduction, is constant *propagation* and *folding*. It is possible that a code segment would initially be rejected, due to use of unwanted variables, while in essence, these variables are nothing but variable representations of results of constant expressions. By *folding* we will attempt to simplify constant expressions by replacing them with their results. Once simplified to a constant value we will then attempt to *substitute* the variables by replacing further uses with said value.

Consider the code in Listing 4.1 below. It would initially be rejected, because the *counter* variable is initialized with a value that is neither a constant nor a parameter.

```
1  bool divide(int p, int n){
2    int variable = 5;
3    variable += 20;
4    int counter = variable;
5    while(counter > 0){
6      counter -= n;
7    }
8    counter += 1;
9    counter -= 1;
10   if(counter == 0){
11     return true;
12   }
13   return false;
14 }
```

Listing 4.1: Example of a function in need of constant propagation

However, due to constant propagation and folding, we can simplify the two statements on line 2 and line 3, so that they become one simplified operation. This can be seen in Listing 4.2.

```c
int variable = 25;
```

Listing 4.2: Result of simplifying operations on *variable*

Now that *variable* is just a constant, we can substitute its value which leads us to the situation given in Listing 4.3.

```c
int variable = 25;
int counter = 25;
```

Listing 4.3: Result of substituting *variable*

Now *variable* has no further benefit for the execution and will be dropped.

```c
bool divide(int p, int n){
  int counter = 25;
  while(counter > 0){
    counter -= n;
  }
  counter += 1;
  counter -= 1;
  if(counter == 0){
    return true;
  }
  return false;
}
```

Listing 4.4: Final version of the simplified function

The counter will not be substituted beyond the initial loop. Whenever a variable gets altered within a conditional scope, the variable is considered to be uncertain, and further folding based on earlier values will not be allowed. Finally, the original issue in Listing 4.1 has now been resolved and the resulting code in 4.4 now satisfies all conditions for conversion to counter automata.

Context-sensitive reduction example: addition folding

As described earlier, we do allow *additions* and *subtractions* of the form $-x, +x, -n$ and $+n$, where $x \in P$ or $n \in \mathbb{N}$ with P the set of parameters. Given that the formation, described in Figure 4.2, occurs in the automaton we would technically not be allowed to accept this, as this is not directly in the required format.

However, the context-sensitive reduction will be able to reduce the earlier specified example to the automaton in Figure 4.3. This automaton will be acceptable, as this is a direct use of an allowed operation.

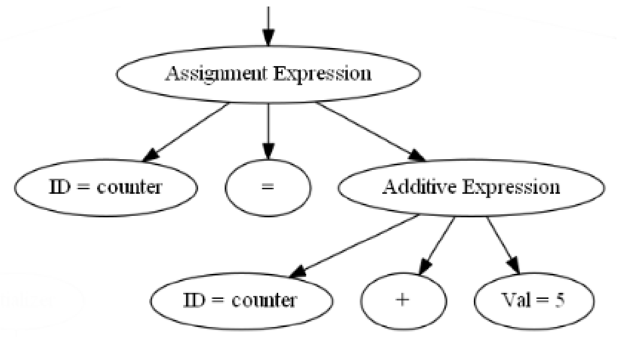

Fig. 4.2: An Assignment Expression with counter addition as value

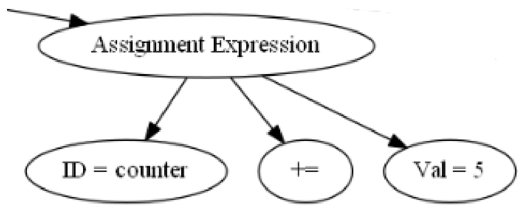

Fig. 4.3: An Assignment Expression with constant as value

4.5 Abstract syntax tree validation

The fourth loop will iterate over the abstract syntax tree resulting from the previous step, and while doing so, it will evaluate the nodes that occur and the context in which they occur.

The majority of the node evaluation is represented in a list of unsupported nodes. Most of these nodes are head nodes for certain kinds of expressions which will never be supported.

$$unsupported = \left\{ \begin{array}{l} Multiplication\ Expression, sizeof, \_Alignof, \\ \&, *, -, +, ., ->, !, \ , Cast\ Expression, \\ Shift\ Expression, Bitwise\ And\ Expression, \\ Bitwise\ Or\ Expression, Bitwise\ Xor\ Expression, \\ *=, /=, \%=, <<=, >>=, \&=, ^{=}, |=, \\ Logical\ And\ Expression, Logical\ Or\ Expression, \\ Additive\ Expression \end{array} \right\}$$

Whenever we encounter a node part of the *unsupported* set, the current situation will be inspected. For example, if a *Multiplication Expression* node occurs, inside a subtree of an *Assignment Expression* node where an assignment to a counter was specified, we know that we are in an invalid state, as counters can only be updated via assignments, additions or subtractions.

The same logic holds for subtrees of conditional expressions. In case we have a conditional expression where the subtree contains a *Bitwise And Expression* node, we know that this is an invalid state, as constant conditions have already been folded in the previous step, this must mean that there is a counter in the condition, but counter comparisons are not allowed to contain and statements.

Furthermore, all variable usage will be tracked. We will allow multiple counters to exist, as long as we can substitute one general counter without having any conflicts, in other words, there cannot be any overlap between counter usages. This is simply tracked by tracking the nodes in which the counter variables occur, with the exception of when they occur within conditional scopes. If this is the case, their first usage is set to the first line of the scope, so that overlap does occur when a different counter is used for the condition of the scope.

Finally, the initial value will be tracked. In case there is a declaration with an initial value, this value will be chosen, otherwise it will be set to 0, which is the default integer value in C. This initial value allows us to skip over declarations in the generator. We can simply set the counter value to the initial value of the used counter at that point in time, just before its first usage.

Another aspect that will be tested, is the function definition which can only have integer parameters. If this not the case, an error status will be added.

These conditions will all be tracked separately for each function. Whenever the status list is empty at the end of the evaluation, we assume that no issues occurred, and we say that the function satisfies the requirements, if there are any issues, no counter automaton will be generated, and the errors will be printed to the terminal.

Important to note is that there is no regard for global statements. These are considered to be variable and undetermined. It could be that they can perfectly be used as counters, and that they will never be altered by other functions, but it is impossible to determine this efficiently. Therefore, a counter must be declared within the function scope.

Validation example

When we go back to the earlier mentioned segment of code, given in Listing 4.5, we obtain the automaton displayed in Figure 4.4.

```
bool divide(int p, int n){
  int counter = p;

  while(counter > 0){
    counter -= n;
  }

  if(counter == 0){
    return true;
  }
  return false;
}
```

Listing 4.5: Example of a ready to evaluate function

We can immediately see that the function conforms to the required function definition, as the parameters are of type int, which can be seen from the parameters defined underneath the *Parameter Type List* node.

From the cleaner we know which variables are counters, and we can see that the first declaration is a counter, but we will skip the declaration, as we know the value which is stored in the initial value, the initialization will happen before the first operation on the counter occurs.

Next, we encounter the *Iteration statement*, for which we will check the condition with the constrained conditional variable enabled. With this variable enabled, we can simply continue traversing the nodes, and in case there are any unsupported nodes, the corresponding error would automatically be generated. However, there will be no corresponding nodes, so we will leave the *Relational Expression* without any error statuses, and we will disable the constrained conditional variable.

Next we will enter the *Compound Statement*, which symbolizes the inner scope of the iteration statement. We will encounter an *Assignment Expression* with a variable that we know is a counter, we will therefore enable the counter assignment variable, while traversing the children of this node. No invalid nodes will appear, and we will disable the counter assignment variable when leaving the *Assignment Expression* node.

The selection statement will be evaluated in the exact same manner as the Iteration Statement was evaluated, and as the jump statement holds no

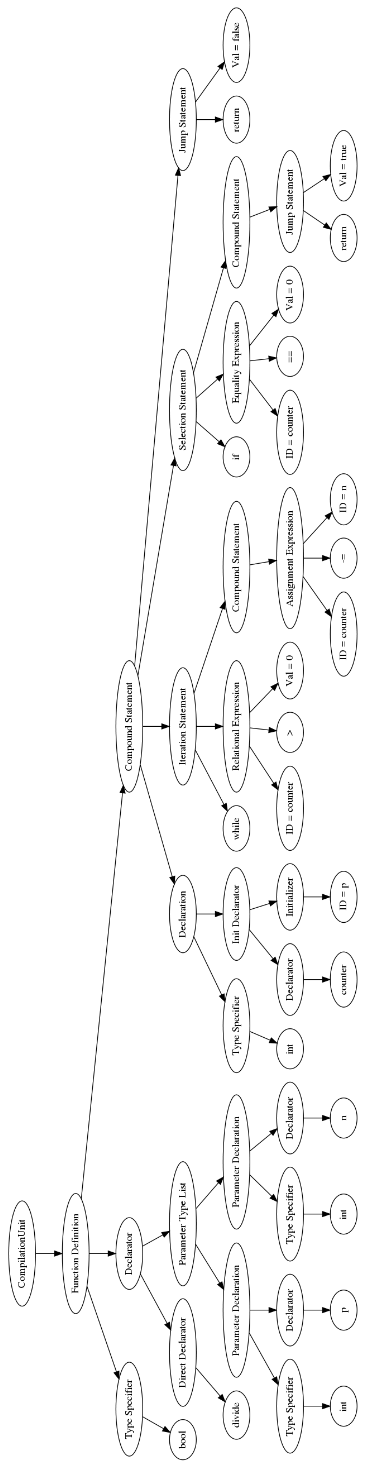

Fig. 4.4: The cleaned Abstract Syntax Tree for the divide function

special operations, we can conclude that this function satisfies the requirements for continuous one counter automaton generation.

4.6 Counter automaton generation

The final step of the conversion will be the counter generation itself. In this loop, there will be no more regards to possible invalid statements. At this point we assume everything to be known, and we can therefore simplify our analysis. Recall that the following list is the collection of allowed labels for the counter automaton.

$$op = \{+c, -c, +p, -p, \leq c, = c, \geq c, \leq p, = p, \geq p : c \in \mathbb{N}, p \in P\} \cup \{\wedge\}$$

This allows us to just check the head nodes related to such expressions.

4.6.1 Functions

First of all, we need the notion of the *Function Definition* statements. The generator will generate a counter automaton for each function, and needs to know when these start.

4.6.2 Assignments

Other than functions, we will also need to know when assignments to counters occur, so that we can add the proper transition labels. We only need to check whether or not we are assigning to a counter, if so, we read the operation used, and the variable that gets assigned. Valid counter assignments need to be of the form $c\ op\ x$ where c is the counter, $op \in \{=,\ +=,\ -=\}$ and $x \in P \vee x \in \mathbb{N}$.

As mentioned earlier, assignments are not in the set of operations, but they can be represented using the operators displayed earlier, as can be seen in Figure 4.5. This can be done by using a reset operation, where we first check whether or not the counter is smaller or equal to the desired value. If so, it is either already satisfied, in which case we can just carry on, or it is smaller than the desired value, for which we will add a self loop with +1 as a label, so that we will eventually reach the desired value.

The same thing can be done when the counter is larger or equal to the desired value, if so, it is either already satisfied, in which case we can just carry

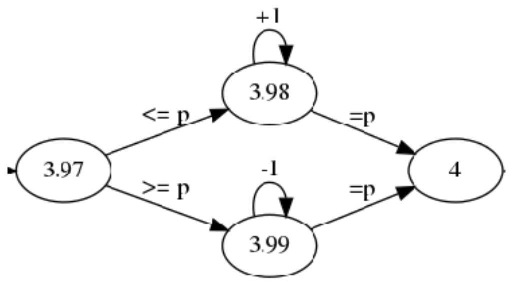

Fig. 4.5: Assignment of a value p to a counter

on, or it is larger than the desired value, at which point we will decrement the counter with a self loop of -1, until the condition is satisfied.

An important thing to remark, is that we are not interested in general properties of runs for these automata. We are interested in the possibility of reachability or in other words, the existence of one path that will take us to a given node. In the given example, there is a path where we would remain in node 3.98 indefinitely but since there is at least one path that would take us to node 4, node 4 is still considered reachable.

4.6.3 Inequality conditions

Another statement we will need to support, are inequality conditions. If we want our conditional statements to be properly functioning, we will need to be able to generate opposing conditions, but the opposing condition of the equality condition, is the inequality condition, which we can not directly model using the given operations.

The following code segment in Listing 4.6 makes use of an if statement which has an equality condition in it.

```c
bool is_ten(int c){
  if(c == 10){
    return true;
  }

  return false;
}
```

Listing 4.6: Example of C code containing an equality condition

When looking at part of a simplified counter automaton in Figure 4.6, we will notice the transitions going from the *start_if* node, towards the *end_if* node. There will not be an *else* node, as there is no *else if* or *else* statement.

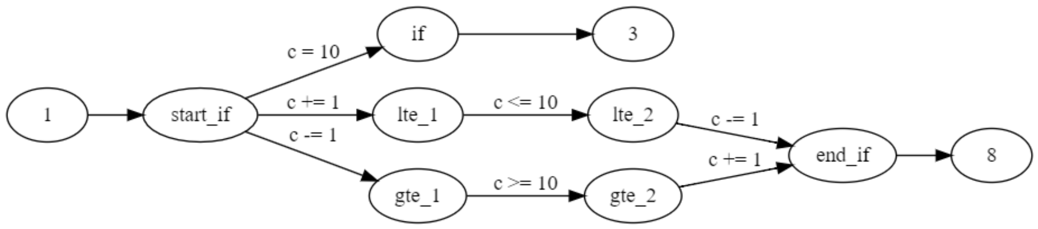

Fig. 4.6: Example of an inequality expression

The first path follows the *lte* nodes, which starts with *c += 1* followed by *c <=10*, followed by *c -= 1*, which results in the same expression as saying the original counter is strictly smaller than 10. The final *c -= 1* is there to reset the counter, so that the counter is back to what it originally was.

The second path follows the *gte* nodes, with the first transition being *c -= 1*, followed by *c >=10*, followed by *c += 1*. This chain states that the counter must be strictly greater than 10.

If either of these can be taken, we can state that the counter must be different to the compared value (in this case 10), as these both express a strict inequality.

4.6.4 Less than and greater than

The final addition to the set of supported statements, are the *less than* and *greater than* expressions. These are needed to model the opposing conditions to greater than or equal and less than or equal respectively.

```c
bool is_ten(int c){
  if(c <= 10){
    if(c < 10){
      return false;
    }
    return true;
  }
  return false;
}
```

Listing 4.7: Example of a code segment using less than and less than or equal conditions

The code segment in Listing 4.7 models both a less than or equal condition and a less than condition. The first condition will need to be opposed with a strictly greater than condition, and the second one with a greater than or equal condition.

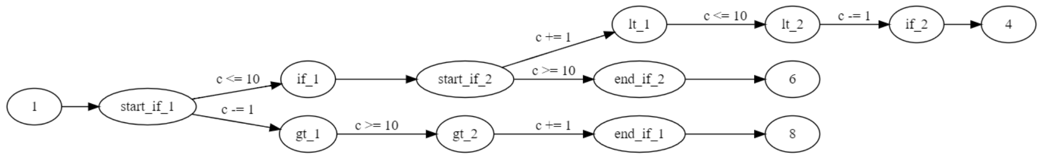

Fig. 4.7: Example of a less than expression and a greater than expression

A simplified version of the counter automaton of this function can be seen in Figure 4.7. This automaton has *start_if_1* as the start of the outer if and it has *if_1* and *end_if_1* as respective if and end nodes. The inner if has *start_if_2* as the start node, *if_2* as the start of the if segment, and *end_if_2* as the end node.

The strict greater than condition, originating from node *start_if_1* and ending in node *end_if_1*, consists of 3 different transitions. The first transitions *c -= 1* and *c >=10*, test if the counter is strictly greater. The third transition *c += 1* is there to make sure that the counter goes back to its original value.

The strict less than condition, originating from node *start_if_2* and ending in node *if_2*, consists of *c += 1* and *c <=a* as their first two transitions, which can only be satisfied in case the counter is strictly less than a. The final condition *c -= 1* is there to make sure the counter goes back to its original value.

4.6.5 Iteration statements

We will also need the notion of iteration statements, which needs 5 additional special nodes to fully support it's functionality. The first node will be the *start of the loop*. The second node is the *pre node* which symbolizes the state after the precondition of the for loop has been evaluated. Next we need *start of the inner segment* to signify the start of the inner segment. This node will be reached by a transition with the condition as a label.

After this node the inner segment will follow, and it will be finalized with the *stop inner segment* node, in case this one is needed. This stop node will symbolize the state after the post expression of the for loop has been evaluated, after this node, a transition will occur to the pre node, to re-evaluate the loop condition.

Finally, one more node is needed, and this node will symbolize the *end of the loop*. This node will always be the last node that gets generated, so that any nodes generated beyond this point will start from this point.

Consider the segment of code in Listing 4.8 for which the code conversion will result in the automaton in Figure 4.8.

```c
bool test(int c){
  for(c = 0; c < 5; c ++){
    continue;
  }
  return true;
}
```

Listing 4.8: Example of an itteration statement

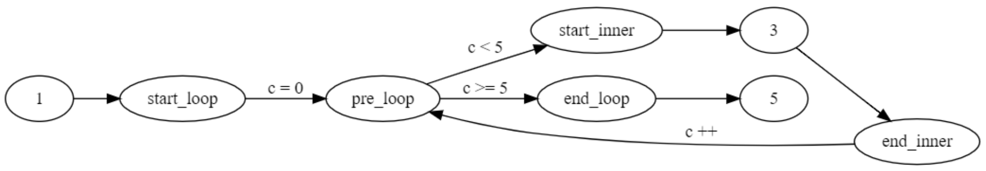

Fig. 4.8: Example of an Iteration Statement

4.6.6 *If statements*

Another important construct are the if statements. These statements will, just as the iteration statements did, require additional nodes to be able to be modelled. If statements require a total of 4 nodes to be modelled.

First of all, we need a *start if* node, from which the conditions will start.

```c
bool test2(int c){
  if(c == 1){
    printf("The counter is equal to 1.");
  }
  else if(c == 2){
    printf("The counter is equal to 2.");
  }
  return true;
}
```

Listing 4.9: Example if statements

The *if* and *else* nodes will be used to signify the start of the if and else segment (if present). Independent of the presence of an *else* node the if configuration will always end in an *end if* node from which point the automaton will carry on to the next line of code.

Consider the function in Listing 4.9. The program will return a more in depth version of the simplified counter automaton displayed in Figure 4.9. The first if statement will start from *start_if_1*, and end at *end_if_1*, the else if will start from *start_if_2*, and end at *end_if_2*.

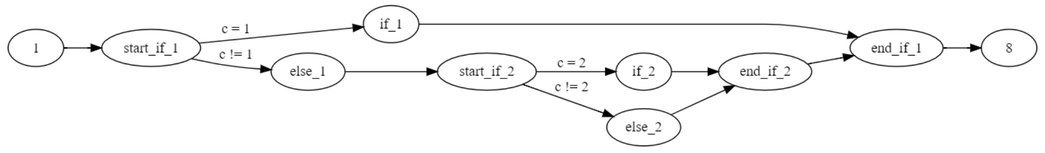

Fig. 4.9: Example of if statements

4.6.7 Switch statements

A second conditional statement that exists in C, is a *switch statement*. This statement allows a variable to be compared against multiple values. What is special about the switch statements, is that if matched with one case, the code of all consequential cases will also be executed, until the end of the switch, or until a jump occurs.

Consider the code snippet in Listing 4.10. In this case, a variable *a* will be compared with multiple values, and depending on the matched case, a number of prints will occur.

```
bool test(int a){
  switch(a){
    case 3:
      printf("The parameter is greater than 2.");
    case 2:
      printf("The parameter is greater than 1.");
    default:
      printf("The parameter value is %d", a);
  }
  return true;
}
```

Listing 4.10: Example of a switch statement

To generate a counter automaton for a switch statement, we will start by generating a chain for the statements in the cases. This is represented by the chain starting from *start_switch*, all the way to *default*, in Figure 4.10.

After the initial node we need to add the conditional branches that correspond to the different cases. A first branch originates from *start_switch*, and can either go to *case_3* in case the counter is equal to 3, or to *not_case_3* in case the counter is not.

There is a second branch from *not_case_3* where we can either go to *case_2* in case the counter is equal to 2, or to *not_case_2* in case the counter is not. Finally, from *not_case_2* we will always go to the *default* node.

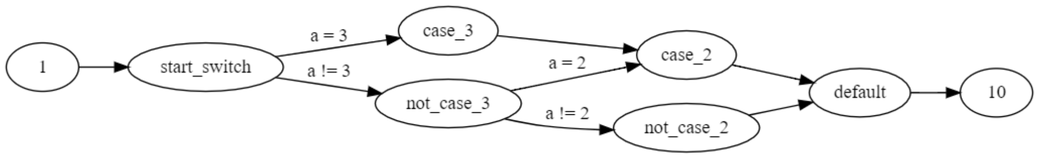

Fig. 4.10: Example of a switch statement

4.6.8 Counter initialization

The counters do not get initialized at the point that they got declared within the code. This is a simple improvement that helps to reduce the number of counters. Consider the code segment in Listing 4.11, in which two different counters are used.

```
bool two_counters(){
  int c = 0;
  int c2 = 0;
  if(c == 2){
    return false;
  }
  else if(c2 == 2){
    return false;
  }
  return true;
}
```

Listing 4.11: Example of a function with two non-overlapping counters

The first counter will be initialized just before the if statement gets used. The second counter will not be initialized at that point, but will only be initialized just before the second if statement gets used. This means that the program will in fact be able to support the given code segment, even though it has two different counters.

4.6.9 Final overview

To give an in depth overview of how the conversion works, we will transform the following code snippet given in Listing 4.12 into a counter automaton. This code consists of multiple if statements, as well as a loop, it makes use of a single counter value, being *counter*, and two parameters. The loop and the final *if* both use inequality expressions, which are representable in a counter automaton, as was described earlier. The first two *ifs* make use of relational expressions with the less than and greater than operators, which are also representable within a counter automaton. The counter is only using additions or subtractions with parameters and none of these parameters will ever be modified, which also conforms to the requirements. Finally, this function has only integer parameters which makes us conclude that this entire function is representable in a counter automaton.

```
bool gcm(int a, int b){
  int counter = a;
  counter -= b;
  int i1 = 1;
  int i2 = 1;
  while(counter != 0){
    if(counter > 0){
      counter -= b;
      i2 += 1;
    }
    else if(counter < 0){
      counter += a;
      i1 += 1;
    }
    if(counter == 0){
      printf("the greatest multiple is equal to ");
      printf("%d * %d = %d * %d = %d.", i1, a, i2, b, i1*a);
    }
  }
  return true;
}
```

Listing 4.12: Example of a more complex function suitable for conversion

The first line is the function definition, this will just be converted to a node with label 1, representing the first line. Since this is the first node added to the counter automaton, it will also be denoted as being the initial node, as can be seen in Figure 4.11.

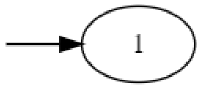

Fig. 4.11: First node in the counter automaton

The second line is of interest to the counter machine, as it is a counter modification. However it will not directly generate any special counter automaton configuration, as declarations of counters are only displayed as soon as a counter is actually used or modified. Since nothing special is generated, it will simply generate a new node representing the current line (being 2), with a transition coming from the last added node before this one, as displayed in Figure 4.12.

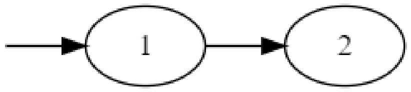

Fig. 4.12: Automaton corresponding to the second line of *gcm*

The third line is an actual modification to the counter, so before we generate any additional nodes regarding the operation specified on this line, we need to add the counter initialization as a *pre* statement for this line. This is represented in the tool by decrementing the current line in steps of 0.01. An assignment operation requires three additional nodes, so we will make use of the nodes 2.97, 2.98 and 2.99 to model the assignment functionality, as visualised in Figure 4.13.

The actual operation specified on line three, is a subtraction of the counter by the parameter b, as is modelled in the sub counter automaton in Figure 4.14.

Line four and five are both statements that are not of relevance to the counter, and will therefore be simply represented as a sequential chain following from the last added node, being node 3, as can be seen in Figure 4.15.

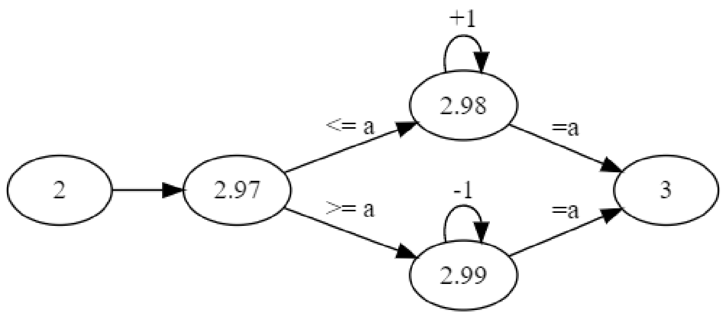

Fig. 4.13: Counter automaton modelling the initial assignment of the counter

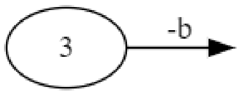

Fig. 4.14: Counter automaton modelling the operation on line two

The sixth line defines a while loop. This requires us to add several nodes that are all related to the statement on line six. The tool will symbolise this by incrementing the current line in steps of 0.01. So 4.01 is a node that is used to help model a statement on line four.

First we need two nodes, one symbolising the start of the inner branch, modelled by the node with label 6.02, and one the symbolise the end of the loop, modelled by the node with label 6.01.

The loop ends as soon as the while condition no longer holds, and therefore the transition from start node, 6, to the end of the loop, 6.01 is a condition checking whether or not the condition of the while loop no longer holds, in this case this condition is represented as equality to zero, being the opposite condition to inequality to zero.

The loop continues as long as the condition holds, which is modelled by the sequence going from the start node, 6.01, to the start of the inner loop, being 6.02. This configuration models an inequality to zero expression, as described in Section 4.6.3 Inequality conditions.

Lines seven until ten define the first selection statement. Selection statements require, as referenced in Section 4.6.6 If statements, three additional nodes to fully model the needed implementation.

The start of the selection statement, is modelled by the node with label 7. From this node, there is a transition going to the node with label 7.03,

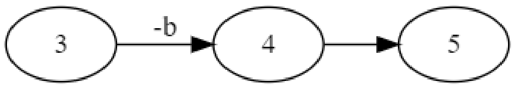

Fig. 4.15: Counter automaton resulting from line four and five

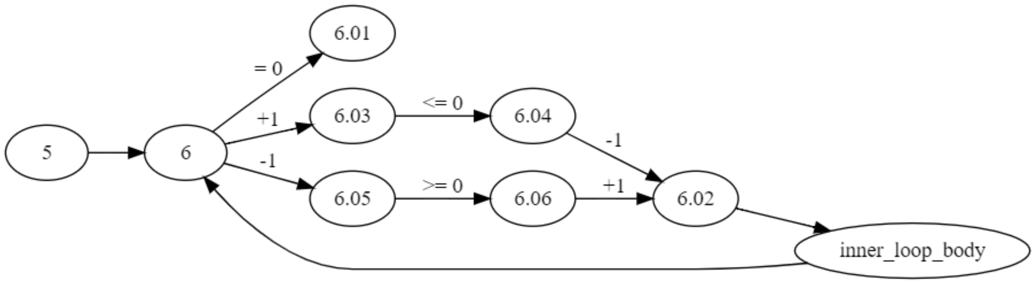

Fig. 4.16: Counter automaton resulting from line six

which symbolises the else branch, and therefore the transition to this node symbolises all counter values for which the if condition would not hold, which in this case are all counters smaller than or equal to zero.

The second transition originating from the start of the selection statement symbolises the cases where the counter satisfies the selection condition. The node 7.01 symbolises the start of the inner body of the if statement. The chain of transitions starting from 7 and going to node 7.01 model strict greater than, as is explained in Section 4.6.4 Less than and greater than.

Finally, there is the *end of if statement* node, being node 7.02. This node symbolises the end of all branches, so if there were any *else* branches, they will also end in this node.

The inner body is simple and does not need any complex structures to be added. The statement on line 8 will result in a transition originating from node 8 with label $- = b$. The statement on line 9 has no effect on the counter, and will just be modelled with an empty transition.

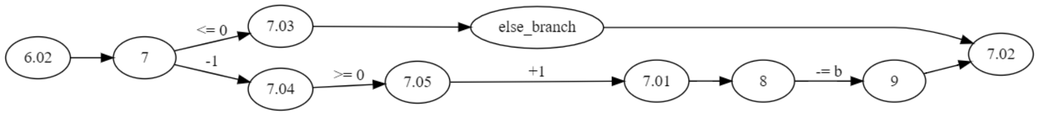

Fig. 4.17: Counter automaton resulting from line seven up until line ten

The eleventh until fourteenth line models the *else* branch of the previous selection statement. This statement follows the same design principles as the previous selection statement did, and will be inserted in the previous counter automaton, in the place where the current *else_branch* label is, as can be seen in Figure 4.18. Note that there is no *else* branch, which was not generated as there are not other *else if* or *else* statements in the code.

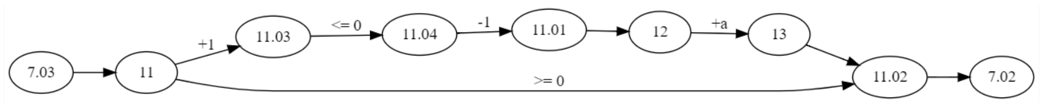

Fig. 4.18: Counter automaton resulting from line eleven up until line fourteen

The final selection statement can be found on line fifteen until line eighteen. This statement includes nothing new, and will simply follow the earlier mentioned specification. This selection statement will end with a transition going from the end of the selection, being the node with label 15.02, to the start of the earlier mentioned while statement, being the node with label 6, as can be seen in Figure 4.19.

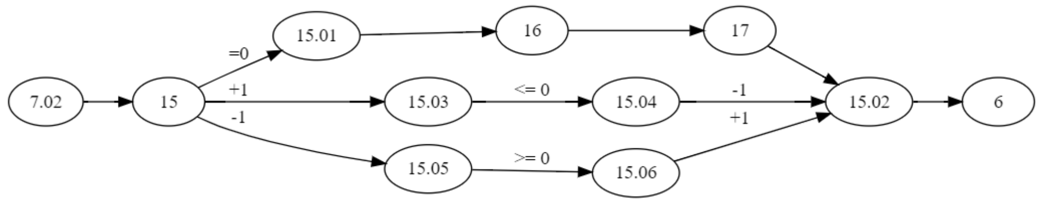

Fig. 4.19: Counter automaton resulting from line fifteen up until line eighteen

After this, all that remains is the return on line twenty. This return will simply follow after node 6.01 from the while configuration specified earlier. After a return, no nodes will be generated no matter what the kind of these statements is.

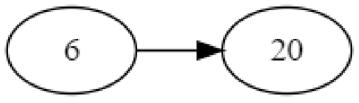

Fig. 4.20: Counter automaton modelling the return on line twenty

The complete overview of the counter automaton can be seen in Figure 4.21. This is a completely correct counter automaton that is suitable for reachability analysis.

4.7 Automaton specification format

For the approach described within this paper, the dot language part of Graphviz[3] was assumed to be used to represent all automata textually. This language is a very versatile and easily understandable format in which automata can be defined. It was also used as the output language in the preceding research which automatically made it the most interesting format to support. No other formats were considered as the conversion from a textual representation to an instance of the *Automaton* class is not considered to be the core of the research. The logic used for the dot language is furthermore identical to the logic that would need to be applied to read any other automaton specification, with the only difference lying in the language specific aspects of the specification.

4.7.1 Edge transformation

Edges are represented within the dot language by a single line per edge using the format specified below. The *origin* and *end* tokens specify the start and end node of the transition. The square brackets encapsulate optional arguments, of which the label and xlabel arguments matter most, as either can be used to specify the label of the transition, and thereby the operation that we want to have attached to the edge.

$$<origin> \rightarrow <end>[label = <label>, xlabel = <label>, ...]$$

This configuration can easily be converted into edge instances, but the labels can still represent several things. To allow any possible automaton specification, we will simply filter the labels using a regex for operations and labels. If it does not match, we assume the label to be an actual name label, if it does match, the label is perceived as a mathematical formula and it will be registered as such.

To closely match the specification of the theoretical proof on which this approach is based [1], we require edges to only contain operations and not conditions. In case an edge is encountered with a condition as a label, this will be resolved by introducing a new node, *inbetween*, to which this condition will be added. This node will be added as an intermediary step and the condition label will be attached to this node. By ensuring that this node will only have one incoming edge and one outgoing edge we preserve the fact that

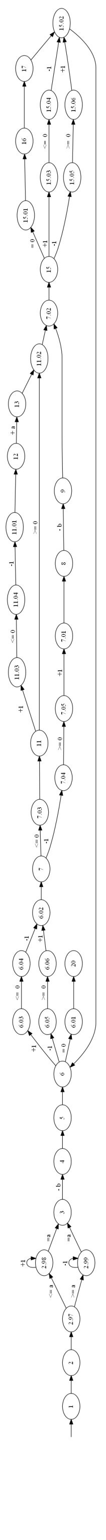

Fig. 4.21: Overview of the final resulting counter automaton

the condition must hold in case one desires to from from the *origin* node to the *end* node.

Below in Figure 4.22 is an example of an edge which does not correspond to the assumptions made within the original paper as the edge between *q0* and *q1* has a conditional label. This is resolved in the image on the right by introducing *_0* without losing the original meaning of the edge in the first image.

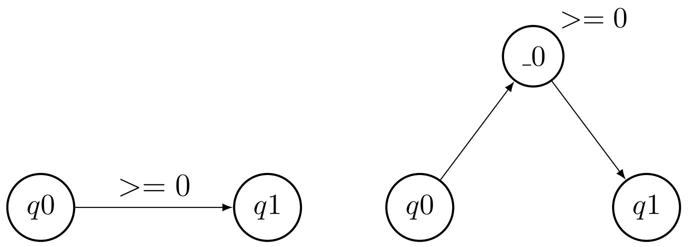

Fig. 4.22: Transformation of an edge with a conditional label

4.7.2 Node transformation

Nodes do not require to be specified using a separate node specification rule, and will simply be generated in case they are used within an edge. Node specifications function specifically for the cases in which additional attributes need to be added to nodes. To support operations as well as general labels, the only two attributes which will be tracked from node specification are the *label* and *xlabel* arguments. These labels will be matched with regexes specifying the supported formats for operations and conditions. In case the label does not match either, it is still tracked as regular node labels are still relevant for reachability reporting. For the approach discussed within this paper the node labels represent the code line from which they were generated and are therefore needed to analyse code line reachability. All node specifications will follow the format given below where the square bracket can contain a number of optional arguments.

$$<node>[label = <label>, xlabel = <label>, ...]$$

To match the specification of the theoretical approach specified by Michael Blondin et Al. [1], nodes can only contain conditions and not operations. In case a node *origin* is found that has an operation as its label we introduce a new node *inbetween*. By connecting a new edge from the *origin* node to the *inbetween* node we can add the label that was originally attached to *origin* to the new edge. To ensure that the automaton does not lose any meaning we will then move every edge that would normally go out from *origin* to *inbetween*.

In Figure 4.23 below, an example can be seen where node *q1* is not conformant to the original specification. This is resolved by introducing *_0* from which the edge that was originally going from *q1* to *q2* gets reattached.

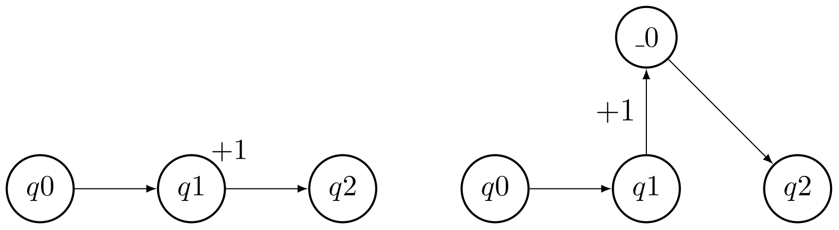

Fig. 4.23: Transformation of a node with an operational label

4.7.3 Initial node

For reachability analysis to be relevant there needs to be an initial node. In automata, the initial node is often represented by a transition going from nowhere towards a single node. This is however not a built in feature within the dot language. One of the solutions to overcome this is to create a node and make it invisible, an edge can then be drawn from this invisible node to the actual initial node. Other approaches are potentially possible but were not explored. Any alternative implementations could easily be introduced instead of the considered format without interchanging any of the other components.

As a consequence of this feature invisible nodes are assumed to exist within the automaton but these are irrelevant to our analysis. Invisible nodes are by definition not relevant for reachability analysis and are therefore not considered as actual nodes.

4.8 Reachability of non-parametric COCA

In this section we will give a description of the solution that was used to determine the reachability for non-parametric continuous one counter automata. This will be done by showing the approach that was taken, as well as an application to show its usage. This approach is a practical implementation of the theoretical approach described by Michael Blondin et al. [1].

4.8.1 Initialization

During the initialization phase, we will instantiate all reachability intervals. The initial node will get an initial reachability interval $[i, i]$ where i represents the initial counter value. This is not a fixed value, but will for the sake of this research always default to 0, simply because all integer values in most programming languages default to 0 at initialization.

All other nodes will get an empty interval assigned to them as to signify that they are initially not reachable.

4.8.2 Interval updates

From the initial state, we will start doing cycles in which we update all intervals once, wherever possible. This will be done using the successor function *succ* below where $R_i : Q \times Q \to 2^{\mathbb{Q}}$ represent the mapping function for the i-th iteration. This function is going from each transition to its corresponding counter reachability interval. The *succ* function will update the interval of one transition, *(p, z, q)*, to its next iteration, based upon the current configuration.

$$\begin{aligned} succ_{p,q}(R_i) := & \bigcup\{R_i(q,l) \mid (q,z,l) \in T\} \\ & \cup \bigcup\{(R_i(p,q) + (0,z]) \cap (q) \mid (p,z,q) \in T, z > 0\} \\ & \cup \bigcup\{(R_i(p,q) + (z,0]) \cap \tau(q) \mid (p,z,q) \in T, z < 0\} \\ & \cup \bigcup\{R_i(p,q) \cap \tau(q) \mid (p,0,q) \in T\} \end{aligned}$$

This formula allows us to iteratively compute the next interval configuration to fully explore which states have a non empty interval. Consider the example given in Figure 4.24. We will initialise R_0 such that all states have empty intervals, with the exception of the first state which we will initialise to $[0, 0]$. Note that these intervals are stored per pair of final and preceding node and not per node in general. This is because this allows us to have a better tracking of which path has which effect, which will help us in later steps. In the end it suffices to know that a node is reachable, independent on the path via which it can be reached.

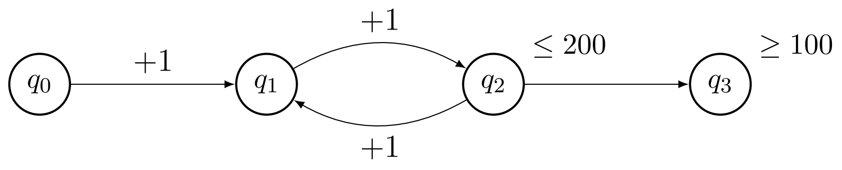

Fig. 4.24: Example of a guarded automaton on which the succ function can be applied

p	q	R_0	R_1	R_2	R_3	R_4	R_5
q_0	q_0	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
q_0	q_1	$\emptyset$	(0, 1]	(0, 1]	(0, 1]	(0, 1]	(0, 1]
q_1	q_2	$\emptyset$	$\emptyset$	(0, 2]	(0, 2]	(0, 4]	(0, 4]
q_2	q_1	$\emptyset$	$\emptyset$	$\emptyset$	(0, 3]	(0, 3]	(0, 5]
q_2	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tab. 4.1: Interval configuration when acceleration is not applied

For the first successor, we will only update for the transition going from q_0 to q_1 as this is the only transition where there is a non empty predecessor. All other transitions will both have an empty interval for the start and end nodes which will therefore trivially result in empty intervals after applying the *succ* function. When we apply the *succ* function on the first interval, we will get $(0, 1]$ as a resulting interval. The 0 is not included as the $(0, z]$ with $z = 1$ which we will add to it does not include 0. As q_1 has no condition the $\tau(q_1)$ will simply be $(-\infty, \infty)$ and will therefore have no effect.

The second successor will update for the transition going from q_1 to q_2. This update will be similar to the update that happened for q_0 to q_1 in every way but the starting interval, which is now (0, 1] and will thus result in an interval of (0, 2].

Next, for the third successor, one would say that we can update both the transition going back form q_2 to q_1 and the transition going from q_2 to q_3.

However, the latter is not updatable as $R_2(q_2) \cap \tau(q_3)$ will result in an empty set, considering that $\tau(q_3) = [100, \infty)$.

As can be seen in Table 4.1 a few more successors were computed and we can clearly notice that we will eventually reach an interval which will exceed the lower bound of 100 which is placed upon q_3. Given that the interval of q_2 goes up by two every other iteration it will take up to cycle 100 before the transition becomes active. Considering that this is only a small example, this is far from desired. Furthermore, given a loop that is not bounded, it is possible that a loop has no fix point at all, which means that our algorithm will evaluate indefinitely. A solution for this is loop acceleration in which we will accelerate our loop based upon the behaviour said loop shows.

4.8.3 Loop acceleration

Unfortunately, the *succ* function is not guaranteed to find the bounds in a finite number of evaluations. For example, the scenario given in Figure 23 would never converge at all. The interval of the loop between q_1 and q_2 will keep expanding indefinitely and the evaluation will therefore never stop.

To perform loop acceleration we need to consider the notion of *expanding cycles*. A cycle or loop is considered to be *positively expanding* if there is a path that follows a state sequence $q_i, q_{i+1}, q_{i+2}, ..., q_{n-1}, q_n$ evaluated in reachability iterations $R_0, R_1, R_2, ..., R_{k-1}, R_k$, where k represents the current iteration, such that:

1. $q_i = q_n$
2. $\forall j \in [i, n-1] : (p_j, z_j, q_{j+1}) \in T$
3. $\forall j \in [i, n-1] : R_k(p_j, q_{j+1}) \neq \emptyset$
4. $\exists (p, z, q) \in T : z > 0$
5. $sup(R_k(q_{n-1}, q_n)) > sup(R_{k-1}(q_{n-1}, q_n))$

Similarly we say that a cycle is *negatively expanding* if the following hold:

1. $q_i = q_n$
2. $\forall j \in [i, n-1] : (p_j, z_j, q_{j+1}) \in T$
3. $\forall j \in [i, n-1] : R_k(p_j, q_{j+1}) \neq \emptyset$

4. $\exists(p, z, q) \in T : z < 0$
5. $inf(R_k(q_{n-1}, q_n)) < inf(R_{k-1}(q_{n-1}, q_n))$

The first condition in combination with the second condition simply implies that we require there to be an actual loop. The third condition states that we should have visited every transition at least once. In case we did not have sufficient information needed to perform the acceleration we can not know the full behaviour of the loop. We will therefore only mark loops as expanding in case we have fully discovered it. This also eliminates the potential that we are looking at loops where some of the nodes are not reachable due to node constraints present within the loop. The fourth and fifth condition are there to ensure that there is an actual expansion. If only the fifth condition holds, it implies that there was a change in supremum due to something outside of the loop, in case only the fourth condition holds but not the fifth, it implies that we have already reached the bound of the loop. In the latter case the loop will not be marked as expanding as this implies that it reached its bound after one iteration.

Given that we know whether or not a cycle is either positively or negatively expanding, we can derive which node will reach its bound first. By doing so we can move the interval to this bound and thus greatly accelerate the evaluation of said loop. By performing one more iteration throughout the loop, all bounds would be reached and no further evaluation of the before mentioned loop will be needed.

To find the node that will be the first to reach a fix point interval, we can simply consider the node for which the current reachability interval's supremum/infimum is closest to the max/min node bound given that we are either trying to accelerate upwards or downwards respectively. In the formula below $reach(q)$ represents a function denoting the set of final counter values for a state q and $\underline{\tau}(q_j)$ and $\overline{\tau}(q_j)$ represent the upper and lower node bounds of node q_j respectively.

Given a loop that is expanding positively. If the following properties evaluate to true:

- $sup(R_i) > sup(R_n)$
- $\exists(p, z, q) \in T : z > 0$
- $\overline{\tau}(q_k) - v_k = min_{i \leq j \leq n} \overline{\tau}(q_j) - v_j$

The following property will apply:

- $[v_k, \overline{\tau}(q_k)] \subseteq reach(q_k)$

Given a loop that is expanding negatively. If the following properties evaluate to true:

- $inf(R_i) < inf(R_n)$
- $\exists(p, z, q) \in T : z < 0$
- $v_k - \underline{\tau}(q_k) = min_{i \leq j \leq n} v_j - \underline{\tau}(q_j)$

The following property will apply:

- $[\underline{\tau}(q_k), v_k] \subseteq reach(q_k)$

In our approach this allows us to traverse the differences between the previous iteration and the current iteration whenever we update a node that we already have an interval for. Given the example that was explored in Section Interval updates we can see that we have a loop going from q_1 to q_2 and back.

p	q	R_0	R_1	R_2	R_3	R_4	R_5
q_0	q_0	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
q_0	q_1	$\emptyset$	(0, 1]	(0, 1]	(0, 1]	(0, 1]	(0, 1]
q_1	q_2	$\emptyset$	$\emptyset$	(0, 2]	(0, 2]	(0, 4]	(0, 4]
q_2	q_1	$\emptyset$	$\emptyset$	$\emptyset$	(0, 3]	(0, 3]	(0, 5]
q_2	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tab. 4.2: Resulting interval configuration before acceleration

In Table 4.2 we can see that in the third reachability interval, R_2, a counter value interval is discovered for the transition going from q_1 to q_2. In the next reachability interval the same happens for q_2 to q_1. In R_4 we will once again update the interval for the transition going from q_1 to q_2 which will result in all conditions for a positive expanding loop evaluating to true. This is because we have a loop of which we know that the supremum of one of the transitions just increased and we have a transition part of this loop with a strictly positive operation.

p	q	R_0	R_1	R_2	R_3	R_4	R_5
q_0	q_0	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]	[0, 0]
q_0	q_1	$\emptyset$	(0, 1]	(0, 1]	(0, 1]	(0, 1]	(0, 1]
q_1	q_2	$\emptyset$	$\emptyset$	(0, 2]	(0, 2]	(0, 200]	(0, 200]
q_2	q_1	$\emptyset$	$\emptyset$	$\emptyset$	(0, 3]	(0, 3]	(0, 201]
q_2	q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	[100, 200]

Tab. 4.3: Resulting interval configuration after acceleration

Applying our previously described steps, we will find that node q_2 is closest to its bound and we will therefore choose to expand the interval of the transition ending in this node, being the transition from q_1 to q_2. The result is shown in Table 4.3.

If we do one additional iteration, we can see that we have now discovered an interval for the transition going from q_2 to q_3. We have furthermore achieved a fix point, as both q_2 and q_3 have met their bounds.

4.8.4 Proof of termination

The mathematical proof of termination has already been given by Michal Blondin et al. [1] but an intuitive proof for the algorithm will also be given here. Given that there are no loops it will trivially terminate within n evaluations, given that n is equal to the total number of nodes within the automaton. If there are loops they must either be positively expanding, negatively expanding, expanding in both directions or not expanding at all.

In the last case the evaluation will trivially terminate as the loops will end in a fix point after their first evaluation. In the first case it will also

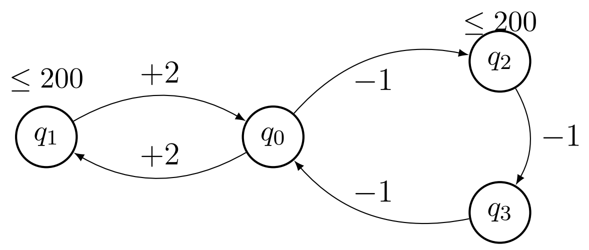

Fig. 4.25: Example of an automaton in which post acceleration evaluation is needed

terminate as loops can only be fully evaluated twice before reaching their fix point. One traversal is needed to discover the loop and a second to make sure that all intervals reach their fix point within said loop.

It is possible that further evaluations require bounds to be recomputed. For example, given the automaton described in Figure 4.25. In this automaton the loop between q_0 and q_1 will be expanded first. The evaluation of the second loop continues and the lower bound of q_0 to q_1 will keep on decreasing, until it eventually gets accelerated, at which moment we achieve a fix point and the evaluation stops. This will always be the case given that we can never have an infinite amount of nodes which will thus result in a finite amount of loops which all accelerate at some point. The effects of said acceleration might need a few more iterations to propagate their effect all across the automaton but once it does the evaluation will end. We can therefore conclude that the algorithm can never run indefinitely.

4.9 Reachability of parametric COCA

Throughout this section we will give an overview of how the reachability for parametric continuous one counter automata can be analysed. This will be done using the approach that was theoretically described by Michael Blondin et al. [1]. This paper will both give a practical approach as well as an application, intended as prove that it works in practise.

In this approach we will attempt to encode the fix-point of the *succ* function defined in the previous section using first order logic which can then be implemented and solved by a SAT solver.

4.9.1 Transition initialization

During initialization we will start by converting all transitions that are defined within the given automaton to their equivalent equations. All transitions are represented by three variables. Two variables are used to represent the start and end point of the transition and the third will represent the operation associated with the transition. The operation will simply be an integer value as we limited our automata to only allow + and - operations. This means that an integer can fully cover every possible operation that can be performed on an edge.

For the example given in Figure 4.26 this would be a conversion to

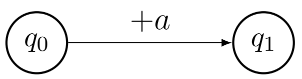

Fig. 4.26: Example of a transition before conversion

$(q_0, +a, q_1)$. In this resulting triple our operation is a variable which represents an integer which will be filled in by the SAT-solver.

Note that there is no need to cover conditions on edges as we resolved all inconsistencies regarding transitions with conditional labels in our automaton input phase.

4.9.2 Node condition initialization

Alongside the transitions we will also initialise the conditions of the nodes in which the transitions end. These are important as these decide whether or not a transition can be taken. Given that we have a transition with an operation equal to $+1$, a current counter value equal to just 0 and an end node with a condition equal to ≤ -1. This example is visualised in Figure 4.27 and we can see that q_1 will not be reachable, given that the initial counter value is equal to 0. Each condition will be represented by two variables, one to indicate the condition type and a second to indicate the value against which it is compared. Just as for the transitions our condition value can either be constant or a variable. The condition type can have three different values, 0 will be used to represent the $\geq$ operation, 1 will be used to represent $\leq$, 2 will represent $=$ and finally 3 will be used in case there is no condition associated with the end node.

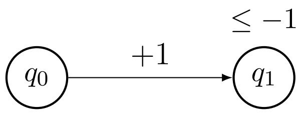

Fig. 4.27: Example of a transition with an unsatisfiable node condition

4.9.3 Interval encoding

Each of the nodes will get reachability intervals assigned to it. These reachability intervals will represent a possible subset of counter values that can be reached in a given node. Using these intervals we can then decide whether or not a next transition can be taken. These intervals will also be used to derive whether or not a node is reachable.

It was proven by Michael Blondin et al. [1] that the number of intervals per node is bounded by $4(|Q|+1)$ where Q is the set of all nodes. This means that we will have to generate as many intervals for each of the nodes. For these intervals we will attempt to keep as many of them empty, as to not needlessly complicate the computation.

Each interval will be represented by 4 variables. The first two variables will represent the lower and upper bound of the interval, the last two will represent whether or not these bounds are inclusive. A 0 indicates not inclusive, a 1 indicates inclusive and a 2 indicates that the bound goes until infinity. For example, an interval with just 0, $[0, 0]$ would be represented by $0, 0, 1, 1$. An interval that goes from negative infinity to 1, $(-\infty, 1]$ would be represented by $0, 1, 2, 1$ and an empty interval, $(0, 0)$ by $0, 0, 0, 0$.

Initially the start node of the automaton will have an interval equal to exactly the starting value of the counter (by default this will be 0). All other intervals of all nodes will get values according to the edges that exist within the automaton. The evolution of these intervals will be discussed in the next sections.

4.9.4 Define the end goal

One by one we will verify for each node whether or not it is reachable, this can be done using the condition below, where e represents the edge taken to generate any of the intervals part of the goal node. In case e is equal to -2 it signifies that no transition was taken and that the interval is empty. If e is not equal to -2 it must imply that the according interval is not empty as will be explained later.

$$\begin{aligned} reachable_i &= e_i \neq -2 \\ reachable &= Or(reachable_i \text{ for all i} \in \text{I}) \end{aligned}$$

4.9.5 msum

The msum function is used to determine the inclusiveness of the bounds in the interval addition. The msum function is defined as:

$$\begin{aligned} msum(i,j) &= 2 && if\ max(i,j) = 2 \\ msum(i,j) &= min(i,j) && otherwise \end{aligned}$$

4.9.6 Emptiness of intervals

We need to be able to verify whether or not an interval is empty in order to create base cases for future operations. An interval is considered to be empty if both bounds are not inclusive and the boundary values are equal to each other as can be seen in the formulae below.

$$\begin{aligned} x_2 &= 0 \\ x_3 &= 0 \\ x_0 &= x_1 \end{aligned}$$

4.9.7 Interval addition

The addition of intervals will be split in three different cases as can be seen in the graph below. In case either operand is an empty interval the resulting interval will trivially equal the other interval. In case they are both not empty we compute the sum by taking the addition of the lower and upper bound of the X and Y intervals and the msum to compute whether or not the resulting bounds are inclusive. An overview of this can be found in Figure 4.28.

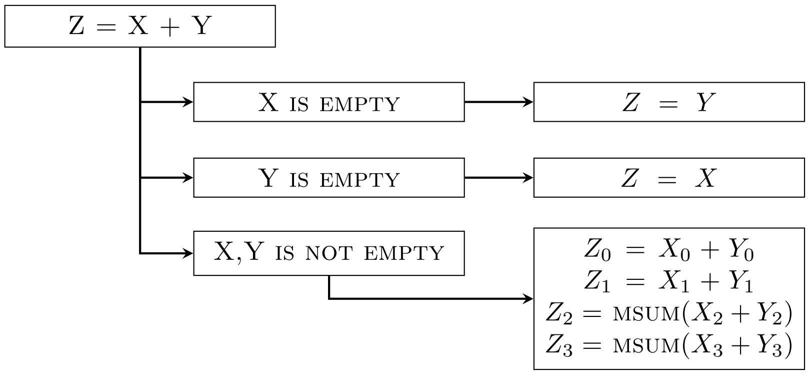

Fig. 4.28: Overview of the interval addition operation

4.9.8 Interval overlap

One important relationship between two intervals is overlap as it directly helps to simplify the generation of unions and intersections of intervals. This can be computed by verifying whether or not any of the endpoints of one interval is within the other interval. Note that it is important to specifically look at the third and fourth variable part of the interval to also consider infinite bounds. An overview of this operation can be seen in Figure 4.29.

4.9.9 Interval intersection

The entirety of the intersection operation is visualised in Image 4.30. In this image all nodes where multiple lines leave are assumed to imply *or* operations between all end nodes. This holds with the exception of the coloured nodes where nodes of the same colour represent an *or* operation which will then be combined in a bigger *and*.

For the intersection of intervals we will first of all consider the cases where the result is an empty vector. This can either occur when there is no overlap or when X or Y is empty. In any of these cases we can simply assign an empty vector to Z, being the resulting interval.

If this is not the case we will separately evaluate both borders of the interval. The lower border is represented by the red colour and the upper border by the blue colour within Image 4.30.

For both sides it holds that if either X or Y goes to infinity in that direction then Z will have the boundary values of the other interval. For example if X would go to infinity on the lower bound then the value of Z will

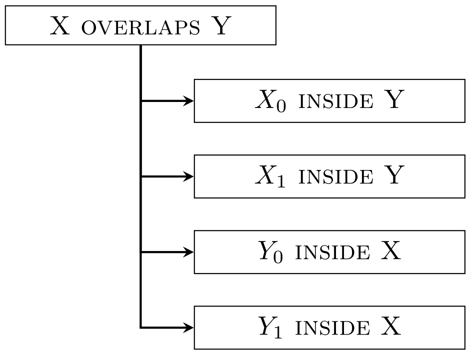

Fig. 4.29: Overview of the interval overlap operation

be equal to the value of Y for both the lower bound value and whether or not this bound is inclusive.

If neither X or Y goes to infinity we will look at whether or not either bound is further out than the other. If this is the case than the most inner side will be chosen to be the representative for that bound in Z.

If both bounds are equal then we look if they are both inclusive, if yes then Z will be inclusive too, if either one is not inclusive then Z will also be not inclusive for said bound. The value of Z will be equal to the bound of X and Y in any of these cases.

4.9.10 Interval union

The union operation is summarised in Figure 4.31 below. We can immediately see that this is very similar to the intersection in the sense that most of the conditional statements are the exact same.

We once again split the evaluation into two near identical evaluations being marked by the blue and red nodes. The red segment relates to the lower bound, the blue segment to the upper bound. In case either side goes to infinity for X or Y, Z will trivially go to infinity too. In case it does not go to infinity we will select the one closest to the bounds. Finally if both bounds are equal we will by default set the boundary value equal to the boundary value of X and Y and make it inclusive if either X or Y are inclusive and exclusive otherwise.

4.9.11 Interval progression

For further intervals to get populated we need conditions to represent the potential traversal of edges. Note that we do not consider an actual path through the automaton, we simply try to guess a configuration. We do not require our solver to give us a unique or optimal solution, we simply require the resulting configuration to represent a subset of all reachable counter values for which the node under test is proven to be reachable.

Each sub interval can either be empty, in case there are no transitions that can be activated, or it holds the result of an update. The schema given in Figure 4.32 describes the logic which is applied on the sub intervals. Given that we are currently evaluating sub interval i of node q we can see that it will be empty in case no transitions can fire. If a transition can still fire the

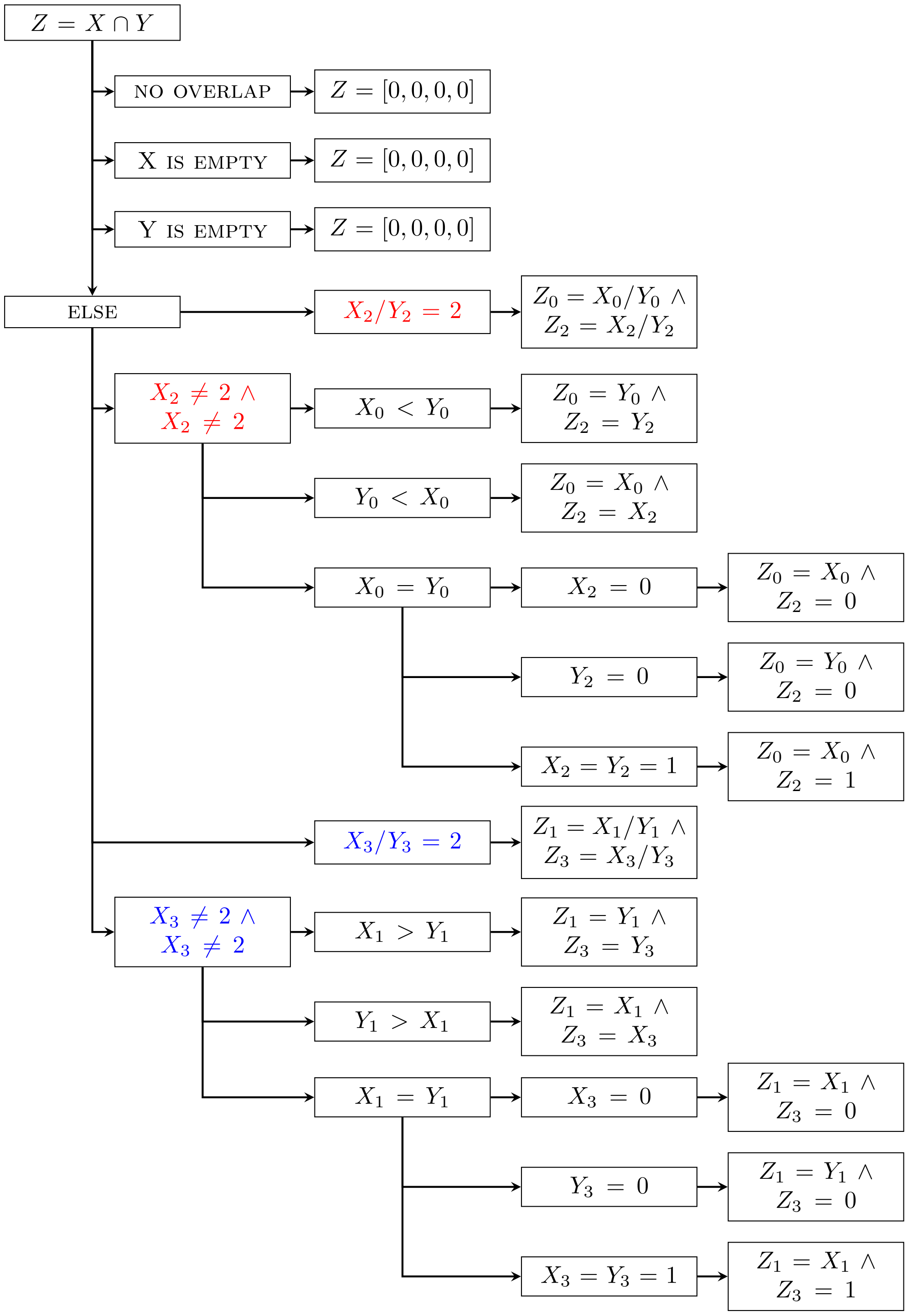
$Z = X \cap Y$
NO OVERLAP
$Z = [0,0,0,0]$
X IS EMPTY
$Z = [0,0,0,0]$
Y IS EMPTY
$Z = [0,0,0,0]$
ELSE
$X_2/Y_2 = 2$
$Z_0 = X_0/Y_0 \wedge Z_2 = X_2/Y_2$
$X_2 \neq 2 \wedge X_2 \neq 2$
$X_0 < Y_0$
$Z_0 = Y_0 \wedge Z_2 = Y_2$
$Y_0 < X_0$
$Z_0 = X_0 \wedge Z_2 = X_2$
$X_0 = Y_0$
$X_2 = 0$
$Z_0 = X_0 \wedge Z_2 = 0$
$Y_2 = 0$
$Z_0 = Y_0 \wedge Z_2 = 0$
$X_2 = Y_2 = 1$
$Z_0 = X_0 \wedge Z_2 = 1$
$X_3/Y_3 = 2$
$Z_1 = X_1/Y_1 \wedge Z_3 = X_3/Y_3$
$X_3 \neq 2 \wedge X_3 \neq 2$
$X_1 > Y_1$
$Z_1 = Y_1 \wedge Z_3 = Y_3$
$Y_1 > X_1$
$Z_1 = X_1 \wedge Z_3 = X_3$
$X_1 = Y_1$
$X_3 = 0$
$Z_1 = X_1 \wedge Z_3 = 0$
$Y_3 = 0$
$Z_1 = Y_1 \wedge Z_3 = 0$
$X_3 = Y_3 = 1$
$Z_1 = X_1 \wedge Z_3 = 1$

Fig. 4.30: Overview of the interval intersection operation

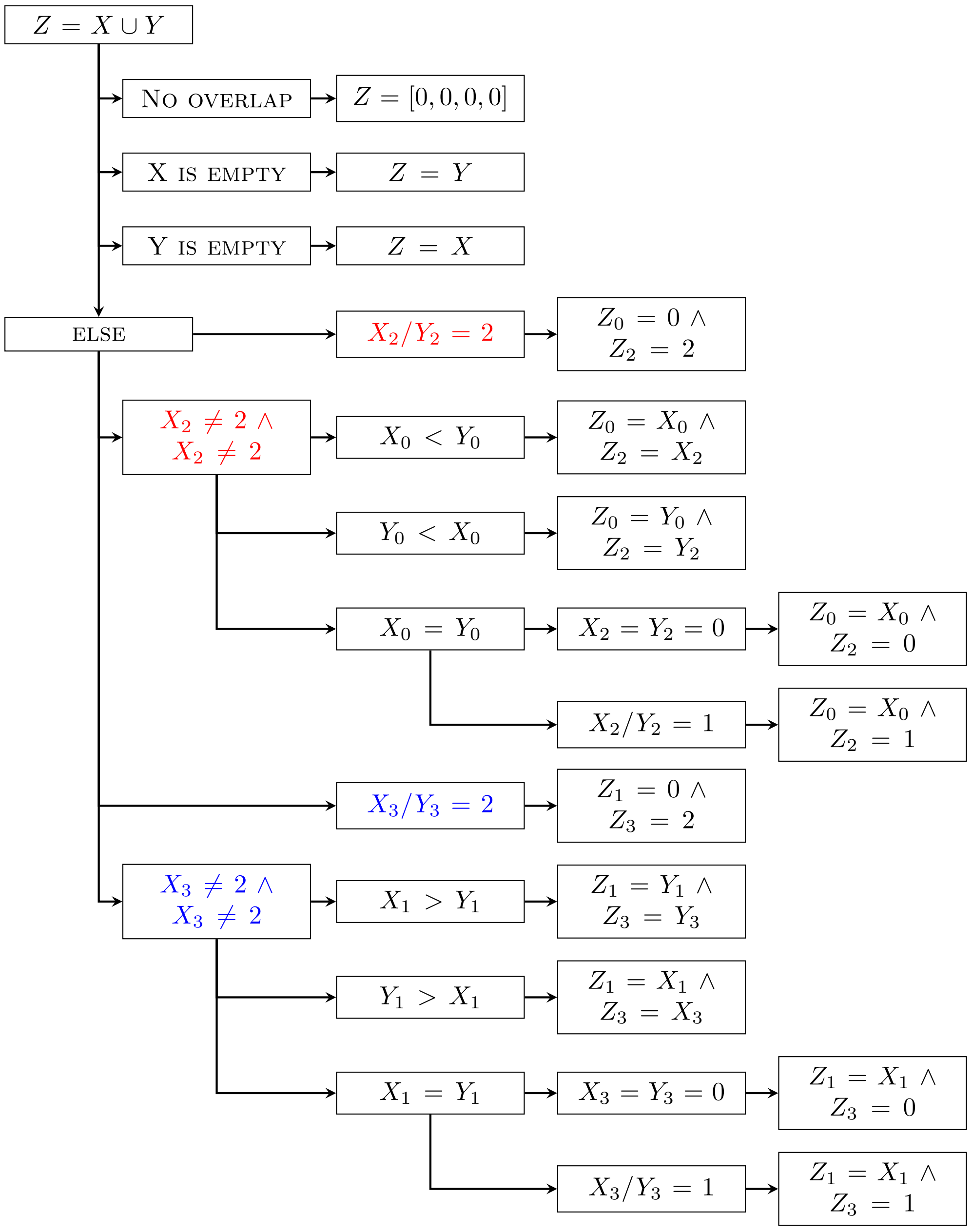
$Z = X \cup Y$
NO OVERLAP
$Z = [0, 0, 0, 0]$
X IS EMPTY
$Z = Y$
Y IS EMPTY
$Z = X$
ELSE
$X_2/Y_2 = 2$
$Z_0 = 0 \wedge Z_2 = 2$
$X_2 \neq 2 \wedge X_2 \neq 2$
$X_0 < Y_0$
$Z_0 = X_0 \wedge Z_2 = X_2$
$Y_0 < X_0$
$Z_0 = Y_0 \wedge Z_2 = Y_2$
$X_0 = Y_0$
$X_2 = Y_2 = 0$
$Z_0 = X_0 \wedge Z_2 = 0$
$X_2/Y_2 = 1$
$Z_0 = X_0 \wedge Z_2 = 1$
$X_3/Y_3 = 2$
$Z_1 = 0 \wedge Z_3 = 2$
$X_3 \neq 2 \wedge X_3 \neq 2$
$X_1 > Y_1$
$Z_1 = Y_1 \wedge Z_3 = Y_3$
$Y_1 > X_1$
$Z_1 = X_1 \wedge Z_3 = X_3$
$X_1 = Y_1$
$X_3 = Y_3 = 0$
$Z_1 = X_1 \wedge Z_3 = 0$
$X_3/Y_3 = 1$
$Z_1 = X_1 \wedge Z_3 = 1$

Fig. 4.31: Overview of the interval union operation

interval will be equal to the direct result of an update starting from the sub interval j of a node p given that there is a transition t going from p to q.

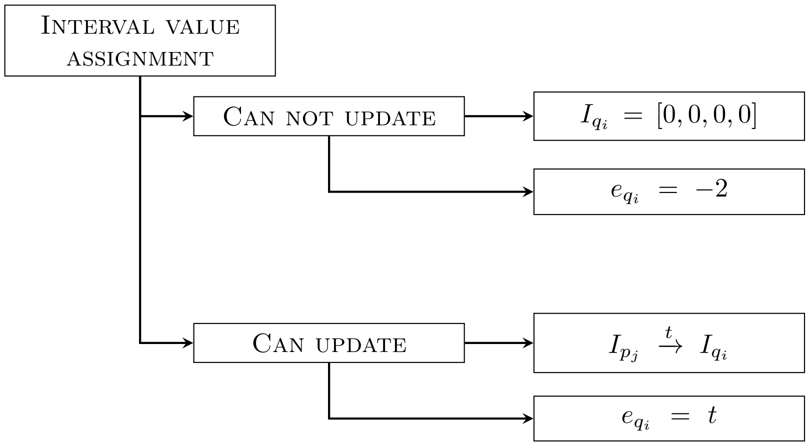

Fig. 4.32: Overview of the interval assignment conditions

We will furthermore ensure that a transition t can only be used once per node q. This is done as to prevent needlessly filling each subinterval.

The *can update* condition will be applied on a preceding interval I_{p_j} and a current interval I_{q_i}. First of all we will check that neither I_{p_j} and I_{q_i} are empty. If either of the two are empty we do not consider it as an update and the update will not be marked as possible.

$$\begin{aligned} succ_{p,q}(R_i) := & \bigcup\{(R_i(p,q) + (0,z]) \cap \tau(q) \mid (p,z,q) \in T, z > 0\} \\ & \cup \bigcup\{(R_i(p,q) + (z,0]) \cap \tau(q) \mid (p,z,q) \in T, z < 0\} \\ & \cup \bigcup\{R_i(p,q) \cap \tau(q) \mid (p,0,q) \in T\} \end{aligned}$$

Next we will follow the same *succ* function as we followed for the reachability of non-parametric COCA. For this function to be applied we will make use of two additional helper intervals called Y and Y2, which will need to be instantiated for each evaluated edge. The first step will be the addition which is dependent on the the z variable representing the edge operation. In Figure 4.33 below this interval is simply denoted as *addend*.

Next we will need to ensure that Y does not exceed any of the bounds. This will be done by first of all intersecting Y with the bounds of the automaton A. The result of this will be stored in Y2 which will then once again be intersected but this time with the bounds of the node q in which the edge ends. The result of this final intersection will be stored in I_{q_i}.

We have thus evaluated whether or not an update is possible. If so, the interval I_{q_i} will trivially hold the result of this update, if not the initial interval assignment condition visualised in Figure 4.32 will let the interval be equal to the empty interval. We can therefore remove the condition saying that I_{q_i} must hold the result of taking the transition t from interval I_{p_j}.

Finally, in case one wishes to compute the reachability interval of an entire node we can simply consider the union of all subintervals.

If we consider the automaton given in Figure 4.34 we start with only q_0 being reachable with one of its subintervals equal to the interval $[0, 0]$ or the configuration $0, 0, 1, 1$ in formula format.

There will be conditions for each of the subintervals I_{q1_i} of node q_1 stating that if there exists an interval I_{q0_j} of node q_0 on which the edge operation $+1$ can be applied, we should do so. Given our current context this is indeed possible and we will therefore assign $0, 1, 0, 1$ to one of the subintervals of q_1 as this will be the resulting value. The corresponding edge value will be set equal to 0, given that the first edge is labelled 0.

The same condition will exist for q_1 but now requiring there to both be a resulting interval and preceding interval part of q_1. Given that this loop only has an addition and no subtractions, any interval going to positive infinity could be used as a fix point for it. In this case $0, 0, 0, 2$ or $(0, \infty)$ was chosen to represent both the preceding and resulting interval.

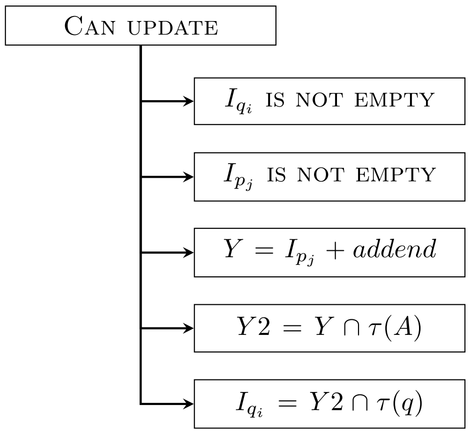

Fig. 4.33: Overview of the can update condition

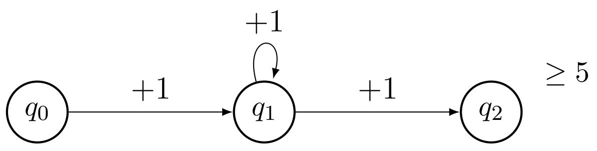

Fig. 4.34: Example of an automaton with a loop

Finally, the edge between q_1 and q_2, labelled edge 2, can be taken if there is a sub interval $I_{q_{1_i}}$ part of q_1 and a sub interval $I_{q_{2_j}}$ part of q_2 for which the conditions hold. If we consider the interval generated from edge 1 we can apply edge 2 to this and get the resulting interval $0, 0, 0, 2$. This does however not satisfy the node bound that exist on q_2 and thus our resulting interval will be $5, 0, 1, 2$ which does satisfy.

After assigning all of these intervals we get the results in Table 4.4 where all intervals are given using the format $<$ *edge* $>:<$ *interval* $>$. Note that there are many more intervals that are simply left out as they are all empty.

4.9.12 Loop progression

In case we only use the conditions above we will still have a problem with loops. Considering the example below in Figure 4.35, node q_2 is not reachable as our initial value is equal to 0 and we only have positive operations. However, with the condition given above any interval with an upper bound equal to ∞ would work as a fix point. We can therefore perfectly select the interval $(-20, \infty)$ and the conditions would be satisfied. We should therefore add an additional condition that restricts fix points for loops.

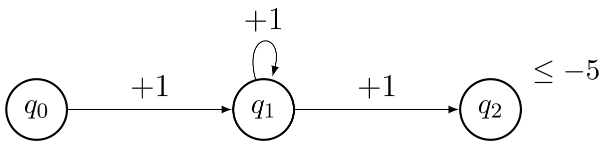

Fig. 4.35: Example of an automaton with a loop that needs to be restricted

	q0	q1	q2
Interval 1	-1: 0, 0, 1, 1	0: 0, 1, 0, 1	2: 5, 0, 1, 2
Interval 2	-2: 0, 0, 0, 0	1: 0, 0, 0, 2	-2: 0, 0, 0, 0

Tab. 4.4: Example interval configuration post evaluation

Considering the definitions of loop expansion given in Section *4.8.3 Loop acceleration* we will differentiate between the different types of expanding loops. Consider a loop l with transitions $t_0, t_1, ..., t_k$ which goes over the nodes $q_i, q_i + 1, ..., q_j$. We furthermore assume that the interval with index 0 of each of the nodes will be used to represent the loop. Finally, we assume to be working in an automaton which has n intervals per node. An overview of the conditions that need to be enforced in case loops are present can be seen in Figure 4.36.

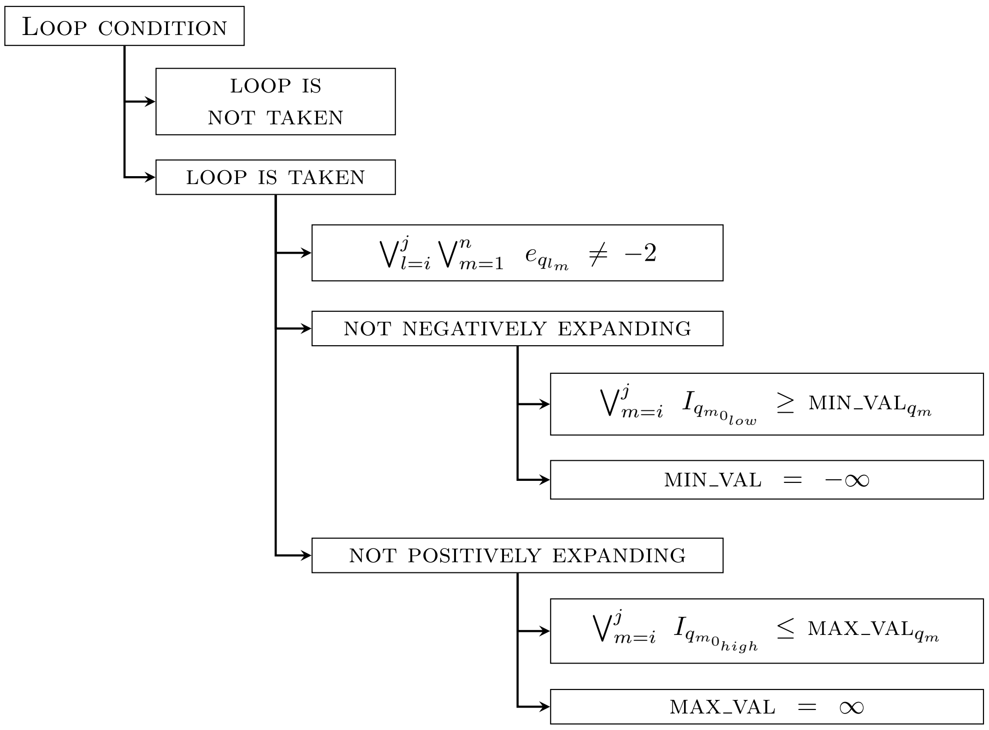
Loop condition
loop is not taken
loop is taken
$\bigvee_{l=i}^{j} \bigvee_{m=1}^{n} \; e_{q_{l_m}} \neq -2$
not negatively expanding
$\bigvee_{m=i}^{j} \; I_{q_{m_{0_{low}}}} \geq$ MIN_VAL$_{q_m}$
MIN_VAL $= -\infty$
not positively expanding
$\bigvee_{m=i}^{j} \; I_{q_{m_{0_{high}}}} \leq$ MAX_VAL$_{q_m}$
MAX_VAL $= \infty$

Fig. 4.36: Overview of the loop condition

In case the loop under consideration is not taken the condition should trivially evaluate to true. In case it is taken there should be at least one node that has an interval that is not related to the loop (the intervals related to the loop are at index 0 and are therefore not considered in the equation). If it is not expanding in either direction it has to originate in another existing interval and it should therefore be bounded by the values part of any other interval. The *min_val* and *max_val* variables are computed using the formulae below.

	q0	q1	q2
Interval 1	-1: 0, 0, 1, 1	0: 0, 1, 0, 1	-2: 0, 0, 0, 0
Interval 2	-2: 0, 0, 0, 0	1: 0, 0, 0, 2	-2: 0, 0, 0, 0

Tab. 4.5: Example interval configuration post evaluation

$$\begin{aligned}\text{min\_val}_q &= min(I_{q1_{low}}, I_{q2_{low}}, ..., I_{q_{n_{low}}})\\ \text{max\_val}_q &= max(I_{q1_{high}}, I_{q2_{high}}, ..., I_{q_{n_{high}}})\end{aligned}$$

In case either the min or max variable go to infinity, the conditions for their respective bounds should trivially be satisfied as the intervals are no longer bounded in that case. The schema representing the loop condition can be found below in Figure 4.36.

When we now reinvestigate the example in Figure 4.35 we can get the intervals given in Table 4.5. Note our edges are numerically labelled here. The label 0 represents the edge from q_0 to q_1, label 1 represents the loop and label 2 represents the edge from q_1 to q_2. Instead of going negative the node q_1 will now go from 0 to infinity and node q_2 will thus not be discovered as being reachable which is as expected. The second interval of q_1 can never go negative as it would then fail to satisfy the loop conditions given above.

5. USE CASE: XRDP

Xrdp[1] is a medium sized c project that provides a graphical login to remote machines using Microsoft Remote Desktop. It was chosen as a use case for this project as it is a real life example of a project that would greatly benefit from dead code elimination. It consists of 120 files containing 94 458 lines of code and there are a total of 1 328 functions (not including defines). It is interesting for dead code analysis as it is highly dependant on efficiency. It is furthermore used in several other projects as this project is forked into 947 repositories thus indicating that this project counts as a core reused in other projects. Inefficiencies in this project would be further extrapolated in the projects who inherit from it and should therefore be prevented at all costs.

lines of code	94 458
number of functions	1 328
number of forks	947

Tab. 5.1: Statistics for the xrdp project

5.1 shortcomings

The code analysis quickly showed flaws in the compiler resulting from the first phase in the analysis.

A first major case that was found rather quickly were cases in which a function is used in conditional statements, rather than variables. This situation was not fully taken into account and will lead to false results. For example, there is an instance where a function is simply used as boolean within if statements. In these statements the validator should have returned that this case is not analysable, as we can not investigate whether or not the invoked function can return a certain value but instead the code will simply

[1] `https://github.com/neutrinolabs/xrdp`

not acknowledge the function as a counter as it is programmed to only handle values. In the example in Listing 5.1 we can see a case where a function is used on line 11 and 14 yet the one counter automaton will be generated but simply without any counter. For this analysis such results were manually removed to avoid analysing trivial cases with our new reachability approach.

```
1  int
2  cleanup_sockets(int display)
3  {
4    LOG(LOG_LEVEL_INFO, "cleanup_sockets:");
5    char file[256];
6    int error;
7
8    error = 0;
9
10   g_snprintf(file, 255, CHANSRV_PORT_OUT_STR, display);
11   if (g_file_exist(file))
12   {
13     LOG(LOG_LEVEL_DEBUG, "cleanup_sockets: deleting %s", file);
14     if (g_file_delete(file) == 0)
15     {
16       LOG(LOG_LEVEL_WARNING,
17        "cleanup_sockets: failed to delete %s (%s)",
18        file, g_get_strerror());
19        error++;
20     }
21   }
22 }
```

Listing 5.1: Example of code using functions as conditional values

Another issue that was found were data structures which the compiler did not recognize. Within the compiler structs are supported and optimizations using the different variables that are part of structs are performed. There is however no optimization for pointers to structs.

When a variable points to a struct you will not access its members with the . operation but rather with the -> operator. This is however not supported in the compiler and use of this operator will lead to exceptions where an optimization function attempts to optimize this without success. Another case were struct pointers have weird effects are cases such as in Listing 5.2. In this case it will once again not detect any counter while in essence we are verifying whether or not the mod variable is initialized by the verification done on line 6.

```c
int
mod_exit(tintptr handle)
{
  struct mod *mod = (struct mod *) handle;

  if (mod == 0)
  {
    return 0;
  }

  g_free(mod);
  return 0;
}
```

Listing 5.2: Example of comparisons not considered by the compiler

Finally, an attempt was made to support dereferencing of pointers as well as pointers of pointers but immediate access during dereference was not considered during the writing of the compiler and will also lead to unhandled cases. In Listing 5.3 the operation on line 4 where the *fip* variable is dereferenced and directly accessed afterwards is not supported by the compiler. The function in Listing 5.3 will lead to a direct crash of the compiler.

```c
1  void xfuse_devredir_cb_enum_dir_done(struct state_dirscan *fip,
2  enum NTSTATUS IoStatus)
3  {
4    struct fuse_file_info *fi = &fip->fi;
5    XFUSE_HANDLE *xhandle = xfuse_handle_create();
6
7    if (xhandle == NULL
8    || (xhandle->dir_handle = xfs_opendir(g_xfs, fip->pinum)) ==
      NULL)
9    {
10     fuse_reply_err(fip->req, ENOMEM);
11   }
12 }
```

Listing 5.3: Example of unsupported access to pointer values

Finally there were other cases where node mappings occurred that were not expected possible. For example, arguments that are not structs are mapped as is visualised in Image 5.1. However, when this type is a struct an entirely different format is used as can be seen in Figure 5.2.

In the initial compiler the first format was assumed and thus several errors occurred due to missing nodes during the evaluation of the project. This is

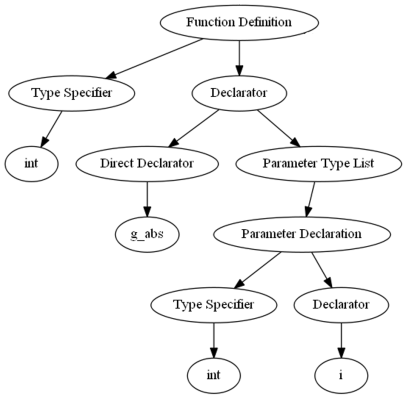

Fig. 5.1: Function with normal argument format

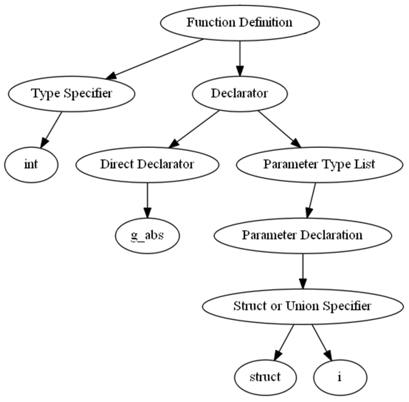

Fig. 5.2: Function with struct argument format

very hard to consider beforehand and cases like this make it near impossible to properly identify every possible case that can ever occur. A significant amount of errors of this kind were resolved, such as this one, but not all.

5.2 Results

As a result of these issues several files were not analysable. An attempt was made to resolve as many issues as possible such as the lack of proper preprocessing. Some of the optimizations were also disabled, such as optimizations for variable types which can not be handled by the compiler but this eventually resulted in other components breaking. This leads to 76 files of the

120 total files being evaluated correctly. Within these files a total of 975 functions were evaluated of which 35 were found to be suitable for counter evaluation. Two of these functions had a function in their conditional statement and should not have been accepted for counter automaton analysis. All other cases were correctly identified and were all ready for further analysis.

Metric	Actual	Found	Percentage
Number of files	120	76	63.33%
Number of functions	1 328	975	73.42%

Tab. 5.2: Statistics for the use case analysis

Of these 33 remaining functions, two functions had variable counters and were analysed using equations. Both of these did not contain any dead code and were correctly identified as fully reachable. The 31 functions that did not contain variables were all found to be fully reachable by the first approach.

It should be noted that a proper time analysis is hard considering the fact that the compiler did take time for files which yielded no results. That being said the two files that had contained parameter operations took a significantly long time to evaluate of up to 15 minutes. This is due to the explosion in the number of variables and is not a direct result of the implementation itself.

Cause	Non-parametric	Parametric
Preceding assignment	Detectable	Detectable
Preceding operation	Detectable	Detectable
Variable can not match	Not applicable	Detectable
Preceding conditional return	Detectable	Detectable

Tab. 5.3: Different cases leading to nodes not being reachable

6. MANUAL VALIDATION

To further identify the correctness tests were performed doing end to end verifications of the tool. We were able to verify that our approach was in fact able to analyse C code including the reachability analysis. We were however unable to prove that it will show lines as unreachable in case there is dead code. To verify this a manual end to end test suite was created which verified mutations of the xrdp project. These mutations specifically introduced different kinds of dead code in an attempt to identify whether or not the approach is indeed able to identify dead code.

A few cases were considered which were both applied to parametric and non-parametric automata as can be seen in Table 5.3. For all of the considered cases both approaches were able to correctly identify the lines that were potentially unreachable. More edge case testing was done with test suites which were written to help with the development of the approach. These will specifically test the edge cases for each of the operations given for both the parametric and non-parametric approach. All of these tests passed.

7. RESEARCH QUESTIONS

The goals of this research are summarised in two research questions which were listed at the beginning of this paper. In this section we will conclude our findings in the form of a reply to the two initial questions.

Question 1: Can we practically determine reachability of continuous one counter automata?
During this research we gave two practical approaches which were both based on theoretical approaches created by Michael Blondin et al. [1].

The first approach can be applied on all COCA that do not contain any parameters. This approach can result in false positives but not in false negatives. It is thus guaranteed that every node flagged as being not reachable is in fact not reachable. The potential false positives are a result of the α parameter which is used to allow transitions to be partially taken. It was able to analyse every case on which it was applied and it was therefore found that this approach is capable of analysing reachability of non-parametric COCA.

The second approach can be applied on all COCA but it is found to be much slower than the first approach. It does have the advantage that it can be applied to parametric COCA. It was able to correctly analyse any case on which it was applied and it correctly identified all cases of dead code which it was given. The earlier mentioned speed drawback is however significant and the first approach should therefore be preferred over the latter for non-parametric COCA. For the parametric COCA it was found to be a viable approach that is capable of identifying the reachability of nodes.

The combination of the two approaches gives a solution that can be applied on any kind of COCA and we can therefore conclude that the combined approach satisfies the first research question.

Question 2: Can we identify dead code in software written in the C language by analysing the reachability of the corresponding counter automata?
During this research we extended a previously performed research during

which a compiler was created capable of converting C code to COCA with a practical approach used to identify the reachability of nodes within a COCA.

We were able to prove that if convertible to COCA, our approach is capable of performing reachability analysis on them. The problem lies with the conversion which was proven to be harder than expected.

To fully conclude that this is a viable approach additional research will need to be performed on alternatives for the compiler that was used. Given that an alternative can be found that is capable of converting C code in a finite amount of time, this combined approach would be capable of identifying dead code. This was proven for the code which the compiler was able to convert, as well as the manually created test cases.

8. FUTURE WORK

As it was originally aimed to be able to fully analyse C code an alternative to the initial compiler is vital. The compiler approach used does work for simple cases but is very hard to get fully operational for more complex cases due to the optimization cycles. These optimization cycles are a utopia and will be impossible to maintain considering that it is almost impossible to correctly identify every possible node configuration. A potential solution for this could be compilation to a lower level language such as LLVM. By doing this the knowledge of which line is unreachable would get lost but the complexity would get reduced and all compiler specific code would be removed. It would furthermore greatly increase the field in which this approach can be applied as almost every language can be converted to LLVM. For simpler languages such as LLVM it is much easier to directly analyse the code which thereby reduces the risk of misinterpretation as was often the case with parse tree analysis.

Another optimisation that should at least be considered is the expansion of the lines that we can map to counter automata. This can either be more code statements that can be mapped to counter automata or more statements supported within the counter automaton itself. As it now stands we were only able to analyse 4% of all the functions of the project. While it is true that those 4% are most often core functions that are reused several times it would be even more useful if we could analyse a larger amount of functions.

Further optimizations with regards to efficiency would also be greatly beneficial to the speed of this analysis. While the exact speed of the entire approach is very hard to correctly analyse due to the issues with the compiler, it was clear that the analysis for parametric one counter automata took very long which is a result of the explosion in the number of nodes. This could be avoided by performing another optimization cycle to the counter automaton before analysis. For example, chains of nodes without operations and conditions could simply be merged into one.

9. CONCLUSION

In this paper we propose an algorithm to analyse the reachability of one counter automata. Two different approaches are given, one for non-parametric COCA and one for parametric COCA. Both make use of the concept of counter value reachability, which allows us to track whether or not there is a set of counter values which can be reached in a given node. if this interval is not empty we know that a node is reachable.

The non-parametric approach makes use of a recursive evaluation of these reachability intervals where each transition that is capable of firing gets fired once per iteration. We also introduced the notion of acceleration in which we identify behaviour of loops by analysing past behaviour. This resulting pattern can then be used to calculate the final value of the loop, thus greatly reducing the number of updates needed before the intervals will reach their final value.

The parametric approach was done by converting all intervals to formulae. Using a capped number of intervals we can then try to look for a mapping for which all intervals satisfy the given conditions. These conditions are converted versions of the conditions applied in the non-parametric approach. The final mapping will then be found by applying a SAT-solver, thus removing the need to find a suiting value for parameters as this will all be resolved by the used SAT-solver.

The proposed algorithm was applied on a project for which it was able to analyse all resulting automata. A total of 975 functions were analysed of which 33 proved suitable for counter automaton analysis. Of these 33 all correctly proved to be fully reachable. There were 2 parametric one counter automata and 31 non-parametric one counter automata. The approach used to do the conversion from C code to automata was however not optimal and a different approach such as LLVM analysis should be considered. For the project under scope no dead code was present and thus additional tests were written to test whether or not the approach is able to identify dead code. For all the designed mutations of the project code the approach was capable of identifying the dead code by directly returning the lines which were found unreachable.

While this is promising there is further need for optimization. The set of analysable functions is very restricted. Further expansion of one-counter automata should be considered. That being said, it was proven that the approach is capable of analysing one-counter automata. It was furthermore proven that it works in a variety of cases and that it is able to identify dead code resulting from unsatisfiable integer conditions in case there are any.

BIBLIOGRAPHY

[1] Michael Blondin et al. “Continuous One-Counter Automata”. In: *arXiv preprint arXiv:2101.11996* (2021).

[2] Yih-Farn R. Chen, Emden R. Gansner, and Eleftherios Koutsofios. “A C++ Data Model Supporting Reachability Analysis and Dead Code Detection”. In: *SIGSOFT Softw. Eng. Notes* 22.6 (Nov. 1997), 414–431. ISSN: 0163-5948. DOI: 10.1145/267896.267924. URL: https://doi.org/10.1145/267896.267924.

[3] John Ellson et al. “Graphviz— Open Source Graph Drawing Tools”. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484. ISBN: 978-3-540-45848-7.

[4] John Fearnley and Marcin Jurdziński. “Reachability in two-clock timed automata is PSPACE-complete”. In: *Information and Computation* 243 (2015). 40th International Colloquium on Automata, Languages and Programming (ICALP 2013), pp. 26 –36. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2014.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0890540114001564.

[5] Haase C.; Kreutzer S.; Ouaknine J.; Worrell J. “Reachability in Succinct and Parametric One-Counter Automata.” In: *Lecture Notes in Computer Science* 5710 (2009). DOI: https://doi.org/10.1007/978-3-642-04081-8_25.

[6] *Reachability Analysis for Annotated Code*. SAVCBS ’07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, 23–30. ISBN: 9781595937216. DOI: 10.1145/1292316.1292319. URL: https://doi.org/10.1145/1292316.1292319.

[7] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Partial Dead Code Elimination". In: *SIGPLAN Not.* 29.6 (June 1994), 147–158. ISSN: 0362-1340. DOI: `10.1145/773473.178256`. URL: `https://doi.org/10.1145/773473.178256`.

[8] Daniel Bundala; Joel Ouaknine. "On parametric timed automata and one-counter machines". In: *Information and Computation* 253, part 2 (2017), pp. 273–303. DOI: `http://doi.org/10.1016/j.ic.2016.07.011`.

[9] B. P. Pokkunuri. "Object Oriented Programming". In: *SIGPLAN Not.* 24.11 (Nov. 1989), 96–101. ISSN: 0362-1340. DOI: `10.1145/71605.71612`. URL: `https://doi.org/10.1145/71605.71612`.

[10] *Evaluation of the impact of code refactoring on embedded software efficiency.* 2010, pp. 145–150.

[11] Yuepeng Wang et al. "Hunter: Next-Generation Code Reuse for Java". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, 1028–1032. ISBN: 9781450342186. DOI: `10.1145/2950290.2983934`. URL: `https://doi.org/10.1145/2950290.2983934`.

[12] Yih-Fam Chen, E. R. Gansner, and E. Koutsofios. "A C++ data model supporting reachability analysis and dead code detection". In: *IEEE Transactions on Software Engineering* 24.9 (1998), pp. 682–694. ISSN: 2326-3881. DOI: `10.1109/32.713323`.