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

Farhaan Fowze
ECE Department
University of Florida
Email: farhaan104@ufl.edu

Tuba Yavuz
ECE Department
University of Florida
Email: tuba@ece.ufl.edu

***Abstract*—In this paper we extend state machine diagrams with a programming concept that is highly utilized in real software: the callback mechanism. A callback is a way to interact with a library and can be instantiated in the form of synchronous or asynchronous mode. Using callbacks speeds up software development at the expense of complicating program comprehension. Introducing the callback concept to a modeling formalism preserves structural similarity between the model and the implementation. This paper presents a formal semantics for this extended formalism to make it amenable to formal verification and concurrency synthesis and to help developers avoid implementation mistakes such as race conditions and deadlocks. We report specification, verification, and synthesis case studies on a device driver.**

I. Introduction

Concurrency is a difficult concept for programmers and testers alike due to its non-deterministic nature and the requirement for an understanding of the global view of the behavior. Programming models that enable software reuse introduces additional complications for reasoning about concurrency related bugs. An example programming model is the callback mechanism: a software component provides custom functionality in the form of callback functions that get registered with a software library. These callback functions serve as entry points to the software component and may be executed synchronously via a chain of calls through the software library or asynchronously based on the events from the environment. The challenge here is to get an accurate picture of the control-flow which is implicit due to the callback mechanism.

Linux operating system is a great example for software that uses callback mechanism extensively, especially for device drivers. We performed a case study on Linux device drivers to find root causes of race conditions. We have selected a sample of 88 device driver bug reports with patches incorporated to the the stable Linux kernel source tree. The sample bugs involve 44 device classes (directories under drivers directory on the kernel source tree), developers (authors and committers) from 45 companies/organizations, and a variety of failure types including system crash, kernel oops/panic, kernel bug warning, performance degradation, and hang/freeze.

Table I categorizes the bugs into six classes and reports the number of bugs that fall into each category. Category a) represents the cases where shared data is not protected by locks at all, which implies an unawareness of the possibility of concurrent accesses. Category b) represents cases where some of the accesses to shared data are not protected with locks, which may imply an inadequate understanding of data races or an oversight. Category c) represents cases where the locking mechanism does not protect against all possible concurrent contexts which implies an incomplete knowledge of possible concurrent contexts such as timers and hardware interrupts. Category d) represents cases where the callback functions become active before the data structures that are accessed in these functions are properly allocated and/or initialized. Category e) represents cases where the callback functions stay active even after the data structures that are accessed in these functions get destroyed/deallocated. Categories d) and e) imply an unawareness of the actual timeline of the callback functions becoming or staying active. Category f) represents a variety of cases such as making data unintentionally global, races with the hardware, etc.

TABLE I
Categories of root causes of race conditions manifested in a sample of Linux device drivers.

Race Condition Type	**Count**
a) Lack of locking	22
b) Inconsistent locking	18
c) Insufficient locking	9
d) Premature resource allocation/registration	14
e) Late resource deallocation/deregistration	4
f) Other	21
Total	88

We think that at least 55% of the race conditions reported in Table I, categories a, c, d, and e, can be avoided by making the concurrent contexts explicit to the developers. An important aspect of this is making the implicit control-flow dependencies introduced by the callback mechanism explicit. To that end, in this paper, we present a formal modeling approach that extends state machine diagrams with the notion of callbacks and synchronous and asynchronous entry points. The proposed approach preserves structural similarity between the model and the implementation. We support the modeling formalism with a semantics that makes it amenable to formal verification and automated synthesis of concurrency. This can help developers avoid implementation mistakes such as race conditions and deadlocks.

The rest of the paper is organized as follows. Section II presents a motivating example on a USB keyboard driver and highlights the difficulties and potential bugs that can exist. Section III presents the modeling formalism, which is associated with a formal semantics in Section IV. Section V presents the synthesis algorithms. Section VI discusses related work and Section VII concludes with future directions.

II. An Example

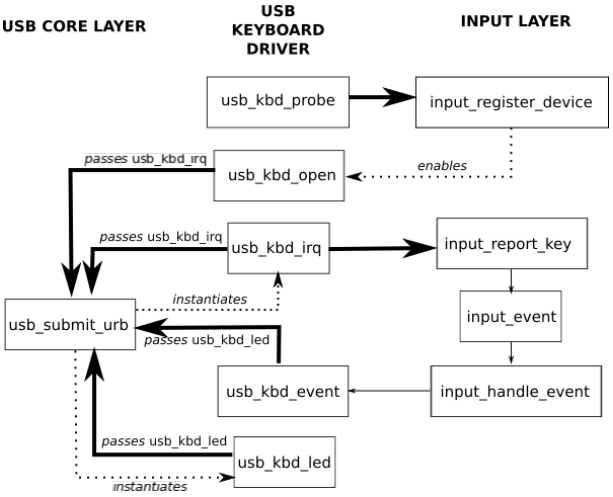

Fig. 1. A run-time view of the interactions between the USB keyboard driver and the USB Core and Input layers. Dotted lines are the asynchronous instantiation or enabling edges. Thick lines exist in the LLVM generated call graph.

In this section, we will discuss a concrete example for a software component that involves callback functions to illustrate the difficulties and the potential problems. We will use this as a running example to demonstrate various aspects of the modeling formalism as well as the output of the concurrency synthesis algorithm.

The example we consider is from the Linux operating system, which structures device drivers into layers to promote modularity and reuse. A device driver developer then needs to identify the specific layers his/her code will need to interact with and provide the necessary callback functions that will be registered with those layers. For instance, a driver for a USB keyboard would need to interact with the Input layer to pass input events to the user processes and with the USB Core layer to communicate with the keyboard.

Figure 1 shows an accurate call graph of Linux USB keyboard driver (`usbkbd` [2]). Here, `usb_kbd_probe` is called by USB Core when the device gets plugged in. The role of this function is to decide whether it can handle the presented device and, if so, to set up various data structures that the driver will use to communicate with the device. Since the driver needs to pass the keyboard events to the Input layer, it registers its input related callback functions by calling `input_register_device` function of the Input layer. After this step, the keyboard device can be accessed by the user processes for input. To simplify the discussion we skip the involvement of the Virtual File System layer and note that `usb_kbd_open` function is called by the Input layer only once regardless of how many processes has opened the device file for input. It basically initiates sensing of the keyboard events by submitting a USB Request Block (URB) to the USB core, which is achieved by calling USB Core's `usb_submit_urb` function and passing the URB as a parameter to this function. The URB stores information such as the specific endpoint of the device that will be polled by the USB core, the frequency of polling for interrupt type endpoints, and the callback function that handles the data received from the device[1]. The completion of URBs is an asynchronous process and keyboard driver will be informed on the status of the URB through the relevant callback function. In this example, the callback function for handling keyboard events is `usb_kbd_irq` function and it may potentially run in *parallel* with `usb_kbd_open` if `usb_kbd_open` has not returned yet by the time that USB Core calls `usb_kbd_irq` upon receiving the data from the keyboard. However, since the pointer to this function is stored in the URB, the fact that `usb_kbd_open` initiates `usb_kbd_irq` is implicit and is mentioned in the call graph through annotation of the relevant edge with the asynchronous function name.

Once `usb_kbd_irq` function starts running, it will process the data to see if there is any new key events, and if so, it will report it to the Input layer, and submit a new URB so that new input events can be received and processed, and so on. So another implicit dependency exists between consecutive instantiations of `usb_kbd_irq` as they may potentially run in parallel. However, the fact that submission of the URB is the last thing performed in the function eliminates that possibility. However, a feasible concurrency scenario is created when it calls Input layer's `input_report_key` function to report the key events. As the call graph shows, `input_report_key` ends up calling `input_handle_event`, which calls the device specific callback function for the relevant input event. It turns out that if the event is an LED event then USB keyboard driver's `usb_kbd_event` is called. So there is an implicit synchronous dependency as `usb_kbd_irq` ends up calling `usb_kbd_event`. This function submits a URB to turn on/off the led on the keyboard. This time the callback function stored in the URB is `usb_kbd_led` function, which means `usb_kbd_irq` may asynchronously instantiate `usb_kbd_led` and, hence, they may run concurrently. What complicates things even further is that once `usb_kbd_led` is called by the USB Core, it also checks whether there has been a new LED related event and, if so, submits a URB to turn on/off the led.

We have generated a call graph of the USB keyboard driver using the LLVM compiler framework [12], which could only generate the thick lines in Figure 1 and missed the regular lines and the dashed lines due to implicit dependencies

[1]Initializations of URBs are generally done in the probe callback function and used in the other entry points of the driver.

through synchronous callback function usb_kbd_event and asynchronous callback functions usb_kbd_irq and usb_kbd_led, each of which will run in an atomic context and in a separate thread of execution . It is important to note that the call graph generated by LLVM misses the implicit dependency between usb_kbd_irq and usb_kbd_event that exists through the input layer.

So a potential conflict exists between functions usb_kbd_event and usb_kbd_led as they may run in parallel and both may submit LED related commands to the keyboard. The reason for running in parallel is due to the possibility of usb_kbd_irq and usb_kbd_led running in parallel. Since submitting LED commands more than necessary would cause an incorrect status on the keyboard [1], these two functions need some type of synchronization so that if a LED command has already been submitted no extra LEDs get submitted. In the rest of the paper, we consider a buggy model for the keyboard to explain how such bugs can be detected at the design level and how correct concurrency can be synthesized using the presented approach.

III. Modeling Language

```
module SpinLock
var l: bool;
SM:(2)  acquire()
    T : {init->init, init->exit};
    [init->init] (guard: l=true);
    [init->exit] (guard: l=false, update: l:=true);
end SM
SM:(2) release()
   T : {init->exit};
   [init->exit] (guard: l := false);
end SM
end module
module InputLayer
uses SpinLock;
sync {input_event(T), input_register_device(T)};
type input_dev {event_lock: SpinLock, vals: dynamic,
                          open: SM, close: SM};
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
SM: input_register_device(dev: input_dev)
 T: {init->alloc, alloc->register_exit};
 [init->alloc] (update: alloc dev.vals);
 [alloc->register_exit] (update: #nable(dev.open),
#nable(dev.close));
end module
...
end module
```

Fig. 2. A model of a spin lock with acquire and release operations and a partial model of the input layer defining three state machines.

In this section, we will use the usbkbd driver to demonstrate how the proposed modeling approach can be used for real software. We will first provide an informal introduction to the state machines with callback mechanism in this section and defer the formal definitions to Section IV.

```
module USBKBD
uses InputLayer, SpinLock;
sync {usb_kbd_probe(T), usb_kbd_disconnect,
usb_kbd_open, usb_kbd_close};
async {usb_kbd_irq, usb_kbd_led};
var CHANGE: bool;
var EV_LED: bool;
type usb_kbd {new: dynamic, old: static,
newleds: static, leds: dynamic,
        led_urb_submitted: bool,
dev: InputLayer.input_dev[usb_kbd_open/open,
usb_kbd_close/close]};
SM: usb_kbd_probe()
    T: {init->register_input,
         register_input->alloc_usb};
    [register_input->alloc_usb] (update: alloc new);
    where register_input =>
       InputLayer.input_register_device(dev);
end SM
SM: usb_kbd_open()
    T : {init->submit_urb, submit_urb->exit};
    [submit_urb->exit] (update:@sync(usb_kbd_irq));
end SM
SM: usb_kbd_event()
  T: {init->r_exit, init->la, la->lr_exit,
  la->submit_urb, submit_urb->lr_exit};
  [init->r_exit]  (guard: !EV_LED);
  [init->la] (guard: EV_LED);
  [la->lr_exit] (guard: led_urb_submitted ||
      (!CHANGE and r leds and r newleds));
  [la->submit_urb] (guard: !led_urb_submitted and
      CHANGE and r leds and r newleds,
      update:w leds, r newleds);
  [submit_urb->lr_exit](update: @sync(usb_kbd_led));
end SM
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
SM:(2) usb_kbd_led()
   T: {init->la, la->lr1_exit, la->update_led,
    update_led->submit_urb, submit_urb->lr2_exit};
   [la->lr1_exit] (guard: !CHANGE and r leds and
    r newleds, update: led_urb_submitted := false);
   [la->update_led] (guard: CHANGE and r leds and
    r newleds, update: w leds, r newleds);
   [submit_urb->lr2_exit]
               (update: @sync(usb_kbd_led));
end SM
...// other SMs
endmodule
```

Fig. 3. A partial model of the usbkbd driver.

A state machine is defined in terms of a set of finite control states and the transitions among them. When a state machine gets instantiated the control starts at one of the initial states and the state machine stays alive until it reaches an exit state. It is possible that a state machine is a reactive one, so it either does not have any exit states or under normal operating conditions does not reach an exit state. In an extended state machine state variables are used to describe the triggering conditions or *guards* as well as the *updates* on those state variables. A hierarchical state machine can be formed by embedding a state

machine sm_1 in another state machine sm_2 by mapping some state in sm_2 to sm_1.

We model functions using extended state machines. We group state machines into `modules` to model software components. For instance, the Input layer and the USB keyboard driver are modeled as modules in Figures 2 and 3, respectively. This paper introduces two new concepts to the extended state machine formalism 1) **generic states** and 2) the distinction between **synchronous** and **asynchronous** state machines.

Generic states help us model the callback mechanism as a function calling a callback function is performing a generic task that can only be interpreted once a specific function is bound to the callback function. As an example, the Input layer functions `input_report_key`, `input_event`, and `input_handle_event` shown in Figure 1 need to be modeled with generic states as they may end up calling the `event` callback function, which is defined as a function pointer in `input_dev` structure. To simplify the discussion, we will model only the `input_event` function out of the mentioned three functions.

Figure 2 shows a model for the Input layer, `module InputLayer`, consisting of two state machines: `input_event` and `input_register_device`. The states of state machine `input_event` is implicitly declared in the transition declaration using the `T:` token followed by a set of transitions. Each transition is represented in the form of `s1 -> s2`, which means existence of a transition from control state `s1` to control state `s2`. The states for `input_event` are `init`, `ih_generic`, and `lr_exit`. Transitions whose behavior are explicitly defined uses the `[s1->s2] (guard: ge, update: ue)` syntax in its most general form. Here ge denotes the guard expression and ue denotes the update expression. We skip the `guard: ge` part if ge is true. Similarly, we skip the `update: ue` part if the transition does not cause any side effect other than changing the control state. If both the guard and the update parts can be skipped, the whole transition definition is skipped.

The subscript `generic` declares that the state is generic. Generic states must be bound to a concrete state machine when the defining state machine gets instantiated. Figure 3 shows a partial model of the `usbkbd` driver. The state machine `usb_kbd_irq` instantiates the `input_event` state machine of the Input layer by mapping 1) the local state `kbd_event` to `input_event` in the `where ... => ...` *binding construct* and 2) the generic state `ih_generic` to `usb_kbd_event` state machine using the *renaming construct* `[new/old]`.

Going back to the second modeling concept introduced into the state machine formalism, marking state machines as synchronous or asynchronous helps us model the control-flow dependencies *precisely and explicitly*. We use the `sync { }` and `async { }` constructs to declare state machines as synchronous and asynchronous, respectively. As an example in Figure 3, `usb_kbd_probe` and `usb_kbd_open` are marked as synchronous whereas `usb_kbd_irq` and `usb_kbd_led` are marked as asynchronous. Synchronous state machines can be enabled by default, which is denoted by `(T)` appended to the name of the state machine, e.g., `usb_kbd_probe(T)`, or it can be enabled explicitly during execution using the `#nable` construct, e.g., `input_register_device` enabling `dev.open` as shown in Figure 2. Once a synchronous state machine is enabled its transitions become enabled, i.e., when the control reaches the control state and the guard evaluates to true, the transition can be executed. An asynchronous state machine gets instantiated explicitly via the `@sync` construct. Each instantiation of an asynchronous state machine represents a new thread of execution.

IV. Formal Semantics

INSTANTIATING A STATE MACHINE

Synch.: $[\![[\mathcal{SM}^S]]\!] \equiv (S, I, R)$ where

$$S \equiv \mathcal{B}^{\lceil log(|\mathcal{SM}.S_{sm}|)\rceil} \times \mathcal{SM}.S_e,$$

$$I \equiv I_e \ \wedge \bigvee_{s \in \mathcal{SM}.I_{sm}} pc_{\mathcal{SM}} = s,$$

$$R \equiv \bigvee_{(s_1,s_2)\in \mathcal{SM}.R_{sm}} pc_{\mathcal{SM}} = s_1 \ \wedge \ pc'_{\mathcal{SM}} = s_2 \wedge$$
$$\mathcal{SM}.G'_e(s_1, s_2) \wedge \mathcal{SM}.U_e(s_1, s_2) \wedge$$
$$\mathcal{IDENTITY}(Unchanged)$$

Asynch.: $[\![[\mathcal{SM}^A]]\!] \equiv (S, I, R)$ where

$$S \equiv \bigwedge_{s \in \mathcal{SM}.S_{sm}} c_s^{\mathcal{SM}} \geq 0 \ \wedge \ \mathcal{SM}.S_e$$

$$I \equiv \bigwedge_{s \in \mathcal{SM}.S_{sm}} c_s^{\mathcal{SM}} = 0 \ \wedge \ \mathcal{SM}.I_e$$

$$R \equiv \bigvee_{(s_1,s_2)\in \mathcal{SM}.S_{sm}} \mathcal{SM}.G_e(s_1, s_2) \ \wedge \ \mathcal{SM}.U_e(s_1, s_2)$$
$$\wedge \ c_{s_1}^{\mathcal{SM}} > 0 \ \wedge \ c_{s_1}^{\mathcal{SM}'} = c_{s_1}^{\mathcal{SM}} - 1 \ \wedge \ c_{s_2}^{\mathcal{SM}'} = c_{s_2}^{\mathcal{SM}} + 1$$

PARALLEL COMPOSITION

$$[\![[\mathcal{SM}_1]|||[\mathcal{SM}_2]]\!] \equiv ([\mathcal{SM}_1].S \cap [\mathcal{SM}_2].S,$$
$$[\mathcal{SM}_1].I \cap [\mathcal{SM}_2].I, [\mathcal{SM}_1].R \cup [\mathcal{SM}_2].R)$$

INSTANTIATING THE MAIN MODULE

$$[\![[\mathcal{M}]]\!] \equiv [\![[\mathcal{SM}_1^S] \ || \ ... \ || \ [\mathcal{SM}_m^S] \ |||[\mathcal{SM}_1^A] \ || \ ... \ || \ [\mathcal{SM}_n^A]]\!]$$

Fig. 5. Semantics of state machine and module instantiations in terms of Kripke structures.

A module, $\mathcal{M}$, is defined as a set of synchronous, asynchronous, and helper state machines. A state machine, $\mathcal{SM}$, is defined in terms of a tuple $(V, S_{sm}, S_e, I_{sm}, I_e, E_{sm}, R_{sm}, G_e, U_e)$, where V denotes the state variables defined by the module that defines $\mathcal{SM}$, S_{sm} denotes the control states, S_e denotes the state space defined by the set of state variables, I_{sm} denotes the set of initial control states, I_e denotes the set of initial states defined by the state variables, E_{sm} denotes the set of exit states, R_{sm} denotes the transitions among the control states, G_e denotes a function that maps transitions of the state machine to the respective guard formula, and U_e denotes a function

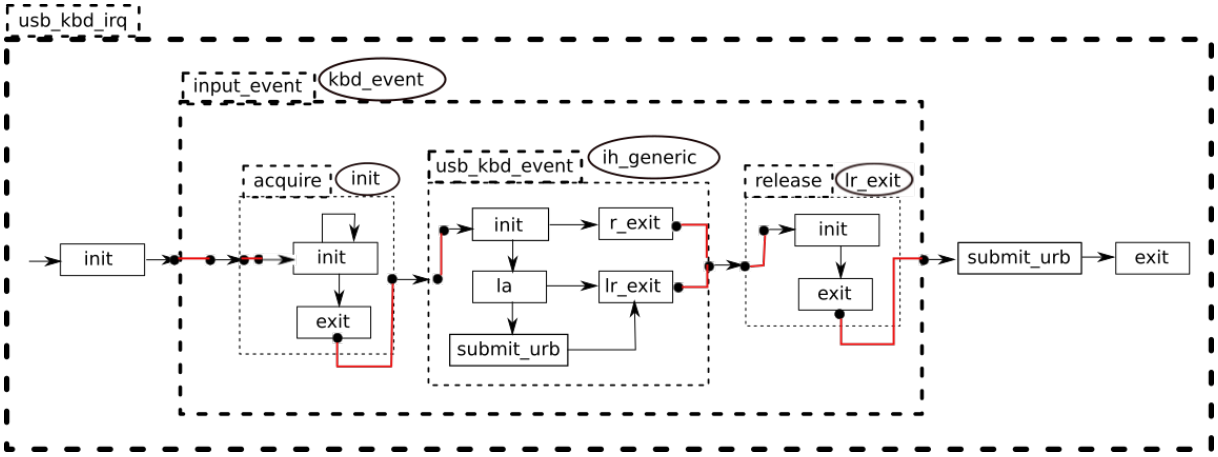

Fig. 4. Flattened version of `usb_kbd_irq` state machine. Dashed boxes, ovals, and red lines connecting the circles represent state machines, mapped states, and redirected incoming and outgoing transitions of the mapped states, respectively.

that maps transitions of the state machine to the respective set of update statements.

In what follows, we present the semantics in a top-down fashion. We denote instantiation operation using square brackets: $[\mathcal{M}]$ and $[\mathcal{SM}]$ represent instantiation of module $\mathcal{M}$ and state machine $\mathcal{SM}$, respectively. Figure 5 presents the semantics of an instantiation via a Kripke structure (S, I, R), where S denotes the set of states, I denotes the set of initial states, and R denotes the transition relation. Below we provide brief explanations.

a) Instantiating the main module: In our modeling formalism, the semantics is defined by instantiation of the `main` module, which is defined by parallel composition of the enabled synchronous state machines of the `main` module and all the asynchronous state machines that are reachable from the enabled synchronous state machines. The state variables that define the Kripke structure $[\mathcal{M}]$ consists of all the state variables of $\mathcal{M}$, the state variables of the modules directly or indirectly used by $\mathcal{M}$ and are reachable from synchronous or asynchronous entry points, and all the meta-variables, e.g., the program counter, $pc_{\mathcal{SM}}$, state machine $\mathcal{SM}$. As an example, consider the scenario defined by the `main` module shown in Figure 7. The state machine `userAction1` models plugging in the USB keyboard device and instantiation of `usb_kbd_probe` state machine of module `USBKBD`. State machines `userAction2` and `userAction3` model unplugging the keyboard and opening of the keyboard device node through the virtual file system, respectively. The semantics of the `main` module given in Figure 7 is represented by asynchronous composition of enabled synchronous state machines `userAction1`, `userAction2`, `userAction3`, and the reachable asynchronous state machines `usb_kbd_irq` and `usb_kbd_led`.

b) Parallel Composition: We use interleaving semantics of concurrency, e.g., only a single transition can be executed at a given time and all possible order of executions are considered. As an example, at a given time an enabled transition defined by `userAction1`, `userAction2`, `userAction3`, or one of the asynchronous entry points (`usb_kbd_irq` or `usb_kbd_led`) will be executed at a given point in time.

```
module main
  uses module USBKBD;
  sync {userAction1(T), userAction2(T),
   userAction3(T)};
  var canAccess: bool;
  SM:(1) userAction1()
     T:{init->exit}
     where init => USBKBD.usb_kbd_probe;
     [init->exit] (update:canAccess := true);
  end SM
  SM:(1) userAction2()
   T:{init->init, init->unplug_exit}
   where unplug_exit => USBKBD.usb_kbd_disconnect;
   [init->init] (guard: true);
   [init->unplug_exit] (guard: canAccess);
  end SM
  SM:(0) userAction3()
   T: {init->init, init->open_exit}
   where open => USBKBD.usb_kbd_open;
  end SM
endmodule
```

Fig. 7. A model of a scenario that executes the `USBKBD` driver.

c) Flattening a Hierarchical State Machine: The flattening process embeds a state machine for each mapped state and in a recursive way each embedded state machine is also flattened. Mapping states to state machines can be achieved using either the `where` construct or the generic state binding construct through renaming. Figure 6 provides an operational semantics for a hierarchical state machines that makes use of callbacks. The key concept is the *embedding operation*, $\mathcal{EMBED}$, which is used in defining the semantics of both constructs. As an example, Figure 4 shows how the state machine `usb_kbd_irq` is flattened.

d) Instantiating a State Machine for a Synchronous Entry Point: For a synchronous state machine, we keep an enumerated meta-variable for each control state. The state space, then, consists of valuations of the variables defining

STATE & TRANSITION DECLARATIONS

$$\frac{\mathcal{C}}{\langle \text{T:\{a->b, list\} body }, SM\rangle \longrightarrow \langle \text{T:\{list\} body}, SM'\rangle}$$

States: $SM'.S_{sm} = SM.S_{sm} \cup \{a, b\}$, Initial states: $\mathcal{C} \to a$ is an initial state : $SM'.I_{sm} = SM.I_{sm} \cup \{a\}$
$\mathcal{C} \to a$ is not an initial state : $SM'.I_{sm} = SM.I_{sm}$, Exit states: $\mathcal{C} \to b$ is an exit state : $SM'.E_{sm} = SM.E_{sm} \cup \{b\}$
$\mathcal{C} \to b$ is not an exit state : $SM'.E_{sm} = SM.E_{sm}$, Transitions: $SM'.R_{sm} = SM.R_{sm} \cup \{(a, b)\}$

TRANSITIONS

$$\frac{}{\langle v ::= expr, list, U\rangle \longrightarrow \langle list, U \wedge v' = expr\rangle}, \frac{}{\langle @sync(\mathcal{SM}), list, U\rangle \longrightarrow \langle list, U \wedge \sum_{s\in\mathcal{SM}.I_{sm}} c_s^{\mathcal{SM}'} = \sum_{s\in\mathcal{SM}.I_{sm}} c_s^{\mathcal{SM}} + 1\rangle},$$

$$\frac{}{\langle \#nable(\mathcal{SM}), list, U\rangle \longrightarrow \langle list, U \wedge en'_{\mathcal{SM}} = true\rangle}, \frac{}{\langle \$isable(\mathcal{SM}), list, U\rangle \longrightarrow \langle list, U \wedge en'_{\mathcal{SM}} = false\rangle}, \frac{o \in \{alloc, init, dealloc\}}{\langle o\ var, list, U\rangle \longrightarrow \langle list, U\rangle}$$

$$\frac{\langle \mathcal{U}, U\rangle}{\langle \text{[a->b] :(guard: } \mathcal{G}\text{, update: } \mathcal{U}\text{) body }, SM\rangle \longrightarrow \langle \text{body}, SM'\rangle}$$

Guards:$SM'.G_e = SM.G_e[(a, b) \mapsto \mathcal{G}]$, Updates:$SM'.U_e = SM.U_e[(a, b) \mapsto U]$

EMBEDDING

$$\frac{\mathcal{C},\ \langle \text{A}[list_1/list_2], SM_1\rangle}{\langle \text{where a=> A}[list_1/list_2],\ list_3, SM\rangle \longrightarrow \langle \text{where } list_3, SM' = \mathcal{EMBED}(SM, a, SM_1, C)\rangle}$$

States: $SM'.S_{sm} = SM.S_{sm} \cup SM_1.S_{sm} \setminus \{a\}$
Initial states: $\mathcal{C} \to a$ is an initial state : $SM'.I_{sm} = SM.I_{sm} \cup SM_1.I_{sm} \setminus \{a\}$, $\mathcal{C} \to a$ is not an initial state : $SM'.I_{sm} = SM.I_{sm}$
Exit states: $\mathcal{C} \to a$ is an exit state : $SM'.E_{sm} = SM.E_{sm} \cup SM_1.E_{sm} \setminus \{a\}$, $\mathcal{C} \to a$ is not an exit state : $SM'.E_{sm} = SM.E_{sm}$
Transitions: $SM'.R_{sm} = SM.R_{sm} \cup \{(c, b) \mid (c, a) \in R_{sm} \ \wedge\ b \in SM_1.I_{sm} \vee (a, b) \in R_{sm} \ \wedge\ c \in SM_1.E_{sm}\} \setminus \{(b, c) \mid a = b \vee c = a\}$
Guards: $SM'.G_e = SM.G_e[(b, c) \mapsto SM.G_e(d_1, d_2)$, where $d_1 = a \wedge d_2 = c \wedge (a, c) \in SM.R_{sm} \wedge b \in SM_1.E_{sm} \vee$
$d_2 = a \wedge b = d_1 \wedge (b, a) \in SM.R_{sm} \wedge c \in SM_1.I_{sm}]$, Updates: Same as Guards.

CALLBACK BINDING

$$\frac{\mathcal{C}, \langle \text{SM: A() } body_A \text{ end SM}, SM^A\rangle, a \in SM^A.S_{sm}, \langle \text{SM: B() } body_B \text{ end SM}, SM^B\rangle}{\langle \text{A[a, } list_1\text{/B, } list_2], SM\rangle \longrightarrow \langle \text{A}[list_1/list_2], SM' = \mathcal{EMBED}(SM, a, SM^B, C)\rangle}, \text{ (see definition of } \mathcal{EMBED} \text{ in EMBEDDING)}$$

INSERTING A TRANSITION, (for synthesis)

$$\frac{\text{SM: A() } body_A \text{ end SM} \Downarrow SM, s_1, s_2 \in SM.S_{sm}}{SM \oplus (s_1, g, u, s_2) \Downarrow SM'}, \text{ where}$$

States: $SM'.S_{sm} = S.S_{sm} \cup \{s_{new}\}$, Transitions: $SM'.R_{sm} = SM.R_{sm} \ \cup\ \{(s_1, s_{new}), (s_{new}, s_2)\} \setminus \{(s_1, s_2)\}$
Guards: $SM'.G_e = SM.G_e[(a, b) \mapsto \mathcal{G}$, where $a = s_1 \wedge b = s_{new} \wedge \mathcal{G} = SM.G_e(s_1, s_2) \vee a = s_{new} \wedge b = s_2 \wedge \mathcal{G} = g]$
Updates: Same as Guards.

Fig. 6. Operational semantics for extended state machines with callbacks formalism.

the control states and valuations of the state variables and any constraints described via the `restrict` keyword. The initial states describe the initial control states and the initial values of the state variables described via the `initial` keyword.

$\mathcal{IDENTITY}(Unchanged)$ keeps all the state and meta-variables that are not changed by the transition (s_1, s_2) the same in the next state. The abstract type operations, e.g., `alloc`, do not impact the semantics. However, as we will show in Section V, they will be instrumental in the generation of safety properties. We explain the meta-variable $c_s^{\mathcal{SM}}$ below.

e) Instantiating a State Machine for an Asynchronous Entry Point: Since an asynchronous entry point can be instantiated dynamically through the `@sync` construct, we use the counting abstraction technique [9] to represent such instantiations. The idea is to keep a counter, $c_s^{\mathcal{SM}}$, for every control location, s, and represent transitions of an asynchronous state machine by manipulating these counters.

V. Verification and Synthesis

In this section, we present our counter-example guided synthesis of concurrency for systems specified using state machines with the callback formalism. Developers can either come up with their solutions to synchronization, specify it as part of their models, and check for correctness. Alternatively, they can leave out the synchronization part and let the synthesis algorithm figure out all the necessary synchronization.

Inspired by the results of the study on Linux device driver bugs reported in Section I, we consider two types of race conditions: 1) those involving allocation, initialization, and deallocation operations, 2) those involving read and write operations. It turns out that each type of race condition requires a different solution. Fixing of type 1 requires reordering of the operations with respect to enabling/disabling operations of the state machines that participate in the race. On the other hand, fixing of type 2 requires using locks.

Figure 8 presents our top-level verification and synthe-

1: **VerifyAndSynthesizeSynchronization**($\mathcal{M}$: Module): (Module, boolean)
2: **global** $\mathcal{M}.protBy : \mathcal{M}.V \rightarrow \mathcal{M}.Locks$, $\mathcal{M}.protBy \leftarrow \lambda d.null$
3: **global** $\mathcal{M}.dep \leftarrow$ ComputeDependencyClasses($\mathcal{M}$)
4: $\phi \leftarrow$ GenerateSafetyProperties($\mathcal{M}$), $j \leftarrow 0$, $\mathcal{M}^j \leftarrow \mathcal{M}$
5: **for** each $\phi_i \in \phi$ **do**
6: **if** $[\mathcal{M}^j] \not\models \phi_i$ **then**
7: Let ϕ_i be the violated property
8: Let ce be the counter-example
9: **if** InvolvesAllocOrInitOrDeallocAction(ϕ_i) **then**
10: Let $type$ denote the action type
11: $\mathcal{M}^{j+1} \leftarrow$ SynthesizeRaceFreeType1($\mathcal{M}^j, ce, \phi_i, type$)
12: **else**
13: $\mathcal{M}^{j+1} \leftarrow$ SynthesizeRaceFreeType2($\mathcal{M}^j, ce, \phi_i$)
14: **end if**
15: $j \leftarrow j + 1$
16: **else if** $[\mathcal{M}^j] \models \phi_i$ is unknown **then return** (null,false)
17: **end if**
18: **end for**
19: **return** ($\mathcal{M}^j$, true)

Fig. 8. Counter-example guided synthesis of synchronization for module $\mathcal{M}$.

sis algorithm **VerifyAndSynthesizeSynchronization** that can handle these two types of race conditions. The algorithm first computes dependency information by considering all the state variables reachable by module $\mathcal{M}$. Two variables x and y are dependent if they appear in the guard or update statement of a transition (s_1, s_2), i.e., $x, y \in \mathcal{SCOPE}(s_1, s_2)$. Equivalence classes are created wrt to dependency relationship. Function $\mathcal{M}.dep : V \rightarrow \mathcal{P}(V)$ maps a state variable to the equivalence class it belongs to and is set to the function returned by algorithm **ComputeDependencyClasses** (line 3).

Taking into consideration the two types of race conditions, the algorithm generates a set of safety properties[2], ϕ, (line 4) that are in the following form:

$$\text{invariant}(\neg(pc_{\mathcal{SM}_1} = s_1 \ \wedge \ pc_{\mathcal{SM}_2} = s_2 \ \wedge \ g_1 \ \wedge \ g_2)), \quad (1)$$

where $\mathcal{SM}_1$ and $\mathcal{SM}_2$ are among the top-level flattened asynchronous and synchronous state machines specified and g_1 and g_2 represent the guard conditions of the qualifying transitions.

For type 1 race condition, the requirement is that s_1 is a state with an outgoing transition that has an update action of type `alloc`, `init`, or `dealloc` and s_2 is a state with an outgoing transition that has a guard action of type read or an update action of type read or write on the same state variable. A read action represents an abstract read (`r`) or a concrete read on a boolean or integer variable. Similarly, a write action represents an abstract write (`w`) or a concrete write in the form of an assignment to a boolean or integer variable.

For type 2 race condition, we require the type of updates in the transitions from both states to be of type read or write and at least one of them being a write type of action.

We cannot use the formula in (1) for a top-level flattened state machine that is an asynchronous point, as we abstract local states of individual instantiations with counting abstraction. However, we can still express the race condition involving an asynchronous entry point $\mathcal{SM}_1$ as

$$\text{invariant}(\neg(c_{s_1}^{\mathcal{SM}_1} > 0 \ \wedge \ pc_{\mathcal{SM}_2} = s_2 \ \wedge \ g_1 \ \wedge \ g_2)), \quad (2)$$

[2]A separate property is generated for each pair of racy transitions.

1: **SynthesizeRaceFreeType1**($\mathcal{M}$: Module, ce: Path, phi: set of Safety Prop, ϕ_i: Safety Prop, $type$: {alloc, init, dealloc}) : Module
2: $data, ce_{sm} \leftarrow$ ExtractData($\phi_i, \mathcal{M}$), Project($ce, \mathcal{M}$)
3: $(\mathcal{SM}_1, \mathcal{SM}_2) \leftarrow$ ExtractRacySM($\mathcal{M}, ce_{sm}, \phi_i$)
4: $(loc_1, sloc_1, loc_2, sloc_2) \leftarrow$ ExtractRaceLocations(ce_{sm}, ϕ_i)
5: where $(loc_1, sloc_1) \in \mathcal{SM}_1.R_{sm}$ and $(loc_2, sloc_2) \in \mathcal{SM}_2.R_{sm}$
6: Let $\mathcal{SM}_1^u$ be the SM that performs $type$ update, u, on $data$ on ce_{sm}
7: Let $\{u\} = \mathcal{SM}_1^u.U_e(loc_1, sloc_1)$
8: Let $g = \mathcal{SM}_1^u.G_e(loc_1, sloc_1)$
9: $\mathcal{SM}_1^u.U_e(loc_1, sloc_1) \leftarrow \mathcal{SM}_1^u.U_e(loc_1, sloc_1) \setminus \{u\}$
10: $\mathcal{SM}_1^u.G_e(loc_1, sloc_1) \leftarrow true$
11: $\mathcal{SM}^{super} \leftarrow$ SuperSM(loc_1, loc_2, ce_{sm})
12: Let $s_1^e \in \mathcal{SM}^{super}.S_{sm}$ s.t. loc_2 reachable from s_1^e
13: **if** $type \in$ {alloc, init} **then**
14: **for** each $pred$ s.t. $(pred, s_1^e) \in \mathcal{SM}^{super}.S_{sm}$ **do**
15: $\mathcal{SM}^{super}.S_{sm} \leftarrow \mathcal{SM}^{super}.S_{sm} \oplus$
16: $(pred, Map(g, \mathcal{SM}^{super}), Map(u, \mathcal{SM}^{super}), s_1^e)$
17: **end for**
18: **else**
19: **for** each $succ$ s.t. $(s_1^e, succ) \in \mathcal{SM}^{super}.S_{sm}$ **do**
20: $\mathcal{SM}^{super}.S_{sm} \leftarrow \mathcal{SM}^{super}.S_{sm} \oplus$
21: $(s_1^e, Map(g, \mathcal{SM}^{super}), Map(u, \mathcal{SM}^{super}), succ)$
22: **end for**
23: **end if**
24: $\phi \leftarrow$ UpdateProperties($\phi, \mathcal{SM}^{super}, \mathcal{SM}_1^u$)
25: return $\mathcal{M}$ with updated state machines

Fig. 9. Synthesizing synchronization to avoid race conditions that involve `alloc`, `init`, and `dealloc` operations.

or if different instances of the same asynchronous entry point race with each other, as

$$\text{invariant}(\neg(c_{s_1}^{\mathcal{SM}_1} > 1 \ \wedge \ g_1)). \quad (3)$$

Algorithm **VerifyAndSynthesizeSynchronization** goes through each safety property (line 5) and analyzes those that are violated to synthesize a synchronization strategy that prevents the violation. The synthesis is guided by the counter-example path that explains the violation. Violations related to type 1 and type 2 race conditions are handled by algorithms **SynthesizeRaceFreeType1** (Figure 9) and **SynthesizeRaceFreeType2** (Figure 10), respectively. Both algorithms start with identifying the state variable involved in the race and projecting the counter-example path onto the original unflattened state machines (line 2 in both algorithms). The next step extracts the control states and identifies the top-level flattened state machines that these control states belong to (line 3 in both algorithms).

Algorithm **SynthesizeRaceFreeType1** takes the type of update action as a parameter. The basic approach to fix this type of race condition is to move the `alloc`, `init`, or `dealloc` type update to leverage the happens-before ordering between these updates and enabling or disabling of the state machine that performs `r` or `w` operation on the same state variable. This requires moving an `alloc` or `init` action before the control state that performs the related `#nable` action (lines 14-17) and moving a `dealloc` action after the control state that performs the related `$isable` action (lines 19-22). This is achieved through performing a special transition insertion operation $\oplus$ as defined in Figure 6.

The algorithm also finds the right place, i.e., the state and the state machine, to move the `alloc`, `init`, or `dealloc` action by walking back on the counter-example path ce_{sm}

1: **SynthesizeRaceFreeType2**($\mathcal{M}$: Module, ce: Path, ϕ: set of Safety Prop, ϕ_i: Safety Prop) : Module
2: $data, ce_{sm} \leftarrow$ ExtractData$(\phi_i, \mathcal{M})$, Project$(ce, \mathcal{M})$
3: $(\mathcal{SM}_1, \mathcal{SM}_2) \leftarrow$ ExtractRacySM$(\mathcal{M}, ce_{sm}, \phi_i)$
4: $(loc_1, sloc_1, loc_2, sloc_2) \leftarrow$ ExtractRaceLocations(ce_{sm}, ϕ_i)
5: where $(loc_1, sloc_1) \in \mathcal{SM}_1.R_{sm}$ and $(loc_2, sloc_2) \in \mathcal{SM}_2.R_{sm}$
6: Let LS_1 (LS_2) be the locks held by $\mathcal{SM}_1(\mathcal{SM}_2)$ at $last(ce_{sm})$
7: Let $l_a = \mathcal{M}.protBy(data)$
8: **if** $l_a \neq null$ and $max(\mathcal{SM}_1.\mathcal{C}, \mathcal{SM}_2.\mathcal{C}) = \mathcal{CNTX}(l_a)$ **then**
9: $\quad l_{sol} \leftarrow l_a$
10: **else**
11: $\quad$ Let $l_c \in LS_1 \cup LS_2$ s.t. acquire of l by $\mathcal{SM}_u$
12: $\quad$ closest to $last(ce_{sm})$.
13: $\quad$ **if** $l_c \neq null$ and $max(\mathcal{SM}_1.\mathcal{C}, \mathcal{SM}_2.\mathcal{C}) \leq \mathcal{CNTX}(l_c)$ **then**
14: $\quad\quad l_{sol} \leftarrow l_c$
15: $\quad$ **else**
16: $\quad\quad l_{sol} \leftarrow newlock$
17: $\quad\quad$ where $\mathcal{CNTX}(newlock) = max(\mathcal{SM}_1.\mathcal{C}, \mathcal{SM}_2.\mathcal{C})$
18: $\quad\quad \mathcal{M}.Locks \leftarrow \mathcal{M}.Locks \cup \{newlock\}$
19: $\quad\quad$ **if** $l_a \neq null$ **then**
20: $\quad\quad\quad$ **for** each s_{la} and s_{lr} that acq./rel. l_a in each $\mathcal{SM}_u$ **do**
21: $\quad\quad\quad\quad \mathcal{SM}_u \leftarrow \mathcal{EMBED}(\mathcal{SM}_u, s_{la}, l_{sol}.acquire))$
22: $\quad\quad\quad\quad \mathcal{SM}_u \leftarrow \mathcal{EMBED}(\mathcal{SM}_u, s_{lr}, l_{sol}.release)$
23: $\quad\quad\quad$ **end for**
24: $\quad\quad$ **end if**
25: $\quad$ **end if**
26: $\quad \mathcal{M}.protBy \leftarrow \lambda d. \begin{cases} l_{sol} & d \in \mathcal{M}.dep(data) \\ \mathcal{M}.protBy(d) & \text{otherwise} \end{cases}$
27: **end if**
28: **for** i: 1 to 2 **do**
29: $\quad$ **if** $l_{sol} \notin LS_i$ **then**
30: $\quad\quad \mathcal{SM}_i.S_{sm} \leftarrow \mathcal{SM}_i.S_{sm} \cup \{s^a_{new}, s^r_{new}\}$
31: $\quad\quad \mathcal{SM}_i.R_{sm} \leftarrow \mathcal{SM}_i.R_{sm} \cup$
32: $\quad\quad\quad \{(loc_i, s^a_{new}), (s^a_{new}, s^r_{new}), (s^r_{new}, sloc_i)\} \setminus \{(loc_i, sloc_i)\}$
33: $\quad\quad \mathcal{SM}_i.G_e \leftarrow$
34: $\quad\quad\quad \lambda x.y. \begin{cases} true & x = s^r_{new} \vee y = s^a_{new} \\ \mathcal{SM}_i.G_e(loc_1, sloc_1) & x = s^a_{new} \wedge y = s^r_{new} \\ \mathcal{SM}_i.G_e(x, y) & otherwise \end{cases}$
35: $\quad\quad \mathcal{SM}_i.U_e \leftarrow$
36: $\quad\quad\quad \lambda x.y. \begin{cases} \emptyset & x = s^r_{new} \vee y = s^a_{new} \\ \mathcal{SM}_i.U_e(loc_1, sloc_1) & x = s^a_{new} \wedge y = s^r_{new} \\ \mathcal{SM}_i.U_e(x, y) & otherwise \end{cases}$
37: $\quad\quad \mathcal{SM}_i \leftarrow \mathcal{EMBED}(\mathcal{SM}_i, s^a_{new}, \mathcal{TYPE}(l_{sol}).acquire)$
38: $\quad\quad \mathcal{SM}_i \leftarrow \mathcal{EMBED}(\mathcal{SM}_i, s^r_{new}, \mathcal{TYPE}(l_{sol}).release)$
39: $\quad\quad LS_i \leftarrow LS_i \cup \{l_{sol}\}$
40: $\quad\quad \phi \leftarrow$ UpdateProperties$(\phi, \mathcal{SM}_i)$
41: $\quad$ **end if**
42: **end for**
43: return $\mathcal{M}$ with updated state machines

Fig. 10. Synthesizing synchronization to avoid race conditions that involve `r` and `w` operations.

and finding the lowest common ancestor, $\mathcal{SM}^{super}$, of the state machines that contain the enabling/disabling state loc_2 and the updating state loc_1 (line 11). Update action is moved to $\mathcal{SM}^{super}$ and gets inserted before (after) the state s^e_1 in $\mathcal{SM}^{super}$ that can reach the enabling (disabling) state loc_2.

Algorithm **SynthesizeRaceFreeType2** maintains a function $protBy : V \rightarrow L$ that maps a state variable to the lock variable that protects it. We recall that this global function is initialized in algorithm **VerifyAndSynthesizeSynchronization** (line 2) by mapping each variable to $null$, i.e., no locking solution is endorsed by the algorithm at the beginning even if the user has provided some in the model. The algorithm updates this function as it determines the most suitable lock that protects a state variable and the equivalence class it belongs to based on a given counter-example.

The key concept in devising a synchronization solution is the *thread context*, which represents scheduling priority of a thread wrt other threads. Examples of contexts in the Linux kernel includes user processes, soft-interrupts, and hardware interrupts. We assume that there is a total order among thread contexts due to the associated scheduling priority. It is important to note that we do not consider thread priorities during model checking stage and perform an over-approximation by considering an interleaved semantics, where all threads have equal priority. This may result in counter-examples that are not realistic such as a low-priority process, e.g., a user process, interrupting a high-priority one, e.g., a hardware interrupt. However, for each such scenario there exists a symmetric scenario in the state space such that the high-priority interrupts the low priority one. So, in terms of detecting races this over-approximation does not introduce imprecision wrt synchronization synthesis, which considers safety properties that express race conditions only. We further assume that each lock type is associated with the highest context it is effective for, i.e., can ensure mutual exclusion without introducing a deadlock due to priority inversion.

If the algorithm in Figure 10 has already chosen a lock l_a that protects the state variable $data$ (and recorded in $protBy$ function) and as long as the context of this lock is at least as big as the maximum of the context priorities of the top-level racy state machines (line 8) then we decide to use this lock as the solution lock l_{sol}. If the context of the lock is not high-enough or if there is no lock associated with $data$ in $protBy$ yet (line 10), it checks if any of the racy state machines has acquired a lock l_c prior to its racy statement (line 11-12). Here, we consider the most recently acquired lock, if any, prior to the violation of the safety property as the state machine may be have acquired multiple locks or both state machines may have acquired some locks.

If the candidate lock l_c's priority is sufficiently high, it is considered as the solution lock. Otherwise, the algorithm has to introduce a new lock $newlock$ and assign it to l_{sol} (lines 15-18). We assume that the synthesis algorithm has access to a lock model for each context priority value and that each lock model has been verified to satisfy the mutual exclusion and deadlock freedom properties. In the case that the new lock replaces a previously endorsed lock l_a (line 19), all acquire and release operations on l_a in any state machine $\mathcal{SM}_u$ need to be replaced with the respective operations of the solution lock l_{sol} (lines 20-23). To eliminate the race condition, the solution lock l_{sol} is applied to the state machine that did not use it by enclosing the guard and update of the racy transition with the acquire and release operations on l_{sol} (lines 28-41). The new lock is introduced to the same custom type definition or the module that $data$ has been defined in.

It is important to note that both synthesis algorithms may change the transitions in the state machines making some of the safety properties generated earlier invalid wrt race checking and introducing new race possibilities to check for. So both algorithms update the set of properties to make sure that the synthesis algorithm check each valid race condition

(line 24 in Figure 9 and line 40 in Figure 10).

A. Correctness

Lemma 5.1: Algorithm **SynthesizeRaceFreeType1** eliminates the given race condition.

Proof It eliminates the race by 1) removing the data alloc/dealloc/init type action (line 9) and 2) adding a state to the state machine that has the enabling or the disabling action so that racy action is performed right before (after) the enabling (disabling) action using the $\oplus$ operator.

Lemma 5.2: Algorithm **SynthesizeRaceFreeType2** eliminates the given race condition.

Proof It eliminates the race by protecting the racy locations either with an existing lock with a suitable priority or adding a new lock.

Theorem 5.3: If algorithm **VerifyAndSynthesizeSynchronization** returns ($\mathcal{M}^j$,true) then $[\mathcal{M}^j]$ is free of *Type 1* and *Type 2* races.

Proof Follows from 1) all racy location pairs are collected and checked for reachability, 2) Lemma 5.1 and Lemma 5.2, 3) updating the safety properties when transitions are changed or added (line 24 in Figure 9 and line 40 in Figure 10) to detect new races that became possible due to the changes in the state machines as well as removing those that are no longer relevant.

Below, we argue that the synthesis algorithm does not introduce any deadlocks under certain conditions.

Definition A lock l_1 encloses another lock l_2 in a state machine $\mathcal{SM}$ if $l_1.acquire \xrightarrow{R^*_{sm}} l_2.acquire \xrightarrow{R^*_{sm}} l_2.release \xrightarrow{R^*_{sm}} l_1.release$, where R^*_{sm} represents the transitive closure of $\mathcal{SM}.R_{sm}$ and $l_i.op$ represents the state of $\mathcal{SM}$ that gets mapped to lock l_i's state machine for operation op.

Lemma 5.4: Algorithm **VerifyAndSynthesizeSynchronization** does not introduce a new lock l_{new} that encloses an existing lock l_{old}.

Proof Follows from the fact that in algorithm **SynthesizeRaceFreeType2** acquire and release operations of each new lock encloses the racy transition's guard and the update. Lock acquire and release operations are defined in terms of state machines and not in terms of abstract operations that can appear in the update of a transition. So the way a new lock is inserted does not cause it to enclose another lock.

Lemma 5.5: Algorithm **VerifyAndSynthesizeSynchronization** does not introduce two new locks l_1 and l_2 such that a flattened top-level state machine $\mathcal{SM}$ in $\mathcal{M}^j$ has a trace in which l_1 encloses l_2.

Proof Follows from 1) Lemma 5.4, 2) a unique lock is assigned to each equivalence class of state variables based on transition level data and control flow dependency, and 3) if an assigned lock is found insufficient in terms of context priority, all acquire and release actions of that lock is replaced with those of a more appropriate new lock.

Theorem 5.6: If the original model does not have any deadlocks, then algorithm **VerifyAndSynthesizeSynchronization** does not introduce deadlocks due to multiple locks acquired in a cyclic manner.

Proof Follows from Lemma 5.4 and Lemma 5.5.

Theorem 5.7: Assuming $[\mathcal{M}^0]$ does not include a priority inversion bug, if algorithm **VerifyAndSynthesizeSynchronization** returns ($\mathcal{M}^j$,true) then $[\mathcal{M}^j]$ does not have a priority inversion bug, i.e., the possibility of a a high priority state machine being blocked on trying to acquire a lock that is held by a low priority state machine.

Proof Follows from the fact that the solution lock candidates are evaluated wrt having a sufficiently high context priority. The solution lock, whether an already endorsed lock, an existing lock, or a new lock introduced by the algorithm, is guaranteed to have a context priority greater than or equal to the context priority of the racy state machines.

B. USB keyboard example

We have used NuXmv [4] model checker for the verification stage and ran it in `check_invar_ic3` mode to perform infinite-state model checking. We generated the model of our running example based on the formal semantics given in Section IV and the safety properties as described in Equations (1), (2), and (3). Below we give examples for both types of races.

Race type 1: The example race we give here is between the top-level synchronous state machine `main.UserAction1` and the top-level asynchronous state machine `USB.usb_kbd_irq` and the violated safety property is invariant($\neg(pc_{\text{usb\_kbd\_probe}} = \text{register\_input} \wedge c^{usb\_kbd\_irq}_{kbd\_event} > 0)$)[3]. The property describes a race as transition `register_input->alloc_usb` in `USBKBD.usb_kbd_probe` allocates variable `new` and the transition `kbd_event->submit_urb` writes `new`. The race happens because the top-level state machine `UserAction1` enables the `usb_kbd_open` state machine in transition `init->register_input` by entering the state machine `USBKBD.usb_kbd_probe` that enters `InputLayer.input_register_device`, which enables `USBKBD.usb_kbd_open`. At this point top-level state machine `main.UserAction3` enters state machine `USBKBD.usb_kbd_open` and instantiates the top-level asynchronous `USBKBD.usb_kbd_irq`. Assuming that `USBKBD.usb_kbd_irq` gets scheduled before variable `new` gets allocated in `usb_kbd_probe` (entered from `main.UserAction1`), in transition `kbd_event->submit_urb`, `new` will be accessed causing a problem such as a kernel panic. The synthesis algorithm in Figure 9 moves `alloc new` before `register_input` state in `usb_kbd_probe` state machine.

Race type 2: One of the violated safety property is invariant($\neg(c^{\text{usb\_kbd\_led}}_{la} > 1 \wedge CHANGE)$), which states that there cannot be more than one instance of the `usb_kbd_led`

[3]Note that both g_1, the guard of transition `register_input->alloc_usb`, and g_2, the guard of `kbd_event->submit_urb`, are true; the latter because of involving abstract operations which are interpreted as true.

asynchronous entry point of module USBKBD that are at la control state when CHANGE holds, i.e., executing the la->update_led transition. The counter-example returned by NuXmv indicates that two instances of the usb_kbd_led can be created by two sequential runs of usb_kbd_irq assuming that each detects a LED related event, i.e., EV_LED holding true. The synthesis algorithm introduces a new lock l_{new} that protects the state variables in the dependency equivalence class {leds, newleds, led_urb_submitted, CHANGE}. Then the algorithm detects a violation for invariant($\neg(c_{la}^{\text{usb\_kbd\_led}} > 1 \wedge \neg CHANGE)$) that corresponds to a race involving the transition la->lr1_exit. Since synthesis algorithm has associated l_{new} with the variables in the scope of this transition, it uses l_{new} to fix this race. Finally, a violation for invariant($\neg(c_{submit\_urb}^{\text{usb\_kbd\_led}} > 1)$) is detected. Since the variables in the scope of transition submit_urb->lr2_exit are not assigned a lock yet, the algorithm creates a new lock to protect this transition.

VI. Related Work

Recently, there has been interest in semantic preserving synthesis for concurrency where correctness is specified in terms of user provided atomic blocks [8], assertions [6], [7], [11], equivalence with the sequential behavior [3], or behaviors such as relative ordering of events possible under non-preemptive scheduling [5]. These approaches transform code whereas our approach works on the presented modeling formalism that extends state machines with callbacks. Our approach is complementary to these; ideally one needs debugging and synchronization synthesis both at the design and the development stages.

The synchronization algorithm in [6] learns from counter-example traces and applies a number of patterns for semantic preserving reordering and their REORDER.RELEASE pattern is similar to our reordering approach. The approach is improved in [7] by also learning from good traces, i.e., those that are feasible under non-preemptive scheduling. Similar to [7], [5] also uses a set of pattern based inference rules for synchronization. Their reordering synchronization handles those errors that pertain to signal/await operations whereas our reordering synchronization handles errors that involve allocation/deallocation operations. [8], [6], [3], [7] and our approach guarantee deadlock freedom by construction of synthesis while [11], [5] do not. Also, unlike the mentioned approaches, our synchronization synthesis handles an *unbounded number of threads* and *context priorities*.

In [10] a state-machine based language, called P, is introduced for event-driven modeling, analysis, and code synthesis. P language allows explicit specification of deferred events, whose handling can be delayed to model asynchrony. In our modeling approach state machines can be labeled as asynchronous instead. Also, in the P framework the model is subjected to systematic testing using explicit-state model checking, which enumerates implicit scheduling choices and explicit modeling choices such as deferred events. So the goal is to find bugs before the full code synthesis is achieved. In our case, we use unbounded model checking to obtain race freedom guarantees to guide the synthesis of concurrency.

VII. Conclusion

We have presented a state machine based formalism that can be used to model software using the callback mechanism. We support both synchronous and asynchronous callbacks and provide a formal semantics using counting abstraction technique. We also present a counter-example guided algorithm that can synthesize concurrency solution with race-freedom guarantee. We show that under certain conditions the synthesis also guarantees deadlock freedom. We have demonstrated various aspects of the approach using the USB keyboard driver as a case study. In future work, we are planning to handle additional synchronization primitives such as wait and notify operations.

Acknowledgement

We would like to thank Naveen Iyer for his help with collecting and categorizing the Linux device driver bugs.

References

[1] Bug Report, HID: usbkbd: synchronize LED URB submission. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/.

[2] Linux USB keyboard driver. http://lxr.free-electrons.com/source/drivers/hid/usbhid/usbkbd.c.

[3] R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Ausserlechner, and R. Spork. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 35–42, 2014.

[4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 334–342, 2014.

[5] P. Cerný, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 180–197, 2015.

[6] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 951–967, 2013.

[7] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free synthesis for concurrency. *CoRR*, abs/1407.3681, 2014.

[8] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. *SIGPLAN Not.*, 43(6):304–315, June 2008.

[9] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[10] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332, 2013.

[11] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach. Succinct representation of concurrent trace sets. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 433–444, 2015.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.