Software Verification
Implementing a bounded model-checker

Ida Tucker

December 15, 2016

Abstract

The goal of this homework is to implement a bounded model-checker based on a depth-first-search exploration of all the feasible paths of the input program. The SMT-solver Z3 is used as an execution engine to compute all the successive states along the exploration. In this report is first described how operators of the program are translated into Z3 propositional formulas via the module `Semantics.ml`. Then the logic behind the implementation of the bounded model checker is laid out and finally a series of additional test cases are explained, with actual and expected results.

1 Implementation of Semantics.ml

The `Semantics.ml` module translates the operators of the program into Z3 expressions in the integer theory module.

1.1 Keeping track of formulas added to the Z3 expression

In order to keep track of the formulas associated to each variable, a hashtable, global to the module, which is reset for every new transition, is used. This hashtable `ht` binds the indexed program variables to the appropriate formula. Separate keys are added if the formula is asserting that a variable is non zero. These keys are of the form `x$i-non-zero`. If the program operation is a guard, the hashtable has one additional key `guard` since the formula for the guard exists alongside formulas added for each variable. Moreover we will only have one guard formula per operation.

Using a hashtable enables us to avoid the duplication of formulas associated to a given variable and to easily update them without having to check their existence.

Thus the keys in the hashtable are:

- $\forall x \in X$, x$i
- $\forall x \in X$ such that we require $x \neq 0$, x$i-non-zero
- if *op* is a guard, guard

1.2 Defining propositional formulas

Now we have established how to keep track of the added formulas, let us see how they are built.

The commands are our program operations. In order to transform the program operation into a list of formulas we first need to determine what command is being applied. Therefore, the `formula` method pattern matches the input command `cmd` with the appropriate sort. According to the the matched sort, the appropriate function `apply_skip`, `apply_guard` or `apply_assign` is called.

```
match cmd with
| Command.Skip ->
   List.iter (apply_skip ctx  i) vars;
   Hashtbl.fold (fun _ v acc -> v :: acc) ht []
| Command.Guard p ->
   apply_guard ctx p i vars;
   Hashtbl.fold (fun _ v acc -> v :: acc) ht []
| Command.Assign (v, e) ->
   apply_assign ctx (v,e) i vars;
   Hashtbl.fold (fun _ v acc -> v :: acc) ht []
```

If the program operation is a *skip*, since the same formula will be applied to each variable we simply iterate through each variable in `vars` and apply skip. The `apply_skip` method is called on a single variable, it ensures that the variable remains unchanged from index i to index $i+1$ by adding a new formula to `ht` associated to $x^{(i)}$:

$$x^{(i)} == x^{(i+1)}$$

The addition of the formula to `ht` is done in the method `add_unchanged`.

The `apply_guard` and `apply_assign` methods, which are applied if the program operation is a *guard* or an *assignment* are detailed below in 1.3 and 1.4. They both also update the hashtable and return the sort `unit`.

Whatever the program operation, after the appropriate function has been called, we iterate through the hashtable to build the returned list of formulas.

1.3 Program operation is a guard

If the command is a guard it is defined as follows:

$$op^{(i)} = g^{(i)} \wedge \bigwedge_{y \in X} (y^{(i+1)} = y^{(i)})$$

And $g^{(i)} = e_1^{(i)} \sim e_2^{(i)}$ where $\sim$ is an operator in $\{=, \leq, <, >, \geq\}$

To satisfy this definition, a formula is added to the hashtable `ht` for the guard g, and for each variable we ensure that the variable remains unchanged from index i to index $i+1$. Ensuring variables remain unchanged is equivalent to calling `apply_skip` on all variables. Thus the `apply_guard` function first adds a formula for the guard, before iterating through each variable in `vars` and applying `apply_skip`.

In order to build the guard formula, the `eval` function evaluates expressions $e_1^{(i)}$ and $e_2^{(i)}$ recursively, in order to build formulas in the Z3 context for the expressions. The `eval` function takes care of adding a `non-zero` key to `ht` if the expression includes a division by a constant, a program variable, or an expression including program variables.

1.4 Program operation is an assignment

If the command is an assignment we have:

$$op^{(i)} = (x^{(i+1)} = e^{(i)}) \wedge \bigwedge_{y \in X \setminus \{x\}} (y^{(i+1)} = y^{(i)})$$

All variables should remain unchanged from index i to index $i+1$ *except* the variable that is being assigned to. Thus `apply_assign` iterates through each variable in `vars` and applies `apply_skip` before adding a formula for the assignment associated with $x^{(i)}$, which will replace the previous binding of $x^{(i)}$ in `ht`. i.e. the binding $\{$`x$i`, $x^{(i)} = x^{(i+1)}\}$ becomes $\{$`x$i`, $x^{(i+1)} = e^{(i)}\}$.

In order to build the assignment formula the expression assigned to $x^{(i)}$ is matched with either a constant, another variable or an expression. Again if $e^{(i)}$ is an expression we use the `eval` function to evaluate the expression.

1.5 Testing division by zero

Tests have been added to `Test_Semantics.ml` for the division by zero.

Test 1: division of an expression by a constant

Expected result: a formula should be added asserting the integer constant is non zero.

Obtained result:

```
[z := (x / 2)]^1: expects `{(= z$2 (div x$1 2)), (= x$1 x$2),
                            (= y$1 y$2), (not (= 2 0))}'
```

PASS

Test 2: division of an expression by an expression that is neither a constant nor a variable.

Expected result: a formula should be added asserting the expression is non zero.

Obtained result:

```
[z := (x / (y + 3))]^1: expects `{(= z$2 (div x$1 (+ y$1 3))), (= x$1 x$2),
                                  (= y$1 y$2), (not (= (+ y$1 3) 0))}'
```

PASS

Test 3: division of an expression by a more complex expression.

```
z := x / ((y * cst 2) + (y / (x + y)))
```

Expected result: formulas should be added asserting the relevant expressions are non zero.

Obtained result: an unexpected variable `a!1` is introduced, but presumably this is the way the Z3 solver deals with such expressions. Nevertheless the obtained result is correct.

```
[z := (x / ((y * 2) + (y / (x + y))))]^1: expects `{(let ((a!1 (div x$1
        (+ ( * y$1 2) (div y$1 (+ x$1 y$1))))))) (= z$2 a!1)),
    (let ((a!1 (= (+ ( * y$1 2) (div y$1 (+ x$1 y$1))) 0))) (not a!1)),
    (= x$1 x$2), (= y$1 y$2), (not (= (+ x$1 y$1) 0))}'
```

PASS

2 Implementation of BoundedModelChecking.ml

2.1 Marking locations as visited

Since the constraints associated to a location evolve as we propagate the search, this implementation of a depth first search does not mark the nodes as visited. If we were to mark them as visited we would need to ensure we have encountered a fixed point.

2.2 Keeping track of path and depth of search

- The global variable `depth` is incremented every time we apply `depth_first_search` to a child node.
- The global variable `path` is a list of pairs `(command, location)` that keeps track of the edges of the CFA taken to get to the current location.
- The global variable `bad_path` is a list of pairs `(command, location)` that lead to the final location, if such a path of depth lower than `bound` exists. If no such path exists (or has been discovered yet) `bad_path` is the empty list.
- `bound_reached` is a Boolean, initially set to `false`, to which we assign the value `true` if there exists a path along which the bound is reached before encountering the final location or an unreachable location.

2.3 Building the path and the path formula

Every time a new branch of the CFA is explored, the `depth_first_search` function checks if `bad_path` is empty, if not it returns.

If `bad_path` is empty, it checks if the `bound` has been reached, if so it returns.

Otherwise we are in the valid situation where the program should continue the depth first search. In this situation the following logic is applied:

- Increase depth by one.
- Create a new scope for the solver by saving the current stack size: pushing the solver will enable us to backtrack later on if we reach the `bound` or an unfeasible path.
- Call `Semantics.formula` on the program operation for the current depth in order to convert the command into a propositional logic formula.
- Add the formula to the solver.
- Add the pair `(command, location)` to the `path` list.
- Check the satisfiability of the solver. At this point, three situations may arise:
  1. The solver is not satisfiable, in which case we backtrack:
     - restore the previous context by popping the solver
     - decrease the depth by one
     - remove the pair `(command, location)` last added to `path`
     - return `unit`.
  2. The solver is satisfiable and we have reached the final location. In this case we assign `path` to `bad_path` and return `unit`.
  3. The solver is satisfiable, thus the location is reachable, and the location is not the final location. In this situation we iterate through all locations accessible from the current location and apply `depth_first_search`.

2.4 Searching for a path to the final state

The `search` function brings all the above functionality together. It first initialises the global variables before calling `depth_first_search` on the initial location with the command skip.

If after calling `depth_first_search` the `bad_path` list is not empty, then the depth first search found a feasible path to the final location and the `Path bad_path` is returned.

If no path to the final state is found, but the bound has been reached, the program may not be safe, we only know that there is no path to the final state in less than `bound` steps. In this case the `bound_reached` variable is true, and `Empty false` is returned.

Otherwise, all feasible paths of have been explored and none lead to the final state. The program is safe and `Empty true` is returned.

3 Tests

The test cases described bellow all take as input an automaton (from the `examples` directory), and expect an output of type `result`:

- a `Path` p where p is a feasible path of the automaton leading to the final state of length at most `bound`
- no feasible path
- no feasible path of length at most `bound`

3.1 New Automatons

The additional automatons created in the `examples` folder test edge cases, the case where no feasible path is found in less than 10 iterations, unsafe programs, potential infinite loops etc.

3.1.1 Initial state is bad state

Test File: `init_is_final.aut`

Test Edge case where the initial and final location are the same node.

Expected result A feasible path that is simply the initial location (no commands).

Obtained result **PASS**

3.1.2 Initial state skips to bad state

Test File: `init_skip_final.aut`

Test Edge case where the initial location skips to the final location.

Expected result A feasible path that is simply a skip from the initial location to the final location.

Obtained result **PASS**

3.1.3 `bound` is 0

Test Test the edge case where the bound is zero and the final location is not the initial location.

```
./bmc.d.byte -bound 0 examples/init_skip_final.aut
```

Expected result No feasible path of length ≤ 0.

Obtained result

```
No feasible path of length <= 0
```

PASS

Test Test the edge case where the bound is zero and the final location is the initial location.

```
./bmc.d.byte -bound 0 examples/init_is_final.aut
```

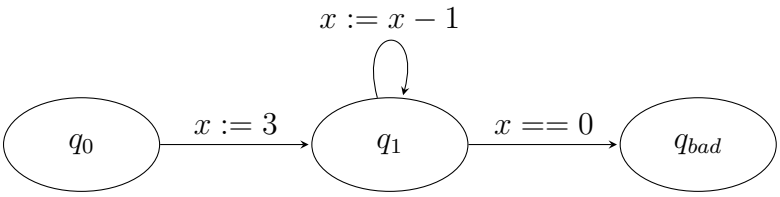
Expected result A feasible path that is simply the initial location (no commands).

Obtained result

```
Feasible path:   q_0
```

PASS

3.1.4 Looping until reaching the final location



Test File: `loop_location_bad.aut`

Test: Ensure the program will loop on a single state in order to reach the final node.

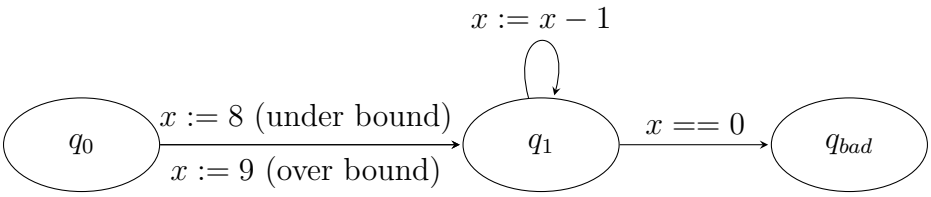
Expected result: A path to the final state.

Obtained result:

```
Feasible path:
   q_0
      >> x := 5 >>
   q_1
      >> x := (x - 1) >>
   q_1
      >> x := (x - 1) >>
   q_1
      >> x := (x - 1) >>
   q_1
      >> x == 0 >>
   q_2
```

PASS

3.1.5 Limit under and limit over `bound`



Test Files: `limit_under_bound.aut`, `limit_over_bound.aut`

Test: `bound` is 10 and:

1. the CFA requires exactly 10 steps to the bad state.
2. the CFA requires exactly 11 steps to the bad state

Expected result:

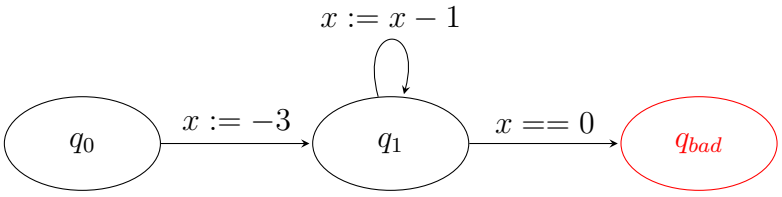
1. A path to the final state.
2. No feasible path of length ≤ 10.

Obtained result: **PASS**

3.1.6 Always negative

Test File: `always_negative.aut`

Test The following CFA



Expected result: No path to the final state should be found, since we have a safe inductive invariant:

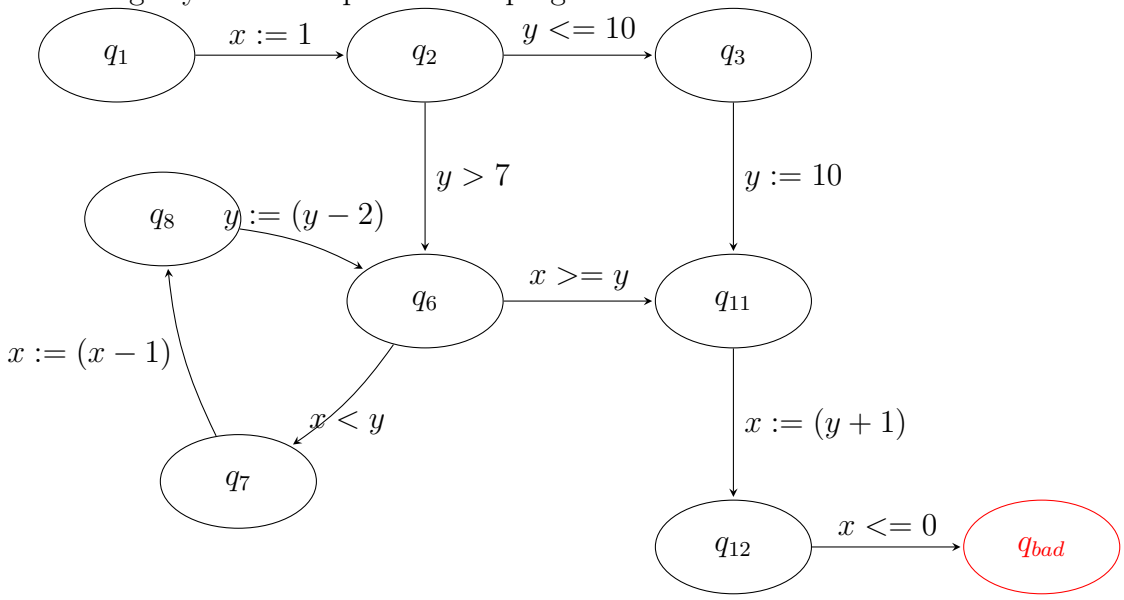
- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = x < 0$
- $\varphi_{q_{bad}} = \text{false}$

Obtained result: The bounded model checker does not guarantee the safety of the the program but whatever the `bound` set, it finds no feasible path of length $\leq$ `bound`.
PASS

3.1.7 Unsafe program

Test File: `unsafe_program.aut`

Test Slightly more complex unsafe program:



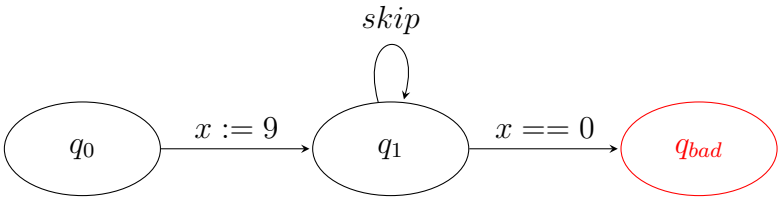
Expected result $\exists B_0$ such that:

- $\forall$ `bound` $< B_0 \rightarrow$ No feasible path of length ≤ 10.
- $\forall$ `bound` $\geq B_0 \rightarrow$: A path to the final state.

Obtained result For a bound of 26 a feasible path is found.
PASS

3.1.8 Recognising fixed points

Test File: `loop_never_sat.aut`



Test If a program that is safe contains a loop, whatever the chosen bound, no feasible path of length $\leq$ `bound` is found.

Expected result $\forall$ `bound` $\rightarrow$ no feasible path of length $\leq$`bound`.

Obtained result No path of length ≤ 1000 is found.
PASS

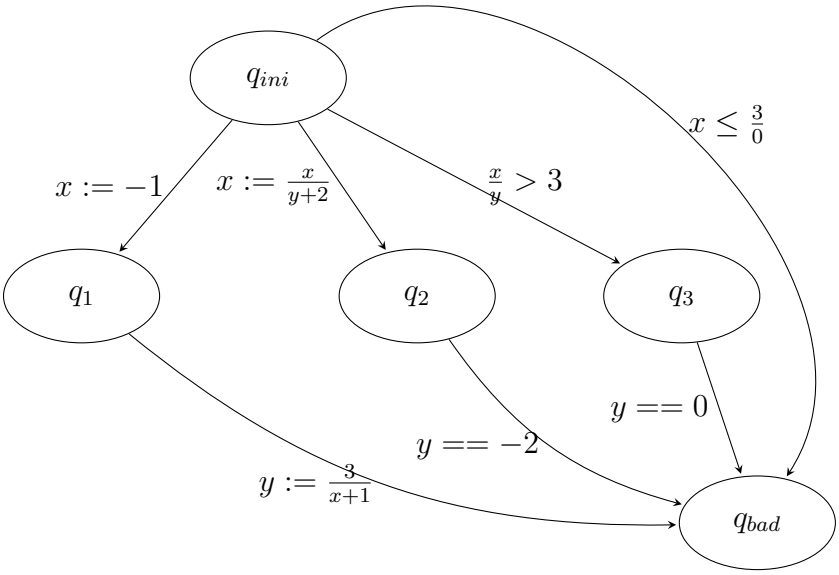
Remark The bounded model checker will never prove that a program that contains loops is safe. It will only prove that there are are no paths of length $\leq$ `bound` to the final location. The objective of a bounded model checker is to find bugs, and not necessarily to prove that the system is safe, so we can consider this to be expected behaviour even though, to a human eye, it is evident that this program is safe, regardless of the bound.

This test demonstrates that the bounded model checker gets stuck in a loop[1] if the state is the same as a previously encountered one. Therefore if a location has been visited and the solver obtained when reaching this location is equivalent to a solver previously obtained at this location, the depth first search still propagates.

Consequently, even though a program is safe, the bounded model checker will often only be able to guarantee that there is no path to the final location of length lower or equal to `bound`, for any positive bound.

3.2 Existing Automaton examples

3.2.1 Division by zero



The division by zero automaton should not lead to a bad state, indeed

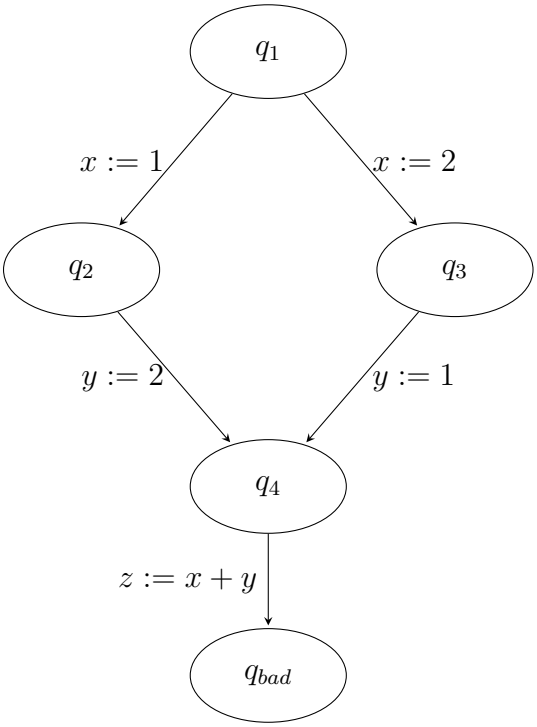
- $q_{ini} \rightarrow q_1$: enforces $x := -1$, so we cannot have $y := \frac{3}{x+1}$ we would have a division by zero, this produces an UNSAT formula.

[1] The model checker will only loop until the bound is reached.

- $q_{ini} \to q_2$: enforces $x := \frac{x}{y+2}$, so we cannot have $y == -2$.
- $q_{ini} \to q_3$: enforces $\frac{x}{y} > 3$, so we must have $y \neq 0$.
- $q_{ini} \to q_{bad}$: dividing by zero produces an UNSAT formula.

The bounded model checker finds that there is no feasible path.
PASS

3.2.2 Constant Propagation



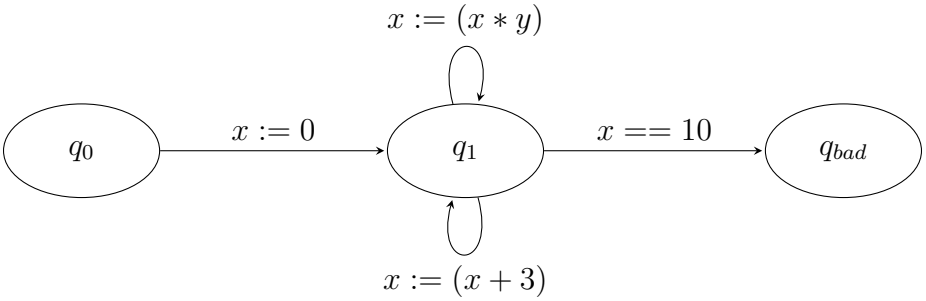
Clearly the constant propagation automaton is unsafe: for any initial value of x and y and for z=3, the final state q_{bad} is reached.

The bounded model checker finds the following feasible path to q_{bad}:

```
Feasible path:
   q_1
     >> x := 1 >>
   q_2
      >> y := 2 >>
   q_4
      >> z := (x + y) >>
   q_5
```

PASS

3.2.3 Inductive Invariants Exercise 1

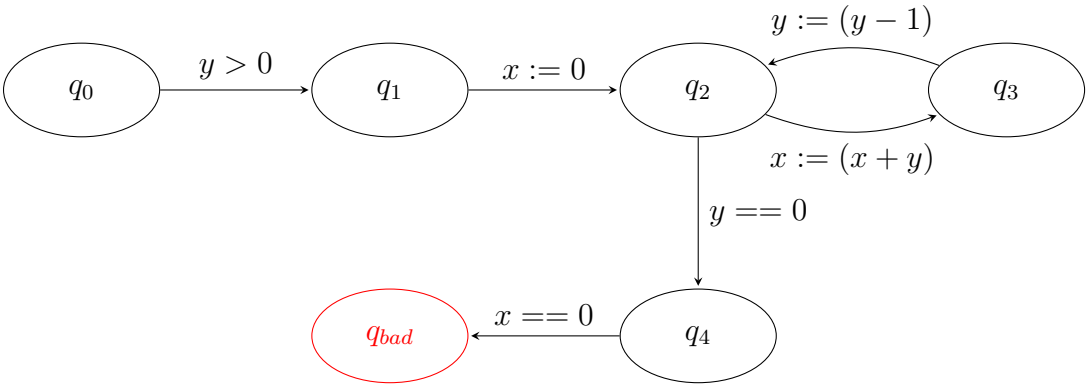


In this exercise we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = \exists k \in \mathbb{Z} : x = 3 * k$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker finds that there is no feasible path of length $\leq$ `bound`.
PASS

3.2.4 Inductive Invariants Exercise 2

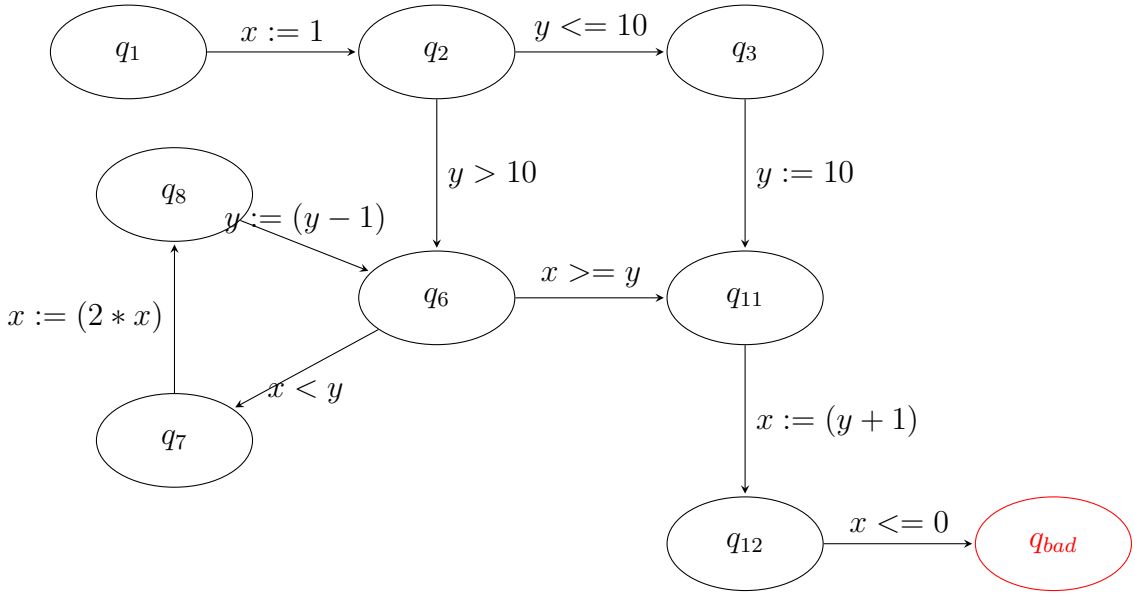


In this exercise we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = y > 0$
- $\varphi_{q_2} = (y > 0 \Rightarrow x \geq 0) \wedge (y = 0 \Rightarrow x > 0)$
- $\varphi_{q_3} = (y \geq 0 \Rightarrow x > 0)$
- $\varphi_{q_4} = (y = 0 \wedge x > 0)$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker finds that there is no feasible path of length $\leq$ `bound`.
PASS

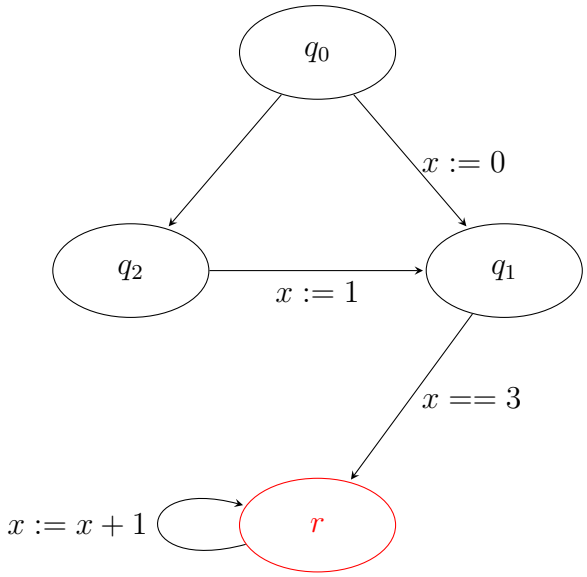
3.2.5 Running Example



With extended sign analysis it can be proved that this program is safe. The bounded model checker finds that there is no feasible path of length $\leq$ `bound`.

PASS

3.2.6 Infinite Descent



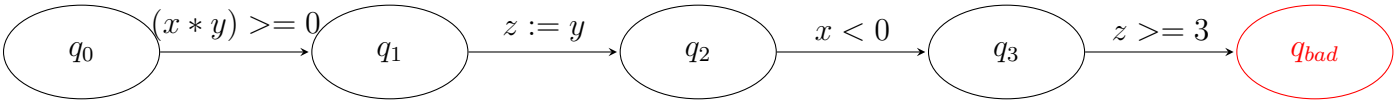
Clearly this automaton has no feasible path, we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = (x = 0) \vee (x = 3)$
- $\varphi_{q_2} = \text{true}$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker also finds that there is no feasible path.
PASS

Remark Since the loop in this example is on the final location, the bounded model checker does not get stuck in the loop. Indeed if it were to reach the loop it would have reached the final location, and found a path to the final location.

3.2.7 Pre Only

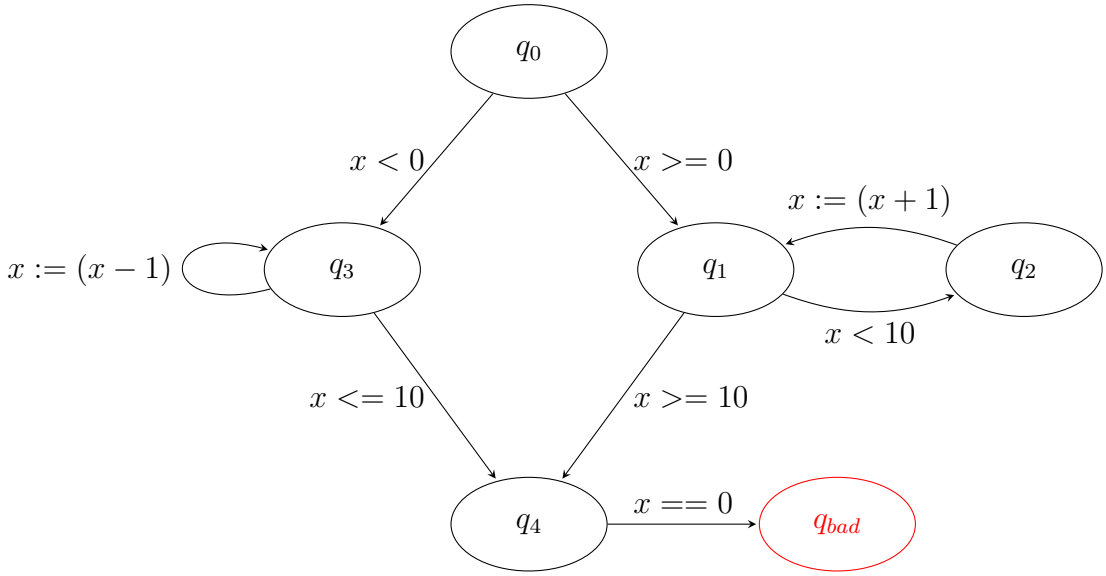


In this test we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = ((x \geq 0) \wedge (y \geq 0)) \vee ((x \leq 0) \wedge (y \leq 0))$
- $\varphi_{q_2} = ((x \geq 0) \wedge (y \geq 0)) \vee ((x \leq 0) \wedge (y \leq 0)) \bigwedge (z = y)$
  $\Leftrightarrow \varphi_{q_2} = ((x \geq 0) \wedge (z \geq 0)) \vee ((x \leq 0) \wedge (z \leq 0)) \bigwedge (z = y)$
- $\varphi_{q_3} = ((x < 0) \wedge (z \leq 0)) \bigwedge (z = y)$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker also finds that there is no feasible path.
PASS

3.2.8 Sign Non Zero Crucial



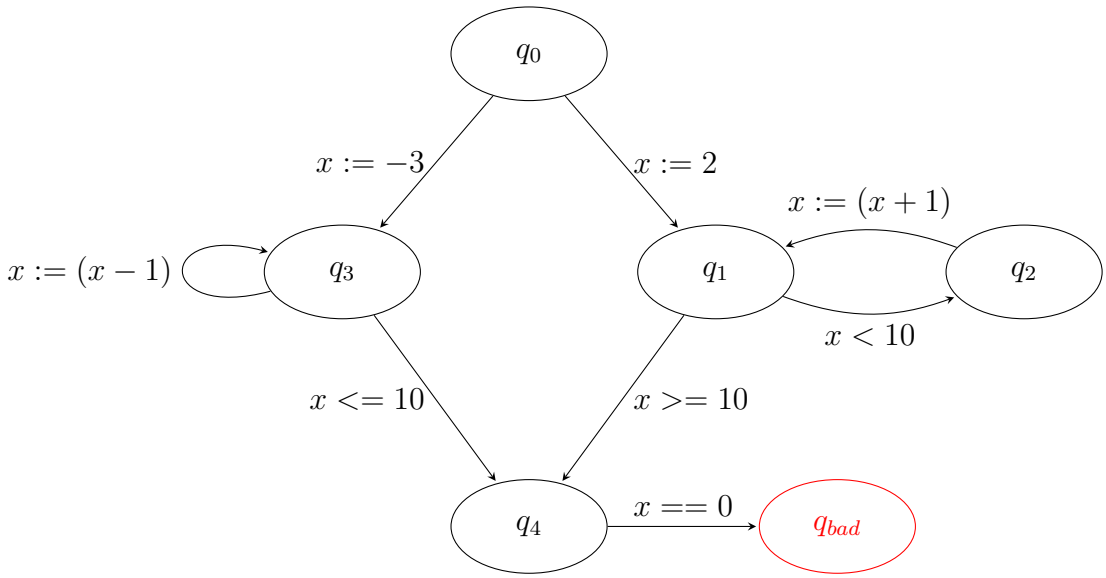
In this test we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$

- $\varphi_{q_1} = (x \geq 0)$
- $\varphi_{q_2} = (x \geq 0) \wedge (x < 10)$
- $\varphi_{q_3} = (x < 0)$
- $\varphi_{q_4} = (x < 0) \vee (x \geq 10)$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker also finds that there is no feasible path of length $\leq$ `bound`.
PASS

3.2.9 Sign Non Zero Crucial Variant



In this test we have a safe inductive invariant:

- $\varphi_{q_0} = \text{true}$
- $\varphi_{q_1} = (x > 0)$
- $\varphi_{q_2} = (x > 0) \wedge (x < 10)$
- $\varphi_{q_3} = (x < 0)$
- $\varphi_{q_4} = (x < 0) \vee (x \geq 10)$
- $\varphi_{q_{bad}} = \text{false}$

The bounded model checker also finds that there is no feasible path of length $\leq$ `bound`.
PASS