Input-based verification and control-flow inference for machine-code systems

Jan Onderka

Czech Technical University in Prague
Faculty of Information Technology

2023-09-13

Introduction

- Three basic levels of digital system descriptions
  - Source code
  - Machine code
  - Hardware description
- All can be transformed to general automata, but verification techniques and formalisms diverge
- Goals of this presentation
  - Relate the levels via *control flow* in three-valued abstraction
  - Show how *inputs* become important within model-checking when abstraction refinement is used

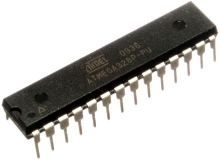
Reminder: Machine code

Source code for processor ATmega328P

```
#include <avr/io.h>
 int main(void) {
     DDRC = 0x07;
     while (1) {
                 uint8_t readval = PIND;
                 uint8_t writeval = ~readval;
                PORTC = writeval & 0x07;
     }
 }
```

is compiled into machine code

```
0C9434000C943E000C943E000C943E00
...
0C943E000C943E000C943E000C943E00
0C943E000C943E0011241FBECFEFD8E0
DEBFCDBF0E9440000C9447000C940000
87E087B989B18095877088B9FBCFF894
FFCF
```



The processor executes the machine code according to its datasheet

Today, we will only use the concept in comparison with other digital systems

Control flow in digital systems

Viewpoint & terminology

- Imperative source code viewpoint
- Hardware descriptions and machine code can be transformed to a virtual-machine source-code program:

```
int main(void) {
    setup();
    while (1) {
        // -> verify here
        perform_one_cycle(); // or instruction
    }
}
```

- Assignment = basic unit of imperative code
- Control flow = order in which assignments are executed (determined by blocks, loops, conditions. . .)
- Trivial control flow = as in the above program where `setup()` and `perform_one_cycle()` are replaced with a sequence of assignments
- Source-code programs can be transformed to trivial control flow by introducing a program counter

Control flow in digital systems

- **Source code**
  - Full control flow corresponding to the programmer's reasoning
- **Machine code**
  - Control flow exists only within execution of one instruction
  - Inference of full control flow is hard
    - disassembly, machine-code translation (Apple Rosetta etc.)...
- **Hardware description**
  - Inherently parallel, control flow does not exist on this level
  - In virtual-machine source code, trivial control flow

Imbalance induced by control flow

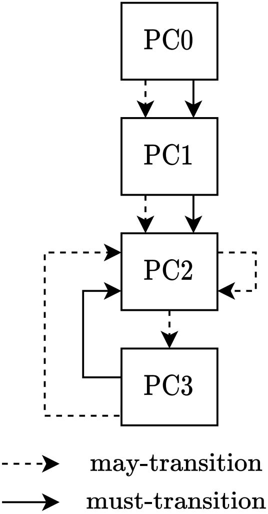
- Imperative source code control flow uses an implicit program counter
    - The program counter is “special” compared to other variables
    - Control-flow-based techniques are closely aligned to it
- But different descriptions have different amounts of control flow. . .
- We may try to balance them by
    - Inferring more control flow
    - Converting to trivial control flow
- Let’s choose a formalism for control flow first
- A “control flow graph” does not tell us how to verify
- Let’s look at things from the point of view of model checking with **three-valued abstraction**
    - Can verify full propositional μ-calculus (including CTL*, CTL, LTL. . .)

Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

- Let's demonstrate verification

Control flow as KTMS

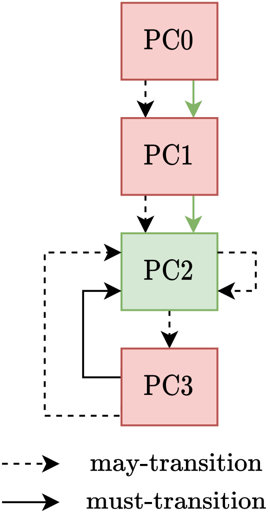


Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

- Let's demonstrate verification
- **EF PC = 2 holds**

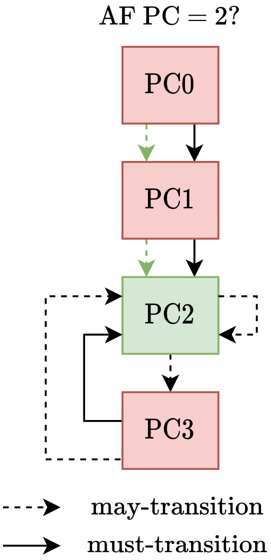
EF PC = 2?



Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

- Let's demonstrate verification
- **EF PC = 2 holds**
- **AF PC = 2 holds**

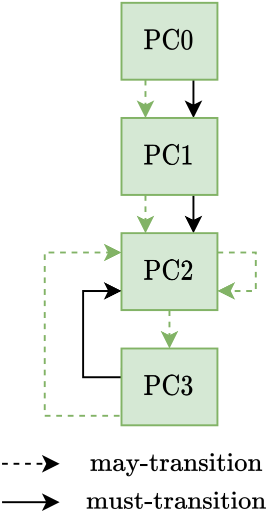


Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

- Let's demonstrate verification
- **EF PC = 2 holds**
- **AF PC = 2 holds**
- **AG PC $\neq$ 4 holds**

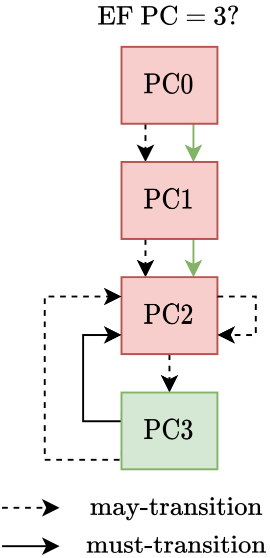
AG PC $\neq$ 4?



Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

- Let's demonstrate verification
- **EF PC = 2 holds**
- **AF PC = 2 holds**
- **AG PC $\neq$ 4 holds**
- **EF PC = 3 unprovable**

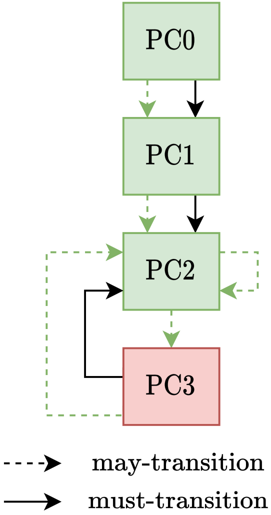


Control flow as a Kripke modal transition structure (KMTS)

```
PC0: bool a = 0;
PC1: bool b = 0;
while(1) {
    PC2: if (input()) {
        PC3: a = 1;
    }
}
PC4: b = 1;
```

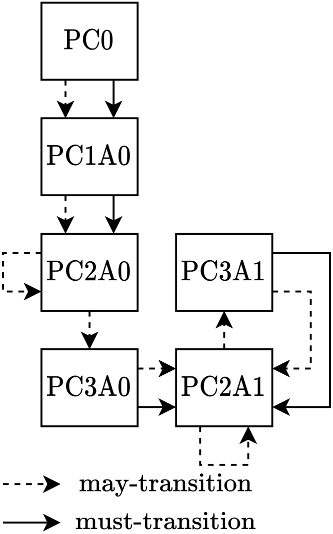
- Let's demonstrate verification
- **EF PC = 2 holds**
- **AF PC = 2 holds**
- **AG PC $\neq$ 4 holds**
- **EF PC = 3 unprovable**
- **AG PC $\neq$ 3 unprovable**

AG PC $\neq$ 3?



Subsuming a variable in control flow

```
PC0: /*bool a = 0;*/ goto PC1A0;
PC1A0: bool b = 0; goto PC2A0;
PC1A1: bool b = 0; goto PC2A1;
while(1) {
    PC2A0: if (input()) {goto PC3A0
        } else {goto PC2A0};
    PC2A1: if (input()) {goto PC3A1
        } else {goto PC2A1};
    PC3A0: /* a = 1; */ goto PC2A1;
    PC3A1: /* a = 1; */ goto PC2A1;
}
PC4A0: b = 1;
PC4A1: b = 1;
```



- We can now verify on values of *a*, e.g.
    - AF $a = 0$ holds
    - AG $(a = 1 \Rightarrow \text{AG } a = 1)$ holds
- However, e.g. AG $a = 0$ is unknown
- We would need to split `input()`

Inferring control flow

- Abstraction refinement can be used to determine variables to subsume
    - Machine-code systems: start with Program Counter register, subsume everything else as needed (branch condition variables, call return addresses. . .)
    - Hardware: no starting point
- Problematic: state space explosion, too complex inferred flow. . .
- If disassembly tools are available for the given platform, it may be best to use them
- **Is this really what we need?**
    - We don't actually care about control flow, but verification
    - Let's try this the other way, getting rid of non-trivial control flow

Trivial control flow & importance of inputs

Trying out trivial control flow

- Non-trivial control flow: branching on abstracted values
    - Corresponds to multiple may-transitions and zero must-transitions in KMTS
    - Leads to a much more compact structure than the state space
- Trivial control flow contains the same may-transitions and must-transitions
    - Variables can be still abstracted away
    - Can be formalized by partial Kripke structures (PKS)
    - Same expressivity as KMTS
- How to generate and refine the PKS?
    - We cannot split states, would lead to different may-transitions and must-transitions
    - We can split inputs, because every input is possible

Abstraction refinement with partial Kripke structure (PKS)

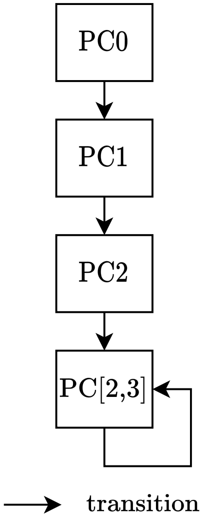
```
PC0: bool a = 1;
PC1: bool b = 1;
while(1) {
    PC2: if (input()) {
        PC3: a = 0;
    }
}
PC4: b = 0;
```

Note: the branch would be actually replaced

with branchless equivalent

- Starting with minimum precision

Control flow as PKS



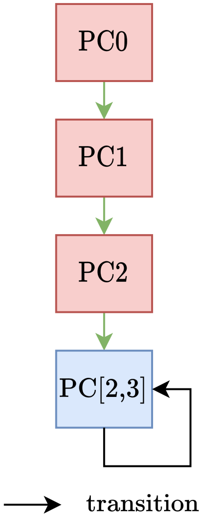
Abstraction refinement with partial Kripke structure (PKS)

```
PC0: bool a = 1;
PC1: bool b = 1;
while(1) {
    PC2: if (input()) {
        PC3: a = 0;
    }
}
PC4: b = 0;
```

Note: the branch would be actually replaced with branchless equivalent

- Starting with minimum precision
- **EF PC = 3 unknown**
- Caused by PC[2,3]
- PC[2,3] caused by PC2 `input()`
- Let's refine...

$\mathbf{EF\ PC = 3?}$

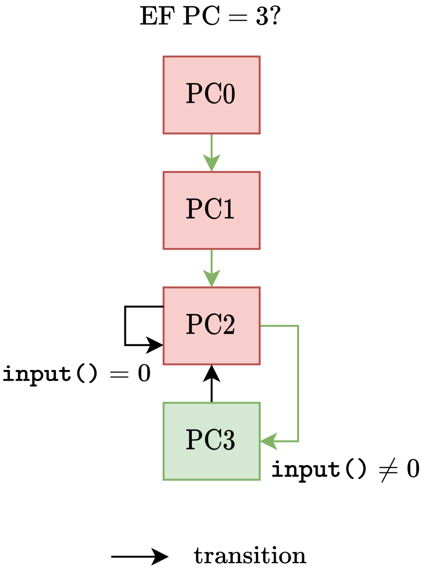


Abstraction refinement with partial Kripke structure (PKS)

```
PC0: bool a = 1;
PC1: bool b = 1;
while(1) {
    PC2: if (input()) {
        PC3: a = 0;
    }
}
PC4: b = 0;
```

Note: the branch would be actually replaced with branchless equivalent

- Starting with minimum precision
- EF PC = 3 unknown
- Caused by PC[2,3]
- PC[2,3] caused by PC2 `input()`
- **EF PC = 3 holds after refinement**



Input-based verification using model checking

- Model checking: formalisms based on Kripke structures
- Unlike general automatons, Kripke structures do not feature inputs
- **Lack of inputs is fine for model-checking, but not abstraction refinement**
- We can sidestep the problem while maintaining compatibility
  - Express the system as an automaton, model-check without inputs
  - Find the state-input combination to be refined in the automaton
- Advantages of input-based three-valued abstraction refinement:
  - Simple algorithms, small soundness-critical core
  - All digital systems and propos. μ-calculus properties can be verified
  - **Good behaviour in machine-code systems**
    - Many unused or unimportant registers
    - Mixing bitwise and arithmetic operations
- Disadvantages
  - Control-flow techniques cannot be used
  - Constructs like countdown loops unroll to large state spaces
- Subject of an upcoming paper

Conclusion

- **Digital system descriptions differ in the amount of control flow**
- Inferring control flow for machine-code systems is problematic
- Proceeding with trivial control flow poses fewer dangers of exponential explosion, and offers simple formalization capable of verifying μ-calculus properties
- Conventional model-checking formalisms do not feature inputs, which poses problems for refinement
- **Is the formalism you use simple and expressive enough?**
  - In source-code verification, the abstract state space and control flow can share the formalism
  - Not considering inputs in the formalism may lead to non-correspondence of theory and tool implementations

Bonus: input-based verification scheme