Reachability analysis for continuous one-counter automata

Lars Van Roy
dept. of Mathematics and Computer Science
University of Antwerp
lars.vanroy@student.uantwerpen.be

May 27, 2021

Abstract

Reachability is a known problem that can heavily affect the efficiency of code. It has been proven that reachability is decidable for continuous one counter timed automata which we will utilise in our approach. The proposed approach will provide a means to convert c code to one counter timed 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. Within this paper, this approach will be repeated here, as the approach is specifically intended to be combined with this predecessor. Finally, we will apply the combination of approaches on an existing code base, as to determine the usefulness of the approach.

Contents

1 Introduction

A well known problem in writing code is the reachability of specific lines in said written code. 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 [8, 5]. Unreachable code is never desired, but is an ever more pressing issue for energy restrained environments such as embedded computer systems [7]. 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 reachability prevention approach is needed.

To analyse the reachability, we will make use of the fact that code is quit 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 represented by one or more nodes and a traversal of an automaton can be directly linked to the traversal of the code itself.

The continuous one counter automata are especially interesting as it has been proven that reachability is decidable for this type of automata [4, 6, 1]. In this paper we will provide a practical implementation for the theoretical approach that was provided by Tim Leys 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 two counter machines. Three counter machines and up have been proven to be undecidable [3].

In this paper we will provide the overview of an approach which will convert c code to a 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. Convert c code into a parse tree, using ANTLR
2. Perform context free reductions on this parse tree
3. Perform context sensitive reductions on this parse tree
4. Validate the resulting parse tree, to be conforming to the code constraints

5. Generate the counter automaton

6. Analyse which nodes are reachable

7. 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 then the described code constraint, functions must also have a boolean return type, and integer parameter types (if any) to be converted into a counter automaton.

2 Motivation

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, one of

```
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 }
```

the more commonly used C IDE's. While it is known that Clion can find dead code that directly results from 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 Clion did not detect this.

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.

The impact of removing this line can be significant if considering cases where bigger numbers are passed. In those cases the loop on line 3 will be executed several times, making the condition on line 4 being quit impactful on the total evaluation time of this function. This clearly shows that there is value in enhanced dead code elimination, compared to the current standards.

3 Preliminaries

This paper will make use of counter automata and more specifically 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
- $\delta : Q \times \Sigma \to Q$ 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. In our analysis the goal will be to find the range of possible counter values that the counter can have in each state $q \in Q$.

The τ parameter represents the mapping from all states to the interval of possible counter values that can exist in that state. For example, if a node q has an associated condition stating ≤ 4 the τ function would have entry for node q equal to $(-\infty, 4]$. 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.

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

Parametric counter automata, as compared to regular counter automata, allow counters to be modified by parameters as well as constants. As the counters need 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.

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

The final state is omitted from this definition as we are not interested in this. In general the reachability of every single line of code will be under

scope and not a single one of the lines. 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 Blonding et al. [1] and will be the basis for the remainder of the approach.

Throughout the evaluation we will be tracking an interval I which will have $|Q|$ entries each corresponding to the counter values that can exist within their respective node.

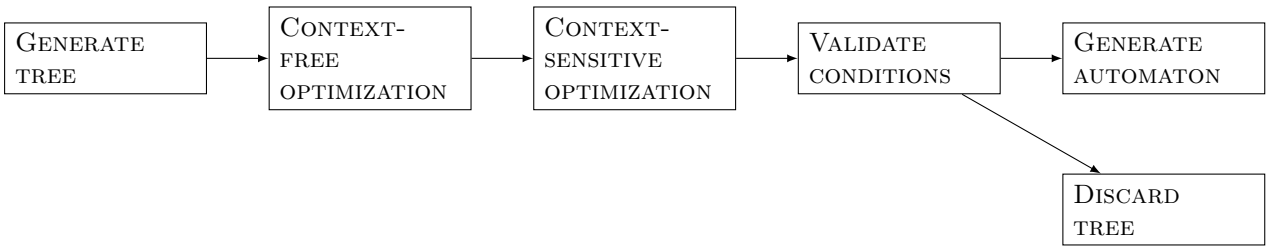
The concept of reachability will be associated with whether or not a certain node has a non empty interval at the end of the evaluation. A non empty interval automatically implies that the node must be reachable cause if a node can have a counter value, it recursively implies that there must be a path π in which all preceding nodes also have non empty intervals.

For this evaluation we will allow transitions to be partially taken. This means that every transition can be scaled by an unknown factor $\alpha \in (0, 1]$. This implies that given that we have a preceding interval $[0, 3]$ and an operation $+2$ our resulting interval will not be $[2, 5]$ but rather $(0, 5]$ as with $\alpha \approx 0$ we would remain at zero and with $\alpha = 2$ we would be able to go up to 5. The zero is not included within the resulting interval as we do not allow our α to actually be zero.

The partial aspect of transitions leads to potential false positives. In the example of $[0, 3] + 2 \rightarrow (0, 5]$ we might later on assume that counter values can be 0, which is not necessarily the case. We do however guarantee that in case we decide that a node is not reachable, this does hold. This implies that we may not be able to prove the reachability of every line, but for every line where we can prove it we know for certain that it is correct.

4 C code to automata conversion

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 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 optimization will help us in the third step, where we will do a more in depth optimization cycle. This third cycle 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 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 it must have a boolean return value, 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 Initialization

The program will start of 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.

4.2 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 a specific kind of node will not be created, 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.

4.3 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. Consider the code snippet below. It would initially be rejected, because the counter variable is initialized with a variable 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 }
```

However, due to constant propagation and folding, we can simplify the two statements on line 2 and line 3, so that they become.

```
1  int variable = 25;
```

Now that the variable is just a constant, we can substitute this variable in the declaration of counter, so that we now have the following.

```
1  int variable = 25;
2  int counter = 25;
```

The new code segment is acceptable, but we will not stop here, as the first line of the code is useless. It has no further benefit for the execution and will therefore be dropped. The final counter assignment will not be folded.

```
1  bool divide(int p, int n){
2    int counter = 25;
3    while(counter > 0){
4      counter -= n;
5    }
6    counter += 1;
7    counter -= 1;
8    if(counter == 0){
9      return true;
10   }
11   return false;
12 }
```

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. The final code segment is as described above, which is trivially accepted to be converted to a counter automaton.

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 1 occurs in the automaton, we would technically not be allowed to accept this, as this is not directly in the specified format.

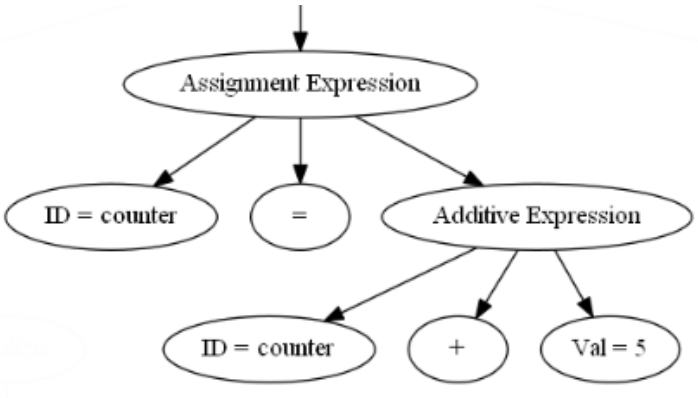


Figure 1: An Assignment Expression with counter addition as value.

However, the context-sensitive reduction will be able to reduce the earlier specified example to the automaton in Figure 2. This automaton will be acceptable, as this is a direct use of an allowed operation.

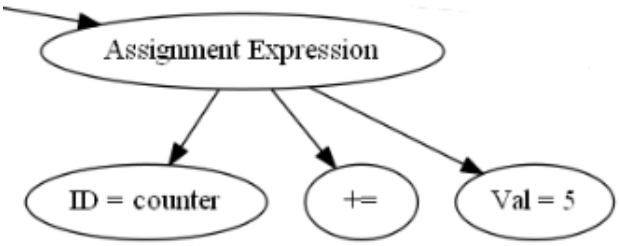


Figure 2: An Assignment Expression with constant as value.

4.3.1 Nodes

After the cleaning cycle has finished, there will only be a relatively small set of nodes remaining. Below a quick overview and their meaning within the syntax tree.

Compilation Unit indicates the root of each Abstract Syntax Tree.
Function Definition indicates the definition of a new function.
Function Specifier indicates specifier of function.
Generic head node for generic specification.
Generic Association indicates a generic association.
Parameter Type List head of the specification of all parameters of a function.
Parameter Declaration head node of the declaration of a single parameter.
Compound Statement indicates a scope defined by { } brackets.
Type Specifier indicates that the child(ren) of this node need to be regarded as being known type(s).
Type Name indicates that the child(ren) of this node need to be regarded as being a type(s).
Type Def Name indicates the definition of a type.
Declaration indicates a declaration, this can contain multiple types and identifiers. In case of a function, the function name and parameter types plus identifiers will all belong to a single declarator node.
Init Declarator defines a declaration of a variable with an initial value.
Direct Declarator defines a declaration of a variable with a second variable.
Declarator head node for the variable name part of the declaration.
Initializer head node for the value part of the declaration.
Primary Expression defines the most basic expression, eg. brackets, identifiers, constants, ...
Postfix Expression defines a postfix expression.
Unary Expression defines a unary expression.
Cast Expression defines a cast expression.
Multiplicative Expression defines a multiplicative expression.
Additive Expression defines an additive expression.
Shift Expression defines a shift expression.
Relational Expression defines a relational expression.
Equality Expression defines an equality expression.
Bitwise And Expression defines a bitwise and expression.
Bitwise Xor Expression defines a bitwise exclusive or expression.

Bitwise Or Expression defines a bitwise inclusive or expression.
Logical And Expression defines a logical and expression.
Logical Or Expression defines a logical or expression.
Conditional Expression defines a conditional expression.
Assignment Expression defines an assignment expression.
Expression head node in case of multiple expressions.
For Declaration head node for the first clause of a for loop.
For Expression head node for a clause of a for loop.
For Condition head node for the condition part of the second clause of a for loop.
Iteration Statement defines an iteration statement (for, while, do while).
Jump Statement defines a jump statement.
return indicates return statement.
Labeled Statement indicates a labeled statement.
Struct or Union Specifier indicates the specification of a struct or union.
Enum Specifier indicates the specification of an enumerator.
Struct Declaration head node for a struct declaration.
Struct Declarator head node for a struct declarator.
Static Assert Declaration defines static assertion.
Enumerator specifies variables part of an enumerator.
Size head for array size.
Default defines the default option for switch statement.
Alignment Specifier defines an alignment restriction to an identifier.
Atomic Type Specifier defines an atomic type.
Arguments head node for arguments part of a function call.
sizeof defines sizeof operation.
_Alignof defines _alignof operation.
Val = <value> defines a constant.
ID = <id> defines a variable name.

4.4 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 can't 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. A function needs to have a boolean return type, and 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. We obtain the automaton displayed in Figure 3.

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

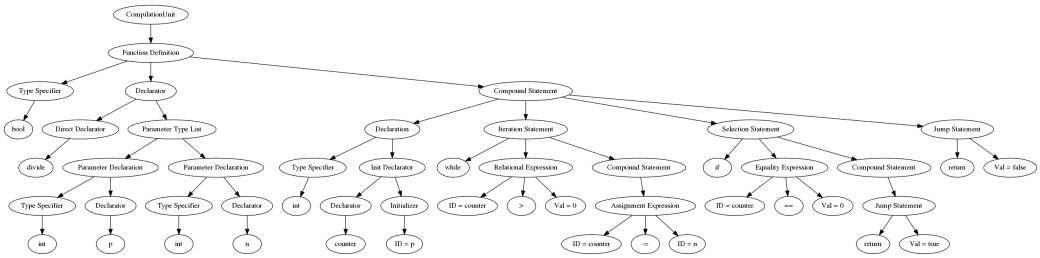


Figure 3: The cleaned Abstract Syntax Tree of the divide function.

We can immediately see that the function conforms to the required function definition, as the return type is boolean, as can be seen from the leftmost

node. The parameters are also 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 special operations, we can conclude that this function satisfies the requirements for one counter automaton generation.

4.5 Counter automaton generation

The final section of the program 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 things drastically. 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.

4.5.1 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.

4.5.2 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. 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.

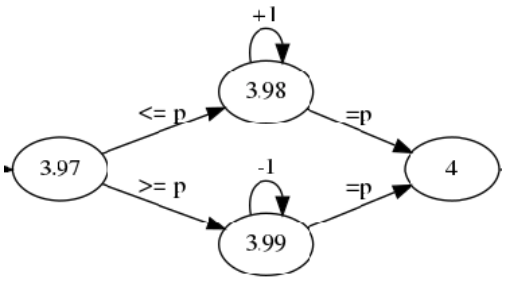


Figure 4: Assignment of a value p to a counter.

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 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.

4.5.3 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 makes use of an if statement which has an equality condition in it, being the edge going towards the else node.

```
1 bool is_ten(int c){
2 if(c == 10){
3 return true;
4 }
5
6 return false;
7 }
```

When looking at part of a simplified counter automaton in Figure 5, 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.

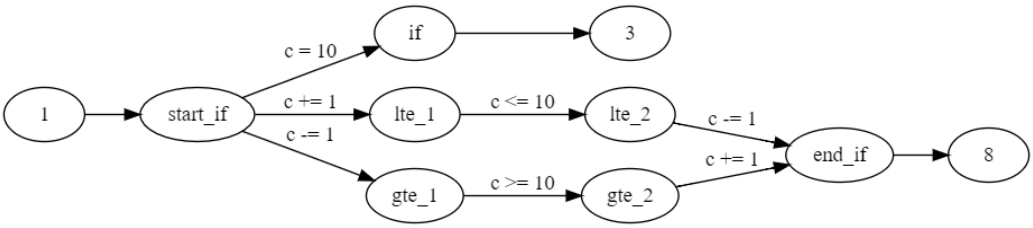


Figure 5: Example of an inequality expression.

The first transition chain 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 transition chain 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.

4.5.4 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.

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

The is_ten code segment 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.

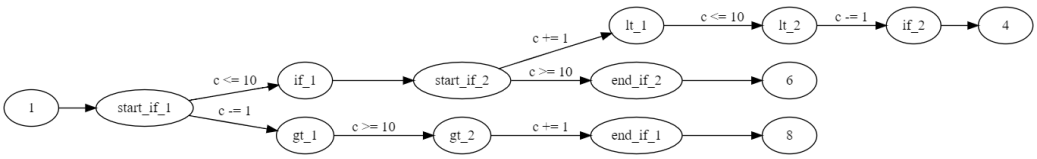


Figure 6: Example of a less than expression and a greater than expression.

A simplified version of the counter automaton of the is_ten function can be seen in Figure 6. 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.

4.5.5 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*, this node symbolizes the state after the precondition of the for loop has been evaluated. Next we need a node to identify the *start of the inner segment*, which 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 following segment of code, this will result in the automaton in Figure 7.

```
1 bool test(int c){
2   for(c = 0; c < 5; c ++){
3     continue;
4   }
5   return true;
6 }
```

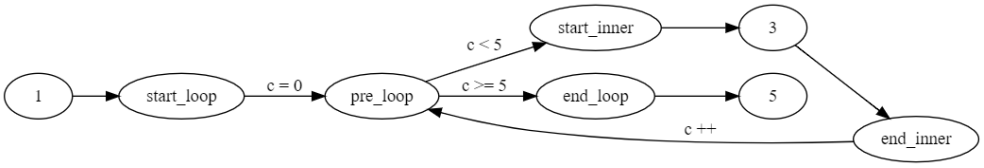


Figure 7: Example of an Iteration Statement.

4.5.6 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.

We will need an *if* node and an *else* node which will both signify the start of the if and possible else segment.

After the inner segment, they will all end up at the *end if* node from which point they will continue with the next line of code.

Consider the test2 function, and the corresponding simplified counter automaton.

```
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;
}
```

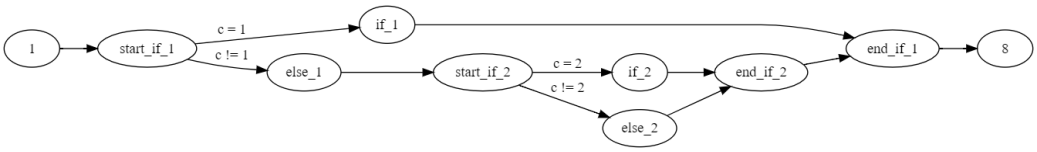


Figure 8: Example of if statements.

The program will return a more in depth version of the simplified counter automaton displayed in Figure 8. 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*.

4.5.7 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 following code snippet. In this case, a variable a will be compared with multiple values, and depending on the matched case, a number of prints will occur.

```
1  bool test(int a){
2    switch(a){
3      case 3:
4        printf("The parameter is greater than 2.");
5      case 2:
6        printf("The parameter is greater than 1.");
7      default:
8        printf("The parameter value is %d", a);
9    }
10   return true;
11 }
```

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 9.

After that 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.

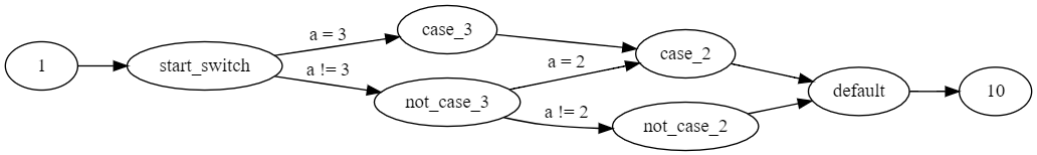


Figure 9: Example of a switch statement.

4.5.8 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 below, 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;
}
```

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.

4.5.9 Final overview

To give a in depth overview of how the conversion works, we will transform the following code 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 $<$ and $>$ 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 and a bool return type, 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;
```

```
4   int i1 = 1;
5   int i2 = 1;
6   while(counter != 0){
7     if(counter > 0){
8       counter -= b;
9       i2 += 1;
10    }
11    else if(counter < 0){
12      counter += a;
13      i1 += 1;
14    }
15    if(counter == 0){
16      printf("the greatest multiple is equal to ");
17      printf("%d * %d = %d * %d = %d.", i1, a, i2, b, i1*a);
18    }
19  }
20  return true;
21 }
```

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 10

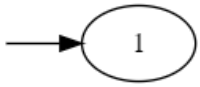


Figure 10: First node in the counter automaton.

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 11.

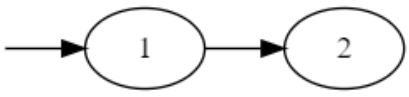


Figure 11: Automaton corresponding to the second line of *gcm*.

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 by 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 12.

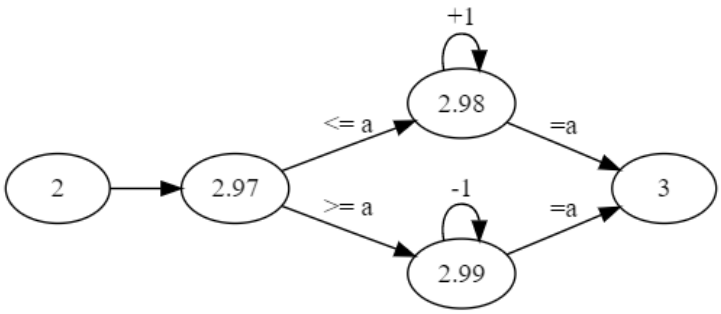


Figure 12: Counter automaton modelling the initial assignment of the counter.

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 13.

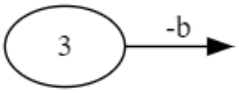


Figure 13: Counter automaton modelling the operation on line two.

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 14.

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.

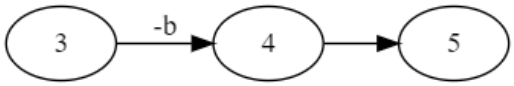


Figure 14: Counter automaton resulting from line four and five.

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 4.5.3, Inequality conditions.

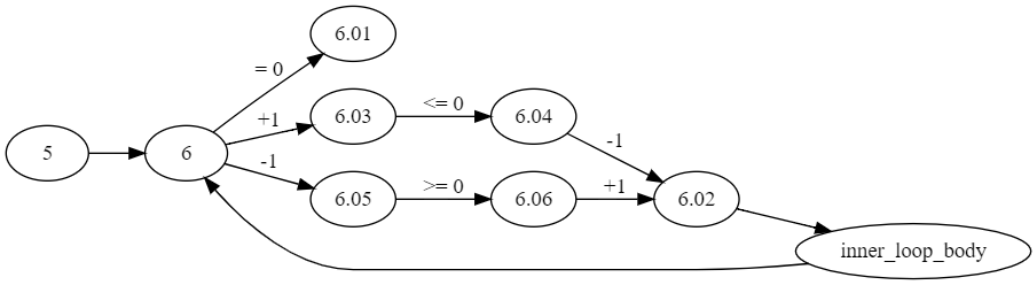


Figure 15: Counter automaton resulting from line six.

Lines seven until ten define the first selection statement. Selection statements require, as referenced in Section 4.5.6, 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, 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 4.5.4, 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.

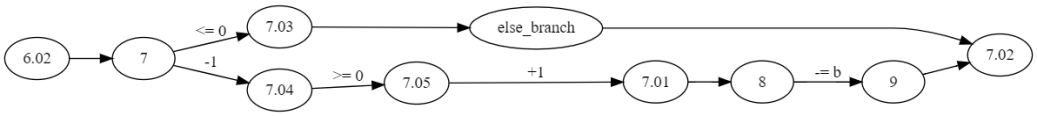


Figure 16: Counter automaton resulting from line seven up until line ten.

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 17. Note that there is no else branch, which was not generated as there are not other *else if* or *else* statements in the code.

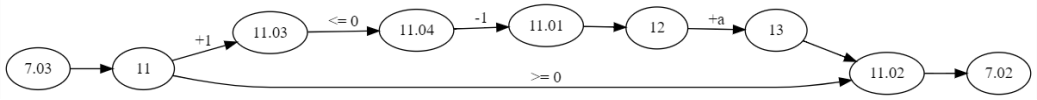


Figure 17: Counter automaton resulting from line eleven up until line fourteen.

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 18.

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.

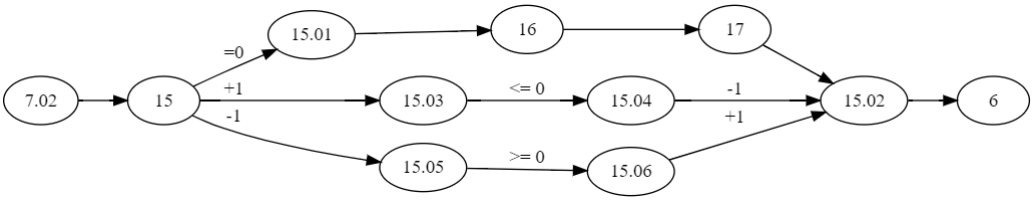


Figure 18: Counter automaton resulting from line fifteen up until line eighteen.

After a return, no nodes will be generated no matter what the kind of these statement is.

Figure 19: Counter automaton modelling the return on line twenty.

The complete overview of the counter automaton can be seen in Figure 20. This is a completely correct counter automaton that is suitable for reachability analysis.

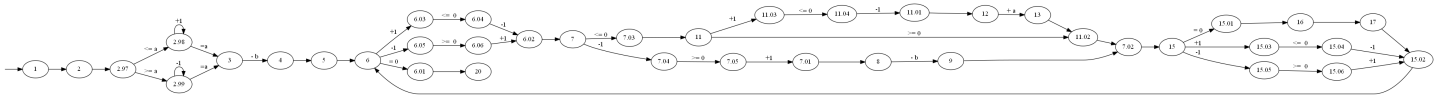


Figure 20: Counter automaton modelling the return on line twenty.

4.6 Automaton input

For the approach described within this paper, an input of an automaton using the dot language part of Graphviz was assumed [2]. 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.

4.6.1 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

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

Below in Figure 21 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.

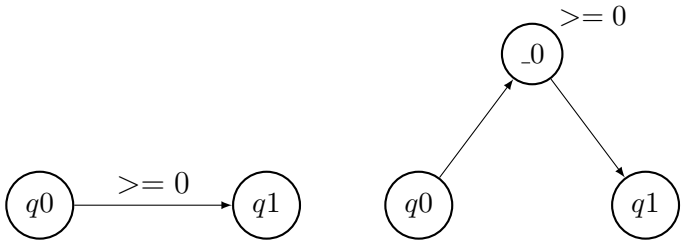


Figure 21: Transformation of an edge with a conditional label.

4.6.2 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 22 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.

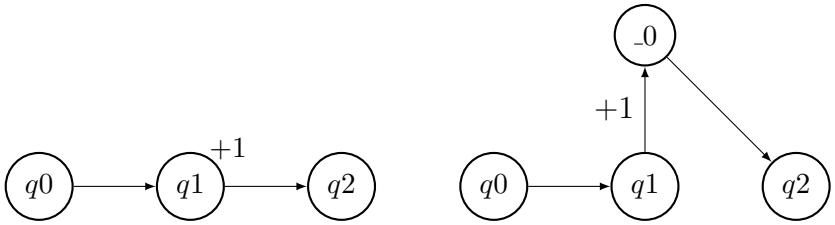


Figure 22: Transformation of a node with an operational label.

4.6.3 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.

4.7 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].

4.7.1 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.

4.7.2 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 $R_i : Q- > 2^{\mathbb{Q}}$ represent the mapping function for the i-th iteration. This function is going from each state to its corresponding counter reachability interval. The *succ* function will update the interval of one state, q, to its next iteration, based upon the current configuration.

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

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 23. We will initialise R_0 to have all empty intervals, other than the first which we will initialise to $[0, 0]$. Note that these intervals are stored per pair of final and preceding node and not, as the formula would suggest, 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.

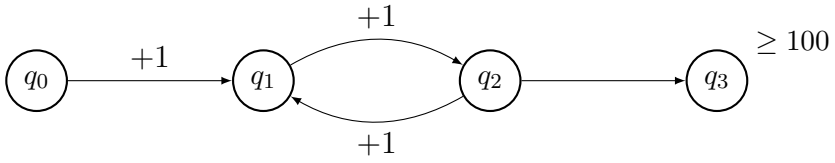


Figure 23: Example of a guarded automaton on which the succ function can be applied.

q	r	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_1	q_0	$\emptyset$	(0, 1]	(0, 1]	(0, 1]	(0, 1]	(0, 1]
q_1	q_2	$\emptyset$	$\emptyset$	$\emptyset$	(0, 3]	(0, 3]	(0, 5]
q_2	q_1	$\emptyset$	$\emptyset$	(0, 2]	(0, 2]	(0, 4]	(0, 4]
q_3	q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1: Decision table for the scenario in which a guest occupies a new cell.

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 preceding node that is not empty. All other transitions will both have an empty interval for the start and ends 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 transitions 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 1 a few more successors were computed and we can clearly notice that we will eventually reach an interval which will exceed the 100 bound which is placed upon q_3. Reaching this interval will however

take us quite some time, and considering that this is only a small example, this is far from desired. Furthermore, this loop is not bounded, this means that it can infinitely expand. A solution for this is loop acceleration in which we will accelerate our loop based upon the behaviour said loop shows. This solution will be discussed within the next sections.

4.7.3 Loop acceleration

Unfortunately, the succ function is not guaranteed to find the bounds in an acceptable 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 state sequence $q_i, q_{i+1}, q_{i+2}, ..., q_n$ such that:

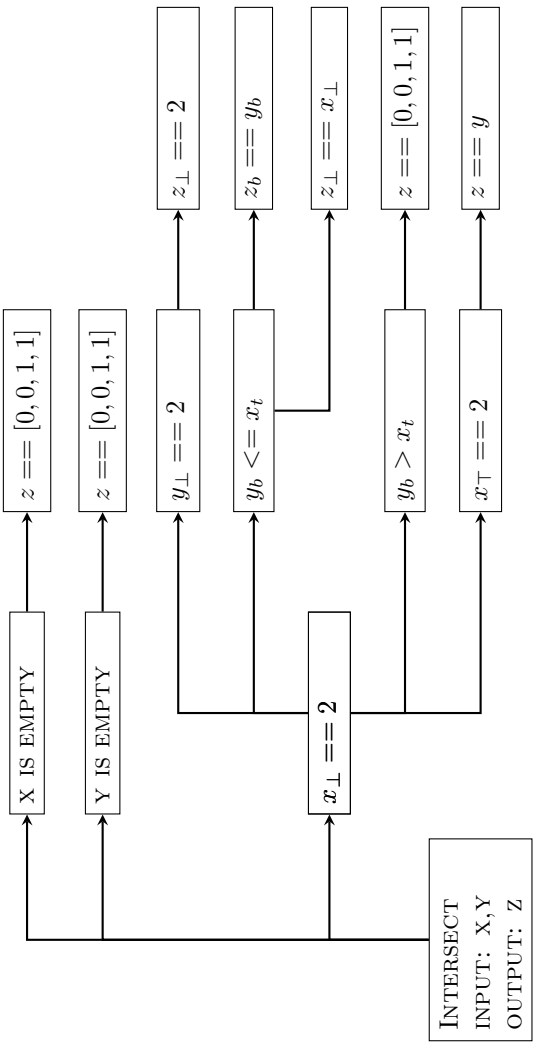
1. $q_i = q_n$
2. $(q_j, z_j, q_{j+1}) \in T$ for all $j \in [i, n-1]$
3. $R_n \subseteq R_i$
4. $sup(R_i) \geq sup(R_n)$

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

1. $q_i = q_n$
2. $(q_j, z_j, q_{j+1}) \in T$ for all $j \in [i, n-1]$
3. $R_n \subseteq R_i$
4. $inf(R_i) \leq inf(R_n)$

Given that we know whether or not a cycle is either positively or negatively expanding, we can look for the strongest bound relative to the direction in which the loop expands.

4.7.4 Efficiency


INTERSECT
INPUT: X, Y
OUTPUT: Z
X IS EMPTY
Y IS EMPTY
$x_\top == 2$
$z == [0, 0, 1, 1]$
$z == [0, 0, 1, 1]$
$y_\top == 2$
$y_b <= x_t$
$y_b > x_t$
$x_\top == 2$
$z_\top == 2$
$z_b == y_b$
$z_\top == x_\top$
$z == [0, 0, 1, 1]$
$z == y$

5 Tool Validation

To determine the corectness of our tool, we created a test suite consisting out of three groups of tests, which are all white box tests. The tests will check the outputted dot and the trace output of the program, against traces and dot files which have been manually generated and verified.

The first group will only test the first loop of the program, and it will check whether or not all nodes get properly generated. This purely tests on the generation of nodes, and will not regard any variation in configuration in which these nodes could occur.

The second group will test the cleaner. It will test most cases (all common cases) in which folding and substituting should occur as well as the cases where it should not fold anything. In this group we will also perform extensive symbol table testing, to check whether or not all values get properly tracked by the symbol table.

The third group will test the full tool. It will validate the dot file resulting from the cleaner, as well as the dot file(s) resulting from the counter automaton generator. It will also evaluate all output regarding the symbol table, and the potential errors of the code validator. There are happy day tests where each conditional statement gets tested separately, there are tests where conditional statements get nested and there are tests to see whether we can still evaluate conditional statements with constant conditions. Furthermore, there are failure tests that test whether or not we can properly detect incorrect counter types, incorrect parameter types, incorrect number of counters, incorrect conditional operations, incorrect counter modifications and properly detect counters defined in global scopes. There are also tests for edge case scenarios, such as returns in every single branch, the cleaner will not be able to detect these, and therefore the generator should be able to stop generating all nodes, once there are no branch that can carry on.

The test coverage was validated using Codecov, which gave a total line coverage of 93% at the end of development. This coverage was used as a metric to determine whether or not there were sufficient tests at each stage of development. The remaining 7% are trivial lines that did not need any testing, such as $\_\_str\_\_$ operators.

6 Conclusion

In this paper we propose an algorithm for converting a small subset of the c language to counter automata, and discuss its implementation. The resulting automata can be used in reachability analysis within the code.

The paper gives a description on all the steps the program goes through in order to achieve this counter automaton, as well as a brief insight in why things are the way they are.

This proposed algorithm can be used to improve existing static code analysis and will furthermore be usable to solve the halting problem for a subset of c programs. It is however only usable in a limited amount of cases, in more complex situations, where reachability analysis would be far more desired, it will most likely not be usable as chances of violated conditions will be high. In the situations where it is applicable, big conditional statements or a large amount of conditional statements can make determining the reachability of certain sub segments hard, and with the addition of this algorithm as a reachability checker, it could definitely lead to improvements in coding.

7 Future work

One of the future aspects that still needs to be worked on, is the precision, efficiency and usefulness of the program. The program is too restrictive to be practically usable, and should therefore be expanded to support further c code.

Furthermore, before the resulting counter automata have any use, there needs to be an expansion to the tool, which will allow the evaluation of counter automata, so that actual reachability conclusions can be made.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] *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.

[6] 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.

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

[8] 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.