Agile Software Design Verification And Validation (V&V) For Automated Driving

Yixiao Li¹, Yutaka Matsubara¹, Daniel Olbrys², Kazuhiro Kajio², Hiroaki Takada¹

¹Nagoya University ²Toyota Research Institute - Advanced Development, Inc.
Verification and Validation (V&V)

V&V must be performed to meet the safety requirements

- Required by the development process standard in the automotive domain (e.g. ISO 26262)
- Verification: check whether the design can meet the requirements
- Validation: check whether the implementation can work properly in the real environment
- Defects and problems found during the validation must be feedbacked to the system designers to modify the requirements or specifications accordingly
Automated Driving System (ADS)

**ADS is much more complex than the traditional system**
- Massive data: HD dynamic maps, sensors (e.g. camera, radar, LiDAR)
- Software: soft/hard real-time tasks, parallelized with many threads, machine learning
- Hardware: multi-core CPUs, many-core GPUs, deep learning accelerators
- Frequent (monthly/weekly) OTA software updates
- New standardized platform (e.g. AUTOSAR Adaptive Platform)
- Virtualization environments (e.g. container on Docker, guest OS on hypervisor)

**The increased complexity of ADS brings new challenges**
- Fixing a problem identified during the validation phases could lead to expensive rework
- The productivity of V&V must be reduced to match the agile development methodologies
Simulation-based V&V can find design problems earlier

- Create a model from the requirements and design during the verification phases
- Generate the approximations of system behavior using a simulator
- Check the requirements during the verification without testing on the real validation environment
- A high accurate model can significantly reduce the expensive rework during the V&V process
- Existing approaches focus on control systems rather than ADS

Issues to enable simulation-based V&V for ADS

- How to properly model the complex ADS applications?
- How to verify and explore the design with the simulator?
Our Proposed Framework

An automated framework to model the ADS software efficiently

User Input
- Design Model
- Application Trace Data

Proposed Framework
- CPU & GPU Timestamps

Feedback
- Timing Model
- Simulation Model
- Verify & Explore Design with the Simulator

User (Developer) provides design & trace as input.

Our framework provides scripts to automatically create a simulation model.

User can verify the current design and/or explore potential improvements on a simulator.
Provide necessary information to schedule the application threads

- Required parameters can be effortlessly collected from the design specifications
- Design configuration files (e.g. ARXML, JSON) can also be used to fill some parameters automatically.
Provide the execution time information of each thread

- Traced on an environment with specification (e.g. CPU, GPU) close to the real target
- Application running in Docker container on a Linux-based AUTOSAR Adaptive Platform
- Tracer: LTTng (Linux Trace Toolkit Next Generation)

The measurement accuracy is vital to the simulation quality

- Minimize interference: isolate the application under test from the system processes
- Reduce overhead: replace default events with manually added tracepoints; use a larger buffer size…
- Avoid overlaps: increase the periodic time of application cycles
Input: The Application Trace Data

The trace data can be very huge
- Difficult to handle directly
- A 90-sec trace of our applications has over 60 million events

Extract necessary timestamps to ease the procedure of creating a timing model
- CPU timestamps: the active CPU time
- GPU timestamps: the time of a thread occupying the GPU resource
- UST timestamps: the time range of one activation (i.e. from triggered to send output messages)
Create the Timing Model

A timing model is a design model with per-thread timing information
- A thread with GPU usage consists of multiple parts with different timing characteristic
- E.g. a thread accesses GPU twice during each execution:

Create an execution time distribution for each part from the trace data

![Diagrams showing execution time distributions for different parts](diagrams.png)
Create the Timing Model

The traced discrete distributions can optionally be fitted using statistical models

- E.g. Gaussian distribution

- Pros: values not included in the trace can also be generated
- Cons: bad-fitting models can lead to poor predictability
- Currently, the user is responsible for choosing proper models according to the actual implementation
Create the Simulation Model

A simulation model consists of the necessary files to execute on a simulator

- Generated from the timing model
- Use INCHRON chronSIM simulator in our paper

Features of INCHRON chronSIM simulator

- Project (IPL) file in XML format, easy to handle by external scripts
- Official Python library to manipulate the simulation model
- Many standard scheduling policies (e.g. SCHED_FIFO, SCHED_RR) supported
- User-defined scheduler and hierarchical scheduler can be used
- Advanced visualization and analysis capabilities: state view, gantt view, event chain, load view, histogram
Create the Simulation Model

The basic task model of chronSIM lacks some features for our application

- Trigger messages cannot be queued
- GPU usage is not supported

Extend the simulation task model implementation

- chronSIM allows us to add C source files to override the default task behaviors in the model
- The execution time distributions are converted to C source for randomly generating simulated values
- Each thread has its data structure to manage the trigger messages with our custom activation logic
- Necessary operations for GPU access (e.g. acquiring, offloading, releasing) are supported
- Visualize the GPU usage with event chains
Evaluation

An industrial case study
- An ADS prototype on AUTOSAR Adaptive Platform from TRI-AD, Inc.
- Use 7 CPU cores and include 50 threads in total
- 30 threads created by the ADS software, 20 threads created by external libraries, 7 threads using GPU
- Trace environment

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-8700K, 3.7 GHz, 12 logical cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce GTX 1080 Ti, 1582 MHz, 3584 CUDA cores</td>
</tr>
<tr>
<td>System</td>
<td>Linux (Ubuntu 16.04 LTS), Docker 19.03.5</td>
</tr>
</tbody>
</table>

- The application is executed for over 300 cycles to collect the trace data

Evaluate the model accuracy and the effectiveness of proposed framework
- Compare event chain end-to-end latency
- Compare CPU usage
Event chain E2E (end-to-end) latency

• Simulate a representative event chain (Camera data output → Multiple threads with different processing algorithms → Result output)
• chronSIM simulator can create a histogram for the end-to-end latency

Identify a design problem

• A significant gap was observed in the initial design (traced 27.5ms vs simulated 38.8ms)
• Use chronSIM simulator to analysis the detailed execution patterns
• An inconsistency between the design and the implementation has been found
• After fixing the problem, the new simulated E2E latency is 27ms
Evaluation

CPU usage comparison

- Per-thread values: very close ($\Delta < 0.1\%$ for each thread except #1,#2)
- Per-core values: prior cores (1~3) higher in simulation, latter cores (4~7) higher in trace

<table>
<thead>
<tr>
<th>Thread#</th>
<th>Trace</th>
<th>Simulation</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34.24%</td>
<td>39.06%</td>
<td>4.82%</td>
</tr>
<tr>
<td>2</td>
<td>19.15%</td>
<td>20.39%</td>
<td>1.24%</td>
</tr>
<tr>
<td>3</td>
<td>21.31%</td>
<td>21.39%</td>
<td>0.08%</td>
</tr>
<tr>
<td>4</td>
<td>1.92%</td>
<td>1.98%</td>
<td>0.06%</td>
</tr>
<tr>
<td>5</td>
<td>1.75%</td>
<td>1.81%</td>
<td>0.06%</td>
</tr>
<tr>
<td>6</td>
<td>0.60%</td>
<td>0.64%</td>
<td>0.04%</td>
</tr>
<tr>
<td>7</td>
<td>1.63%</td>
<td>1.60%</td>
<td>-0.03%</td>
</tr>
<tr>
<td>8</td>
<td>0.74%</td>
<td>0.72%</td>
<td>-0.02%</td>
</tr>
<tr>
<td>9</td>
<td>0.45%</td>
<td>0.47%</td>
<td>0.02%</td>
</tr>
<tr>
<td>10</td>
<td>0.16%</td>
<td>0.18%</td>
<td>0.02%</td>
</tr>
<tr>
<td>11</td>
<td>1.87%</td>
<td>1.89%</td>
<td>0.02%</td>
</tr>
<tr>
<td>12</td>
<td>1.05%</td>
<td>1.04%</td>
<td>-0.01%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>50</td>
<td>1.93%</td>
<td>1.93%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Comparison of per-thread CPU usage (sorted by absolute value of $\Delta$)

<table>
<thead>
<tr>
<th></th>
<th>Trace</th>
<th>Simulation</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>14.71%</td>
<td>20.11%</td>
<td>5.40%</td>
</tr>
<tr>
<td>CPU2</td>
<td>18.27%</td>
<td>28.52%</td>
<td>10.25%</td>
</tr>
<tr>
<td>CPU3</td>
<td>22.00%</td>
<td>24.36%</td>
<td>2.36%</td>
</tr>
<tr>
<td>CPU4</td>
<td>11.99%</td>
<td>9.30%</td>
<td>-2.69%</td>
</tr>
<tr>
<td>CPU5</td>
<td>13.22%</td>
<td>9.15%</td>
<td>-4.07%</td>
</tr>
<tr>
<td>CPU6</td>
<td>13.96%</td>
<td>9.68%</td>
<td>-4.28%</td>
</tr>
<tr>
<td>CPU7</td>
<td>25.85%</td>
<td>19.71%</td>
<td>-6.14%</td>
</tr>
<tr>
<td>Mean</td>
<td>17.14%</td>
<td>17.26%</td>
<td>0.12%</td>
</tr>
<tr>
<td>StdDev</td>
<td>4.76%</td>
<td>7.35%</td>
<td>2.59%</td>
</tr>
</tbody>
</table>

Comparison of per-core CPU usage
Per-core CPU usage differences

- Reason: system threads in the trace environment frequently migrate applications threads to other cores
- System threads relate to many factors (e.g. devices, kernel, services…), and thus are difficult to simulate
- Instead, we simulate the interference by adding random migrations
- The updated model (RandomStartCore) shows much smaller Δ.

<table>
<thead>
<tr>
<th></th>
<th>Trace</th>
<th>Simulation</th>
<th>RandomStartCore</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>14.71%</td>
<td>20.11%</td>
<td>14.11%</td>
</tr>
<tr>
<td>CPU2</td>
<td>18.27%</td>
<td>28.52%</td>
<td>16.08%</td>
</tr>
<tr>
<td>CPU3</td>
<td>22.00%</td>
<td>24.36%</td>
<td>18.86%</td>
</tr>
<tr>
<td>CPU4</td>
<td>11.99%</td>
<td>9.30%</td>
<td>12.99%</td>
</tr>
<tr>
<td>CPU5</td>
<td>13.22%</td>
<td>9.15%</td>
<td>15.16%</td>
</tr>
<tr>
<td>CPU6</td>
<td>13.96%</td>
<td>9.68%</td>
<td>16.85%</td>
</tr>
<tr>
<td>CPU7</td>
<td>25.85%</td>
<td>19.71%</td>
<td>24.21%</td>
</tr>
<tr>
<td>Mean</td>
<td>17.14%</td>
<td>17.26%</td>
<td>16.89%</td>
</tr>
<tr>
<td>StdDev</td>
<td>4.76%</td>
<td>7.35%</td>
<td>3.47%</td>
</tr>
</tbody>
</table>
Conclusion

• Propose an automated framework for modeling the ADS applications based on the design and trace data
• Support commercial simulator chronSIM for design verification and exploration
• Evaluation results show high accuracy of the simulation results
• Simulation-based V&V enables the early/efficient problem identification during agile software development

Future work

• Simulate the scheduling of virtualization environments (e.g. containers, guest OSes on hypervisor)
• Analyze the threads created by external libraries (e.g. OpenCV, SOME/IP) to further improve accuracy
• Automatic design optimization (e.g. adjusting the design parameters to balance CPU load)
• Characterizing different applications with quantitative metrics
• Unified modeling framework for both ADS and traditional (e.g. AUTOSAR Classic Platform) applications