OS-in-the-loop CEGAR for Multitasking Embedded Control Software

Dongwoo Kim, Yunja Choi
kdw9242@gmail.com
Outline:

- Background
  - Multitasking embedded software
  - Model checking
- Limitations of existing methods
- The proposed verification method: OiL-CEGAR
  - Formal OS model and
  - OiL-CEGAR process
- Experiments
- Conclusion & Future work
Multitasking embedded software

- Each ECU mounts 1 software.
- Each software compiled with 1 OS, 1 App, and 1 Configuration.

A car has hundreds of ECUs
Verification of multitasking software

• Multitasking is used in most embedded software
  ✓ usually written in C language
  ✓ uses multiple tasks
  ✓ e.g., brake pedals, engines, sensors, actuators, etc.
  ✓ safety-critical
  ✓ require comprehensive verification

• Model checking is suitable for comprehensive verification
  ✓ rigorously verify software systems
Model checking

- Method for checking whether model $M$ meets a given specification $\phi$.

Finite state system $M$:

Property $\phi$ (a given specification): (e.g. system never reaches an error state)

- Model checking can be applied to a model or a **program code** (C, Java, etc)
- However, model checking on **multitasking embedded software** is very challenging.
Properties

• Boolean property (invariants) (it should be satisfied in all states)
  e.g., The running state should never be reached

• Assertion property (it should be satisfied in a state)
  e.g., Variable v cannot have value after statement 32

• Temporal property (specifies dynamic behavior) (it should be satisfied in every path)
  e.g., All task must be ready eventually

• Monitoring automata (it should not remain in error states, infinitely)
  e.g., When it receives a forward signal, it must move forward in 100 ticks.

• API-call constraint (a type of monitoring automata having API-call events)
  e.g., API-calls f1 and f2 shall be called in pairs.
Limitation: Model checking multitask program code with OS

- An OS implementation and application program code are can be directly verified.

Problems:
1. An enormous load of verification cost is required as it consumes time and memory exponential to the size of the program.
2. Usually, performs bounded model checking (cannot verifies the whole system)

---

Limitation: Model checking multitask program code w/o OS

- Complexity can be reduced by using highly abstracted OS. (allow all possible context switch)
- Most of reported traces are **false alarms** having incorrect task execution order

Problems:
1. False-alarm will be reported as there are no scheduling policy considered
2. Also, performs bounded model checking (cannot verifies the whole system)

References:
Necessity of operating system

- Task scheduling involves multiple objects of the OS kernel, including:
  - Tasks, API functions, resources, events, alarms, and ISRs, etc.
- A sound OS is necessary to improve the verification accuracy

Structure of an embedded OS (OSEK/VDX OS)
Insight: use of a sound OS model

- **OS model** correctly schedule an application and remove false alarms
- Model-level verification is efficient as it exclude all the details of programming language.
- Modeling language supports for concurrency, atomicity, and blocking.

List of sound OS models


- The correctness of generated OS model is verified based on the OSEK/VDX specification.
Insight: use of a sound OS model (cont.)

- The application code has to be translated into an application model.
- Informally, abstraction groups a set of states into a state.
- **Abstraction** is necessary due to the heterogeneity (between languages used for a model and program code)
  - Reduce verification complexity
  - Results in **high false alarm rate**
- False alarms shall be automatically identified and removed.

```
Concrete model   Abstract model
                
Running \( \xrightarrow{T} \) Suspended
| PT  \( \xleftarrow{A} \) | Ready \( \xrightarrow{PT} \) Running
|

Traces reaching Running:
- \( \text{SUS} \xrightarrow{A} \text{RDY} \xrightarrow{PT} \text{RUN} \)

Traces reaching Running:
- \( \text{NOT} \xrightarrow{PT} \text{RUN} \)
- \( \text{NOT} \xrightarrow{A} \text{NOT} \xrightarrow{PT} \text{RUN} \)
- \( \text{NOT} \xrightarrow{PT} \text{RUN} \xrightarrow{PD} \text{NOT} \xrightarrow{PT} \text{RUN} \)
- ... 
```
Counterexample-Guided Abstraction Refinement (CEGAR)


- Benefits
  - Scalable
  - Automated false alarm reduction
  - Automated false alarm reduction

- Problems
  - Certain types of false alarms may remain
  - Difficulty of feasibility checking for embedded software

Diagram:

- $\tilde{M} \models \phi$
- $\tilde{M} \leftarrow \text{refine}(\tilde{M}, \tau)$
- Is $\tau$ feasible?
  - no
  - yes
- $M \not\models \phi, \tau$
Our approach: OS-in-the-loop CEGAR
OS-in-the-Loop CEGAR (OiL CEGAR)

\[ M = M_{os} \parallel M_{app} \]

Model Checking

\[ M \models \varphi \]

NuSMV

satisfiable

\[ M_{os} \parallel M_{app} \models \varphi \]

Model Refinement

\[ M \leftarrow M \circ A(\tau') \]

unsatisfiable

a counterexample \( \tau \)

Executability Checking

\[ \widehat{M_{os}(\tau)} \parallel M_{app} \not\models^{R}_n \]

CBMC

yes

\[ M_{os} \parallel M_{app} \not\models \varphi, \tau \]
OS-in-the-loop CEGAR

- **A verified OS model** is used to enhance the accuracy of the property checking.
- **A mini-OS** is constructed from the counterexample trace for executability checking with improved accuracy.
- Model refinements are performed through **trace composition**.
- Utilized two different model checkers, NuSMV and CBMC, suitable for the two different purposes.
Formal OS models used in this work

- A pattern-based OS model generation framework* is reused


Typical construction of embedded software

```
MODULE main
VAR
t1 : Task(1, TRUE, ...);
t2 : Task(2, FALSE, ...);
rq : ReadyQueue(...);
evt : Event(...);
env : Environment(...);
...
```

NuSMV model generation
Application model construction

1. CFG construction for each task

2. Control abstraction
   - Blocks together a sequence of statements to be executed without interrupts

3. Data abstraction
   - Abstracts visible statement with a unique symbol
   - Major sources of false alarms
   - But greatly helps to reduce the complexity

4. Conversion into a task statemachine
   - Each transition is guarded to check the scheduling status of the task

5. Parallel composition of task statemachines
Composition model

Application Task t1

Task t1

Application Task t2

ReadyQueue

Event

Infinitely waiting state
Model checking using NuSMV

- Boolean properties and temporal logic properties in CTL and LTL can be checked
- Automatically generates a counterexample trace
- We can supply another module for monitoring automata with Boolean or temporal properties.
  - We verify assertion and API-call constraint checking as special types of monitoring automata

No infinitely waiting state
Static executability checking

- Executability of a counterexample is confirmed by checking the reachability of each application block
Model refinements

- The model is refined through trace composition with refinement base.

- A refinement base is a state machine constructed from a subtrace of the counterexample up to the non-executable statement block.

- Trace composition of two state machines A and B, retains a trace in A only if an equivalent trace without leading to an error state exists in B.
False alarm reduction for cycles

- **Cycles** in the composite model make infinite traces reaching error state.
- As refinement base remove one trace at a time, these traces can be refined infinitely.
- These traces can be removed if post-condition of the new trace already tested.

Counterexamples with cycles:

\[ \tau_0 = s_0 \quad s_1 \quad s_2 \quad s_4 \quad s_3 \]

\[ \tau_1 = s_0..s_{26} \quad s_0 \quad s_1 \quad s_2 \quad s_4 \quad s_3 \]

\[ \tau_2 = s_0..s_{26} \quad s_0..s_{26} \quad s_0 \quad s_1 \quad s_2 \quad s_4 \quad 3 \]

\ldots

extract post-condition

\neg (\text{post}' \rightarrow \text{post})

Unsatisfiable

SMT Solver
Experiments
Experiments 1 & 2

Objective
• Experiment 1-1: To evaluate effectiveness of property checking
• Experiment 1-2: To evaluate effectiveness of API-call constraint checking
• Experiment 2: To compare the verification accuracy of OiL-CEGAR

Applications
• TS1. Two example programs running on Erika OS (small scale / 3 tasks / tens of LoC)
• TS2. An object-follower and a platoon running on Lego Mindstorms NXT (realistic / 3~4 tasks / hundreds of LoC)
• TS3. Application programs running on Lego Mindstorms NXT (small scale / 2~3 tasks / ~87 LoC)
• TS4. Test programs from a commercial conformance test suite (complex / comes from domain experts / 5+tasks / hundreds of LoC)
Effectiveness of OiL-CEGAR: Property checking (2)

- TS2. An object-follower and a platoon running on Lego Mindstorms NXT (realistic / 3~4 tasks / hundreds of LoC)
- Assuming these are real vehicles, we verify properties that should be satisfied on real vehicles.
- There are 13 properties for
  - vehicle rollovers,
  - sharp turns,
  - sudden stops,
  - or liveness properties.
- Compared with testing method which is good at identifying presence of bugs

<table>
<thead>
<tr>
<th>prop.</th>
<th>prop. kind</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Boolean</td>
<td>During high-speed driving at 150 km, sudden turns shall not permitted, so a vehicle should not rollover.</td>
</tr>
<tr>
<td>p2</td>
<td>Boolean</td>
<td>During high-speed driving at 60 km, sudden turns shall not permitted, so a vehicle should not rollover.</td>
</tr>
<tr>
<td>p3</td>
<td>LTL</td>
<td>Do not decelerate beyond a certain force when making a sudden stop.</td>
</tr>
<tr>
<td>p4</td>
<td>Boolean</td>
<td>Steering beyond a certain level should not occur.</td>
</tr>
<tr>
<td>p5</td>
<td>Boolean</td>
<td>A value for the vehicle to go straight can be assigned to the motor.</td>
</tr>
<tr>
<td>p6</td>
<td>Boolean</td>
<td>It can receive input values from sensors.</td>
</tr>
<tr>
<td>p7</td>
<td>Monitor</td>
<td>When the control task receives a forward signal, it must move forward.</td>
</tr>
<tr>
<td>p8</td>
<td>Monitor</td>
<td>When the control task receives a backward signal, it must move backward.</td>
</tr>
<tr>
<td>p9</td>
<td>Boolean</td>
<td>Sensor input for moving forward from the vehicle in front can be transmitted to the sensor, and the vehicle control variable shall be updated.</td>
</tr>
<tr>
<td>p10</td>
<td>Assertion</td>
<td>When the vehicle moves forward and steers, the desired steering degree should be reflected in the motor considering the maximum motor output.</td>
</tr>
<tr>
<td>p11</td>
<td>Assertion</td>
<td>When the vehicle moves backward and steers, the desired steering degree should be reflected in the motor considering the maximum motor output.</td>
</tr>
<tr>
<td>p12</td>
<td>Assertion</td>
<td>When the vehicle moves forward, all wheels must rotate forward.</td>
</tr>
<tr>
<td>p13</td>
<td>Assertion</td>
<td>When the vehicle moves backward, all wheels must rotate backwards.</td>
</tr>
</tbody>
</table>
Effectiveness of OiL-CEGAR : Property checking (2)

<table>
<thead>
<tr>
<th>App.</th>
<th>prop.</th>
<th>Expected result</th>
<th>OiL-CEGAR verification</th>
<th>OiL-CEGAR exec. chk.</th>
<th>Total time(s)</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time(s)</td>
<td>Mem(MB)</td>
<td>len(trace)</td>
<td>#R</td>
<td>Time(s)</td>
</tr>
<tr>
<td>p1</td>
<td>satisfied</td>
<td>5,428</td>
<td>207</td>
<td>106 558</td>
<td>10,976</td>
<td>1201 18,000</td>
</tr>
<tr>
<td>p2</td>
<td>violated</td>
<td>6</td>
<td>398</td>
<td>36 4 15 268 37</td>
<td>violated</td>
<td>46</td>
</tr>
<tr>
<td>p3</td>
<td>violated</td>
<td>18</td>
<td>47</td>
<td>67 12 69</td>
<td>631 127</td>
<td>violated</td>
</tr>
<tr>
<td>p4</td>
<td>violated</td>
<td>1</td>
<td>36</td>
<td>16 2</td>
<td>violated</td>
<td>18 16 1,777 62</td>
</tr>
<tr>
<td>p5</td>
<td>violated</td>
<td>5</td>
<td>926</td>
<td>109 14</td>
<td>violated</td>
<td>92 109 1,344 44</td>
</tr>
<tr>
<td>p6</td>
<td>violated</td>
<td>2</td>
<td>51</td>
<td>47 147 10</td>
<td>violated</td>
<td>18 147 1,777 62</td>
</tr>
<tr>
<td>p7</td>
<td>violated</td>
<td>2</td>
<td>917</td>
<td>94 142 2</td>
<td>violated</td>
<td>92 94 1,646 46</td>
</tr>
<tr>
<td>p8</td>
<td>violated</td>
<td>6</td>
<td>641</td>
<td>76 113 20</td>
<td>violated</td>
<td>65 76 1,646 46</td>
</tr>
<tr>
<td>p9</td>
<td>violated</td>
<td>8</td>
<td>34</td>
<td>48 164 46</td>
<td>violated</td>
<td>34 48 1,646 46</td>
</tr>
<tr>
<td>p10</td>
<td>violated</td>
<td>5</td>
<td>654</td>
<td>63 122 2</td>
<td>violated</td>
<td>65 63 1,22 2</td>
</tr>
<tr>
<td>p11</td>
<td>violated</td>
<td>5</td>
<td>-</td>
<td>- 146 0</td>
<td>violated</td>
<td>- - 146 0</td>
</tr>
<tr>
<td>p12</td>
<td>violated</td>
<td>8</td>
<td>-</td>
<td>- 146 0</td>
<td>violated</td>
<td>- - 146 0</td>
</tr>
<tr>
<td>p13</td>
<td>violated</td>
<td>5</td>
<td>-</td>
<td>- 146 0</td>
<td>violated</td>
<td>- - 146 0</td>
</tr>
<tr>
<td>p1</td>
<td>violated</td>
<td>5</td>
<td>110</td>
<td>22 90 157</td>
<td>violated</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>violated</td>
<td>5</td>
<td>8</td>
<td>16 88 714</td>
<td>violated</td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td>violated</td>
<td>6</td>
<td>-</td>
<td>- 90 0</td>
<td>violated</td>
<td>- - 90 0</td>
</tr>
<tr>
<td>p4</td>
<td>violated</td>
<td>5</td>
<td>7</td>
<td>16 100 86</td>
<td>violation</td>
<td></td>
</tr>
<tr>
<td>p5</td>
<td>violated</td>
<td>7</td>
<td>13</td>
<td>17 101 957</td>
<td>violation</td>
<td></td>
</tr>
<tr>
<td>p6</td>
<td>violated</td>
<td>1</td>
<td>26</td>
<td>4 1</td>
<td>violated</td>
<td>1 4 54 790</td>
</tr>
<tr>
<td>p7</td>
<td>violated</td>
<td>604</td>
<td>90</td>
<td>66 64 348</td>
<td>violated</td>
<td>25 604 1130</td>
</tr>
<tr>
<td>p8</td>
<td>satisfied</td>
<td>13,875</td>
<td>261</td>
<td>106 588</td>
<td>2414</td>
<td>337 18,000</td>
</tr>
<tr>
<td>p9</td>
<td>violated</td>
<td>1</td>
<td>29</td>
<td>6 42</td>
<td>violated</td>
<td>1 29 54 58</td>
</tr>
<tr>
<td>p10</td>
<td>violated</td>
<td>31</td>
<td>46</td>
<td>32 11 15 114 78</td>
<td>violated</td>
<td>1 114 115 329</td>
</tr>
<tr>
<td>p11</td>
<td>violated</td>
<td>11,739</td>
<td>255</td>
<td>116 808</td>
<td>4087</td>
<td>342 18,000</td>
</tr>
<tr>
<td>p12</td>
<td>violated</td>
<td>11,291</td>
<td>215</td>
<td>116 909</td>
<td>4250</td>
<td>319 18,000</td>
</tr>
<tr>
<td>p13</td>
<td>violated</td>
<td>11,435</td>
<td>214</td>
<td>67 904</td>
<td>4137</td>
<td>298 18,000</td>
</tr>
</tbody>
</table>

OiL-CEGAR

- Founds all the property violations while testing cannot find 3 violations (testing finds 86% of prop. violation)
- But two method cannot verify the 5 properties (expected to be satisfied)
- OiL-CEGAR sometimes finds property violations faster than testing method

Test environment:
- Ubuntu Linux-based machine with a 3.3Ghz Intel Xeon Gold 6234 CPU and 192 GB of memory.
- NuSMV version 2.6.0 with dynamic variable reordering & cone-of-influence reduction.
- CBMC version 5.10 for executability checking

STAAR Workshop 2022
Conclusion

- This thesis proposed a model checking technique for the verification of multitasking embedded applications.
  - Model checking is applied to the formal OS model and the abstraction model.
  - Executability of a trace was checked on the application code with a mini-OS.
  - In model refinements, a refinement base is introduced to remove false alarm traces.

- TORCHE is accurate and efficient for verifying multitasking embedded applications.
  - TORCHE is accurate as it includes not only formal operating system but also, application program code.
  - TORCHE is also efficient as it abstracts applications and operating systems in model checking and executability checking, one at a time.
Future work

- Improving performance
  - Reuse of NuSMV checking data after refinement
  - Configuration slicing
  - Removing missed true alarm
  - Improving executability checking

- Reducing false alarms
  - Relative timing
  - Hardware abstraction
  - Use of non-deterministic interrupts
  - Use of finite counterexample trace

- Infinite refinements
  - Verification over fixed-size memory
  - Verification of infinite spaced program

- Support for more platforms
  - Support other OS and platforms
  - Support general OS such as Linux or Windows...
Thank you!!