Runtime Verification for Real-Time Automotive Embedded Software

S. Cotard, S. Faucou, J.-L. Béchennec, A. Queudet, Y. Trinquet

10th school of Modelling and Verifying Parallel processes (MOVEP)
Motivating example

Safety constraint: \( T_2 \) requires the data from \( b_1 \), but also reads \( b_0 \) in order to perform a plausibility check. \( T_2 \) has to read the same instance of data.

Requirement: consistency checking

Correctness property: when \( T_2 \) starts reading, the buffers are synchronized and stay synchronized until \( T_2 \) completes its execution.
A possible solution: diversification
A possible solution: diversification
A possible solution: diversification
A possible solution: diversification
Context

Objectives

- based on formal methods
- compatible with functional and industrial constraints
  - small and deterministic detection latency
  - small and deterministic overheads (execution time, memory footprint)
  - compatible with multi-tiers system design process (the provided source code is not always modifiable)

Proposed solution

- runtime verification
- injection of monitors in the kernel
Formal methods

Model $M$ of a system $S$

- Model checking: all runs of $M$ satisfy $\phi$? (design time)
- Tests: some runs of $S$ satisfy $\phi$? (design time)
- Runtime verification: does this run satisfy $\phi$? (online analysis)
  $\rightarrow$ generate a monitor from $M$ and $\phi$ that outputs a verdict in $\{\top, \bot, ?\}$
Our approach: runtime verification: step 1

[Bauer et al, 2011] solution

For $\phi$ and $\neg \phi$
1) Compute NBAs
2) Emptiness checking per state (derived F)
3) Compute NFAs using F
4) Compute DFAs
5) DFAs synchronization
Our approach: runtime verification: step 1

Property

\[ \phi = G ((m_{t2}.firstb0 \lor m_{t2}.firstb1) \implies (m\_sync.sync U m_{t2}.begin)) \]

1) Computation of the NBAs

\[ A^\phi \]

\[ \neg m_{t2}.begin \]

\[ \neg m_{t2}.begin \lor \neg m\_sync.sync \]

\[ \neg m_{t2}.begin \land \neg m\_sync.sync \]

\[ true \]
Our approach: runtime verification: step 1

Property

\( \phi = G ((m_{t2}.firstb0 \lor m_{t2}.firstb1) \implies (m_{sync}.sync U m_{t2}.begin)) \)

1) Computation of the NBAs

2) Emptiness checking per state

\( F^\phi = \{S_0, S_1\} \)

\( F^{-\phi} = \{S'_0, S'_1, S'_2\} \)
Our approach: runtime verification: step 1

\[ \phi = G((m_{t2}.firstb0 \lor m_{t2}.firstb1) \implies (m_{sync}.sync \cup m_{t2}.begin)) \]

3) Computing NFAs using F and completes automata
Our approach: runtime verification: step1

Property

\[ \phi = G ((m_{t2}.firstb0 \lor m_{t2}.firstb1) \implies (m_{sync}.sync U m_{t2}.begin)) \]

4) Determinization $\rightarrow$ Composition $\rightarrow$ Minimization

The intermediate monitor \( M^m \) reacts to changes in the values of the atomic propositions used in \( \phi \).
Our approach: runtime verification: step 1

Intermediate monitor \((M^m)\)

The intermediate monitor is the Moore machine given by

\[ M^m = (Q^m, i^m, \rightarrow_m, \gamma^m) \]

over \(2^{AP}\), the set of intercepted events.

- \(Q^m\) is the finite set of states
- \(i^m\) is the initial state
- \(\rightarrow_m \subset (Q^m \times 2^{AP}) \mapsto Q^m\) is the transition function
- \(\gamma^m \subset Q^m \mapsto \mathbb{B}_3 = \{\top, \bot, ?\}\) is the output function
Our approach: runtime verification: step2

Input: system model + properties

\[ G \left( (m_{t2}.firstb0 \lor m_{t2}.firstb1) \implies (m_{sync}.sync \cup m_{t2}.begin) \right) \]
Our approach: runtime verification: step 2

Model of the system \((A^s)\)

The model of the system is given by \(A^s = (Q^s, i^s, \rightarrow_s)\) over \(\Sigma^s\), the set of intercepted events.

- \(Q^s\) is the finite set of states
- \(i^s \in Q^s\) is the initial state
- \(\rightarrow_s \subseteq (Q^s \times \Sigma^s) \mapsto Q^s\) is the transition function

We denote \(\lambda^s \subseteq Q^s \mapsto 2^{AP}\), the labeling function that maps each state of the DFA to the set of atomic proposition true in this state.
Our approach: runtime verification: step2

**Final monitor computation (\( M' \))**

The final monitor is defined by \( M' = (Q', i', \to, \gamma') \) over \( \Sigma^s \)

- \( Q' = Q^s \times Q^m \)
- \( i' = (i^s, i^m) \)
- \( \to \subseteq (Q' \times \Sigma^s) \mapsto Q' \) where \( (q^s, q^m) \overset{\sigma}{\rightarrow} (r^s, r^m) \) iff \( q^s \overset{\sigma}{\rightarrow}_s r^s \) and \( q^m \overset{u}{\rightarrow}_m r^m \) and \( u \subseteq \lambda^s(r^s) \) and \( \gamma^m(q^m) =? \)
- \( \gamma' \subset Q' \mapsto \mathbb{B}_3 \) where \( \gamma'(q^s, q^m) = \gamma^m(q^m) \)
Our approach: runtime verification: step2

Output: a monitor

\[ s_{11}, r_{10} \]
\[ s_{11}, r_{10} \]
\[ s_{00}, r_{20} \]
\[ s_{11}, r_{10} \]
\[ s_{00}, r_{21} \]
\[ s_{11}, s_{00} \]
\[ s_{11} \]
\[ s_{11}, s_{00} \]
\[ s_{00} \]
\[ s_{00} \]
\[ r_{20}, r_{21} \]
\[ r_{20}, r_{21}, r_{10} \]
**Enforcer: A tool for monitor synthesis**

- **System model** (transition system)
- **Properties** (LTL formulae)
- **Enforcer tool**
- **Output sources**
  - Monitor (*.c *.h)
  - Transition table

- Inconclusive state
- True state
- False state

LTL $\rightarrow$ NBA $\rightarrow$ Intermediate Moore machine $\rightarrow$ Monitor

- **step 1**
- **step 2**

$l tl\_rule = "Always (a Until b)"$
Injection of the monitors in the kernel

The Trampoline compilation chain (open-source implementation of AUTOSAR OS)
Architecture
Evaluation: computation overhead

- target running at 60 MHz
- composition of the overhead
  - 1µs to identify the event
  - 2.4µs to react per monitor interested in the event
## Evaluation: memory footprint

<table>
<thead>
<tr>
<th>Transition table</th>
<th>Monitor descriptor</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROM</strong></td>
<td><strong>RAM</strong></td>
<td><strong>ROM/RAM</strong></td>
</tr>
</tbody>
</table>

- **30 bytes** dominates the monitor size, depending on the monitor.  
  3 optimizations have been proposed.
- **15 bytes** is a constant per monitor.
- **152 bytes** (monitor update) is a constant.  
  **16 bytes** (event handler) depends on the number of monitors per event.
Conclusion

- approach has been implemented in a tool: Enforcer
  - freely available (see paper for URL)

- results show that runtime verification can be affordable for (static) industrial real-time embedded systems
  - kernel instrumentation allows to achieve (guaranteed) low detection latency
  - static code and data generation allows to achieve low execution time overhead
  - system designer can pay time for memory

- future works
  - compute the theoretical bound on the size of the monitors (given the size of $M$ and $\phi$)
  - multicore extension (not only a matter of implementation)
Thank you for your attention