Synthesizing Device Drivers with Ghost Writer

Bingyao Wang
University of British Columbia
Vancouver, Canada
bingyao@cs.ubc.ca

Sepehr Noorafshan
University of British Columbia
Vancouver, Canada
snoora@cs.ubc.ca

Reto Achermann
University of British Columbia
Vancouver, Canada
achreto@cs.ubc.ca

Margo Seltzer
University of British Columbia
Vancouver, Canada
mseltzer@cs.ubc.ca

Abstract

Device drivers are components that enable operating systems to interact with devices. Unfortunately, they are the main source of bugs in operating systems, because writing a driver is an intricate and error-prone process that requires extensive knowledge of devices and operating systems. Furthermore, supporting new devices and accommodating kernel revisions require significant development effort. To facilitate the development of device drivers, we present Ghost Writer, an end-to-end toolchain that allows developers to synthesize correct-by-construction device drivers from high-level specifications. Ghost Writer supports control and data plane operations (e.g., handling DMA transactions). It makes synthesis tractable by 1) modeling the device interface as a set of virtual registers that abstract the hardware details and 2) leveraging behavior trees to model operations on virtual registers and synthesize complex operations from simpler ones. Our prototype can synthesize `putc` for the PL011 UART device and `send_packet` for the VirtIO network device. We believe that Ghost Writer can be the foundation towards automating the development of correct-by-construction device drivers.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Automatic programming**.

ACM Reference Format:
Bingyao Wang, Sepehr Noorafshan, Reto Achermann, and Margo Seltzer. 2023. Synthesizing Device Drivers with Ghost Writer. In *12th Workshop on Programming Languages and Operating Systems (PLOS '23), October 23, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3623759.3624545

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PLOS '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0404-8/23/10.
https://doi.org/10.1145/3623759.3624545

Figure 1. Illustration of the burdensome and error-prone process of developing device drivers for a particular operating system.

1 Introduction

Modern computers can support numerous devices, such as GPUs, SSDs, and Wi-Fi cards, each of which requires a driver. Device drivers are the interface between the operating system and devices, enabling a seamless exchange of commands and data; they are crucial components and comprise most of the operating system code [19].

However, device drivers are difficult and tedious to write. Developers need to understand the device manual, the protocol specification, and the kernel APIs as shown in Figure 1; otherwise, they might misconfigure the device [20], violate the protocol [2], or introduce performance bugs [28]. Since most drivers run in kernel space, a faulty driver makes a system vulnerable [12, 32]. As such, researchers have proposed various techniques to isolate drivers [22] and tolerate faults in them [27]. Commodity operating system vendors, such as Apple, have moved certain drivers, such as Ethernet and USB, to user space [3]. Unfortunately, none of these approaches eliminates the vast number of bugs found in device drivers [6].

Device drivers are tightly coupled with a particular operating system, since they rely on kernel functions and programming models such as physical memory allocators and synchronization primitives [16]. Therefore, porting a driver from one operating system to another requires either a significant rewrite or a compatibility layer that emulates the source operating system's APIs [14]. However, emulation comes at the cost of performance and maintenance effort. The ported driver might use the target operating system's APIs in a suboptimal way as shown in Figure 2. Moreover, since operating systems' APIs are not stable, developers need

```
void* kzalloc(size_t size, gfp_t flags)
{
    return IOMallocZero(size);
}
```

Figure 2. A naïve emulation of Linux's `kzalloc()` using `IOMallocZero()` on macOS. The implementation ignores the second argument, `flags`, which specifies the type of memory to allocate. If the ported driver intends to allocate memory for DMA transactions, it should use `IOBufferMemoryDescriptor` to avoid creating bounce buffers [4].

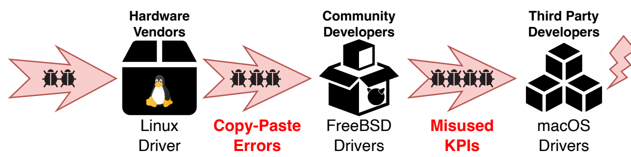

Figure 3. Illustration of the dependency chain caused by porting device drivers to multiple operating systems.

to keep the compatibility layer up to date to support new kernels while ensuring backward compatibility.

Practically speaking, device driver developers often treat Linux drivers as reference implementations, because official device manuals are not always available. If that reference driver is buggy, ported ones likely inherit the bugs. Additionally, developers need to keep track of upstream changes and, if necessary, integrate them into the ported driver, which creates a long dependency chain shown in Figure 3.

To facilitate writing and porting device drivers, we introduce Ghost Writer, an end-to-end toolchain that transforms a high-level device specification into a C, C++, or Rust driver implementation. Ghost Writer leverages a novel device driver synthesis technique based on behavior trees (Section 3). Its user-guided search algorithm exploits the layered architecture present in modern device drivers and synthesizes drivers hierarchically (Section 4.1). This approach decouples the semantics of high-level device operations from their implementation and enables search space reductions that make the synthesis process efficient (Section 4.2).

Our prototype, while not yet feature complete, provides preliminary evidence about the feasibility of the approach and is already capable of synthesizing simple driver functions (e.g., transmitting a character over a UART) and DMA transactions (e.g., sending a network packet over a VirtIO network card). Ghost Writer provides the foundation for the next steps toward automating the development of correct-by-construction device drivers (Section 5).

2 Background

A behavior tree is a mathematical model of task planning based on observations in a system [10]. They are widely used in video game development to model the behavior of non-player characters [18] and in robotics to control robots in a dynamic environment [23]. Behavior trees have been shown to generalize traditional control structures such as finite state machines, hierarchical state machines, and decision trees [8]. However, their key strength is enabling hierarchical construction of complex tasks from simple ones.

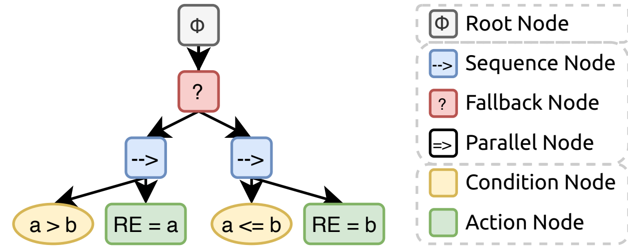

Figure 4. Illustration of a behavior tree in standard notation, which can be used to write the maximum of two numbers, `a` and `b`, to the register `RE`.

A behavior tree is composed of four types of nodes: 1) A single *root node* denotes the entry point of a behavior tree and has exactly one child. 2) *Control flow nodes* are internal nodes that define an execution policy for their children, such as sequence, fallback, and parallel. 3) *Execution nodes* are leaf nodes that perform operations: condition nodes evaluate predicates, while action nodes execute user-defined instructions. 4) *Decorator nodes* have a single child and define custom behaviors, such as loops and negation. Figure 4 shows a behavior tree that models computing the maximum of two numbers and writing the result to a register.

Execution in behavior trees resembles function calls. When a behavior tree is executed, the root node invokes its child, which then executes and returns `success` if the execution succeeds or `failure` otherwise. Control flow nodes use the status a child returns to determine whether to execute the next child. Specifically, a sequence node executes its children in turn and immediately returns `failure` if a child fails (i.e., it implements a logical AND of all its children). A fallback node is similar to a sequence node but immediately returns `success` as soon as any child succeeds and returns `failure` only if no child succeeds (i.e., it implements a short-circuit OR among its children). A parallel node runs all its children in parallel and returns `success` if a developer-specified number of its children succeed.

3 Ghost Writer

Ghost Writer (Figure 5) is composed of three components: specifications, a synthesizer, and a code generator. To illustrate how these components interact, we synthesize `putc(char ch)` for the PrimeCell UART PL011 [5] as a running example.

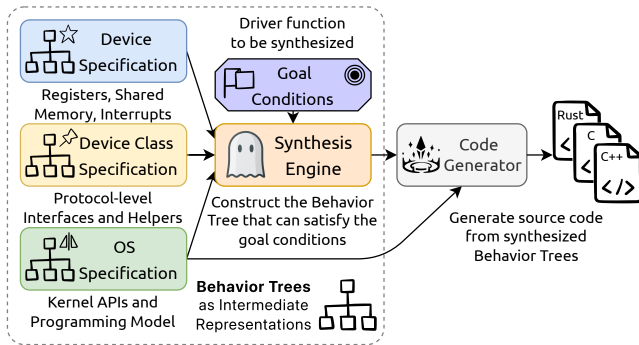

Figure 5. Illustration of interactions between Ghost Writer components while synthesizing a driver function.

3.1 Specifications

Ghost Writer uses three different types of specifications: **a device specification**, which describes a device's behavior, **a device class specification**, which describes the behavior common to all devices in a particular category (e.g., UART), and **an operating system specification**, which describes the kernel programming interface. Ideally, these specifications are provided by the device vendors, the standards organizations, and the operating system vendors respectively.

3.1.1 Device Specifications. A device specification specifies the device interface and is composed of two parts: a **register definition file** and a **behavior definition file**.

The register definition file defines a set of *virtual registers*, which represent device registers, individual bitfields, and in-memory data structures. Each virtual register has a single responsibility and a limited set of possible values. The top half of Figure 6 defines two registers, UARTDR and UARTFR, which expose ten virtual registers. UARTDR stores the character to be transmitted and exposes itself as a virtual register. UARTFR stores the device status and exposes each of its bitfields as a virtual register.

The behavior definition file specifies the operations on virtual registers as primitive action nodes, such as *read*, *write*, and *wait*. Primitive action nodes provide the synthesizer with a unified interface for accessing device registers and in-memory data structures. The bottom half of Figure 6 defines the write operation on UARTDR and the wait operation on UARTFR.BUSY. The device can transmit a character only if it is not busy transmitting data. We specify this behavior using a precondition (Section 4.2.2), which indicates that the value of the bitfield BUSY must be zero before the driver writes a character to UARTDR. The driver uses the wait action on BUSY to wait until the device is no longer busy. Once the driver has written the character to UARTDR, the device transmits the character. We use a postcondition (Section 4.2.2), UARTCharTransmitted, to indicate that the character has been transmitted.

```
// Register Definition File
registers MMIO
{
    // Define the 8-bit data register at 0x000
    register rw UARTDR: UInt8 @ 0x000;

    // Define the 16-bit flag register at 0x018
    register ro UARTFR: LittleUInt16 @ 0x018
    {
        // Bits 9:15 are reserved
        bitfield ro CTS  @ 0:0;
        bitfield ro DSR  @ 1:1;
        bitfield ro DCD  @ 2:2;
        bitfield ro BUSY @ 3:3;
        bitfield ro RXFE @ 4:4;
        bitfield ro TXFF @ 5:5;
        bitfield ro RXFF @ 6:6;
        bitfield ro TXFE @ 7:7;
        bitfield ro RI   @ 8:8;
    };
};
-------------------------------------------------------------
// Behavior Definition File
WriteRegister(MMIO.UARTDR, value)
    - Preconditions: MMIO.UARTFR.BUSY == 0
    - Postconditions: UARTCharTransmitted(value)

WaitForRegister(MMIO.UARTFR.BUSY, 0)
    - Postconditions: MMIO.UARTFR.BUSY == 0
```

Figure 6. A simplified device specification for the PL011. Note that the concepts of virtual registers and behavior trees are independent of the specification syntax.

3.1.2 Device Class Specifications. A device class specification is similar to a device specification but defines virtual registers and primitive action nodes common to devices belonging to a particular category. Additionally, it may define predicates that can be used in pre- and post-conditions. These predicates connect the device to the operating system without exposing device internals. For example, all UART devices [11], including the PL011, support transmitting characters. Thus, the UART device class specification defines the predicate UARTCharTransmitted, which acknowledges operating system requests while hiding how a particular device transmits data.

3.1.3 Operating System Specifications. An operating system specification describes the interface to kernel service routines, such as physical memory allocators and timers, and the interface the driver must implement. For example, all UART device drivers must implement the interface putc(char ch) that the operating system uses to transmit a character while remaining unaware of how the driver satisfies the request. As such, putc has a postcondition of UARTCharTransmitted(ch) as shown in Figure 7. This guarantees that the character ch has been transmitted after the function returns.

```
// Operating System Specification
putc(char ch)
    - Postconditions: UARTCharTransmitted(ch)
```

Figure 7. Specification of the interface `putc` that a UART driver must implement.

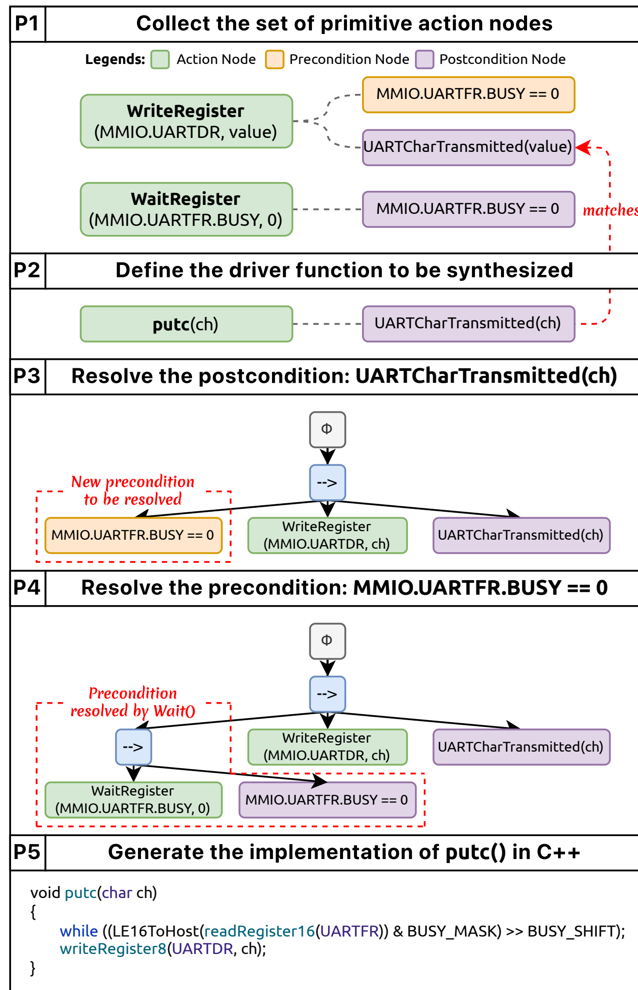

Figure 8. Illustration of the process of synthesizing `putc()` for PL011.

3.2 Synthesizer and Code Generator

Ghost Writer uses behavior trees as intermediate representations during synthesis and code generation. The synthesizer is responsible for constructing a behavior tree from primitive action nodes. The code generator then compiles the synthesized behavior tree to an implementation. Figure 8 depicts an example of this process by synthesizing `putc` for the PL011 in C++.

Given a set of primitive action nodes (e.g., `write` and `wait` in P1) and a driver function to be synthesized (e.g., `putc` in P2), the synthesizer constructs a behavior tree that represents the function implementation. The synthesizer first finds a primitive action node that can satisfy the postcondition of the driver function (P2). It then places the found action node along with its pre- and post-conditions in the behavior tree using a sequence node (P3). After that, the synthesizer recursively resolves the preconditions of the found action node using other primitive action nodes, expanding the behavior tree until all preconditions are satisfied (P4).

In the PL011 example, the synthesizer finds `write(UARTDR)` to satisfy `putc`'s postcondition. However, this `write` action has a precondition that `UARTFR.BUSY` must be zero, so the synthesizer finds `wait(UARTFR.BUSY)` to satisfy that precondition. Since the `wait` action has no precondition, the synthesizer finishes constructing the behavior tree. Finally, the code generator takes the synthesized behavior tree along with the specification of the target operating system and generates an implementation in C++ (P5).

Behavior trees exhibit three key characteristics that make them well-suited for synthesizing device drivers. First, behavior trees differ from abstract syntax trees, which are widely used in traditional syntax-guided synthesis [1], in that they do not incorporate programming language specifics within their structure. This draws a distinct line between the synthesizer that discovers the correct driver logic and the code generator that ensures correct translation to a driver implementation. Second, behavior trees are reusable and composable, allowing the synthesizer to synthesize simple driver functions and use them as building blocks to synthesize higher-level complex functions (Section 4.1). Third, behavior trees are expressive and flexible. Existing control flow and execution nodes are sufficient to model common device and driver behavior, but when needed, Ghost Writer can customize them and introduce new nodes to make the synthesis more tractable (Section 4.2).

4 Methodology

4.1 User-Guided Hierarchical Synthesis

Modern device drivers have a layered architecture as depicted in Figure 9. Developers need to implement the interface required by an upper layer (e.g., block storage) using the interface provided by a lower layer (e.g., PCIe transport). A well-structured architecture allows developers to reuse as much code as possible and maintain the driver easily. Ghost Writer follows the same paradigm: Developers decompose a driver into multiple layers, specify the interface of each layer, and use the synthesizer to synthesize the implementation of each interface. Figure 10 demonstrates a driver decomposed into $N + 1$ layers, from layer 0, which provides access to the device, to layer N, which defines the operating system interfaces the driver must implement.

The synthesizer uses the primitive action nodes provided by some layer K where $0 \le K \le N$ (1) to construct a behavior tree for a function defined in layer $K + 1$ (2). When

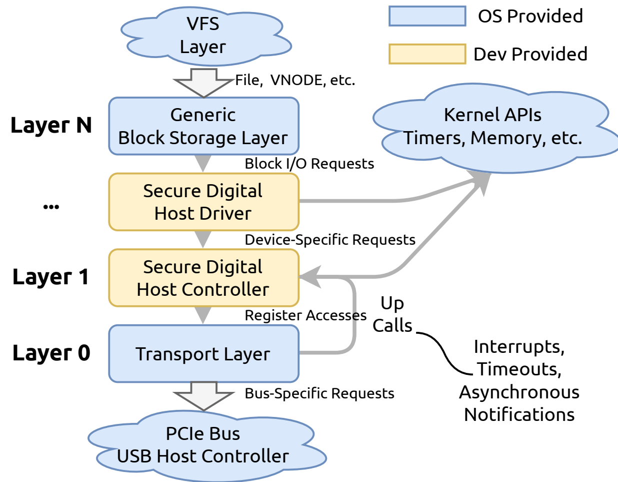

Figure 9. Illustration of a conventional driver stack. Developers use the APIs provided by the transport layer to access the Secure Digital (SD) host controller device and implement the block storage interface for an SD card reader.

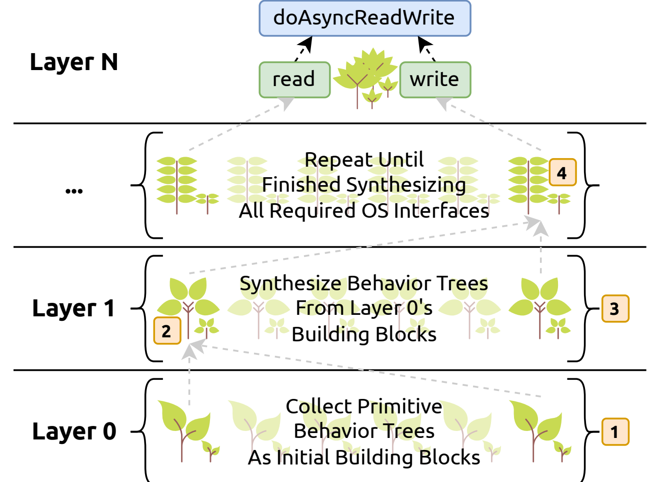

Figure 10. Illustration of a driver synthesis stack that resembles the conventional driver stack. The synthesizer uses the primitive action nodes provided by a lower layer to synthesize functions defined in an upper layer.

K is zero, the primitive action nodes are imported from the device and the device class specifications. After synthesizing all the functions in layer $K + 1$, the synthesizer uses them as primitive action nodes (3) to construct behavior trees for layer $K+2$ (4). To expose a behavior tree as a primitive action node, we encapsulate it into an *Action View* node, which can be associated with specialization constraints, preconditions, and postconditions (Section 4.2.3). Additionally, the synthesizer only needs the interface of each primitive action node during synthesis, allowing developers to synthesize multiple layers in parallel. Our hierarchical synthesis approach makes it possible to synthesize individual driver functions efficiently and at scale.

4.2 Constrained Search Space

One major challenge in program synthesis is the combinatorial explosion of the search space [9]. In our design, the search space is the set of all behavior trees that can be assembled from primitive action nodes. Since validating all the possible candidate behavior trees is neither practical nor efficient, we prune the search space by 1) using virtual registers to model the device interface while leaving the implementation of their access methods to the code generator (Section 4.2.1), 2) leveraging behavior trees to model driver operations instead of transitions between driver states (Section 4.2.2), and 3) extending behavior trees with new sets of predicates to impose additional constraints on the search space (Section 4.2.3).

4.2.1 Modeling Device Interfaces. Device drivers interact with devices through registers, shared memory regions, and hardware interrupts. Hardware vendors tend to make these interfaces as compact as possible, so it is common that a single device register serves multiple purposes. Additionally, these interfaces encompass hardware implementation details, such as endianness, width, and access mode, which the driver must handle. For example, the PL011 driver can query the device status by reading `UARTFR`'s value and examining its individual bits. Checking a particular bit involves four operations: a register read, a byte order conversion, a bitwise AND, and a right shift. Synthesizing the access method using these four operations would introduce unnecessary implementation details to the search space. Instead, we want the synthesizer to focus on the value of each bit. Thus, the synthesizer generates device driver functionality in terms of primitive actions on virtual registers, while the code generator implements the primitive actions with respect to the register definition file (Section 3.1.1).

4.2.2 Leveraging Behavior Tree. Recall that behavior trees generalize finite state machines (Section 2). Modeling the device driver as a finite state machine requires all states and transitions to be defined explicitly. Each state captures a combination of values for all virtual registers, so the synthesizer must explore all possible sequences of transitions from the initial state to the goal state. In the PL011 example, a state machine would consist of 512 states for the nine virtual registers that comprise `UARTFR`, half of which have the `BUSY` bit set to 1. To make a transition to a state where `BUSY` is 0, the synthesizer would need to consider up to 65536 possibilities. In contrast, behavior trees emphasize actions instead of states, so the synthesizer considers only the action nodes that affect the virtual registers involved in a particular operation. When we apply this to the PL011, the synthesizer must

establish the precondition that the device is not busy before transmitting a character; it is interested in the value of the BUSY virtual register only. The synthesizer then searches among the three primitive action nodes (i.e., *read*, *write*, and *wait*) that reference the BUSY virtual register and finds the *wait* action node that it can use to wait until BUSY becomes zero.

4.2.3 Extending Behavior Trees.

To further reduce the search space, we associate action nodes with three different sets of predicates: specialization constraints, preconditions, and postconditions. A *specialization constraint* imposes constraints on the type of value an action node can write to a virtual register, similar to C++20's concepts [25] or Rust's type traits [21]. For example, all physical memory addresses written to ADDR must be aligned to a 4096-byte boundary. We express this requirement in the type specification of the physical memory address. A specialization constraint can also impose constraints on the access mode of the virtual register that an action node references. For example, the *write* action node can be used on virtual registers that are writable, whereas *wait* can only be used on read-only virtual registers.

Recall that the synthesizer needs to recursively resolve a precondition using other primitive action nodes when constructing a behavior tree (Section 3.2). The synthesizer starts with a subset of developer-specified primitive action nodes whose postcondition matches the precondition being resolved. The synthesizer will discard a primitive action node if it cannot satisfy the node's specialization constraints and will not place the node in the behavior tree if it cannot resolve the node's preconditions. As such, the synthesizer excludes all the behavior trees containing this particular action node from the search space.

5 Current Status and Next Steps

Our prototype can handle control plane operations and has preliminary support for manipulating in-memory data structures and handling DMA transactions. Our register definition language also allows specification writers to describe relationships, such as grouping and ordering, between registers. While we demonstrated that it is feasible to synthesize device drivers using behavior trees, we now highlight some key challenges and discuss potential strategies to address them.

Obtaining Specifications. Ghost Writer's effectiveness depends on the quality of the specifications provided by various vendors. Writing these specifications could be as challenging as writing the driver code, but it is a one-time effort, because our toolchain can use the same device specification to synthesize device drivers for multiple operating systems. Besides, hardware vendors, such as ARM, **have begun** to release machine-readable and executable specifications for their CPUs and ISAs [26]. These specifications enable the verification of programs, compilers, and operating systems. Similarly, drivers can benefit from the device specifications released by vendors.

Modeling Standardized High-Level Protocols. Devices might expose a high-level, standardized command interface instead of registers. For example, NVMe SSDs support the DEALLOCATE command [13], which the driver uses to inform the device about logical blocks the operating system no longer needs. We can potentially expand the device class specification to model such high-level commands using behavior trees.

Modeling Kernel APIs. The compatibility layer in ported device drivers suggests a common set of kernel interfaces that drivers use. We believe that it is possible to standardize these interfaces, similarly to how POSIX standardized OS APIs, and model them as behavior trees, allowing the synthesizer to be agnostic to the kernel implementation. The operating system specification defines the mapping between the standardized interfaces and the kernel APIs, which can be used to translate the usage of kernel services without introducing a compatibility layer.

Modeling Data Processing. Device drivers can process data while handling I/O requests [19]. For example, an ethernet driver might need to calculate the checksum of a packet, while an audio driver might need to process the audio stream before sending it to the audio card. We can potentially model the data processing pipeline in the driver as a sequence of primitive transformers, each of which does some calculation on the data and passes the transformed data to the next one.

Dealing with Complex Devices. A complex device can be viewed as a collection of logical units with well-defined interfaces. For example, a conventional GPU has units that are responsible for rendering (e.g., managing framebuffers), acceleration (e.g., decoding HEVC streams), handling displays (e.g., performing link training for DisplayPort), etc. Each unit can access only a subset of the device interfaces, which limits the scope of the synthesis. The synthesizer treats each unit as an independent mini device, synthesizes a mini driver for each mini device, and assembles the complete driver from the mini drivers. We envision that this divider-and-conquer technique will be effective for handling complex devices.

Generating Optimized Code. Currently, the synthesizer focuses on finding the correct steps to accomplish the goal of a driver function, and the code generator translates the synthesized tree into an implementation. However, the synthesized tree can be further optimized to exploit hardware implementation details. For example, a sequence node with writes to multiple virtual registers representing different bits of the same physical register can be replaced by a single write action node. We envision adding an optimizer between the synthesizer and the code generator to perform such optimizations.

Supporting Memory Consistency Models. The synthesizer is unaware of the memory consistency model and

assumes that every write to a virtual register is visible to the device immediately. To ensure that the driver implementation is correct with respect to the memory consistency model, we envision adding a secondary synthesizer that is only concerned with finding the correct places to insert memory barriers.

Dealing with Concurrency. The current prototype assumes all driver functions are non-reentrant and will be executed in a single-threaded environment. There are two main concurrent scenarios in device drivers. First, the kernel might have multiple threads that invoke the same driver function, whereas the device can handle one I/O request at a time. As such, it is the driver's responsibility to coordinate requests submitted by each thread and serialize the access to the device. Second, the device might provide a mechanism (e.g., a circular queue) that handles I/O requests asynchronously. In this case, the driver must not overwrite the request that has not yet been processed. We plan to encapsulate common concurrent scenarios as behavior tree templates and letting the synthesizer fill in the missing pieces.

Handling Errors. Device drivers must be able to handle failures and notify the kernel accordingly. Currently, the synthesizer assumes that all nodes will return success when constructing behavior trees. However, it can use a fallback node to bind a node that could fail to an error-handling node. The error-handling node can return `success` if the driver function should resume execution after recovering from errors or `failure` if the execution should abort. We envision extending behavior trees for the synthesizer to generate meaningful and flexible error handlers.

6 Related Work

Device driver generation and synthesis has been an active research area for years. We discuss how Ghost Writer draws on prior work in these areas and how video games and robotics communities synthesize behavior trees, highlighting the potential applications of their methodology in our design.

6.1 Device Driver Generation and Synthesis

Early device driver synthesis work focused on embedded microcontrollers, which have a limited set of interfaces and run bare metal or specific real-time operating systems [24, 35]. Since these approaches do not draw a distinction between the device interface and the kernel interface, developers must specify complete driver behavior for each device. Chen et al. overcome this drawback with a model-driven approach to generating drivers for multiple operating systems [7]. While they can automatically generate about 70% of driver code, on average, their approach requires developers to specify correct register programming sequences. In contrast, Ghost Writer generates this sequence from device specifications.

Modern device driver synthesis targets more complex devices and general-purpose operating systems. Termite [29] and Termite 2 [30] model both devices and operating systems as finite-state machines and synthesize drivers using a game theoretic algorithm. Their algorithm recursively computes all states reachable from the goal state after a single transition and suffers from state space explosion for devices with many states. More importantly, device drivers are not limited to translating operating system requests into a series of device-specific commands; some perform complex computations that cannot be modeled as simple state machines [19]. Neither Termite nor Termite 2 supports manipulating in-memory data structures and allocating memory, while Ghost Writer overcomes these drawbacks using virtual registers and behavior trees.

6.2 Behavior Tree Synthesis in Games and Robotics

The video games and the robotics communities proposed different approaches for synthesizing behavior trees, because constructing them from scratch is burdensome and error-prone. These approaches leverage traditional program synthesis techniques with a focus on reactivity and safety in a highly dynamic environment [17]. For example, Trained Behavior Tree [31] exploits the idea of inductive synthesis [36] and can be used to generate a behavior tree for a non-player character (NPC) from the traces of the game designer controlling the NPC manually. French et al. [15] present a similar approach for robots to learn behavior trees from humans' demonstrations. Scheide et al. [33] leverage both inductive synthesis and stochastic search [34] to synthesize behavior trees from a formal grammar that defines the search space of well-structured behavior trees.

Ghost Writer is motivated by the use of behavior trees in the video games and the robotics industries, but we focus on search space reduction and hierarchical behavior tree construction. Synthesizing drivers directly from traces might not be feasible, but we imagine deriving device specifications from the traces and leveraging Ghost Writer to synthesize the driver. Additionally, we envision using stochastic search to prune the synthesized behavior trees, thereby reducing code size and improving runtime performance.

7 Conclusion

We presented a novel approach for synthesizing device drivers in a hierarchical, modular fashion using virtual registers and behavior trees. Virtual registers enable the decomposition of device interfaces while hiding the hardware implementation details. Behavior trees allow the synthesizer to reason about operations on individual virtual registers and assemble complex operations from simpler ones. To the best of our knowledge, Ghost Writer is the first project that brings behavior trees to the operating system community and uses them to model the behavior of an operating system component. We continue to explore the possibilities enabled by behavior trees in operating systems.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Anonymous. 2023. Audio cuts with 192 khz 8 channel 16 bit and 24p. https://gitlab.freedesktop.org/drm/intel/-/issues/8276

[3] Apple. 2023. DriverKit. https://developer.apple.com/documentation/driverkit

[4] Apple. 2023. IOKit Fundamentals. https://developer.apple.com/documentation/kernel/iokit_fundamentals/memory

[5] ARM. 2017. PrimeCell UART PL011 Technical Reference Manual. https://developer.arm.com/documentation/ddi0183/latest/

[6] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 255–268. https://www.usenix.org/conference/atc19/presentation/bai

[7] Hui Chen, Guillaume Godet-Bar, Frederic Rousseau, and Frederic Petrot. 2014. Device driver generation targeting multiple operating systems using a model-driven methodology. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*. 30–36. https://doi.org/10.1109/RSP.2014.6966689

[8] Michele Colledanchise and Petter Ögren. 2017. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. *IEEE Transactions on Robotics* 33, 2 (2017), 372–389. https://doi.org/10.1109/TRO.2016.2633567

[9] Wikipedia contributors. 2022. Combinatorial explosion. https://en.wikipedia.org/wiki/Combinatorial_explosion

[10] Wikipedia contributors. 2023. Behavior tree. https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

[11] Wikipedia contributors. 2023. UART. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[12] National Vulnerability Database. 2022. CVE-2022-32811. https://nvd.nist.gov/vuln/detail/CVE-2022-32811

[13] NVM Express. 2022. NVMe Command Set Specifications. https://nvmexpress.org/developers/nvme-command-set-specifications/

[14] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. ACM, New York, NY, USA, 38–51. https://doi.org/10.1145/268998.266642

[15] Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. 2019. Learning Behavior Trees From Demonstration. In *2019 International Conference on Robotics and Automation (ICRA)*. 7791–7797. https://doi.org/10.1109/ICRA.2019.8794104

[16] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 613–631. https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe

[17] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. 2022. A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems* 154 (2022), 104096. https://doi.org/10.1016/j.robot.2022.104096

[18] Damian Isla. 2005. GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI. (3 2005). https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

[19] Asim Kadav and Michael M. Swift. 2012. Understanding Modern Device Drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2150976.2150987

[20] Roland Kletzing. 2012. MAC 00:00:00:00:00:00 with natsemi DP83815 after driver load. https://bugzilla.kernel.org/show_bug.cgi?id=51791

[21] The Rust Programming Language. 2023. Generic Types, Traits, and Lifetimes. https://doc.rust-lang.org/book/ch10-00-generics.html

[22] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 269–284.

[23] Petter Ogren. 2012. *Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees*. https://doi.org/10.2514/6.2012-4458

[24] M. O'Nils, J. Oberg, and A. Jantsch. 1998. Grammar based modelling and synthesis of device drivers and bus interfaces. In *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, Vol. 1. 55–58 vol.1. https://doi.org/10.1109/EURMIC.1998.711776

[25] C++ References. 2023. Constraints and concepts. https://en.cppreference.com/w/cpp/language/constraints

[26] Alastair Reid. 2016. Trustworthy Specifications of ARM® V8-A and v8-M System Level Architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 161–168.

[27] Matthew J. Renzelmann and Michael M. Swift. 2009. Decaf: Moving Device Drivers to a Modern Language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 14.

[28] Cristian Romero. 2017. wlp1s0: Driver has suspect GRO implementation, TCP performance may be compromised. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1664072

[29] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 73–86. https://doi.org/10.1145/1629575.1629583

[30] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 661–676.

[31] Ismael Sagredo-Olivenza, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín, and Pedro Antonio González-Calero. 2019. Trained Behavior Trees: Programming by Demonstration to Support AI Game Designers. *IEEE Transactions on Games* 11, 1 (2019), 5–14. https://doi.org/10.1109/TG.2017.2771831

[32] Takashi Sakamoto. 2022. FireWire: Fix potential use-after-free. https://tinyurl.com/linux-firewire-uaf

[33] Emily Scheide, Graeme Best, and Geoffrey A. Hollinger. 2021. Behavior Tree Learning for Robotic Task Planning through Monte Carlo DAG Search over a Formal Grammar. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 4837–4843. https://doi.org/10.1109/ICRA48506.2021.9561027

[34] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[35] Shaojie Wang, S. Malik, and R.A. Bergamaschi. 2003. Modeling and integration of peripheral devices in embedded systems. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. 136–141. https://doi.org/10.1109/DATE.2003.1253599

[36] Patrick H Winston. 1970. Learning structural descriptions from examples. (1970).