(Formal) Software Verification via Logic

(using One-Counter Automata)

Ritam Raha

October 21, 2022

1. Formal Verification

Motivation

```
...

def dummy(x:int,y:int):
    z = 0
    if x>0 & y>0:
        z=x
    return z
...
# end program
```

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

- *Function Coverage*: “dummy” has been called once
- *Statement Coverage*: dummy(1,1)

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

- *Function Coverage*: “dummy” has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

- *Function Coverage*: “dummy” has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

- *Function Coverage*: “dummy” has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

How much testing needed to be done?

- *Function Coverage*: "dummy" has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

How much testing needed to be done?

- *Function Coverage*: "dummy" has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

- What about codes that heavily depend on user inputs or outside inputs?
- How do we know, we are not missing any critical corner cases?

Motivation

```
...

def dummy(x:int,y:int):
    z = 0
    if x>0 & y>0:
        z=x
    return z
...
# end program
```

How much testing needed to be done?

- *Function Coverage*: "dummy" has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

- What about codes that heavily depend on user inputs or outside inputs?
- How do we know, we are not missing any critical corner cases?
- How to prove your software is doing what it is supposed to do and nothing more!

Motivation

```
1 ...
2
3 def dummy(x:int,y:int):
4     z = 0
5     if x>0 & y>0:
6         z=x
7     return z
8 ...
9 # end program
```

How much testing needed to be done?

- *Function Coverage*: "dummy" has been called once
- *Statement Coverage*: dummy(1,1)
- *Branch Coverage*: dummy(0,1), dummy(1,1)
- *Condition Coverage*... etc

- What about codes that heavily depend on user inputs or outside inputs?
- How do we know, we are not missing any critical corner cases?
- How to prove your software is doing what it is supposed to do and nothing more!

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." - E. Dijkstra

What can go wrong?

What can go wrong?

- A bug in the code controlling the *Therac-25 radiation therapy machine* killed five patients.: problem in code (race condition)

What can go wrong?

- A bug in the code controlling the *Therac-25 radiation therapy machine* killed five patients.: problem in code (race condition)

- The software error of a MIM-104 Patriot resulting in failure to intercept an incoming Iraqi Al Hussein missile, killing 28 Americans :
  system clock drifted by one third of a second

What can go wrong?

- A bug in the code controlling the *Therac-25 radiation therapy machine* killed five patients.: problem in code (race condition)

- The software error of a MIM-104 Patriot resulting in failure to intercept an incoming Iraqi Al Hussein missile, killing 28 Americans :
  system clock drifted by one third of a second

- AT&T telephone network outage resulting in 9 hrs of outage of US telephone network: wrong interpretation of break statement in C.

What can go wrong?

- A bug in the code controlling the *Therac-25 radiation therapy machine* killed five patients.: problem in code (race condition)
- The software error of a MIM-104 Patriot resulting in failure to intercept an incoming Iraqi Al Hussein missile, killing 28 Americans :
  system clock drifted by one third of a second
- AT&T telephone network outage resulting in 9 hrs of outage of US telephone network: wrong interpretation of break statement in C.
- A lot more at `https://en.wikipedia.org/wiki/List_of_software_bugs`

Formal Verification

Formal Verification

Formal verification is the process of proving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Formal Verification

Formal Models

- Finite State Machines
- Vector Addition Systems
- Timed Automata/ Hybrid Automata
- Markov Decision Processes

Formal Verification

Formal Models

- Finite State Machines
- Vector Addition Systems
- Timed Automata/ Hybrid Automata
- Markov Decision Processes

Requirements/Formal Specifications

- Reachability
- Safety
- Temporal Logic (LTL, CTL etc.)

Formal Verification

Formal Models

- Finite State Machines
- Vector Addition Systems
- Timed Automata/ Hybrid Automata
- Markov Decision Processes

Requirements/Formal Specifications

- Reachability
- Safety
- Temporal Logic (LTL, CTL etc.)

Advantages of Formal Verification

- Formally proving correctness and ensure safety
- Significantly reduces the verification time

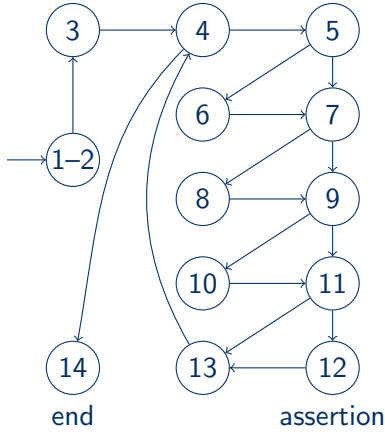
2. One-Counter Automata

Using the control flow graph (CFG)

```
1  skip = 2
2  retake = 3
3  retake += skip
4  while retake >= 0:
5      if retake == 3:
6          print("You get a reminder")
7      if retake == 4:
8          print("You get soft warning")
9      if retake == 5:
10         print("You get hard warning")
11     if retake >= 6:
12         assert("God forbid!")
13     retake -= 1
14 # end program
```

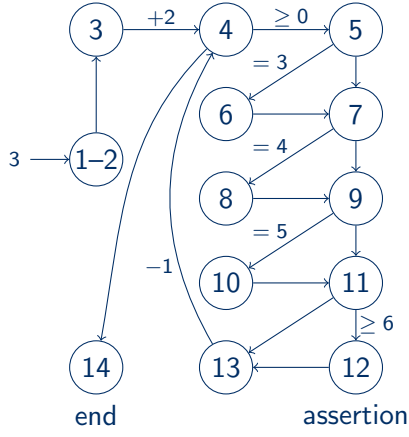
Using the control flow graph (CFG)

```
1  skip = 2
2  retake = 3
3  retake += skip
4  while retake >= 0:
5      if retake == 3:
6          print("You get a reminder")
7      if retake == 4:
8          print("You get soft warning")
9      if retake == 5:
10         print("You get hard warning")
11     if retake >= 6:
12         assert("God forbid!")
13     retake -= 1
14 # end program
```



Extending the CFG with a counter

```
1  skip = 2
2  retake = 3
3  retake += skip
4  while retake >= 0:
5      if retake == 3:
6          print("You get a reminder")
7      if retake == 4:
8          print("You get soft warning")
9      if retake == 5:
10         print("You get hard warning")
11     if retake >= 6:
12         assert("God forbid!")
13     retake -= 1
14 # end program
```



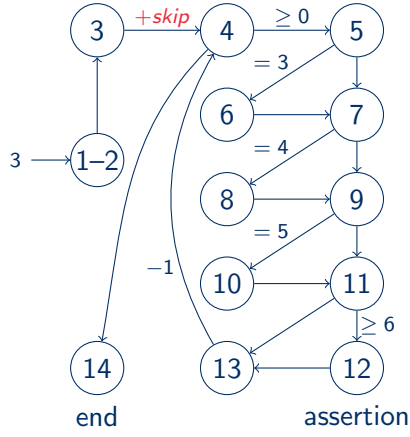
Counter : = retake

Parametric one-counter automata

```
def CanSkip(skip):
    retake = 3
    retake += skip
    while retake >= 0:
        if retake == 3:
            print("You get a reminder")
        if retake == 4:
            print("You get soft warning")
        if retake == 5:
            print("You get hard warning")
        if retake >= 6:
            assert("God forbid!")
        retake -= 1
    # end program
```

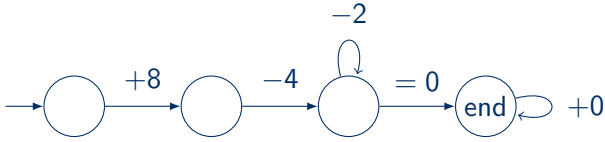
Parametric one-counter automata

```
def CanSkip(skip):
    retake = 3
    retake += skip
    while retake >= 0:
        if retake == 3:
            print("You get a reminder")
        if retake == 4:
            print("You get soft warning")
        if retake == 5:
            print("You get hard warning")
        if retake >= 6:
            assert("God forbid!")
        retake -= 1
    # end program
```

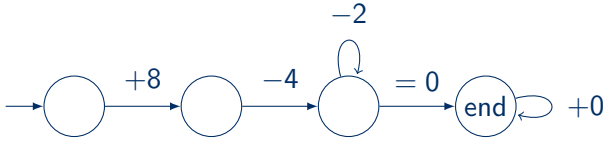


Counter:= retake

(Parametric) One-Counter Automata



(Parametric) One-Counter Automata

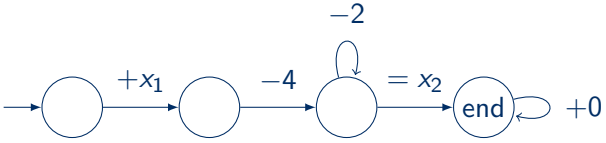


Counter Value has to be non-negative all the time!

(Parametric) One-Counter Automata

Natural-valued parameters

$X = \{x_1, \ldots, x_n\}$



Counter Value has to be non-negative all the time!

Decidability Questions

Definition (Parameter-value Reachability)

Is there some valuation $V : X \to \mathbb{N}$ such that there is some run of $\mathcal{A}$ that reaches/avoids a good/bad state?

Decidability Questions

Definition (Parameter-value Reachability)

Is there some valuation $V : X \to \mathbb{N}$ such that there is some run of $\mathcal{A}$ that reaches/avoids a good/bad state?

Definition (Parameter-value Synthesis)

Is there some valuation $V : X \to \mathbb{N}$ such that all runs of $\mathcal{A}$ reach/avoid a good/bad state? (reach, LTL etc)?

Decidability Questions

Definition (Parameter-value Reachability)

Is there some valuation $V : X \to \mathbb{N}$ such that there is some run of $\mathcal{A}$ that reaches/avoids a good/bad state?

Definition (Parameter-value Synthesis)

Is there some valuation $V : X \to \mathbb{N}$ such that all runs of $\mathcal{A}$ reach/avoid a good/bad state? (reach, LTL etc)?

Non-parametric Versions of the above also

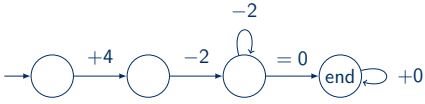
3. Logic

Towards Logic

"Have a problem? Encode it into a logic with decidable theory."

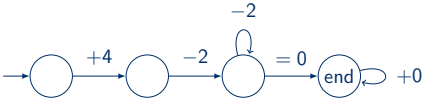
Towards Logic

"Have a problem? Encode it into a logic with decidable theory."



Towards Logic

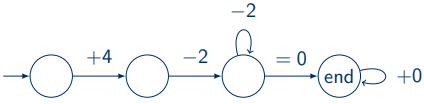
"Have a problem? Encode it into a logic with decidable theory."



Logical formula:
$\exists k(4 - 2 - 2k = 0)$

Towards Logic

"Have a problem? Encode it into a logic with decidable theory."

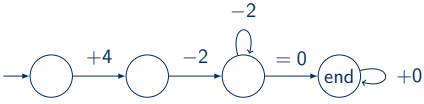


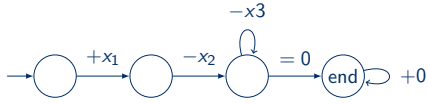
Logical formula:
$\exists k(4 - 2 - 2k = 0)$

Presburger Arithmetic : $\mathrm{FO}(\mathbb{Z}, 0, 1, +, <)$

Towards Logic

"Have a problem? Encode it into a logic with decidable theory."



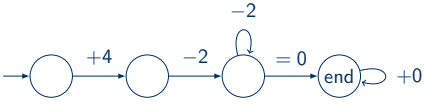


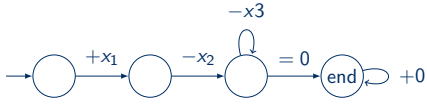
Logical formula:
$\exists k(4 - 2 - 2k = 0)$

Presburger Arithmetic : $\mathrm{FO}(\mathbb{Z}, 0, 1, +, <)$

Towards Logic

"Have a problem? Encode it into a logic with decidable theory."





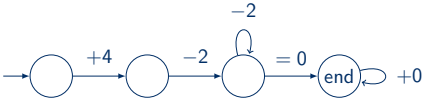
Logical formula:
$\exists k(4 - 2 - 2k = 0)$

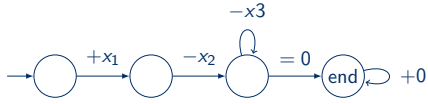
Logical formula:
$\exists x_1, x_2 x_3\ (x_1 \geq 0 \wedge x_1 \geq x_2 \wedge x_3 | x_1 - x_2)$

Presburger Arithmetic : $\mathrm{FO}(\mathbb{Z}, 0, 1, +, <)$

Towards Logic

"Have a problem? Encode it into a logic with decidable theory."





Logical formula:
$\exists k(4 - 2 - 2k = 0)$

Logical formula:
$\exists x_1, x_2 x_3\ (x_1 \geq 0 \wedge x_1 \geq x_2 \wedge x_3 | x_1 - x_2)$

Presburger Arithmetic : $\mathrm{FO}(\mathbb{Z}, 0, 1, +, <)$

Presburger Arithmetic with divisibility:
$PA\ +\ |$
$(a \mid b \iff \exists c \in \mathbb{Z} : b = ac)$

Complexity and Decidability

- Non-parametric Reachability: **NP** (Presburger Arithmetic/PA)
- Non-parametric Synthesis: **coNP** (Reduction complement to Non-parametric Reach)
- Parametric Reachability: **NEXP** (Existential PAD)
- Parametric Synthesis: **N2EXP** (BIL : a fragment of one-alternation PAD)

4. Conclusion

Overview

- Have a system: try to model it formally
- Have requirements in head: try to write it formally (specifications)
- If both of them work, prove/disprove correctness

Overview

- Have a system: try to model it formally
- Have requirements in head: try to write it formally (specifications)
- If both of them work, prove/disprove correctness

Research:

- Continuous One-counter automata, VASS
- Markov Decision Process
- Hybrid Automata