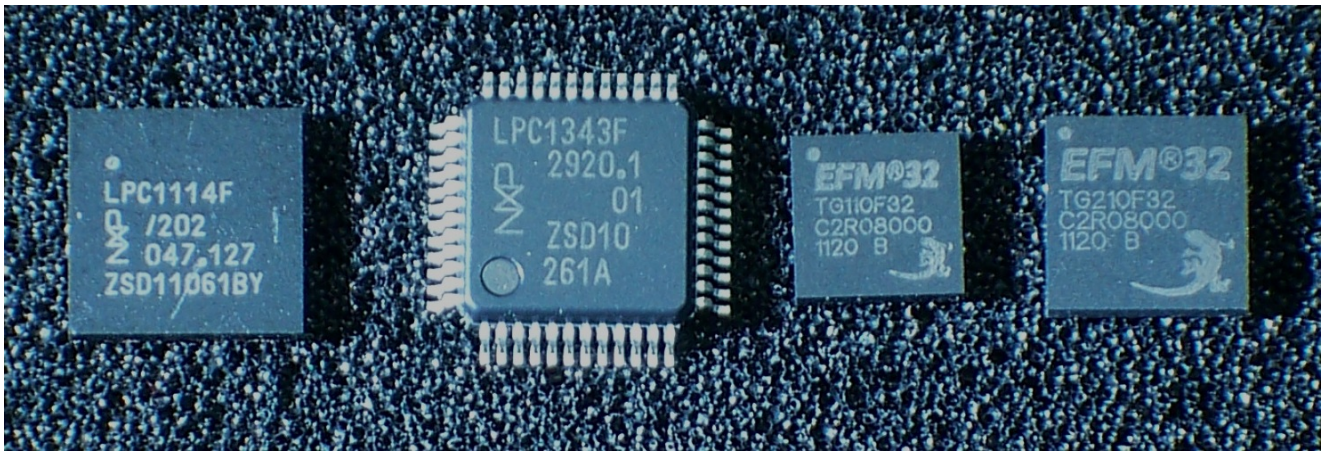
Formal Verification of Machine-Code Systems by Translation of Simulable Descriptions

Jan Onderka

Czech Technical University in Prague
Faculty of Information Technology

2024-06-11

Digital systems executing machine-code programs



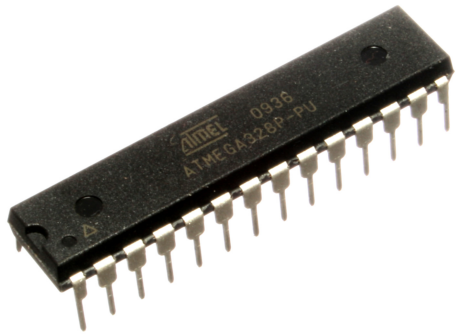
Example: Source code vs machine code

Source code for ATmega328P microcontroller

```
1 #include <avr/io.h>
2 int main(void) {
3   DDRC = 0x07;
4   while (1) {
5     uint8_t readval = PIND;
6     uint8_t writeval = ~readval;
7     PORTC = writeval & 0x07;
8   }
9 }
```

is compiled into machine code

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



which is loaded into the microcontroller and executed after each reset

Verification

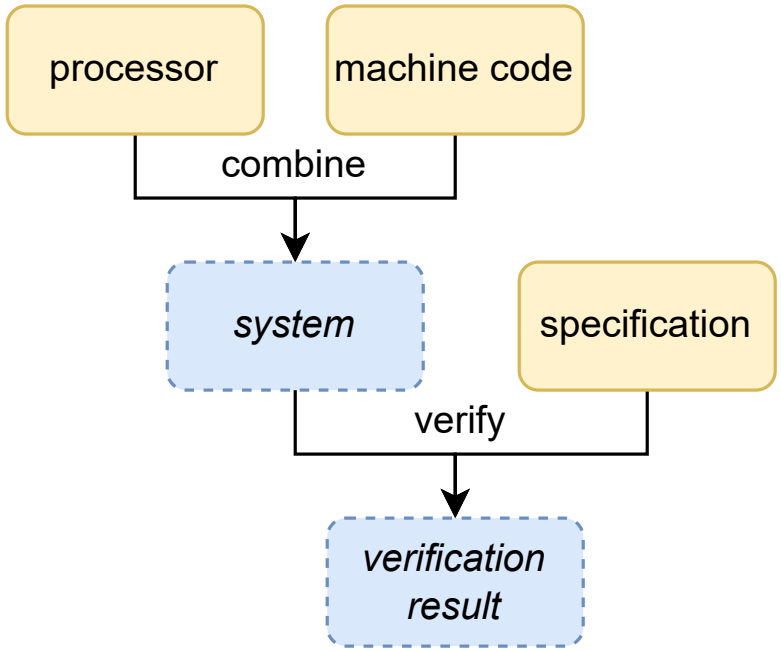
- Informal × **formal**
  - Informal verification can find bugs
  - **Formal verification guarantees correctness**
- Digital systems
  - Hardware
  - **Software**
    - ★ **Machine code**
    - ★ Source code
- Many tools for hardware and source-code verification, none publicly available for machine-code verification
- I have developed the formal verification tool **machine-check**
  - the first publicly available, free and open-source tool that can formally verify **machine-code systems**
  - can also verify other digital systems

What can we verify in machine-code systems that we cannot in source-code systems?

- Proper use of peripherals
  - Are you communicating correctly with your temperature sensor?
- Mitigation of single point of failure in compilation
  - CompCert compiler: dozens of bugfixes
- Real-time systems: best-case and worst-case timings
  - Audio processing takes too long $\Longrightarrow$ your Hi-Fi audio drops out
- Detection of triggering known hardware bugs
- ...

Why was formal machine-code verification not available?

- Some abortive research attempts at machine-code verification
- Two main problems that interact with each other:
  - Standard formal verification problem: many system states
  - Unique to machine-code systems: how to describe processors



How **machine-check** solves the problems

- Main state-of-the-art technique for tackling many system states: *model checking with abstraction refinement*
- We can write a simulator of the processor if we have the documentation
  - Practically infeasible to introduce abstraction refinement manually
- **Novel approach**: describe the processor as a simulable finite-state machine and use meta-programming to introduce abstraction refinement
  - i.e. take the simulable processor description code and transform it to its abstract and refinement analogues
- **Machine-check** uses macros in the Rust programming language to perform the meta-programming
- **Write the description for simulation, get formal verification capability for free**

Simplified RISC processor example 1/3: data structures

- Program counter, four 8-bit registers, 256 bytes of data memory, 128 bytes of program memory

```
#[machine_check::machine_description]
mod machine_module {
  pub struct Input {
    gpio_read: BitvectorArray<4, 8>,
    uninit_reg: BitvectorArray<2, 8>,
    uninit_data: BitvectorArray<8, 8>,
  }
  impl ::machine_check::Input for Input {}
  pub struct State {
    pc: Bitvector<7>,
    reg: BitvectorArray<2, 8>,
    data: BitvectorArray<8, 8>,
  }
  impl ::machine_check::State for State {}
  pub struct System {
    pub progmem: BitvectorArray<7, 12>,
  }
(...)
}
```

Simplified RISC processor example 2/3: init state

- Initialize program counter to 0, registers and data cells are uninitialized

```
1 fn init(&self, input: &Input) -> State {
2   State {
3     pc: Bitvector::<7>::new(0),
4     reg: Clone::clone(&input.uninit_reg),
5     data: Clone::clone(&input.uninit_data),
6   }
7 }
```

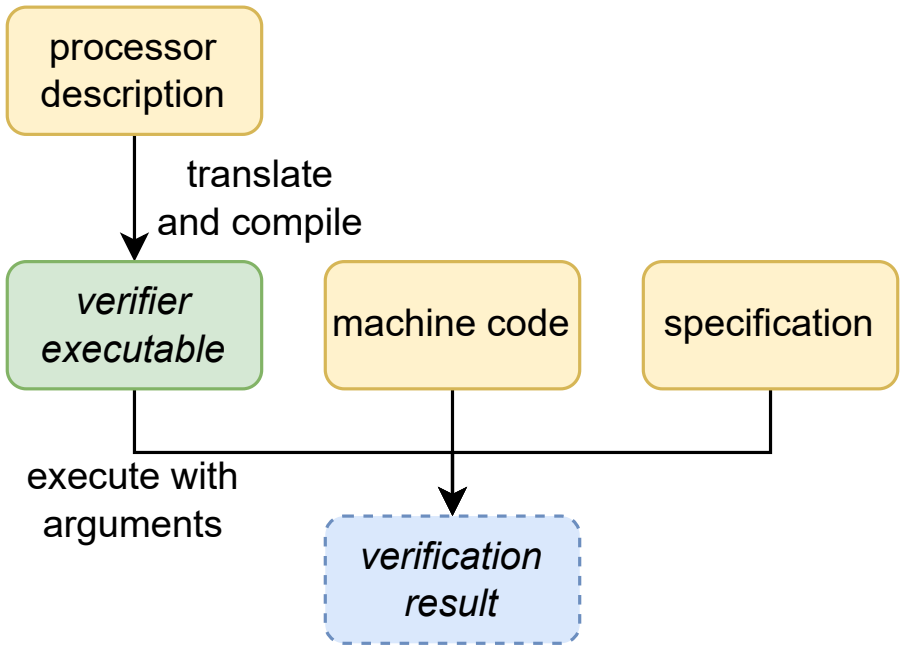
Simplified RISC processor example 3/3: next state

- Fetch instruction, increment program counter, execute

```
1  fn next(&self, state: &State, input: &Input) -> State {
2    let instruction = self.progmem[state.pc];
3    let mut pc = state.pc + Bitvector::<7>::new(1);
4    let mut reg = Clone::clone(&state.reg);
5    let mut data = Clone::clone(&state.data);
6    ::machine_check::bitmask_switch!(instruction {
7      "00dd_00__aabb" => { // add
8        reg[d] = reg[a] + reg[b];
9      }
10     "00dd_01__gggg" => { // read input
11       reg[d] = input.gpio_read[g];
12     }
13     "00rr_1kkk_kkkk" => { // jump if bit 0 is set
14       if reg[r] & Bitvector::<8>::new(1)
15         == Bitvector::<8>::new(1) {
16         pc = k;
17       };
18     } (...) // other instructions skipped for conciseness
19   });
20   State { pc, reg, data }
21 }
```

Putting it together in **machine-check**

- The processor description is compiled together with verification algorithms, translation to verification equivalents occurs during compilation
- Typically, the executable receives the machine code file and a Computation Tree Logic specification from the command line
- **No need for the description writer to know about advanced formal verification techniques**



Conclusion

- Novel technique of translation of simulable descriptions resolves previous problems with formal machine-code verification
- Already implemented in my Rust tool **machine-check**[1]
  - Not stable and ready for serious use yet
  - Initial stable version planned later this year
  - Finishing my dissertation, further development subject to funding
- For truly safe and secure systems, we should aim to close the blind spot in machine-code verification

[1] The version discussed here is available at `https://crates.io/crates/machine-check/0.2.0`, development on GitHub