Specification, Verification, and Synthesis using Extended State Machines with Callbacks

Farhaan Fowze and Tuba Yavuz

University of Florida

MEMOCODE 2016, Kanpur, India
Overview

1. Problem
   - A Study on Linux Device Drivers
   - Example: Linux usbkbd driver

2. Approach
   - Modeling: State Machines with Callbacks (SMACK)
   - Formal Semantics
   - Synchronization Synthesis

3. Conclusion
   - Related Work
   - Summary & Future Work
Software Reuse is Useful *but* ..

- Modern software is built with extensibility as a design goal
- Programming models with callback mechanism
  - Linux kernel
  - Android Framework
  - Robotic Operating System
- + New software can be developed with less effort
- - Debugging becomes difficult as the control-flow becomes implicit
  - Especially when coupled with concurrency!
A Study on Linux Device Driver Race Conditions

- 88 drivers from 44 device classes
- Developed by programmers from 45 different companies/organizations
- Patches incorporated into the Linux stable kernel source tree

<table>
<thead>
<tr>
<th>Race Condition Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Lack of locking</td>
<td>22</td>
</tr>
<tr>
<td>b) Inconsistent locking</td>
<td>18</td>
</tr>
<tr>
<td>c) Insufficient locking</td>
<td>9</td>
</tr>
<tr>
<td>d) Premature resource allocation/registration</td>
<td>14</td>
</tr>
<tr>
<td>e) Late resource deallocation/deregistration</td>
<td>4</td>
</tr>
<tr>
<td>f) Other</td>
<td>21</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>88</strong></td>
</tr>
</tbody>
</table>

Farhaan Fowze and Tuba Yavuz (UF)
Running Example: USB keyboard driver (usbkbd)

- Receives keyboard events and passes to the input layer
  - `usbkbd_irq` processes keyboard events including LED related keys, e.g., CAPSLOCK, and sends LED commands to the device
- Sends control commands to turn on/off LEDs
  - `usbkbd_led` receives acknowledgement of performed LED commands
- **Complication:** Both `usbkbd_irq` and `usbkbd_led` may send LED commands to the device and need to synchronize
  - Submission of LED commands in `usbkbd_irq` is implicit due to a callback function!
Example for the Implicit Control-Flow - USB keyboard driver

USB KEYBOARD DRIVER

external node

usb_kbd_open
usb_kbd_led
usb_kbd_event
usb_kbd_irq

usb_submit_urb
input_report_key

external node

USB CORE LAYER

usb_kbd_probe
usb_kbd_open
usb_kbd_irq

usb_kbd_event

INPUT LAYER

input_register_device
input_report_key
input_event
input_handle_event

Static Call Graph

Run-time Dependencies
Dealing with Concurrency at the Model Level

- Model the concurrent components of a software system making the programming model visible
- Verify the formal semantics of the model for correctness including race and deadlock freedom
- Synthesize correct concurrency
Overview

1 Problem

2 Approach
   - Modeling: State Machines with Callbacks (SMACK)
   - Formal Semantics
   - Synchronization Synthesis

3 Conclusion
State Machines with Callbacks (SMACK)

- Extended state machine formalism
  - Separation bw control locations and the states over extended variables
  - Hierarchical state machines

- Modeling of callbacks as state machines
  - Synchronous: Embedding an instance
    - Models function calls
  - Asynchronous: Instantiating an instance
    - Models creation of a thread of execution

**Goal:** To provide

- A modeling formalism that can express the dependencies among software components explicitly.
- An associated formal semantics to provide automated analysis for concurrency bugs
Modeling simplified usbkbd with SMACK
// Model for the driver
module USBKBD
    uses InputLayer, SpinLock;
    sync {usb_kbd_probe(T), usb_kbd_disconnect,
            usb_kbd_open, usb_kbd_close, usb_kbd_event(T)};
    async {usb_kbd_irq, usb_kbd_led};
    ...
end module

// Model for the Input Layer of the kernel
module InputLayer
    uses SpinLock;
    sync {input_event(T), input_register_device(T)};
    ...
end module
SM: input_event(dev: input_dev)

T : {init->ih_generic, ih_generic->lr_exit};

where init => dev.event_lock.acquire,
lr_exit => dev.event_lock.release;

end SM
A SMACK Model for usb_kbd_irq

SM: (2) usb_kbd_irq()

T: {init->kbd_event, kbd_event->submit_urb,
    submit_urb->exit};

[init->kbd_event] (guard: r new and r old);

[kbd_event->submit_urb] (update: w new, r old);

[submit_urb->exit] (update:@sync(usb_kbd_irq));

where  kbd_event =>
    InputLayer.input_event[usb_kbd_event/ih_generic];
end SM
State machines with generic and mapped states

Mapping of states: `kbd_event` $\rightarrow$ `input_event`, `init` $\rightarrow$ `acquire`, `lr_exit` $\rightarrow$ `release`.

Binding generic states: `ih_generic` $\rightarrow$ `usb_kbd_event`.
Mapping of states: `kbd_event` $\rightarrow$ `input_event`, `init` $\rightarrow$ `acquire`, `lr_exit` $\rightarrow$ `release`.

Binding generic states: `ih_generic` $\leftrightarrow$ `usb_kbd_event`.
Asynchronous Instantiation
Overview

1 Problem

2 Approach
   - Modeling: State Machines with Callbacks (SMACK)
   - Formal Semantics
   - Synchronization Synthesis

3 Conclusion
Semantics of the Main Module

INSTANTIATING THE MAIN MODULE

\[[\mathcal{M}]\] ≡ \[[[S\mathcal{M}^S_1] \ | \ ... \ | \ [S\mathcal{M}^S_m] \ | [S\mathcal{M}^A_1] \ | \ ... \ | \ [S\mathcal{M}^A_n]]\]

where

- \([S\mathcal{M}^S_i]\): Synchronous state machine
  - One of the top synchronous state machines of the main module
- \([S\mathcal{M}^A_j]\): Asynchronous state machine
  - One of the state machines instantiated by at least one of the flattened top synchronous machines of the main module
- \(\mid\mid\): Asynchronous composition
Semantics of a Synchronous State Machine

\[ [[\mathcal{SM}^S]] \equiv (S, I, R) \text{ where} \]

\[ S \equiv B^\lceil \log(|\mathcal{SM}.S_{sm}|) \rceil \times \mathcal{SM}.S_e, \ I \equiv I_e \land \bigvee_{s \in \mathcal{SM}.I_{sm}} pc_{\mathcal{SM}} = s, \]

\[ R \equiv \bigvee_{(s_1, s_2) \in \mathcal{SM}.R_{sm}} pc_{\mathcal{SM}} = s_1 \land pc'_{\mathcal{SM}} = s_2 \land \mathcal{SM}.G_e(s_1, s_2) \land \mathcal{SM}.U_e(s_1, s_2) \land IDENTITY(Unchanged) \]  

- Synchronous state machines are represented concretely, i.e., a concrete control state variable per instance.
- Transitions can be executed if the state machine is enabled, i.e., \( en_{\mathcal{SM}s} = true \).
- \#nable (SM): \( en'_{\mathcal{SM}} = true \), $isable (SM): \( en'_{\mathcal{SM}} = false \).
[[S.M^A]] ≡ (S, l, R) where

\[ S \equiv \bigwedge_{s \in S.M.S_{sm}} c_s^{S.M} \geq 0 \land S.M.S_e, \ l \equiv \bigwedge_{s \in S.M.S_{sm}} c_s^{S.M} = 0 \land S.M.l_e \]

\[ R \equiv \bigvee_{(s_1, s_2) \in S.M.S_{sm}} S.M.G_e(s_1, s_2) \land S.M.U_e(s_1, s_2) \land c_{s_1}^{S.M} > 0 \land c_{s_1}^{S.M'} = c_{s_1}^{S.M} - 1 \land c_{s_2}^{S.M'} = c_{s_2}^{S.M} + 1 \]  

(2)

- Asynchronous state machines are abstracted using counting abstraction
- \( c_{s_1}^{S.M} \) keeps track of the number instances of state machine \( SM \) in state \( s_1 \)
- \( @sync(SM) : \sum_{s \in S.M.l_{sm}} c_s^{S.M'} = \sum_{s \in S.M.l_{sm}} c_s^{S.M} + 1 \)
Overview

1. Problem

2. Approach
   - Modeling: State Machines with Callbacks (SMACK)
   - Formal Semantics
   - Synchronization Synthesis

3. Conclusion
Counter-example Guided Synchronization Synthesis

- **Type 1 Race:** Involves abstract operations alloc, init, and dealloc and relies on existence of #nable and $isable
- **Type 2 Race:** Otherwise, i.e., no #nable and $isable or among read/write or write/write operations
  - Operations can be concrete, e.g., $CHANGE, or abstract, e.g., w leds
- Safety property for potential race conditions between $s_1$ of $SM_1$ and $s_2$ of $SM_2$: Both $SM_1$ and $SM_2$ are synchronous:
  \[
  \text{invariant}(-((pc_{SM_1} = s_1 \land pc_{SM_2} = s_2 \land g_1 \land g_2)), \quad (3)
  \]
  If $SM_1$ is asynchronous:
  \[
  \text{invariant}(-((c_{SM_1}^{S} > 0 \land pc_{SM_2} = s_2 \land g_1 \land g_2)), \quad (4)
  \]
  If $SM_1$ and $SM_2$ are the same asynchronous state machine:
  \[
  \text{invariant}(-((c_{SM_1}^{S} > 1 \land g_1)), \quad (5)
  \]
Type 1 Race bw usb_kbd_probe and usb_kbd_irq
Fixing Type 1 Races

- The fix involves moving the alloc, init (dealloc) operation before (after) the enabling (disabling) action of the state machine that performs the read/write.
- The challenge is to identify the right level in the hierarchy of state machines the alloc, init, or dealloc operation should be moved to.
Establishing happens-before via `#enable`
Type 2 Race bw different instantiations of usb_kbd_led
Fixing Type 2 Races

- The fix involves enclosing the racy transition with acquire and release operations of a lock model.
- Creates equivalence classes of variables based on data-flow and control-flow dependency.
  - Two variables belong to the same equivalence class if they appear in the guard or update of the same transition.
  - Each equivalence class is associated with at most one lock.
- The algorithm checks if there is a candidate lock that can be used.
  - The lock held closest to the race location is considered as a candidate.
- Introduces a new lock if there is no candidate or the candidate does not satisfy the minimum context priority requirement.
Establishing happens-before via locking
Correctness

- **Race-freedom**: Due to undecidability of infinite-state model checking, the synthesis algorithm may not terminate. However, if it terminates the generated model is guaranteed to be race free.
  - All types of races are considered.
  - Comprehensiveness of the race properties
  - Updating the set of properties as the model gets updated.

- **Deadlock freedom**: If the original model does not have a deadlock scenario then the synthesis does not introduce new deadlock scenarios.
  - Each equivalence class is assigned a unique lock.
  - Acquire and release of an assigned lock does not enclose acquire/release of another lock.
Overview

1. Problem

2. Approach

3. Conclusion
   - Related Work
   - Summary & Future Work
Related Work

- **Concurrency Synthesis:**
  - Synthesis at the code level
  - Various interpretation of correctness: user provided atomicity, equivalence with sequential behavior, relative order of events under non-preemptive scheduling
  - Learning from bad/counter examples, good examples, and patterns
  - Guarantee for deadlock freedom
  - Number of threads

- **Modeling Asynchronous Events**
  - P is also a state machine based modeling language. Its deferred events models asynchronous behaviors. The goal is verifying responsiveness.
Summary & Future Work

- A state machine based modeling formalism that can make synchronous and asynchronous calls of the programming model visible.
- Goal is to provide automated verification support for getting shared memory concurrency right.
- Automated concurrency synthesis for unbounded number of threads.

**Future Work:**

- Incorporation of more synchronization primitives
- A mechanism to specify constraints introduced by the programming model
- Optimizations
THANK YOU