Hardware-based runtime verification with Tessla

Martin Leucker
University of Lübeck

Joint work with Normann Decker, Cesar Sanchez, Torben Scheffel, Malte Schmitz, Daniel Thoma et al.
Runtime Verification

- Monitor analyzes the execution of the system
- Synthesized from high-level specification
- Has to see the execution
- Used for finding bugs
Aging Bugs – Mandelbugs – Heisenbugs

• Some bugs only occur
  • after a long execution time
  • under “weird” circumstances

See what’s going on?
• Program annotation?
• Program annotation changes timing

Change of the underlying system

• Run what you test and test what you run
Code annotation – Program cooperates

<table>
<thead>
<tr>
<th>Original source code</th>
<th>Instrumented source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo()</td>
<td>char inst[15];</td>
</tr>
<tr>
<td>{</td>
<td>void foo()</td>
</tr>
<tr>
<td>bool found=false;</td>
<td>{</td>
</tr>
<tr>
<td>for (int i=0; (i&lt;100) &amp;&amp; (!found); ++i)</td>
<td>bool found=false;</td>
</tr>
<tr>
<td>{</td>
<td>for (int i=0;((i&lt;100)?inst[0]=1:inst[1]=1,0) &amp;&amp;</td>
</tr>
<tr>
<td>if (i==50) break;</td>
<td>((!found)?inst[2]=1:inst[3]=1,0); ++i)</td>
</tr>
<tr>
<td>}</td>
<td>{</td>
</tr>
<tr>
<td>if (i==20) found=true;</td>
<td>if ((i==50?inst[4]=1:inst[5]=1,0))</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>inst[6]=1; break;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>inst[9]=1; found=true;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>inst[12]=1; found=true;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>inst[13]=1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;foo\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>printf(&quot;foo\n&quot;);</td>
<td>printf(&quot;foo\n&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>inst[14]=1;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

- Slowdown typically not acceptable for production code
- Unless done in a clever way? Printf??
- Here: Use additional hardware resources instead
Hardware-based Runtime Verification
SoC approach – Architecture

The Observer Entity is plugged to the platform where the functional system software components execute, and components, enabling the verification in runtime that its properties are being fulfilled and that no design assumption is being violated.

2.2 Observer Entity

The demand for non-intrusive observability justifies, the interest in hardware-based methods, powered by: the usage of reconfigurable logic, supported on FPGA special-purpose observers; the row availability of integrated processing, enabling observer per se, detecting events, storing the dynamically captured all physical bus activity, such as bus transfers or actions does not disturb the execution of the functional system, therefore ensuring the necessary and sufficient mechanisms for the runtime verification of any functional system. TeSSLa reasons over asynchronous input streams by applying functions to already existing streams. A divide and conquer strategy is used in the definition and has already been used to build monitors for Runtime Verification. TeSSLa [5] is a temporal stream-based specification language that allows to provide the necessary and sufficient mechanisms for the runtime verification of any functional system. A set of basic monitors, encompassing essential runtime verification mechanisms for the synthesis of runtime verification mechanisms. A set of

3 An introduction to TeSSLa

A stream can be defined declaratively as can be seen in the following example of a TeSSLa specification:

```
def maximum := max(x1, x2)
def def a>b := a>0 && b<0
```

A stream is only allowed to be defined for a finite number of timestamps, while a signal stream defines a value for every downstream block awaiting for that event. An unique identifier (e.g., identifier (obsID, v, t) = obsID, v, t) is assigned to each observed event, being obsID, the corresponding observed value (e.g., instruction code obs), the address observed from the functional system bus interface that matches a given event specification; v, the variables of interest; t, the attached timestamp.

2.3 System Observing Mechanisms

These monitors can be instantiated as required. Additional monitoring mechanisms for the runtime verification of any functional system.

2.4 Monitoring Mechanisms

An unique event is detected, it is timestamped with the instant of occurrence, and has already been used to build monitors for Runtime Verification. The right combination of these building blocks complement and enlarge the functionality provided by the basic building blocks (selectors, transformers and past-time event registers) actions, in both value and time domains, is detailed in [11].

SoC approach – Architecture

Non-intrusive Runtime Verification within a System-on-Chip, RUME 2018
José Rufino, António Casimiro, Felix Dino Lange, Martin Leucker, Torben Scheffel, Malte Schmitz, Daniel Thoma
The Observer Entity is plugged to the platform where the functional system software components execute, and components execution. Probing the processor-cache interfaces should allow an higher accuracy in the observation of software actions does not disturb the execution of the functional system non-intrusively, meaning execution of runtime verification is completely non-intrusive and can be observation resources [9, 10]. By nature, hardware-based purpose observers [6, 7, 8]; the row availability of integrated usage of reconfigurable logic, supported on FPGA special-interest in hardware-based methods, powered by: the interest in hardware-based methods, powered by: the basic concept of TeSSLa is deriving internal or output signals and event streams. An event stream is only allowed to be defined for a finite number of timestamps in a finite interval, while a signal stream defines a value for every point in time. TeSSLa supports signals and event streams. An event stream should be able to provide the necessary and sufficient mechanisms for the runtime verification of any functional system. TeSSLa reasons over asynchronous input streams by applying functions to already existing streams. TeSSLa [5] is a temporal stream-based specification language which is designed for monitoring real-time signals and has already been used to build monitors for Runtime Verification [12].

2.3 System Observing Mechanisms

These monitors can be instantiated as required. Additional monitors specified in TeSSLa can observe events, that are obtained from the Time Base module, and supplied to eventing or data value); an event composed by the tuple:

\[ \text{obs} = (\text{obsID}, \text{vcod}, \text{tobs}) \]

where:\n
- \( \text{obsID} \) is the address observed from the functional system.
- \( \text{vcod} \) is the address observed from the functional system.
- \( \text{tobs} \) is the attached timestamp.

In the following example of a TeSSLa specification:

\[ \text{max}(a,b) := \begin{cases} a & \text{if } a > b \\ b & \text{else } \end{cases} \]

\[ \text{maximum} := \text{max}(x1, x2) \]

2.4 Monitoring Mechanisms

A set of basic monitors, encompassing essential runtime verification mechanisms for the synthesis of runtime verification mechanisms. A set of monitoring mechanisms for the runtime verification of any functional system. TeSSLa 

3 An introduction to TeSSLa

System Observer Management Interface System Observer Configuration Time Base Monitor Bus Interfaces System Clock System Bus
SoC approach with TeSSLa specifications

C code -> C compiler -> binary

TeSSLa specification -> TeSSLa compiler

analyser -> debug symbols

Leon processor -> data

system observer -> events

hardware platform (FPGA) -> monitor

monitor in Verilog

report

dependency graph

observation points

Volume xx, Number y, May 2018 Ada User Journal
Fixed Monitors for RV

Debugging

Requirements
• Quick loop for synthesizing new properties on the test system
• Still long-term observability useful
• Change monitoring focus dependent on previous outcome
SoC approach with TeSSLa specifications

- Use interpreter of monitor
- Synthesize code/table for interpretation
- Loadable into memory of interpreter

Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, Richard Lasslop:
The COEMS approach
Continuous Online Observation for Embedded Multi-core Systems
EU Horizon 2020 project
The COEMS Consortium

University of Lübeck

Accemic Technologies

Thales Romania

Thales Austria

Høgskulen på Vestlandet / Western Norway University of Applied Science

Airbus
Objectives

- Increase test efficiency
- Increase debug efficiency
- Increase test effectiveness
- Improve embedded systems performance

- For embedded multicore systems (ARM etc.)
- Running (multithreaded) C programs
- Perhaps on Linux
Applications – Semi-Formal Verification

• Finding Data Races
• Finding Timing Bugs
• Finding Functional Bugs
• Measuring Coverage
• Measurement of Worst-Case Execution Time and Worst-Case Response Time
COEMS set-up
System Overview
Rapidly adjustable Embedded Trace Online Monitoring – RETOM
Observation Specification

*Specification of atomic artefacts to be observed*

- Elements of Source Code
  - Program Line
  - Entering / Leaving a Function / Exception
  - Reading / Writing variables

- Elements of the Binary
  - PC Address
  - Calls / Returns
  - Specific Operations
    (e.g. Floating Point Operations)

- Hardware Supported Instrumentation
  - ITM, STM
Observation Specification

C Code

```c
int main() {
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum = add(sum, sub(i, 2));
    }
    sum = add(sum, add(21, 21));
    for (int i = 0; i < 5; i++) {
        sum = add(sum, sub(i, 2));
    }
}

int add(int x, int y) {
    return x+y;
}

int sub(int x, int y) {
    return x-y;
}
```

Specification

```python
def add_event := function_call("add")
def add_count := eventCount(add_event)
def sub_event := function_call("sub")
def sub_count := eventCount(sub_event)
def diff := add_count - sub_count
def error := diff >= 2

def output_diff
    out diff

def output_error
    out error
```
Monitor Specification Language (TeSSLa)

- **Event-Stream-Analysis**
  Event ordering constraints, timing constraints and quantitative analysis

- **Declarative style**
  Describe correctness criterion or analysis goal without having to think about the algorithmic check

- **Modularity**
  Allowing abstractions based on few primitives

- **Time**
  As first-class citizen

- **Events & signals**
  Based on one common abstraction

- **Finite memory**
  Allowing execution on FPGA

**Specification**

```python
def add_event := function_call("add")
def add_count := eventCount(add_event)
def sub_event := function_call("sub")
def sub_count := eventCount(sub_event)
def diff := add_count - sub_count
def error := diff >= 2
out diff
out error
```
Specification Languages for RV
Streams

Concurrency/Distribution
Streams

Time? Synchrony/Ticks
Equational specifications, data, time, concurrency

**LOLA**

[D’Angelo et al.]

\[
\begin{align*}
  s_1 &= \text{true} \\
  s_2 &= t_3 \\
  s_3 &= t_1 \lor (t_3 \leq 1) \\
  s_4 &= ((t_3)^2 + 7) \mod 15 \\
  s_5 &= \text{ite}(s_3, s_4, s_4 + 1) \\
  s_6 &= \text{ite}(t_1, t_3 \leq s_4, \neg s_3) \\
  s_7 &= t_1[+1, \text{false}] \\
  s_8 &= t_1[-1, \text{true}] \\
  s_9 &= s_9[-1, 0] + (t_3 \mod 2) \\
  s_{10} &= t_2 \lor (t_1 \land s_{10}[1, \text{true}])
\end{align*}
\]
Streams

Time triggered systems

Event-triggered

Time? Synchrony/Ticks
TeSSLa’s Streams
Streams

Time? Events
Streams of Programs - After Discretization

Values
e.g., of a program variable \( x \)

Program events
e.g., call to \( \text{my\_func()} \)

\begin{align*}
5 & \quad 6 \\
1 & \quad 2 \\
\hline
\end{align*}
Streams

Values
e.g., of a program variable $x$

| 5 | 6 | 1 | 2 |

Program events
e.g., call to `my_func()`
Streams here for RV

Observations (Input streams)

Derived streams (definable)

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value x</td>
</tr>
<tr>
<td>Event irq4</td>
</tr>
<tr>
<td>Event (with value) f</td>
</tr>
</tbody>
</table>

x > 1023
changeOf(x)
f inPast <=10ms

I compute information from observations
I formulate and monitor complex correctness properties
I define complex triggers
TeSSLa by Example

\[
\begin{align*}
  a & = 5 \\
  b & = 3 \quad 1 \quad 4 \\
  c & = 8 \quad 5 \quad 3 \quad 6 \\
\end{align*}
\]

\textbf{def} \ c \ := \ a + b

\[
\begin{align*}
  e & \\
  x & = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \\
\end{align*}
\]

\textbf{def} \ x \ := \ \text{eventCount} \ (e)
TeSSLa by Example

def c := a + b

def x := eventCount(e, reset = r)
TeSSLa – Burst Pattern (Macros)

\[
\begin{align*}
\text{def } c &= a > b \\
\text{def } p &= \text{if } c \\
&\quad \text{then noEvent}(e, \text{since} = \text{rising}(c)) \\
&\quad \text{else bursts}(e, \text{burstLength} = 2s, \\
&\quad \quad \text{waitingPeriod} = 1s, \\
&\quad \quad \text{burstAmount} = 3, \\
&\quad \quad \text{since} = \text{falling}(c))
\end{align*}
\]
TeSSLa – Burst Pattern (Macros)

def c := a > b
def p := if c
    then noEvent(e, since = rising(c))
    else bursts(e, burstLength = 2s,
                waitingPeriod = 1s,
                burstAmount = 3,
                since = falling(c))
def c := a > b
def p := if c
  then noEvent(e, since = rising(c))
  else bursts(e, burstLength = 2s,
             waitingPeriod = 1s,
             burstAmount = 3,
             since = falling(c))
TeSSLa’s operators

**default, defaultFrom**
- Initialize streams
- Start of recursion

**time**
- Get timestamps of stream
- Replaces data values with timestamps
- Only way to read timestamps

**last**
- Refers to previous value of a stream
- Recursion

**delayedLast**
- Only way to create events
- Takes a stream and delays events by its current value
- Output events have the previous value of another given stream

**lift**
- Lifts standard functions to streams
- Used to manipulate data, events, ...
TeSSLa’s fragments

![TeSSLa's fragments diagram]

- **Data**:
  - **TeSSLa**: Temporal Stream-based Specification Language
  - **TeSSLa + Data**: Includes data structures and computations on timestamps

- **Timestamps**:
  - **TeSSLa**
  - **TeSSLa + delay**: Allows for arbitrary computations on timestamps

- **Ordering**:
  - **TeSSLa**
  - **TeSSLa + Data**: Restricted to boolean streams

- **Comparison**:
  - **TeSSLa**: Restricted to expressions of clocks following adjustments
  - **TeSSLa + Data**: Extended clock constraints

- **Creation**:
  - **TeSSLa**
  - **TeSSLa + Data**: Additional clock constraint for event values and timestamps

- **Unbounded**:
  - **TeSSLa + Data**: unrestricted

- **Bounded**:
  - **TeSSLa**: Restricted to boolean streams

- **None**:
  - **TeSSLa + delay**: Limited to boolean streams

- **DTFST**
  - Deterministic Timed Finite State Transducer
  - Same run as DFST with timestamps

- **TeSSLa**
  - Translates DTFST to TeSSLa

- **Theorem 8**
  - For a DTFST, the inverse encoding functions for decoding timestamps
  - Both representations are now isomorphic
  - Can encode words as streams and vice versa, but this time

- **Translating DTFST to TeSSLa**
  - The current time in the input and output word
  - Additional clock constraint for event values and timestamps

- **Temporal distance**
  - Related to how clock constraints in timed automata work

- **Stream constraints**
  - Translates stream constraints to boolean streams

- **Temporal distance (g)**
  - Temporal distance of the current events of two streams
  - Directly related to how clock constraints in timed automata work

- **Constants (c)**
  - Represented as timestamps

- **TeSSLa fragments**
  - Restricted regarding a) data structures and b) event values and available data structures

- **DTFST and TeSSLa**
  - Similar expressiveness
  - Same temporal distance of the current events of two streams

- **Consistency**
  - TeSSLa fragments are consistent with how clock constraints in timed automata work.
EU Horizon 2020 Project: COEMS
The COEMS Consortium

University of Lübeck
Accemic Technologies
Thales Romania
Thales Austria
Høgskulen på Vestlandet / Western Norway
University of Applied Science
Airbus
COEMS set-up

system-on-chip

CPU 1  CPU 3
CPU 2  CPU 4

non-intrusive observation

COEMS
online processing (FPGA based)

online reconstruction
online analysis

output of results
PC

debugging property

42
Hardware Platform

• Hardware
  • Virtex-7 series FPGA (available)
  • Zynq Ultrascale+ SOC (under development)
  • RLDRAM3 memory for fast lookup tables
  • Interface to Aurora (Nexus, HSSTP)
  • VPX / FMC form factor

• Functionality
  • Online trace data processing
    (Coresight trace data -> event stream)
  • Supported architectures:
    ARM Cortex-A9, ARM Cortex-A53*, QorIQ PPC*, Infineon Aurix*
  • Online processing of event stream

*under development
Web IDE – with trace
Web IDE with C-code – software backend
Web IDE – with C-code – hardware backend
Conclusions
Summary

- Sometimes software annotations not acceptable
- Usage of trace functionality of modern processors feasible
- Extra hardware may be used to monitor non-intrusively
- Sophisticated ideas necessary to make overall approach feasible
- Hardware-based RV might be a game changer
Future Work

• Abstractions in TeSSLa
• Precise Relation of TeSSLa fragments to STL
• Partial-Order Semantics

• Support for ITM traces
• Increase performance of implementation
• Achieve TRL6
• Enhance training material