Synthesizing Operating System Based Device Drivers in Embedded Systems

Shaojie Wang
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA
wsj@ee.princeton.edu

Sharad Malik
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA
sharad@ee.princeton.edu

ABSTRACT

This paper presents a correct-by-construction synthesis method for generating operating system based device drivers from a formally specified device behavior model. Existing driver development is largely manual using an ad-hoc design methodology. Consequently, this task is error prone and becomes a bottleneck in embedded system design methodology.

Our solution to this problem starts by accurately specifying device access behavior with a formal model, *viz*. extended event driven finite state machines. We state easy to check soundness conditions on the model that subsequently guarantee properties such as bounded execution time and deadlock-free behavior. We design a deadlock-free resource accessing scheme for our device access model. Finally, we synthesize an operating system (OS) based event processing mechanism, which is the core of the device driver, using a disciplined methodology that assures the correctness of the resulting driver.

We validate our synthesis method using two case studies: an infrared port and the USB device controller for an SA1100 based handheld. Besides assuring a correct-by -construction driver, the size of the specification is 70% smaller than a manually written driver, which is a strong indicator of improved design productivity.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program verification – *Formal method*; D.1.2 [**Programming Techniques**]: Automatic Programming – Program synthesis; D.2.1 [**Software Engineering**]: Requirements/Specifications – *Methodologies*; D.2.2 [**Software Engineering**]: Design Tools and Techniques – *computer aided software engineering*.

General Terms: Algorithms, Design, Experimentation, Verification.

Keywords: Correct-by-construction, device driver, operating system based software synthesis, embedded system software.

CODES+ISSS'03, October 1-3, 2003, Newport Beach, California, USA.

1. INTRODUCTION

Device drivers are the glue software providing the bridge between peripheral devices on one side and the upper layers of the operating system and the application software on the other. They are critical software elements that significantly affect the quality and productivity of system development tasks. They are also notoriously hard to design and debug. Complex hardware behaviors and complicated system synchronization requests and mechanisms are the two main contributors to driver errors. Most modern operating systems are multi-tasking. As part of the kernel, the driver software must handle concurrent accesses from multiple tasks, which is difficult to design and debug. This, coupled with the complexity of hardware-software interactions makes the development and design of drivers laborious and error-prone.

The low productivity and unreliability of manual driver development provides a strong motivation for the correct-by-construction methodology proposed in this paper. This methodology is based on a formal specification of a device access model with easy to specify semantics. The correctness of the driver generation algorithm is then easy to demonstrate.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 explains general concepts of device drivers; Section 4 presents issues related to choosing appropriate preemption prevention mechanisms; Section 5 provides an overview of the device driver synthesis framework; Section 6 presents the formal device access model essential to the proof to the correctness of the code synthesized; Section 7 discusses the synthesis procedure in detail and the correctness proof; Section 8 describes our case studies; and finally Section 9 provides some concluding remarks.

2. RELATED WORK

We briefly discuss related research work in the area of device driver development methodology and operating system based software generation methodology. As part of the methodology described in our earlier publication [1], our effort distinguishes itself by the formal specification of device access behavior, the design of the resource sharing scheme and the synthesis of an event processing mechanism based on OS primitives which is provably correct according to our formal model specification. In this paper we provide the details of this specification and methodology.

Early device driver synthesis techniques, as part of hardware/software co-design efforts, focus on generating simple

interrupt handlers synchronizing the software and the device [5, 6]. Recently, attempts have also been made to generate operating system based device drivers [1, 2]. These methodologies propose a high-level abstract model and synthesize operating system based device drivers. By using a library for each supported OS and an automatic selection mechanism, they generate device drivers for a range of operating systems. O'Nil [2] proposes a Yacc-like language ProGram [8] to model the communication protocol between the device and the software. Finite state machines are extracted from the specification to enable code synthesis. Iris [1] proposes to separate the device programming interface, device control and data path accesses to provide a higher level abstraction which enables cleaner specification and more efficient checking. While our earlier publication [1] gives an overview of the overall methodology, this paper focuses on the synthesis technique of correct-by-construction OS based device drivers.

Traditional system level design languages are based on hardware description languages, and tend to describe static architectures. More recently, researchers have realized the importance of dynamic behavior and propose to include it in system level design models. Such dynamic features are essentially services provided by an OS. SoCOS [3] proposes a concurrency model including asynchronous, reactive and synchronous communications and generates embedded system software based on a real-time operating system (RTOS) through several refinement steps. Gerstlauer [4] adopts a similar approach. They propose an RTOS model, which is effectively a set of commonly used RTOS services, to extend the original specC language's ability to handle the interleaved execution behavior of dynamic schedulers.

3. BACKGROUND

Device drivers are a collection of functions and data structures, used to set up and initialize the device, transfer data between the device and the software, configure the device, monitor and trace the status of the device, reset the device, and shut down the device as requested. These functions are called *entry* functions, which are usually part of the kernel. User space programs access device drivers through the OS input-output (I/O) system calls, which are file system calls in the case of the UNIX operating system. The OS is responsible for routing user space requests contained in system calls to appropriate entry functions.

Operating systems provide device driver models reflecting device-processor data transfer patterns and data processing patterns. For example, UNIX [7] provides four basic device driver models: character (for byte-stream low volume data), terminal (a special type of character driver for terminal functions including tab expansions), block (high volume data such as storage devices), and stream (for packet based network data).

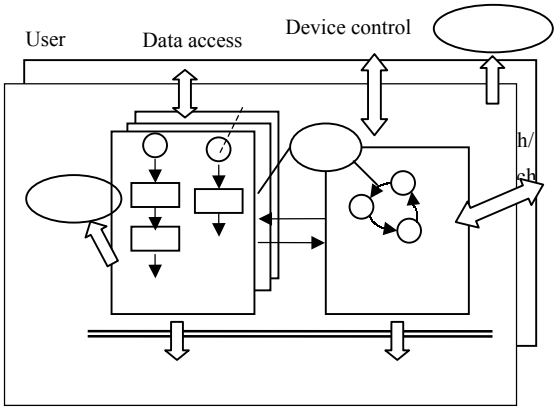
4. OPERATING SYSTEM EXECUTION LEVELS AND PREEMPTION PREVENTION MECHANISMS

Operating systems typically define multiple execution levels. The code executing at a higher execution level preempts the code executing at a lower level by default. For example, the execution level of interrupt handlers is higher than that of a normal piece of code and interrupts preempt those codes by default. However, preemption can corrupt the integrity of the data (critical sections) shared between the code and the interrupt handler. Thus, a preemption prevention mechanism is required to ensure the correct execution of the program. Operating systems define preemption prevention primitives. Concurrent programs must select appropriate preemption prevention mechanisms to protect their critical sections. For example, Linux defines interrupt disable/enable mechanisms to prevent interrupt intrusion. Locking mechanisms are used to protect critical sections shared by the kernel parts of user processes. For the older version of the Linux kernel, timer handlers are executed by timer interrupt handlers, the executions of normal interrupt handlers and timer handlers do not interleave. This is not true for the newer kernel in which timer handlers are executed by *tasklet*, an execution entity managed by the kernel and interruptible by the interrupts. Clearly, the correct preemption mechanism needs to be selected based on the execution levels of the codes sharing the critical section and functionality range of the primitives provided by the OS. We provide a formal mechanism to specify the preemption prevention primitives. With limited space we choose to omit its description in favor of the main driver synthesis framework.

5. OVERVIEW OF DEVICE DRIVER SYNTHESIS FRAMEWORK

An overview of our device driver synthesis framework is provided in our earlier publication on this [1]. We briefly review it here. Device driver entry functions are partitioned into three parts: core functions, platform functions, and registry functions. The core functions are the platform (processor and OS) independent part of the driver. The others are the platform dependent part of the driver. The core functions are synthesized from a device specification, which is based on a device access model. The platform functions and registry functions are implemented as libraries and templates. They are mapped to a particular operating system automatically according to the driver configuration. To demonstrate the correct-by-construction code synthesis technique, we describe the specification of the device access model in this section.

Figure 1 shows an overview of the device access model. We partition a device into multiple sub-devices according to the function they implement. Sub-devices may share a physical interface (ITF). The access of each sub-device is modeled in two



layers. The first layer is the device programming interface model. In this layer, the device memory and register accesses are modeled through aggregating data structures and access functions. For example, a UART has 12 interface registers controlling its communication with the processor. The read-only register, Receive Buffer Register (RBR), is the read-point of the UART receiving FIFO. The FIFO is modeled at the device programming interface layer and the access function of the FIFO is automatically synthesized from the model. The second layer is the device behavior model. The device behavior model has two parts: device control and data channels.

- Device control models the logical states and state transitions of a device with an extended event driven finite state machine.
- Data channels model the physical data exchange channels of the device with an event driven finite state machine that conforms to one of a small set of templates. This is sufficient since these behaviors tend to be limited to a small set of possibilities. A sub-device may possess multiple parallel data channels competing for system resources such as DMA channels. A data channel has at most two ports: *read* and *write*. Each port access is modeled as a series of data unit accesses. A *data unit* access is a series of hardware data transfer operations performed without any intervening synchronization. Because they have multiple data unit accesses, data port accesses (read and write) of a single channel are concurrent. The data channel is shared at the granularity of a data unit. Three data port access modes are defined: *blocked*, *buffered*, and *asynchronous*. Blocked and buffered mode accesses are initiated by the user while the asynchronous mode accesses are initiated by the device. For blocked mode accesses, the data accumulation buffer is provided by the caller. For the other access modes, the buffer is managed by the driver.

A data port is available only when the device controller is in certain states. A data port may share data (SH_{data}) with event handlers of the device controller.

The device controller and data port accesses communicate synchronously or asynchronously. A data port pushing data to the device controller communicates with the controller asynchronously. The controller accessing a blocking data port communicates with the data port synchronously.

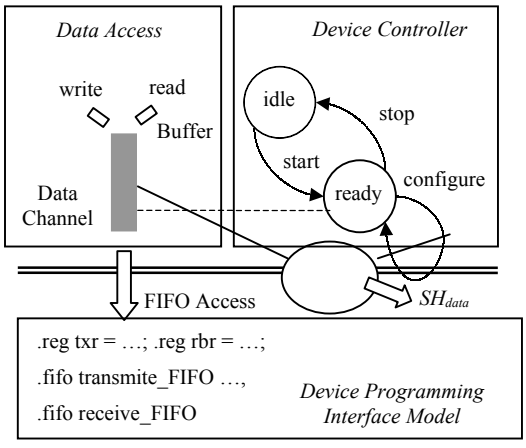


Figure 2 illustrates the device access model of a half-duplex UART. It has only one sub-device. The first layer models the register accesses of the UART including a transmit FIFO and a receive FIFO. The second layer models UART behavior.

- The device controller state machine has two states: *idle* and *ready*. State *idle* is the start state. A UART in the *idle* state is not initialized by any software and is not operating. It moves to *ready* state after being initialized, and returns to *idle* state when the software issues the *stop* event and shuts down the device. The initialization is triggered by event *start*. Software can configure a UART in *ready* state.
- The UART has one data channel with two ports: read and write. The data unit of the read port reads from the receive FIFO until it is empty. The data unit of the write port writes to the transmit FIFO until it is full (or there is no more data).

The data port accesses are available only when the device is in *ready* state. Configuration data, baud rate setting, is shared between the *configure* event handler and the channel. The half-duplex feature of the UART is modeled elegantly by our data channel sharing model.

6. FORMAL MODELING OF THE DEVICE BEHAVIOR

The device access behavior model we propose is based on a formal model: extended event driven finite state machine. This section describes the extended event driven state machine model, the control function model, the synchronization and communication model of the state machines, and the timing model formally. The correctness of the implementation of the model and the soundness of the model are also defined.

6.1 Extended Event Driven Finite State Ma-chine Model

The extended event driven state machine is defined as a 5-tuple (S, S_0, E, Δ, C). S represents the state set. S_0 represents the initial state set. E is the event set. Each event is a pair, (e_t, e_d), where e_t represents the type of the event and e_d represents the data associated with it. C is a set of event handlers, invariant to state and event space of the state machine, i.e. the code in the event handler does not influence the state transition or the output events emitted. Δ is the set of transitions satisfying $\Delta : S \times E \to S \times 2^E \times C$, where 2^E is the power set of E.

A transition, $\delta_1 : s_1 \times e_0 \to s_2 \times E_{out} \times c_{\delta_1}$ is enabled if the state machine is in state s_1 and event e_0 (enabling event of the transition) appears. E_{out} is the set of output events. c_{δ_1} is the event handler, which is a piece of executable code not synchronized with any events. The firing of an enabled transition has three steps: (1) e`xecute the event handler, (2) emit the output events, E_{out}, and (3) change the state of the state machine from s_1 to s_2. The enabling event of a transition is consumed when it is fired. The firing of a transition is atomic.

Each event has a priority. A transition inherits the priority from its enabling event. An enabled transition of the highest priority is

fired. If there are multiple instances of the same event, the one that arrived the earliest is consumed. If multiple instances appear simultaneously, a random one is picked. Because the firing of a transition is atomic, new events are blocked. Besides priority, an event has the additional attributes of *stickiness*, *reliability*, and *recognition type*. An event is either *sticky* or *non-sticky*. Assume the events visible to a state machine before the firing of transition δ_1 are E_1, after the firing are E_2. Obviously, $E_{out} \subseteq E_2$. An event e is *sticky* iff $\forall \delta_1, e \in E_1, e \neq e_0 \Rightarrow e \in E_2$, that is, a sticky event remains visible unless it is consumed. An *unreliable* event may not be captured by the state machine even if it occurs, while a *reliable* event is guaranteed to be captured. Event reliability models the features of a hardware signal. Recognition type has two possibilities. We can define an event to be recognized by polling. By default, an event notifies its appearance by itself.

Input events are sticky. An output event is *internal* if it is invisible outside the state machine; otherwise it is *external*. An output event is always reliable. Internal output events are non-sticky because they represent execution status rather than synchronization states. External output events are sent to a particular state machine.

6.2 Control Function Model

A control function is defined as a series of state transitions of a single state machine. Because events being exchanged between different state machines enable state transitions, an execution of a control function might involve several state machines. We define two forms of control functions.

F1) A pair of an event and a set of destination states. The control function is issued when the event is emitted to the state machine and completes when the state machine reaches a state in the destination state set.
F2) A pair of events. The first event enables the request. It can be empty. The second event is an output event. When the state machine emits the output event, the control function ends.

A control function is referred to as a *request* in the following sections as it models a request from the device by the software. Request functions can overlap because a request function takes several transitions to complete and events are visible at the states. Five policies to coordinate two requests of the different types, and different instances of the same type, are defined. They are declaratively specified for the control functions.

P1) Coexist.
P2) Block the new request until the previous one finishes.
P3) Abort the previous request and execute the new request from the current state.
P4) Abort the previous request and execute the new request from the state where the previous request starts.
P5) Suspend the previous request and resume it after the new one completes its execution.

Assume requests A and B coexist. Request C is blocked by A and aborts B. When C appears while A and B are executing, a dilemma occurs: whether to execute or to block B.

- Consider A and C first. C is blocked while A executes. Then consider B. Because A and B coexist, the result is that A and B execute and C is blocked.
- Consider B and C first. B is blocked and C executes. Then consider A. Because C is blocked by A, the result is A executes, B and C are blocked.

We say coordination policies are *incompatible* when there is an inconsistency in deciding whether to execute a request, or transition to the next state of the state machine. Compatibility can be checked by the synthesizer.

6.3 Synchronization and Communication Model of the State Machines

Two message-passing based communication modes, asynchronous and synchronous, are defined.

- A state machine M_1 communicates *asynchronously* with a state machine M_2 if it calls a request of M_2 and does not block for the end of the request. Such requests are typically F2 type requests.
- A state machine M_1 communicates *synchronously* with state machine M_2 if its event handler calls a request of M_2 and blocks for the end of the request. Such requests are typically F1 type requests.

To call a request, an output event is emitted and the state machine accepts an input event at a certain point. The original event handler is defined to be invariant of state and event space of the state machine. To simplify the description, we say an event handler calls a request by putting output event emission (and blocking for the end of request, for synchronous communication) in the handler. We say a request f is *nested* in a request r if an event handler of r calls f. The communication between event-driven finite state machines may enforce an execution order between them. When two state machines are not ordered, their execution order is arbitrary. Clearly, the execution order changes as the execution progresses.

The event driven finite state machines are synchronized when they share resources. Resources are shared between event handlers of different state machines. Each resource is managed by a state machine. The resource users communicate with the resource management state machines.

6.4 Timing Model of Event Handlers

Because of the complexity and speed of software execution, we cannot ignore the execution time of event handlers. We capture the timing behavior of event handlers by its worst case execution time (WCET) because our main concern is the relationship between atomic event handler execution and event occurrence.

We define chained execution of event handlers as executing several handlers without processing other events. For example, an event handler h may call a request r synchronously. If we execute r after h without processing a new event when h ends, we say we chain h and r. Chaining happens only when request nesting occurs. Let $W_H(h)$ denote the WCET of chained event handles started by h. $T_{WCET}(c)$ denotes the WCET of a piece of code c. Because of chaining,, $W_H(h)$ is not simply $T_{WCET}(h)$. When the state machine is in different states, the handler enabled by an

event e differs. For an event e, define H(e) as the set of event handlers enabled by event e, and define W_E as

$$W_E(e) = \underset{\forall h \in H(e)}{MAX}(W_H(h)) .$$

The timing model of chained event handlers, started by h, $W_H(h)$, is defined as:

$$W_H(h) = \begin{cases} W_E(e), & h = sync(e,S) \\ T_{WCET}(h) + W_E(e), & h = async(e) \\ T_{WCET}(h), & Otherwise. \end{cases}$$

When the event handler contains a synchronous request (type F1), the time cost is estimated as the maximum WCET of all the event handlers enabled by the request event. If the event handler generates an output event, the WCET of the event handler is sum of the WCET of the handler code itself and the maximum time cost of all the event handlers enabled by the output events. Otherwise, the time cost of an event handler without synchronization is defined as the WCET of the code. Clearly, if the chaining has a loop, $W_H(h) = \infty$.

6.5 Modeling Device Access Behavior with Extended Event Driven Finite State machines

As illustrated in Section 5, device access behavior has two parts: data channel and device control. Both of these parts are modeled by extended event driven finite state machines, though a data port access function conforms to one of a set of templates.

We define five types of input events according to their appearance and functionality: *interrupt*, *timer*, *task request*, *interrupt request*, and *asynchronous input*. A *timer* event occurs when the timer expires. An *interrupt request* event occurs when an interrupt needs to be serviced. A *task request* event is emitted when a task, user (or kernel) process (or thread), issues a service request. Other interrupts are grouped together as *interrupt* events. An *asynchronous input* is the output of a type F2 request of a state machine. A timer event is *obsolete* if the state machine no longer waits for it. The obsoleteness of the timer event models event occurrence timeout. Moreover, a state has at most one up-to-date timer event.

To assign priorities to the input events, we classify the input events according to their appearances. An asynchronous input inherits the priority of the event enabling the transition which emits it. The interrupt related events, including interrupt and interrupt request events, have the highest priorities. The task request events have the lowest priority. The timer events have a priority in the middle. Because of the obsoleteness of the timer event, all the timer events share the same priority. The task requests share the same priority though we can prioritize them. Interrupt events may have multiple priority levels. This event priority assignment scheme agrees with the execution level definition of operating systems. Because most operating systems define similar execution levels for interrupts, timer and system calls, this scheme is compatible with a range of operating systems.

An event could be unreliable. It explains the mysterious behavior of a line printer driver as highlighted in [7]: the driver hangs although the program is logically perfect and the printer works fine. The line printer model has a channel with a write port. The synchronization event of the write port data unit is an interrupt indicating that the buffered data are printed. Unfortunately, the interrupt is easily lost, i.e., not captured by the processor. The data access state machine is stuck in a state blocking for the interrupt. To avoid this situation, we associate a time out value with an unstable event indicating its latest arrival time if it appears.

6.6 Composition of Extended Event Driven Finite State Machines

We define a *sub-device* as a set of device controller state machines and data port access state machines. Each state machine belongs to exactly one sub-device. According to our experience with device drivers, we define four types of resources shared by the device controller and data port access state machines. The shared resources defined are the interface (ITF) shared between sub-devices, data (SH_{data}) shared by data channels and device controller requests, system resources such as dynamically allocated DMA shared by data channels, and physical data channel (P_{CH}) shared by the two port accesses of the same channel. The communication between the state machines is also constrained because M_{data} is responsible for transfering data while M_{ctrl} is responsible for assigning the data transfer task and coordinating the entire device processor interaction process. The communication and synchronization between the device controller state machine (M_{ctrl}) and data port access state machine (M_{data}) complies with the following rules.

- Cmp1) Requests defined in M_{data} do not nest in each other.
- Cmp2) An M_{data} (including its nested requests) only communicates with a M_{ctrl} in the same sub-device.
- Cmp3) M_{data} share DMA. An M_{data} requires at most 1 DMA.
- Cmp4) Each state machine requires one ITF.
- Cmp5) SH_{data} are shared between M_{ctrl} requests and M_{data} requests.
- Cmp6) Two M_{data} (read, write) of the same channel share a P_{CH}. A M_{data} requires one P_{CH}.

6.7 Correct Implementation Model

The correct-by-construction code synthesis technique requires a proof of the correctness of the implementation of the event processing mechanism. We say an event processing mechanism is correct if it satisfies the following conditions. The event processing mechanism designer (synthesis technique) should guarantee that the conditions are satisfied.

- E1) The sticky events are queued.
- E2) The enabled transition, the enabling event of which is at the highest priority, is fired.
- E3) A state machine with an enabled transition is executed at most *T* time units after the transition is enabled (the time the enabling event occurs, not the time it is checked). T is a constant for the state machines. However, the transition that fires is not required to be the first transition enabled.

E3 guarantees the fairness of the implementation, i.e., a state machine with an enabled transition is executed in bounded time. E3 also limits the highest frequency of the events of high priority.

6.8 Model Soundness

To guarantee the deterministic behavior and finite request execution time, we impose certain conditions on the model. A model is *sound* if and only if it satisfies the following:

S1) Request nesting is not conditional.
S2) Request coordination policies are compatible.
S3) The request terminates in finite time assuming all the events are stable and the input events occur eventually.
S4) Composition rule (Section 6.6) of event driven finite state machine is satisfied.
S5) No circular shared resource requirements.
S6) The execution result is not affected by the execution order of the state machines.

Rule S1 constrains our synthesis method to a statically defined communication of the state machines. Rule S2 is necessary to guarantee deterministic behavior of the model. Rule S3 is necessary to guarantee finite request execution time. Rule S4 constrains the network of event driven finite state machines to one suitable to device modeling. Rule S5 is necessary to ensure deadlock free. The soundness constraint S6 avoids the following dilemma. The state machine M_1 is in state S_1 and event e_1 occurs at t_1. However, M_1 is not executed until event e_2 occurs at time t_2. The priority of e_2 is higher than e_1. If M_1 executes between t_1 and t_2, e_1 is consumed. After t_2, e_2 is consumed. If the correctness is dependent on which event is consumed, the model is not sound. Given a correct implementation scheme and a model which does not satisfy soundness constraint S6, we either restructure the state machine or require $t_2 - t_1 > T$ to make it sound.

Rule S1 and S2 are checked statically. S3 is reducible to the reachability problem and can be checked by tools for state space traversal such as SMV [9]. Rule S4 items are either enforced by the specification or checked statically. The checking of rule S5 is discussed in Section 7.1. S6 reduces to equivalence under asynchronous communication. Formally this can be checked using formal verification techniques. However, informally the designer may be able to convince themselves of this based on explicit use of synchronization when needed, much as is done in parallel programming. We assume the latter case for now.

7. OPERATING SYSTEM BASED DEVICE DRIVER SYNTHESIS

The device driver synthesis procedure generates data structures and functions for a sound model. As part of the methodology mentioned in [1], we focus on synthesizing core functions modeled by event driven finite state machines. It contains three parts: handling unstable events, implementing the request coordination policy, and designing event processing mechanisms and resource sharing schemes. The unreliable events of the original finite state machines are switched to reliable ones by inserting new transitions, (S_1, S_{err}), enabled by time out events, where S_{err} is an error state. The time out value is the maximum time out value of the unstable events on the out edges of the state S_1. The coordination policy implementation includes request state definition and processing specific to each coordination rule. Because of the lack of space, we omit the details of the algorithms.

7.1 Resource Sharing Scheme

According to Section 6.6, four types of shared resources, ITF, SH_{data}, DMA, and P_{CH}, are defined. The resource acquiring and releasing operations follow the following rules.

R1) An ITF is acquired right before a transition fires. The current ITF is released and a new ITF is acquired when we execute a transition in another state machine with a different ITF.
R2) If a request requires a SH_{data}, it acquires SH_{data} after the ITF of the first transition is obtained and holds SH_{data} until the request ends. If SH_{data} is not obtained, the ITF is released.
R3) Dynamically allocated DMA is acquired when data unit executes (after SH_{data} is acquired). It is held until data unit execution finishes.
R4) A P_{CH} is required at the start of the data unit execution. If DMA is used, it is required after DMA is obtained. It is held until the data unit execution ends.

Given the resource sharing scheme and soundness rule S4, we conclude that the soundness rule S5 is guaranteed.

Proof:
Assume we have a circular resource requirement, (A_1, A_2, …, A_K, A_1), where A_i (i=1, …, K) is a shared resource.

- A P_{CH} is not in the circle. Data unit operations are essentially event handlers of data port access state machines. According to Cmp1 and Cmp2, a M_{data} never requires an ITF after SH_{data} is acquired. Given this fact, according to R3, a request never requires an ITF or SH_{data} after a DMA is held. Hence, according to Cmp6 and R4, P_{CH} is always the last resource to be acquired.
- A DMA is not in the circle. According to R3, R4 and Cmp3, the only possible resource required while holding a DMA is P_{CH}. Because a P_{CH} is not in the circle, DMA is not in the circle.
- According to Cmp1, a request never requires a SH_{data} while holding one. According to Cmp2, a request never requires a SH_{data} while holding an ITF. Because of the previous 2 items, SH_{data} is not in the circle.
- According to R1, there is not an ITF circle.

Hence the circular resource requirement cannot exist.

7.2 Event Processing Mechanism

Because the device behavior is modeled as multiple communicating event-driven finite state machines, our main objective is to design the event processing mechanism. Each state machine is a schedulable entity managed by the event processing kernel. A traditional event processing kernel queues the events and executes the schedulable entities. Figure 3 shows the sketch of the code. It has a background kernel thread dedicated to event processing. The events in the event queue are ordered according to their priorities. A key feature of our solution is to take advantage of the services provided by the operating system and remove the explicit central event manager and the event queue. The code sketch is shown in Figure 4. It is based on the observation that a lower level event has no chance to be issued when a higher level transition fires on a single CPU because the event priority assignment agrees with the operating system

```
Interrupt (Timer) handler, User request: {
  Enqueue the event to the dest FSM atomically;
  Block for the end of the request if it is a request;
}

Event process kernel (a kernel thread):
For each state machine {
      For each event {
            if (enabled transitions exist)
                  Fire it;
                  Enqueue the output events;
                  Delete the enabling event;
                  Unblock a request ends;
            else
                  if (event is non-sticky) remove it;
    }
}
```

Figure 3. Traditional event processing kernel

execution level definition. Sticky events are simulated by registering a timer handler simulating the occurrence of the event repeatedly, that is, we poll for the consumption of the event. For the multiple CPU case, primitives such as spin locks are used. The preemption mechanism is calculated according to the discussion in Section 4.

The scheme shown in Figure 4 has lower cost and predictable execution time. Besides the saving of the queue management operations, we do not maintain a background thread and save context switches. Moreover, the gap between when a background thread becomes ready and it executes is not predictable for a non-real-time operating system. This is hazardous for a device driver. For example, the USB device controller receives a command from the host. If the command is not processed in time, a protocol error occurs. Figure 4 shows an extreme case of zero synchronization time and NULL queue maintenance which is useful for a class of drivers (such as devices in our case studies). We prove that for a model satisfying soundness rule S1, the scheme (Figure 4) is a correct implementation scheme when the following constraints are satisfied. These constraints should be checked by the synthesizer.

C1) There are no internal output events.

C2) Let f_p denote the highest frequency of the events of priority p. Define W_p as the longest execution time of the code executing at priority p without processing a new event, that is,

$$W_P(p) = \max_{\forall e, P(e)=p}(W_E(e)) \ .$$

The input event sequence satisfies the following condition. If two events can overwrite each other, they have the same priority, say P_0. Their occurrence interval, T_{inter}, satisfies

$$T_{inter} > \max_{\forall p, p<P_0}(W_P(p)) \Big/ 1 - \sum_{\forall p, p \geq p_0}(W_P(p) \times f_p)$$

when preemption can be disabled.

C3) The caller of an F2 type request always blocks for the end of it.

C4) When an F1 type request of a state machine is called, the machine is always ready to receive the request.

C5) $1 > \sum_{\forall p}(W_P(p) \times f_P)$ is satisfied.

Proof:

C1 guarantees E1 for internal events. Because the event priority definition agrees with operating system code execution level definition, S1, C3 and C4 guarantee E2 and E1 for the algorithm shown in Figure 4.

```
Execute FSM with event e: {
     Fire the enabled transition of highest priority;
     if (F2 typed request (r) ends)
            execute FSM(r's caller) with the output event;
     else if (F1 types request (r) is activated)
             execute the FSM the request resides
             with the request evt;
}

Interrupt (Timer) Handler: {
   Require shared resources if necessary;
   Disable preemption;
   Execute FSM with interrupt (or timer) events;
   Enable preemption;
   Release shared resources if necessary;
}

User request: {
   Request coordination and start a request;
   Require shared resources if necessary;
   Disable preemption;
   Execute FSM with event (user request);
   Block for destination & Enable preemption;
   Release interface and data shared if necessary;
   Request coordination (cleanup operations);
}
```

Figure 4. OS based event processing

Assume the minimum interval between two over-writable events is T_{inter}. In this period, there are at most (T_{inter}*f_p) instances of events of priority p. Two types of code may block the execution of the event handlers of event with priority p: event handlers of higher (or the same) priority events and lower priority event with preemption disabled. Because an event handler with a higher priority is selected each time a handler finishes, the event handler of higher priority is blocked by at most one event handler of lower priority. Hence, we have

$$T_{inter} > \max_{\forall p < p_0}(W_P(p)) + \sum_{\forall p, p \geq p_0}(T_{inter} \times f_P \times W_P(p)) \ ,$$

which results in C2 guaranteeing E1 for input events, that is, input events are not lost.

Similarly, in time period T mentioned in E3, there are at most (T*f_p) instances of events of priority p. The WCET of the event handlers satisfies $T > \sum_{\forall p}(T \times f_p \times W_P(p))$, i.e., C5 guarantees E3.

8. CASE STUDIES

We demonstrate our device driver synthesis techniques with two case studies: the infrared port (IRDA) for the Intel Pentium III mobile processor and the USB device controller (UDC) for an Intel StrongArm SA1100 based handheld. These peripherals are among the most popular peripherals for SoC embedded processors.

IRDA An infrared port is modeled as a single sub-device consisting of a data channel and a device controller. The channel has two ports, read and write, which are asynchronous and

blocking respectively. The device controller has three control functions: *start*, *stop*, and *configure*. Besides the request events, *start*, *stop*, and *configure*, a poll event (recognition type) representing the emptiness of the transmitting buffer is defined. The baud rate setting data is shared between the device controller and the port access state machines. Line discipline (a special terminal interface) and network interface are defined for the driver. Because the infrared port matches the model of the line discipline model perfectly, low level access functions such as register FIFO accesses are already included in the line discipline library functions. Hence the device programming interface is not necessary.

To test the correctness of the driver, we bind the IRDA protocol stack on top of the driver and setup the PPP connection between an IBM T23 (Redhat 8 with linux kernel 2.4.18) and a DELL Inspiron 5000 (RedHat 9.0 with linux kernel 2.4.20). The correctness of driver is verified successfully by executing the *scp* command correctly.

UDC The USB device controller of SA1100 is modeled as a single sub-device consisting of a device controller and 3 data channels, ep0, ep1, and ep2. Channel ep0 has two ports, read and write, which are asynchronous and blocking respectively. Channel ep1 has only an asynchronous read port. Channel ep2 has only a blocking write port. Channel ep1 and ep2 have pre-allocated DMA channels. The device controller includes the specification of the USB device controller setup protocol. The read port access of ep0 communicates asynchronously with the device controller. The device controller communicates synchronously with the write port access of ep0. A network interface is defined for it.

To test the correctness of the driver, we bind the IP socket layer on top of the driver and set up the USB connection between the HP iPaq3600 handheld (Familiar v0.5.3 with linux kernel 2.4.17) and a 686 based desktop (Redhat 7.2 with linux kernel 2.4.9). The correctness of the driver is tested successfully by executing the standard TCP/IP command *ping* correctly.

Table 1 Line count of IRDA and UDC drivers

Device	Iris Specification	Linux source code	Synthesized (library excluded)
IRDA	30	775	283
UDC	620	2002	971

We use the reduction of line count as a metric to evaluate our methodology's ability to increase productivity. Although line count is not a perfect measure of software complexity, it is a first order estimate of how much effort is required to develop the software. Table 1 shows the line count comparison of the specification, the original Linux driver (may include several files), and the synthesized code (library and templates are excluded). From Table 1 we see a significant reduction in the size of the specification compared to the original Linux driver which is a strong indication of increased productivity. Moreover, our specification style helps the developer to understand the device behavior and thus is more suitable to driver development compared with a general language such as C. Further, the structure of the driver is provably correct because both the models are sound and satisfy implementation correctness constraints (C1 to C5).

9. CONCLUSIONS

This paper addresses the issue of modeling device access behavior with a formal model, *viz*. extended event driven finite state machines, and using it to synthesize a correct-by-construction operating system based device driver. We apply this methodology to develop device drivers for two devices: an infrared port and the USB device controller of a SA110 based handheld. The experiments show a 70% reduction of specification size which is an indication of increased productivity.

10. REFERENCES

[1] Shaojie Wang, Sharad Malik, and Reinaldo A. Bergamaschi, Modeling and Integration of Peripheral Devices in Embedded Systems, DATE 2003.

[2] Mattias O'Nil, and Axel Jantsch, Device Driver and DMA Controller Synthesis from HW/SW Communication protocol specifications, Design Automation for Embedded Systems, Vol 6, 2001. pp 177-205.

[3] Dirk Desmet, D. Verkest, and Hugo De Man, Operating system based software generation for system-on-chip, DAC 2000.

[4] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski, RTOS Modeling for System Level Design, DATE 2003.

[5] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, Hardware-Software Co-Design of Embedded Systems: The Polis Approach, Kluwer Academic Press, 1997.

[6] I. Bolsen, H. J. De Man, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest, Hardware/software co-design of digital telecommunication systems, Proceeding of the IEEE, Vol 85, 1997. pp391-418.

[7] George Pajari, Writing UNIX Device Drivers, Addison-Wesley Publishing Company, Inc., 1992.

[8] Johnny Oberg, Anshul Kumar, and Ahmed Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols", In Proc. of the 9th ISSS, Nov. 1996, pp14-19

[9] K. L. McMillan, Symbolic Model Checking: An approach to the state explosion problem, CMU Tech Rpt. CMU-CS-92-131.