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 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.

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.

<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>
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

![Call Graph Diagram]

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.

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 [2], which could only generate the thick lines in Figure 1 and missed the regular lines and the dashed lines due to implicit dependencies.

1Initializations 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

```plaintext
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
```

#### module InputLayer

```plaintext
uses SpinLock;
sync {input_event(T), input_register_device(T)};
type input_event {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->allocate, allocate->register_exit};
  [allocate->allocate] (update: alloc dev.vals);
  [allocate->register_exit] (update: #nable(dev.open), #nable(dev.close));
end SM
```

---

### III. Modeling Language

```
SM: usb_kbd_irq()
  T : {init->usb_kbd_probe, usb_kbd_disconnect};
  [usb_kbd_probe] (guard: usb_kbd_open, usb_kbd_close);
  [usb_kbd_disconnect] (guard: usb_kbd_open, usb_kbd_close);
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};
  [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_irq));
end SM
SM: usb_kbd_close(close);
```

---

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 $s1 \rightarrow s2$, 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 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 $\#$able 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.: $\langle [SM^\delta] \rangle \equiv \langle S, I, R \rangle$

$S \equiv B^{log([SM, sm_{\text{m}}])} \times SM.S_e,$
$I \equiv I_e \land \bigvee_{s \in SM.I_{\text{sm}}} pc_{SM} = s_i,$
$R \equiv \bigvee_{(s_1, s_2) \in SM.R_{\text{sm}}} pc_{SM} = s_1 \land pc_{SM} = s_2 \land$

$SM.G_e(s_1, s_2) \land SM.U_e(s_1, s_2) \land$ $\text{IDENTITY}(\text{Unchanged})$

Asynch.: $\langle [SM^A] \rangle \equiv \langle S, I, R \rangle$

$S \equiv \bigwedge_{s \in SM.S_{\text{sm}}} c^S_{SM} \geq 0 \land SM.S_e,$
$I \equiv \bigwedge_{s \in SM.S_{\text{sm}}} c^S_{SM} = 0 \land SM.I_e,$
$R \equiv \bigwedge_{(s_1, s_2) \in SM.S_{\text{sm}}} SM.G_e(s_1, s_2) \land SM.U_e(s_1, s_2)$

$\land c^S_{s_1} \geq 0 \land c^{S'}_{s_1} = c^S_{s_1} - 1 \land c^{S'}_{s_2} = c^S_{s_2} + 1$

**PARALLEL COMPOSITION**

$\langle [SM_1] \rangle \parallel \langle [SM_2] \rangle \equiv \langle SM_1.I \cap [SM_2].S, [SM_1].I \cap [SM_2].I, [SM_1].R \cup [SM_2].R \rangle$

**INSTANTIATING THE MAIN MODULE**

$\{M\} \equiv \langle [SM_1^G] \rangle \parallel ... \parallel [SM_{\text{m}}^G] \parallel [SM_2^G] \parallel ... \parallel [SM_A^G] \rangle$

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

A module, $M$, is defined as a set of synchronous, asynchronous, and helper state machines. A state machine, $SM$, is defined in terms of a tuple $(V, S_{sm}, S_I, I_{sm}, I_c, E_{sm}, R_{sm}, G_e, U_e)$, where $V$ denotes the state variables defined by the module that defines $SM$, $S_{sm}$ 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, $I_c$ 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...
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: \([M]\) and \([SM]\) represent instantiation of module \(M\) and state machine \(SM\), 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 \([M]\) consists of all the state variables of \(M\), the state variables of the modules directly or indirectly used by \(M\) and are reachable from synchronous or asynchronous entry points, and all the meta-variables, e.g., the program counter, \(pc_{SM}\), state machine \(SM\). 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.

\(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 \(embedding operation, EMBED\), 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
the control states and valuations of the state variables and any constraints described via the \texttt{restrict} keyword. The initial states describe the initial control states and the initial values of the state variables described via the \texttt{initial} keyword.

\textbf{IDENTITY(\texttt{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., \texttt{alloc}, do not impact the semantics. However, as we will show in Section \[\text{V} \] they will be instrumental in the generation of safety properties. We explain the meta-variable \(c^s_{SM}\) below.

e) \textbf{Instantiating a State Machine for an Asynchronous Entry Point:} Since an asynchronous entry point can be instantiated dynamically through the \texttt{async} construct, we use the counting abstraction technique \[\text{F} \] to represent such instantiations. The idea is to keep a counter, \(c^s_{SM}\), for every control location, \(s\), and represent transitions of an asynchronous state machine by manipulating these counters.

\textbf{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 \[\text{I} \] 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 \[\text{F} \] presents our top-level verification and synthesis.
1. VerifyAndSynthesizeSynchronization:\(M : \text{Module}) : (\text{Module, boolean})
2. global \(M, \text{protBy} : M \mapsto M, \text{Locks}, M, \text{protBy} \leftarrow \lambda x.\text{null}
3. global \(M, \text{dep} \leftarrow \text{ComputeDependencyClasses}(M)
4. \phi \leftarrow \text{GenerateSafetyProperties}(M), j \leftarrow 0, M^j \leftarrow M
5. for each \(\phi_i \in \phi\) do
6. if \([M^j]_i \neq \phi_i\) then
7. Let \(\phi_i\) be the violated property
8. Let ce be the counter-example
9. if \(\text{InvolvesAllocOrInitOrDeallocAction}(\phi_i)\) then
10. Let type denote the action type
11. \(M^{j+1} \leftarrow \text{SynthesizeRaceFreeType1}(M^j, ce, \phi_i, \text{type})\)
12. else
13. \(M^{j+1} \leftarrow \text{SynthesizeRaceFreeType2}(M^j, ce, \phi_i)\)
14. end if
15. \(j \leftarrow j + 1\)
16. if \([M^j]_i = \phi_i\) unknown then return (null,false)
17. end if
18. end for
19. return \((M^j, \text{true})\)

Fig. 8. Counter-example guided synthesis of synchronization for module \(M\).

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 \(M\). 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., \(x, y \in \text{SCOPE}(s_1, s_2)\). Equivalence classes are created wrt to dependency relationship. Function \(M, \text{dep} : \mathcal{V} \mapsto \mathcal{P}(\mathcal{V})\) 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 transitions, the algorithm generates a set of safety properties for \(\phi\), (line 4) that are in the following form:

\[\text{invariant} \left( \neg \left( p_{SM_1} = s_1 \land p_{SM_2} = s_2 \land g_1 \land g_2 \right) \right)\] (1)

where \(SM_1\) and \(SM_2\) are 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 \((\varepsilon)\) or a concrete read on a boolean or integer variable. Similarly, a write action represents an abstract write \((\omega)\) 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 \(SM_1\) as

\[\text{invariant} \left( \neg \left( c_{SM_1} > 0 \land p_{SM_2} = s_2 \land g_1 \land g_2 \right) \right)\] (2)

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

\[\text{invariant} \left( \neg \left( c_{SM_1} > 1 \land g_1 \right) \right)\] (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 flattened 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 \(\varepsilon\) or \(\omega\) 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 $is able 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 \(ce_{sym}\)
1:  
2: \textbf{SynthesizeRaceFreeType2} : \textbf{M} : Module, ce: Path, \( \phi \): set of Safety Prop, \( \phi_i \): Safety Prop \} -> Module  
3: \textbf{data, ce,sm}, \textbf{extractData}(\( \phi_i \), \textbf{M}), \textbf{Project}(\textbf{ce, M})  
4: \textbf{SM1, SM2} : ExtractRacySM(\textbf{M}, ce,sm, \( \phi_i \))  
5: \textbf{loc1, loc2, sm1, sm2} : ExtractRaceLocations(ce,sm, \( \phi_i \))  
6: \textbf{SM1, SM2} : \textbf{Locks} be the locks held by \textbf{SM1, SM2} at last(ce,sm)  
7: \textbf{SM1, SM2} : \textbf{SM} \in \textbf{Locs} \} \textbf{newlock}  
8: \textbf{SM1, SM2} : \textbf{max} \textbf{(SM1, SM2)} = CNTX(l_c) then  
9: \textbf{l}_\text{sol} \leftarrow l_c  
10: \textbf{else}  
11: \textbf{let} \textbf{l}_c \in \textbf{LS1} \cup \textbf{LS2} s.t. acquire of \textbf{l}_c \textbf{by SM}_u \textbf{do} \textbf{end for}  
12: \textbf{if} \textbf{l}_c \neq null \textbf{then} \textbf{end for}  
13: \textbf{SM}_u \leftarrow \textbf{EMBED}(\textbf{SM}_u, s) \leftarrow  \textbf{l}_\text{sol, acquire} \textbf{end for}  
14: \textbf{end if}  
15: \textbf{end if}  
16: \textbf{SM}_u \leftarrow \textbf{EMBED}(\textbf{SM}_u, s, l_{1, sol, release}) \textbf{end for}  
17: \textbf{Return} \textbf{M, protBy} \leftarrow \textbf{M, protBy}  
18: \textbf{for} \textbf{v} \textbf{from 1 \ to 2 \ do}  
19: \textbf{end if}  
20: \textbf{end for}  
21: \textbf{return} \textbf{M} \textbf{with updated state machines}  

Fig. 10. Synthesizing synchronization to avoid race conditions that involve \( \varepsilon \) and \( \omega \) operations.  

and finding the lowest common ancestor, \( \textbf{SM}^{\text{super}} \), of the state machines that contain the enabling/disabling state \textit{loc2} and the updating state \textit{loc1} (line 11). Update action is moved to \( \textbf{SM}^{\text{super}} \) and gets inserted before (after) the state \( s_i \) in \( \textbf{SM}^{\text{super}} \) that can reach the enabling (disabling) state \textit{loc2}.  

Algorithm \textbf{SynthesizeRaceFreeType2} maintains a function \textbf{protBy} : \textbf{V} \textbf{to L} that maps a state variable to the lock variable that protects it. We recall that this global function is initialized in algorithm \textbf{VerifyAndSynthesizeSynchronization} (line 2) by mapping each variable to \textit{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.
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 race 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 \( (M^j, \text{true}) \) then \( [M^j] \) 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 \( SM \) if \( l_1.\text{acquire} \rightarrow R^*_sm \rightarrow l_2.\text{acquire} \rightarrow R^*_sm \rightarrow l_2.\text{release} \rightarrow l_1.\text{release} \), where \( R^*_sm \) represents the transitive closure of \( SM.R_sm \) and \( l_i.op \) represents the state of \( SM \) that gets mapped to lock \( l_i \)’s state machine for operation \( op \).

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

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 \( SM \) in \( M^j \) 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 \( [M^0] \) does not include a priority inversion bug, if algorithm VerifyAndSynthesizeSynchronization returns \( (M^j, \text{true}) \) then \( [M^j] \) does not have a priority inversion bug, i.e., the possibility of 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 race 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\( (\neg (pc_{\text{usb_kbd_irq}} = \text{register_input} \land (pc_{\text{kbd_event}} > 0))) \). 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 2 moves alloc new before register_input state in usb_kbd_probe state machine.

Race type 2: One of the violated safety property is invariant\( (\neg (pc_{\text{kbdLed}} > 1 \land CHANGE)) \), which states that there cannot be more than one instance of the usb_kbdLed

\(^3\) 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\text{new} that protects the state variables in the dependency equivalence class \{leds, newleds, led_usb_submitted, CHANGE\}. Then the algorithm detects a violation for invariant(\neg (\neg (e_{la}^{\text{usb_kbd_led}} > 1 \land \neg \text{CHANGE})) that corresponds to a race involving the transition la->lr1_exit. Since synthesis algorithm has associated l\text{new} with the variables in the scope of this transition, it uses l\text{new} to fix this race. Finally, a violation for invariant(\neg (e_{submit_u rb}^{\text{led}} > 1)) is detected. Since the variables in the scope of transition submit_u rb->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 counterexample 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. [9], [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