The Future of Runtime Verification for Cyber-Physical Systems Development

Runtime Verification Conference
James Kapinski
Toyota Research Institute of North America
November, 2018
Topics

• Importance of CPS and verification
• A vision for RV applied to CPS development
• Recent RV developments, tech transfer experience, and open challenges
Who we are?

Toyota Research Institute of North America

- **Vision:** Focused research to enable breakthrough applications for a sustainable mobility society
- **Mission:** Through in-house, collaborative and partnership activities in NA create the research foundation for Toyota’s advanced product development
CPS applications are everywhere!

- Basic research for CPS supports many applications
  - Service robots
  - Robot swarms
  - Warehouse automation
  - Disaster assistance
  - Autonomous vehicles
  - IoT
  - Smart buildings, smart grids, smart cities
  - Security
  - Medical devices
  - Personal mobility
CPS development is hard!

- Cost of developing software controllers for cars is large
  - 40% of the cost to develop a premium car could be from software and electronics, with software development alone contributing to about 13% [1]
- Huge cost for rework due to problems:
  - Cost of rework can be over 50% on very large projects
  - Rework cost much smaller (by factors of 50 to 200) in the earlier phases [2]
- Cost of testing significant
  - Not just verifying functionality, but lots of calibration work and tuning for performance
- Improved methods have potential to decrease cost, not to mention increase reliability

CPS is safety critical!

The FDA has issued 23 recalls of defective devices during the first half of 2010, all of which are categorized as “Class I,” meaning there is “reasonable probability that use of these products will cause serious adverse health consequences or death.”
RV Technologies and CPS Development

- Strict interpretation of RV applies to systems in late development stages (or deployed)
RV Technologies and CPS Development

- We consider a broader interpretation of RV, to include any RV-related technology
- We apply to aspects of CPS development process
A Vision for RV Applied to CPS Development

- We envision RV technologies improving CPS development activities
  - Requirement engineering
  - Design debugging
  - Test & calibration
  - Runtime verification
Recent Advances

• Recent efforts
  – Requirements engineering
    • ST-Lib
    • STL-2-SMT
  – Design debugging
    • Falsification
    • LUT’s and Stateflow
  – Test & calibration
    • Interface-aware STL
  – Runtime verification
    • STL-2-FPGA
    • OBD considerations

• Let’s explore each in more detail, see results from application efforts, and describe ongoing challenges…
Introduction to STL

• Signal Temporal Logic (STL)
  – Specify timed behaviors of systems, containing:
    • Logic operators (∧, ¬, ∨, →)
    • Temporal operators (□: “always”, ◊: “eventually”, and 𝜉: “until”)
    • Atomic constraint formula ($f(x) \geq 0$)

Introduction to STL

- Signal Temporal Logic (STL)
  - Specify timed behaviors of systems, containing:
    - Logic operators ($\wedge$, $\neg$, $\lor$, $\rightarrow$)
    - Temporal operators ($\square$: “always”, $\Diamond$: “eventually”, and $\mathcal{U}$: “until”)
    - Atomic constraint formula ($f(x)\geq 0$)
  - Examples

\[ \square_{[0,100]} (\text{boost pressure} < 250) \]

\[ \square_{[0,100]} \left( (\text{gear} = 1 \land \Diamond_{[0,\epsilon]} \text{gear} = 2) \rightarrow \square_{[\epsilon, \tau+\epsilon]} (\text{gear} = 2) \right) \]
Introduction to STL

• Signal Temporal Logic (STL)
  – Key STL-related technologies
    • Given STL formula $\varphi$ and any behavior of system $\mathcal{M}$, $\phi(\mathcal{M}, p, u)$, we use the following tools
      – **Monitoring**: we can efficiently determine whether $\phi(\mathcal{M}, p, u)$ satisfies $\varphi$
      – **Robust Semantics**: we can efficiently compute how well $\phi(\mathcal{M}, p, u)$ satisfies or does not satisfy $\varphi$
Requirements Engineering

• **ST-Lib**\(^{[1]}\):
  – **Idea**: provide support for control engineers to create formal specifications
  – A library for specifying and classifying signal patterns of system behaviors
  – Uses STL to identify signal patterns interesting to design engineers:
    • Ringing
    • Spikes and glitches
    • Excessive overshoot or undershoot
    • Slow response time (settling, rising, or falling)
    • Undesirable timed relation behaviors
    • Steady state or tracking error

• Let’s look at some examples…

---

Requirements Engineering

**Example ST-Lib Requirements**

**Ringing**

- Unacceptable: \( x(t + d) - x(t) > a \)
- Acceptable: \( x(t) - x(t + d) > a \)
- Ringing once: \( \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{diff}} < -m) \land \Diamond_{[0, \infty]} (x_{\text{diff}} > m)) \)
- Ringing twice: \( \Diamond_{[0, T]} (\Diamond_{[0, 0]} \land \Diamond_{[0, \infty]} (x_{\text{diff}} < -m) \land \Diamond_{[0, \infty]} (x_{\text{diff}} > m)) \)

**Spike**

- Unacceptable: \( y(t + e) - y(t) > r \)
- Acceptable: \( y(t) - y(t + e) > r \)

\[ \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{ref}}) \land \Diamond_{[0, \infty]} (x - x_{\text{ref}} > c)) \]

**Overshoot**

\[ \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{diff}} < \beta \land |x_{\text{diff}}| < \beta \land \Box_{[0, \infty]} (x_{\text{ref}} - x_{\text{ref}} > c)) \]

**Settling Time**

\[ \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{ref}}) \land \Diamond_{[0, \infty]} (|x - x_{\text{ref}}| > a)) \]

**Rise Time**

\[ \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{ref}}) \land \Diamond_{[0, \infty]} (x_{\text{ref}} - x < \beta \land \Box_{[0, \infty]} (x_{\text{ref}} - x_{\text{ref}} > c)) \]

**Steady State Error**

\[ \Diamond_{[0, T]} (\Diamond_{[0, \infty]} (x_{\text{ref}}) \land \Diamond_{[0, \infty]} (x_{\text{ref}} - x_{\text{ref}} > a)) \]

**Note:** Above formulas specify “bad” behaviors.
Requirements Engineering

• ST-Lib application example
  – Example: used the following version of the overshoot requirement

OVERSHOOT := □((STEPUP and □[dt,sstime]¬STEP) => □[dt,sstime](OVERSHOOTLIMIT))

STEPUP := in[t+dt]-in[t] > StepThresh
STEPDOWN := in[t]-in[t+dt] > StepThresh
STEP := STEPUP or STEPDOWN

OVERSHOOTLIMIT := out[t] < 1.1*(in[t])

Comment:
• No other step should be present when checking the overshoot

sstime: Time over which steady-state is assumed to be reached
dt: Small constant, comparable to a sampling step size
Requirements Engineering

- ST-Lib application example: Hydrogen Fuel Cell Vehicle
- Overshoot requirement performance
  - Requirement monitor performance for hypothetical model, using engineered input patterns

Comments:
- Overshoot values are appropriately detected
- New fault localization tool use to highlight instants when faults occur
Requirements Engineering

- ST-Lib application example: Hydrogen Fuel Cell Vehicle
- Challenge:
  - Bad (unexpected) requirement performance for real data

Comments:
- Many unexpected behaviors are identified
- Other behaviors are mischaracterized
Requirements Engineering

- **ST-Lib**
  - **Problem:** Many important behaviors are do not fall into typical class of steps, steady-state, etc.
    - These behaviors should fall under some other category of inputs, like an input \textit{ramp}, with corresponding requirements

**Comment:**
- This is appropriately identified as an overshoot failure

**Comments:**
- These behaviors are not steep enough to be steps but not small enough to be steady-state
  - So the behaviors are not constrained in any way
- Want to make sure that all behaviors are somehow evaluated
Requirements Engineering

- **ST-Lib**
  - Other challenges
    - Engineer had a notion of an ideal response (included a time-shifted, rate-limited version of the command signal)
      - Not easily captured in STL
      - Addressed by a priori defining a new signal
Requirements Engineering

• ST-Lib shortcomings
  – Step, steady-state, etc. are not the only meaningful input classifications
    • Want to impose constraints on output corresponding to step, ramp, steady-state, etc.
  – For real data, need to define a partition on the input and relate a corresponding behavioral constraint on output
    • Can’t allow unconstrained output behaviors
    • Every moment of output behavior should be evaluated/constrained
  – Need to allow for more subjective classification of reference signal class (step, ramp, SS, others…)

Takeaway: Need to constrain all output behaviors, but difficult to classify corresponding input behaviors
Requirements Engineering

• ST-Lib
  – What’s the underlying problem?
    • There’s some expected mapping from inputs (as they relate to a canonical inputs) to outputs
    • The expectation:
      – When inputs similar to a step are applied, outputs should be similar to a step-response

![Nominal step-response I/O behaviors](image)

![Behaviors similar to nominal step-response](image)
Requirements Engineering

• ST-Lib

  – Qualitative differences hard to capture
  – Skorokhod metric one way to do that\textsuperscript{[1]}
    \[ \mathcal{D}_\mathcal{S}(x, y) \text{ is least of the following for all } \text{retiming functions } r(t): \]
    \[ \max \left( \sup_{t \in [0,T]} |r(t) - t|, \sup_{t \in [0,T]} \mathcal{D}(x(r(t)), y(t)) \right), \]
    where \( \mathcal{D} \) a pointwise metric

  – Provides way to compare time-series data
  – Allows for time distortions
  – Efficient methods to compute
  – Can we use to capture distance to canonical behaviors?

Requirements Engineering

- **STL-2-SMT**[1]
  - Created method to generate example traces from STL requirements
    - Uses Z3 SMT solver
  - Use: debug formal requirements

Consider the following examples for an Overshoot STL requirement:

\[
\text{Overshoot} := \square_{[0,10]} (\text{Step}\_\text{Up} \Rightarrow \square_{[0,1]} \text{out}(t) < 1.1 \cdot \text{in}(t))
\]

\[
\text{Step}\_\text{Up} := \text{in}(t + dt) - \text{in}(t) > \text{Step\_Thresh}
\]

**Takeaway:** Example traces can identify bugs in the requirement

Requirements Engineering

- **STL-2-SMT**\(^1\)
- **Open challenges:**
  - Scalability (in equation length and number of switching instants)
  - Traces are piecewise linear
- **Note:** STLInspector tool addresses similar problem and can provide realistic-looking traces\(^2\)

---


Requirements Engineering

• Other topics
  – Learning STL requirements from models[1]
  – Learning STL requirements from data[2,3]

Design Debugging

Optimizer:
Minimize robust satisfaction value

Falsification by optimization

Falsification supported by both S-TaLiRo and Breach tools
Design Debugging

• Falsification
  – Big impact
    • Tech transfer continuing to multiple groups
  – Considering lots of strategies
    • Trace “splicing”\(^\text{[1]}\), coverage-based\(^\text{[2]}\), Bayesian optim.\(^\text{[3]}\), discrete-search\(^\text{[4]}\)

---

Design Debugging

• Falsification
  – Applications to autonomous driving[5,6]

Results from Tuncali et al.

Design Debugging

- **Localization of faults in Lookup Table entries**\(^1\)
  - Statistical approach
  - Generalizing software fault localization tool approaches (Tarantula)
    - Characterizes LUT elements based on STL requirement robustness
    - Produces heatmaps that identify problematic LUT entries
  - Tool (Vajra) developed
- **Related work for Stateflow models**\(^2\)

---


Design Debugging

- Localization of faults in Lookup Table entries\(^1\)
- Related work for Stateflow models\(^2\)
- Ongoing challenges: robustification of tools

---


Test & Calibration

- **Interface-Aware STL**\(^{[1]}\)
  - **Idea**: engineers need help evaluating test data
  - **STL shortcoming**: does not capture system context
  - Consider STL specification:
    \[ \text{alw}((\text{req} \geq 4) \rightarrow \text{ev}_{[0,2]} (\text{gnt} \geq 4)) \]
  - Classical robustness
    - Intuition: \( \text{rob}(\{\text{req}, \text{gnt}\}, S) = -3 \)
    - Actual: \( \text{rob}(\{\text{req}, \text{gnt}\}, S) = -1 \)
  - Classical robustness combines
    - How bad is the lack of grant
    - How good is the request
  - Measure of how good is the request wins over measure of how bad is the grant
  - Problem is: no distinction between in/out

Test & Calibration

- **Interface-Aware STL**[1]
- **Interface-Aware STL Benefits:**
  - Can find input vacuity
  - Can determine worst case output behavior
  - Can localize (in time) where worst case output behavior occurs[2]
  - Provides more useful robustness values (for sensitivity analysis)
- **Implemented in Breach**
- **Ongoing challenges:**
  - Can we use IA-STL to do compositional reasoning?

Example result of fault localization

Test & Calibration

• **STL-based clustering/classification of time-series data**
• **Challenge:** Difficult to bring engineering insight to characterize time-series data in big-data context
• **Solution:** New methods that leverage parameterized STL (PSTL) to characterize behaviors[^1,^2]
  • Used to cluster and identify anomalies

Runtime Verification

• **STL-2-FPGA**
  
  – STL monitoring possible on runtime platforms, like FPGAs\(^1,2\)
  
  – We hold out hope for this, but we have no applications yet
  
  – A word about On-board Diagnostics (OBD) for powertrain monitoring…


Runtime Verification

• Considerations from the “frontlines” of runtime verification
  – On-board diagnostics (OBD)
    • Used for powertrain system monitoring
    • Mandated by the California Air Resources Board (CARB)
    • Monitors emissions-related behaviors
    • Since 1996, almost all cars equipped with OBD II systems
    • Estimated to account for 20-30% of processing power
    • Most functionality is hand-coded in C
    • OEMs required to provide clear description of OBD functions to CARB (i.e., flowcharts)
Runtime Verification

• Considerations from the “frontlines” of runtime verification
  – Opportunities for the RV community
    • There is a desire to reduce calibration effort and code complexity
    • A strong interest in “prognosis” methods
  – Challenges for the RV community
    • Creating formalisms/languages that can be easily defined/tuned by engineers
    • How to clearly communicate monitor functionality to regulators?
      – If we are auto-generating monitors from requirements, do we need to go through a compiler-certification process?
A Vision for RV Applied to CPS Development

- RV tech will improving CPS development process
  - RV will be deeply embedded into many aspects of CPS development
    - Increasing confidence in system correctness
    - Simplifying the process (by removing ambiguities in the design and debug processes)
    - Decreasing development cost
Summary

• CPS applications are: everywhere, hard to develop, and safety critical
• RV technologies can support the CPS development process
  – Requirements engineering
    • **ST-Lib**: Library of STL formulas for CPS applications
    • **STL-2-SMT**: Automatic generation of traces for STL formulas
  – Design debugging
    • **Falsification**: Automated bug-finding
    • **LUT’s and Stateflow**: Automated diagnosis
  – Test & calibration
    • **Interface-aware STL**: STL for systems
  – Runtime verification
    • **STL-2-FPGA**: Automatic generation of monitors
Thanks for your attention!

• Questions? Comments?