Hardware Support for Efficient Testing of Embedded Software

Thomas Reinbacher[1], Andreas Steininger[1], Tobias Müller[2]
Martin Horauer[2], Jörg Brauer[3] and Stefan Kowalewski[3]

[1] Institute of Computer Engineering
Vienna University of Technology, Austria

[2] Department of Embedded Systems
University of Applied Sciences Technikum Wien, Austria

[3] Embedded Software Laboratory
RWTH Aachen University, Germany

Outline

Motivation

"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."

[Henk Barendregt]

- Explosion of the Ariane 5 launcher on its maiden flight (1996)
- Loss of the NASA Mars Climate Orbiter (1999)
- US-Northeast blackout (2003)
- Toyota Prius software causes stopping and stalling on highways (2005)
- Microsoft Excel multiplication bug (2007)
- ÖBB train ticketing machine selling single fare tickets for 3720.8 € (2008)
- A1 mobile network breakdown (2011-08-21) due to a SW bug

"Are we building the product right?"
Does the embedded systems software conform to its specification?

Dynamic Verification vs. Formal SW Verification

Dynamic Verification (Testing)

. . . finds cases where code does **not meet its specification**

- can never be exhaustive & may miss errors
- generate and run test-cases is labor and time intensive

Formal SW Verification (Model Checking, Abstract Interpretation)

. . . **formally proving** that the program satisfies its specification

- suffers from scalability issues (e.g., state-explosion problem)
- scalability is often traded for precision
- is exhaustive, at best a "push-button" solution

A trivial Example (bug can be found by model-checking)

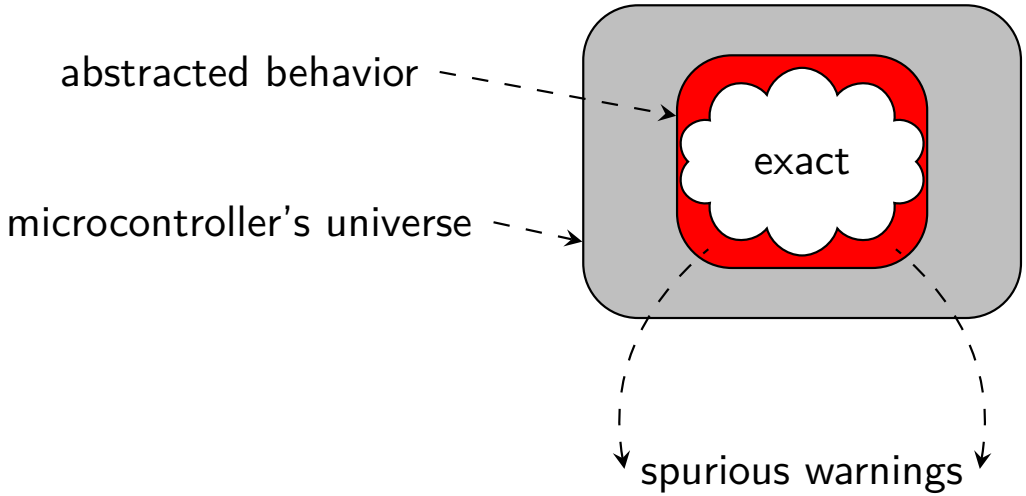
```
#include <avr/io.h>
#include <avr/interrupt.h>

UINT16 event;

ISR (TIMER0_OVF_vect)
{
  if (event == 0x00)
    /* ... */
}

void main(void)
{
  event = 0x01;

  init();
  while(1)
  {
    if (pulse_from_sensor() == 0x01)
      event++;
    if (event == 0x0100)
      event = 0x01;
  }
}
```

Problems and Challenges in Model Checking

- **State-explosion problem** (execution time, memory consumption)
  - Abstraction Techniques (Dead Variable Reduction, Delayed Nondeterminism, Nondeterministic Program Status Word, Lazy Interrupt/Stack Evaluation, Path Reduction, etc.)
  - Static Analyses (Control Flow Analysis, Stack Analysis, Reaching Definitions Analysis, Interrupt Flag Analysis, Live Variable Analysis, Dead Variable Reduction, Path Reduction, Register Bank Analysis etc.)
- Validity of the **system model**
  - [MC]SQUARE: Verification of the MCU simulators
- Specification of **system properties**
  - GUI to guide the creation of CTL / CTL* formulas
- **False Negatives** – invalid counterexamples

  ⇝ CounterExample Validation and Test Case Generation Framework (CevTes)

Abstract Interpretation for Test-Case Generation



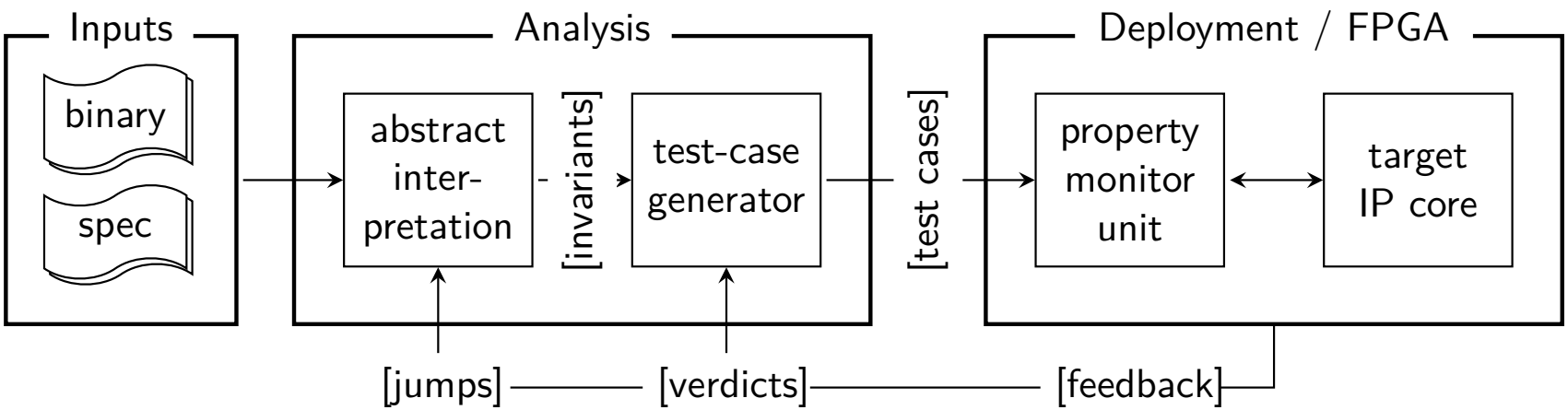
- AI computes an over-approximation of the exact behavior
- thus, found bugs may be spurious
- how to separate real bug reports from spurious ones?

Our goals:

- derive real counterexample traces for binary programs using Abstract Interpretation
- verify counterexamples by a dedicated hardware unit

CevTes Approach

1. use AI to derive an over-approximation of the reachable states
2. find program locations where the specification is violated
3. backward analysis derives counterexamples (test-cases)
4. interface a hardware unit attached to the SUT to replay a CE and automatically identify spurious warnings



Specification Language

. . . Assertions figure strongly in Microsoft code. A recent count discovered more than a quarter million of them in the code for its Office product suite; (C.A.R. Hoare 2003) . . .

Our industrial case study showed that the full expressive power of temporal logics is often not understood/needed by test engineers.

- local assertions $\mathcal{A}(pc, \varphi)$ hold on certain program locations
- global invariants $\mathcal{I}(\varphi)$ hold on every program location

Properties φ are a form of two-variable-per-inequality constraints:

$$\alpha \cdot m_1 + \beta \cdot m_2 \bowtie C$$

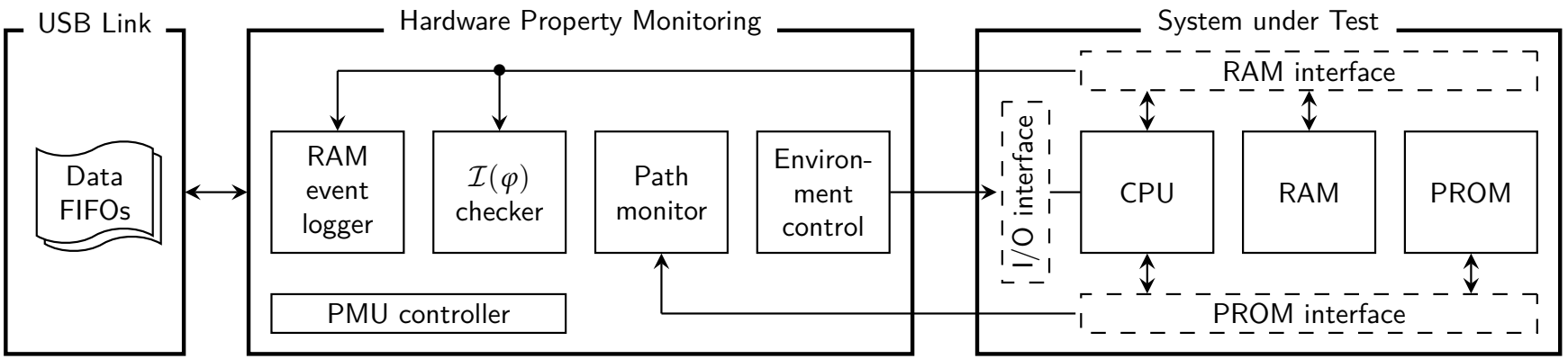
$\alpha, \beta \in \{0, \pm 2^n \mid n \in \mathbb{N}\}$, m_1, m_2 are locations within RAM,
$\bowtie \in \{<, >, \leq, \geq, =, \neq\}$, and $C \in \mathbb{Z}$ is a constant.

Property Monitoring Unit (PMU)

Input

- dedicated, simple specification language
- assertions derived from the high-level representation of the program

$\rightsquigarrow$ test suite Γ with a finite number of test cases n



1. initialization phase: loading the program image, loading the test case, setup of the property checker, invocation of the MCU
2. execution phase: RAM event logger $\delta = \langle \Delta, @, pc \rangle$, invariant checker, path monitor (execution == CFG path ?)

Property Monitoring Unit

Online Monitoring

- global invariants are monitored on-the-fly

Offline Monitoring

elaborate properties and local assertions are checked at the host $\rightsquigarrow$ PMU transfers RAM updates δ in temporal order

$\rightsquigarrow$ the PMU executes all n test cases T and reports

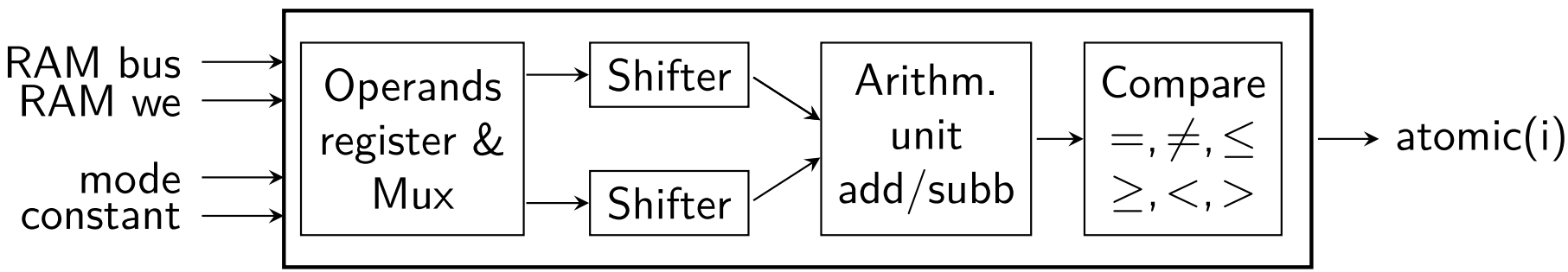
- spurious (property could not be affirmed)
- violation (global invariant failed)
- infeasible (test case left the intended control flow)
- feasible

Runtime Feedback

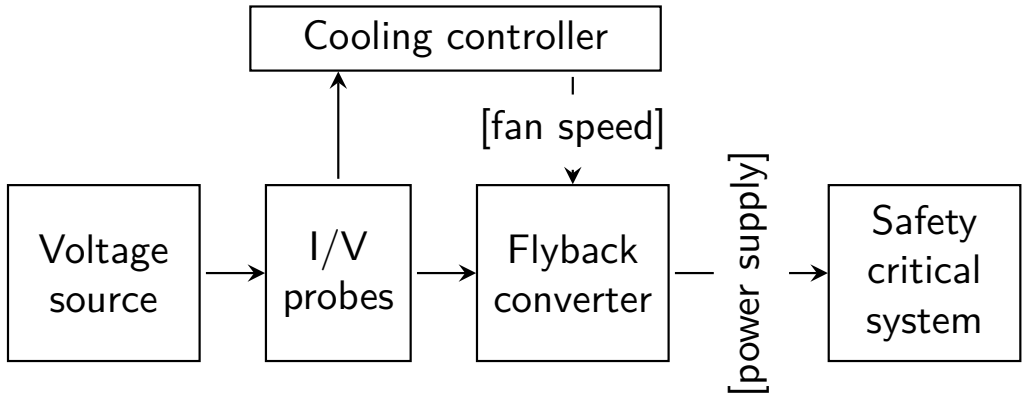
abstract interpretation starts with an incomplete control flow graph (CFG) $\rightsquigarrow$ "path monitor" deviation + iJMP $\Rightarrow$ add new edges to CFG

Invariant Checker: Hardware

- ripple carry adder: $\mathbf{Add}(\langle a \rangle, \langle b \rangle, c)$
- subtraction of $\langle a \rangle - \langle b \rangle$ is equivalent to $\mathbf{Add}(\langle a \rangle, \langle \overline{b} \rangle, 1)$
- relational operators are similar



Example - Cooling Control of a DC/DC Converter

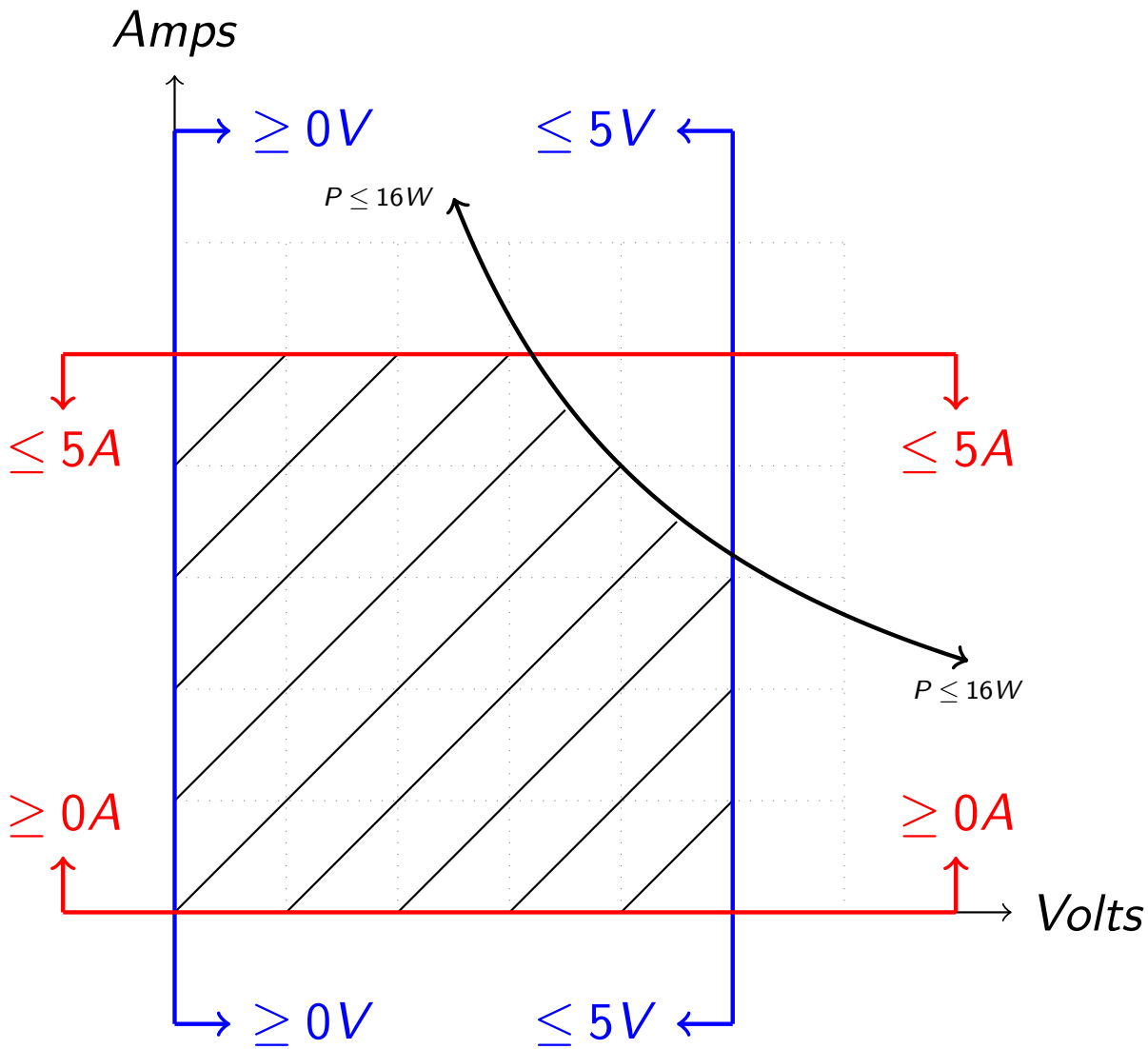


- Specification:

$$\begin{array}{lccccc} \text{Req1:} & 0A & \leq & I & \leq & 5A \\ \text{Req2:} & 0V & \leq & V & \leq & 5V \\ \text{Req3:} & 0W & \leq & V \times I & \leq & 16W \end{array}$$

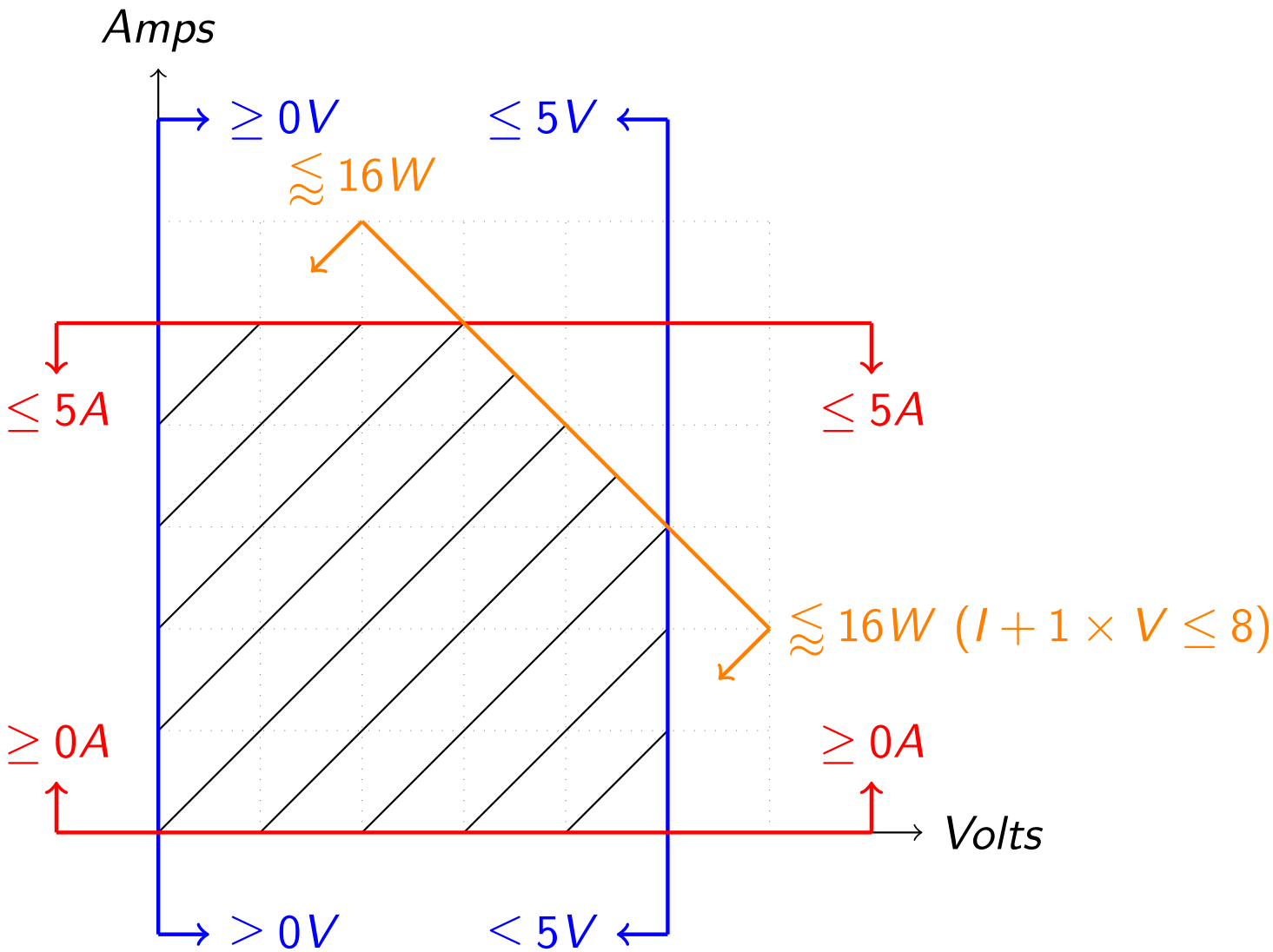
- Assume:

  Analysis found a property violation of Req3 and derived a test case. But how to verify it is a real bug? How to replay the test case on the SUT?


Amps
$\geq 0V$
$\leq 5V$
$P \leq 16W$
$\leq 5A$
$\leq 5A$
$P \leq 16W$
$\geq 0A$
$\geq 0A$
Volts
$\geq 0V$
$\leq 5V$

What we can Monitor in Hardware

$\pm 2^n \cdot r_i \pm 2^m \cdot r_j \leq C$



Conclusion

"Formal verification approaches in combination with testing may pave the way for exhaustive tests of embedded systems software."

- combine formal verification strategies with testing and online monitoring
- automatically rule out spurious warnings / test cases
- property checking
  - online (on-the-fly) while running the test case on the target hardware (property monitoring unit)
  - offline (on a host computer)

Future work:

- combine with run-time verification approaches (e.g., past time LTL)
- extend flexibility of online checking, allow for more complex properties