Past Time LTL Runtime Verification for Microcontroller Binary Code

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

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

[2] Embedded Software Laboratory
RWTH Aachen University, Germany

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

Outline

1. Requirements for Practical Applicability
2. Past Time LTL
3. Non-Intrusive Monitoring
    1. Online: Observer synthesized from VHDL
    2. Offline: Java application on host computer
4. Example

1. Embedded software mostly not in plain ANSI C
   - side effects, embedded assembler, direct hardware register access
2. Who verified your compiler?
   - GCC 4.3.5 has 8M loc (2.5M C, 1.5M C++, 1.5M Java, 60k Assembler, ...)
   - Good SW has about 1 error in 250 loc $\rightarrow \frac{8M}{250} = 33k$ flaws
   - Proving correctness of the compiler is typically infeasible
   - Even translation validation is hard (and not really widespread in industry)
3. Binary code is as close as possible to the actual execution

RV is less ambitious than pure formal SW verification. Aim is to prove conformance of a single execution w.r.t. a specification.

Testing is based on a check and guess paradigm.

- Guess a configuration of the program's input
- Check the result of an individual test run

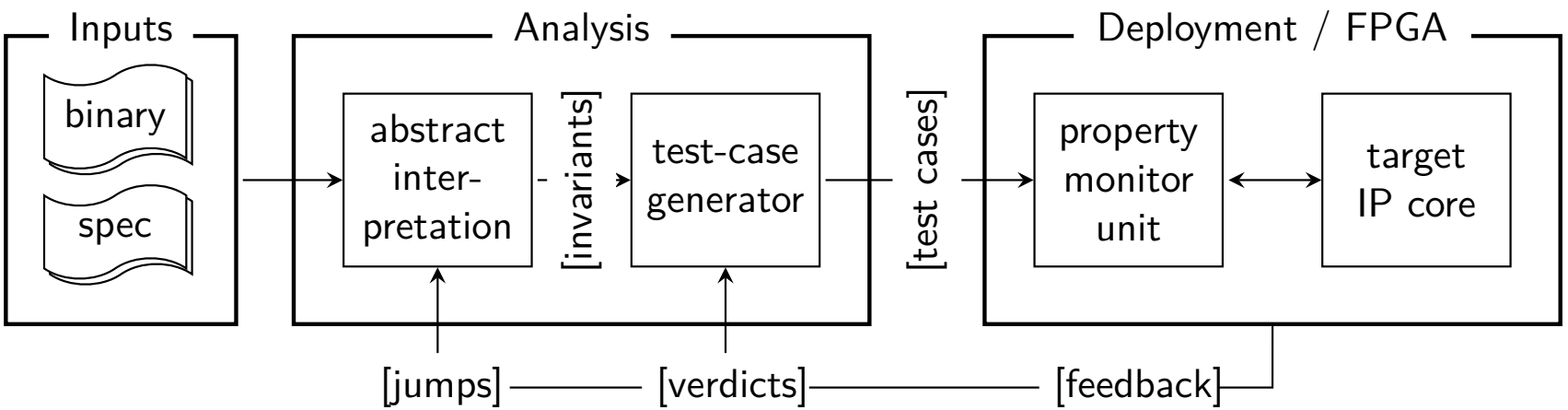
RV bridges the gap between rigorous software verification and dynamic testing.

Intuition:

- Use formal methods to derive a set of test-cases (guess)
- Execute the test cases, use RV to check validity of test-case (check)

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



Requirements to RV of Microcontroller Code

1. Generality
   $\rightsquigarrow$ not bound to a certain high-level language
2. No Code instrumentation
   $\rightsquigarrow$ extract event sequences without code instrumentation
3. Evaluate Atomic Propositions
   $\rightsquigarrow$ expressiveness vs. complexity of evaluation
4. Automated Observer Synthesis
   $\rightsquigarrow$ push button solution
5. Usability
   $\rightsquigarrow$ applicable to industrial SW development process

Systems Computing Embedded

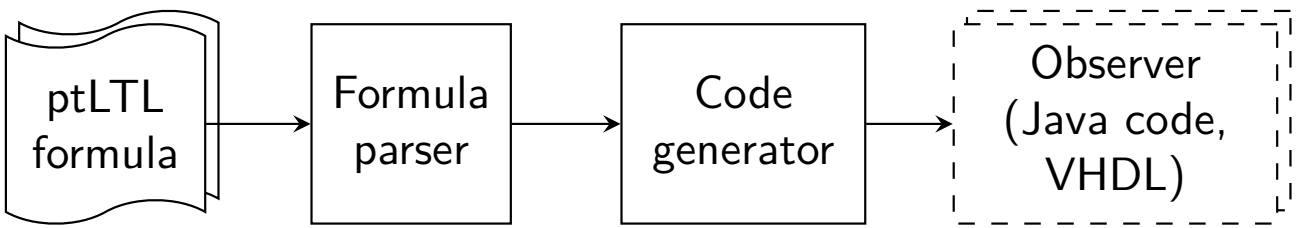
Past time LTL (ptLTL) [Emerson; Handbook of Theoretical CS'90]

$$\psi ::= true \mid false \mid AP \mid \neg\psi \mid \psi \bullet \psi \mid \odot\psi \mid \diamond\psi \mid \boxdot\psi \mid \psi\ S_s\ \psi \mid \psi\ S_w\ \psi$$

Monitoring operators

$$\psi ::= \uparrow\psi \mid \downarrow\psi \mid [\psi,\psi)_s \mid [\psi,\psi)_w$$

Approach



1. Synthesize HW+SW monitors for ptLTL [Havelund&Rosu; TACAS'02]
2. Evaluate Atomic Propositions AP of ψ with our invariant checker

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

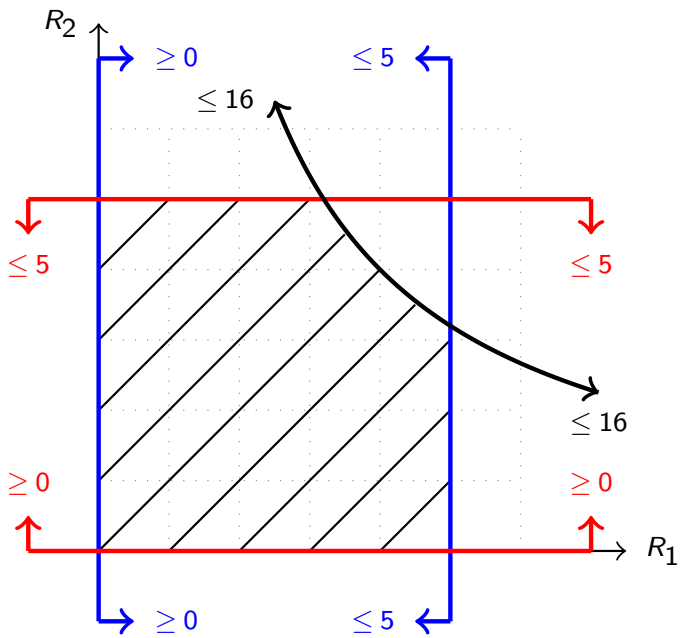
$$\alpha \cdot m_1 + \beta \cdot m_2 \bowtie C$$

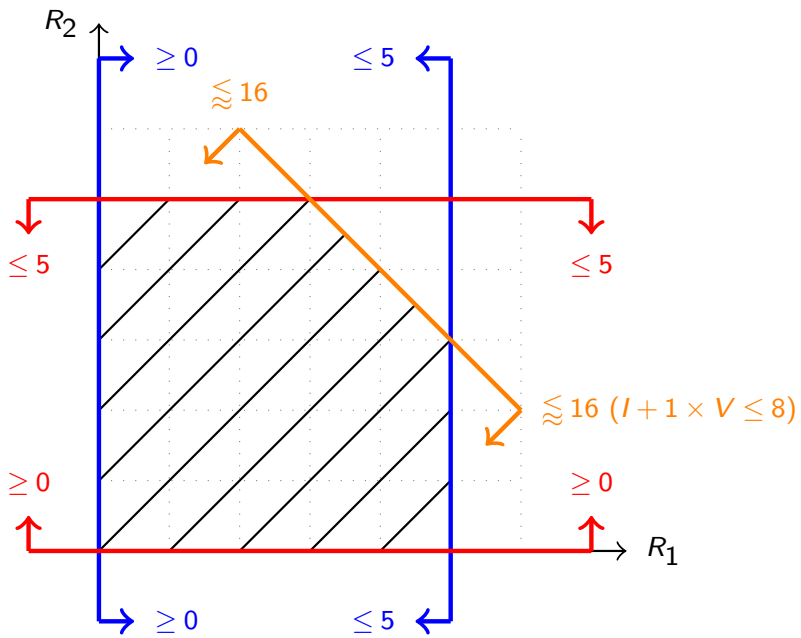
where:

- $\alpha, \beta \in \{0, \pm 2^n \mid n \in \mathbb{N}\}$
- $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$

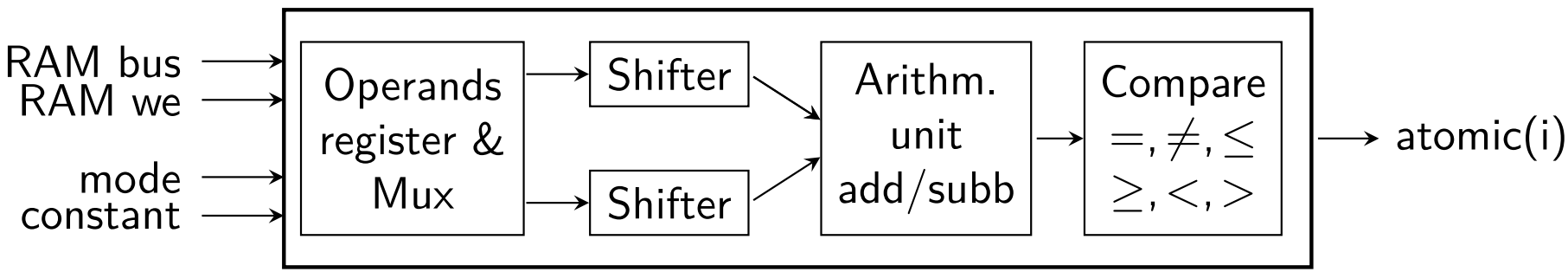
m_1, m_2 are locations within RAM

$C \in \mathbb{Z}$ is a constant



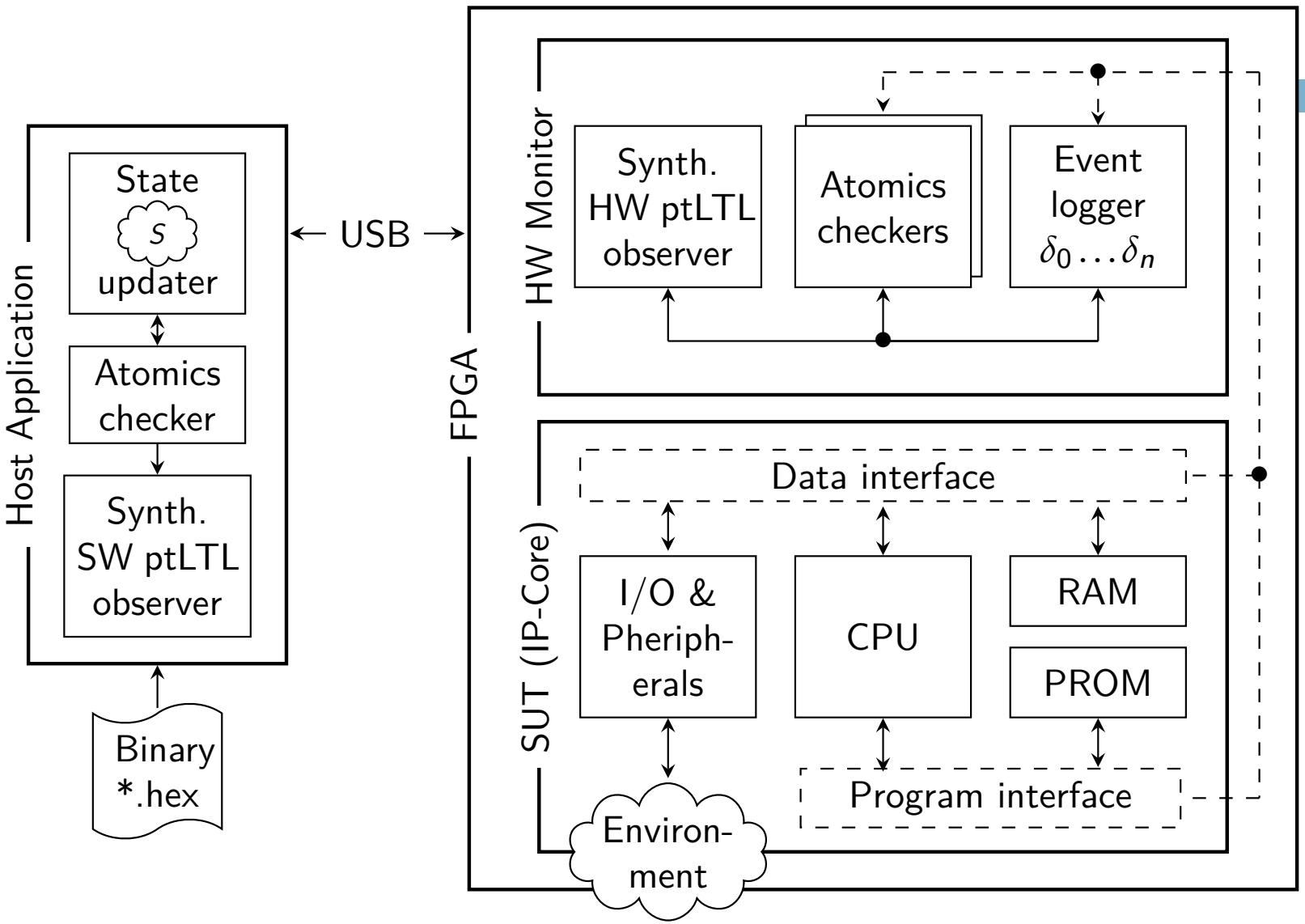


- ripple carry adder: $\mathbf{Add}(\langle a \rangle, \langle b \rangle, c)$
- subtraction of $\langle a \rangle - \langle b \rangle$ is equivalent to $\mathbf{Add}(\langle a \rangle, \langle \overline{b} \rangle, 1)$
- relational operators are similar



System Overview

1. SUT is a COTS microcontroller IP core
2. Invariant checker is attached to the SUT on an FPGA



Offline vs. Online Monitoring

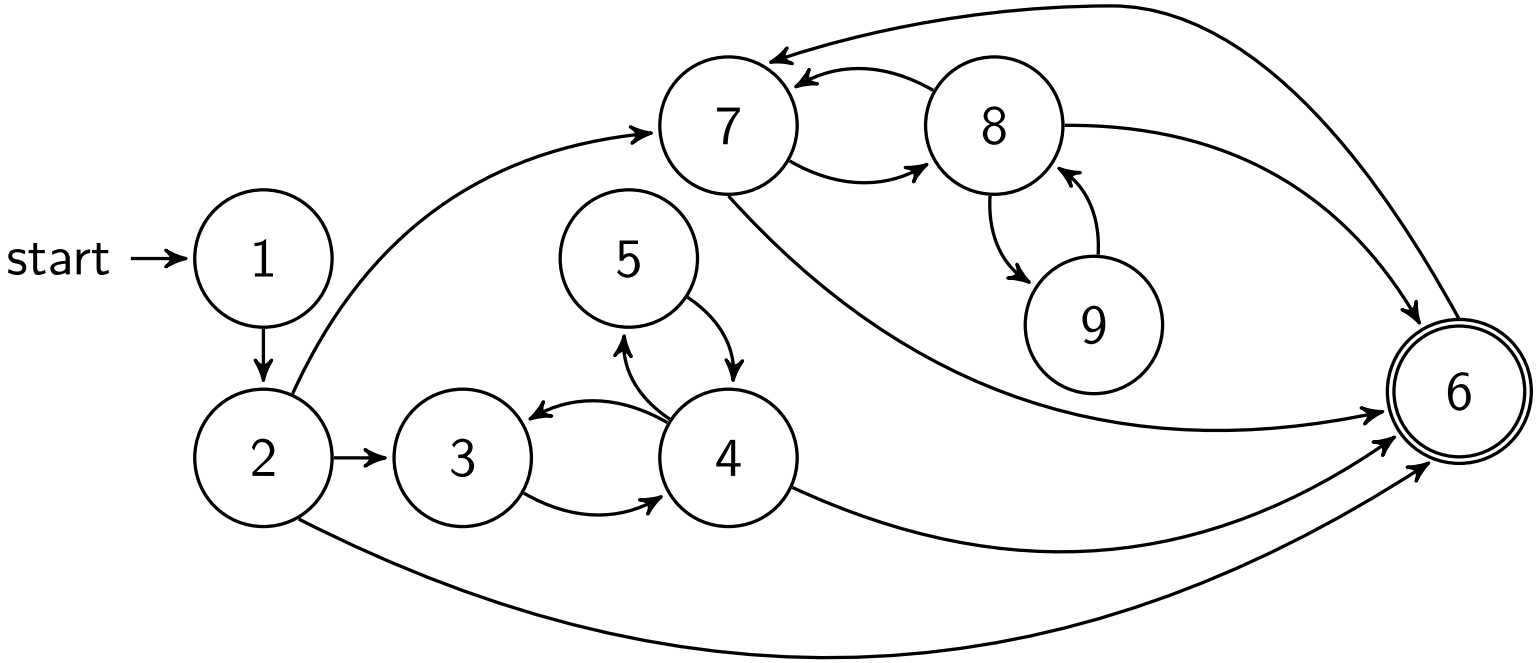
Online mode:

- synthesize VHDL code for ptLTL evaluation
- evaluate specification in parallel to execution

Offline mode:

- send event updates to host computer
- store microcontroller states to evaluate (arbitrary) APs
- evaluate specification on the execution trace (delayed)

Emergency Stop (IEC 61131-3)



ptLTL properties:

$$
\begin{aligned}
\psi^1 &:= \uparrow(\Theta_6) \rightarrow \big[\uparrow(\Theta_2 \vee \Theta_4 \vee \Theta_7 \vee \Theta_8),\ \uparrow(\Theta_1 \vee \Theta_3 \vee \Theta_5 \vee \Theta_9)\big)_S \\
\psi^2 &:= \uparrow(\Theta_5) \rightarrow \downarrow(\Theta_4) \\
\psi^3 &:= \uparrow(\Theta_9) \rightarrow \downarrow(\Theta_8)
\end{aligned}
$$

e.g., $\Theta_8 \triangleq (\mathsf{currState} == \texttt{ST\_WAIT\_FOR\_RESET2})$

Emergency Stop (IEC 61131-3)

```
1 case ST_WAIT_FOR_ESTOPIn2:
2   Ready = true;
3   S_EStopOut = false;
4   Error = false;
5   DiagCode = 0x8004;
6   if (!Activate)
7     currState = ST_IDLE;
8   if (S_EStopIn && !S_AutoReset)
9     currState = ST_WAIT_FOR_RST2;
10  if (S_EStopIn && S_AutoReset)
11    currState = ST_SAFETY_OUTP_EN;
12 break;
```

```
case ST_WAIT_FOR_ESTOPIn2:
  Ready = true;
  S_EStopOut = false;
  Error = false;
  DiagCode = 0x8004;
  if (!Activate)
    currState = ST_IDLE;
  if (S_EStopIn && !S_AutoReset)
    currState = ST_WAIT_FOR_RST2;
  if (S_EStopIn && S_AutoReset)
    currState = ST_RST_ERR2;
break;
```


- The ptLTL observer will check the validity of ψ^1
- The invariant monitor will trace the state changes of the variable `currState`

The transition $ST\_WAIT\_FOR\_ESTOPIn2 \rightarrow ST\_RST\_ERR2$ is illegal and causes ψ^1 to evaluate to false.

Conclusion & Future Work

- Non-intrusive monitoring framework for ptLTL
  - Atomic propositions evaluated in hardware
  - Code for ptLTL monitor synthesized in Java or VHDL
- Monitoring an IP-core running on FPGA
  - Handles unmodified embedded programs
  - on off-the-shelf IP core

- More recent work
  - Reprogrammable μCPU for checking properties (see RV'11)
  - CFG recovery a priori instead of on-the-fly (see EMSOFT'11)
  - ptLTL verification for PLCs (with S. Biallas)
- Orthogonal but related future work: automatic test-case / trace generation