Deadline Verification Using Model Checking

Author: Ing. Jan Onderka
Supervisor: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Motivation

- Fulfillment of actions within set deadlines is **crucial** for real-time embedded systems
- Non-critical systems are mostly verified using non-exhaustive testing
- Formal verification possible, **always** detects noncompliance
- Using **machine code** for verification is ideal for microcontroller-based embedded systems due to extensive peripheral interaction
- **Latency guarantees** are also possible in machine code through known clock cycles per instruction
- Current tools for formal verification on machine code basis are **severely limited**, prompting me to design a new one

Tool design philosophy

- Formal verification through model checking, in which the **space of possible microcontroller states** is built and processed; this also aids visualisation
- Support for multiple microcontrollers and future extensibility through usage of a custom, **newly designed processor description language**
- Focus on **generality** and orthogonality of advanced techniques used to prevent state space explosion that occurs in naïve model checkers

```
Uint8 R[32]; // General Purpose Registers
(...)
void step() {
    Uint16 instruction = fetchInstruction(PC);
    PC = PC + 1;
    (...)
    masked_case (instruction) {
    (...)
        // AND
        "0010_00rd_dddd_rrrr":
                // logical and
                R[d] = R[d] & R[r];
                set_status_logical(R[d]);
(...)
        }
}
```

Figure: A short excerpt of AVR ATmega328P described in the custom description language. The tool uses the processor description and a program in machine code to build the state space. This eliminates the shortcoming of current machine code verification tools in which the addition of a new microcontroller is highly complex and time-consuming.

Implementation considerations and advanced techniques

- Each bit of state is represented by value ‘0’, ‘1’, ‘X’, or ‘U’
- Value ‘X’ means that both 0 and 1 are possible, **useful for inputs**
- Value ‘U’ means the value is undefined and manipulating it results in an error, useful for memory with unknown initial state
- Aggressive propagation of ‘X’ values through bit operations by default, keeps state space size reasonable, but results in **possible spurious counterexamples**
- A **novel** *value propagation* technique is introduced and implemented, reducing the amount of spurious counterexamples and **vastly increasing usefulness**
- *Path reduction* is also implemented, reducing the amount of states in programs with busy delays

Results

- Simple action-reaction programs have been **successfully verified**
- More complex programs are still problematic due to state space increase
- Fulfillment of actions within set deadlines can be guaranteed with **high precision**
- State space graphs, counterexamples, and worst-case rule satisfaction paths can be exported and **visualised**
- The introduced *value propagation* technique was crucial for checking usefulness and **shows great promise** for future improvement

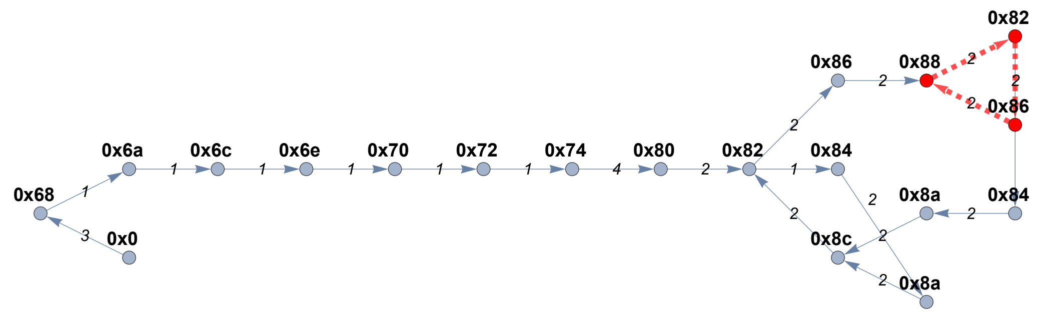

Figure: Visualised state space of a simple program for AVR ATmega328P that sets the value of an output pin according to the value of an input pin. The tool found a counterexample to a rule stating that the value of the output pin must be set to 0 within 30 processor cycles after being set to 1. The counterexample path forms a cycle, visualised in the graph in red with dashed lines. Program counter values are shown adjacent to the states (vertices), cycle counts elapsed between states are shown on the edges.