User-Guided Device Driver Synthesis[*]

Leonid Ryzhyk[1,2] Adam Walker[2] John Keys[3] Alexander Legg[2] Arun Raghunath[3]
Michael Stumm[1] Mona Vij[3]

[1]University of Toronto
[2]NICTA[†] and UNSW, Sydney, Australia
[3]Intel Corporation

Abstract

Automatic device driver synthesis is a radical approach to creating drivers faster and with fewer defects by generating them automatically based on hardware device specifications. We present the design and implementation of a new driver synthesis toolkit, called Termite-2. Termite-2 is the first tool to combine the power of automation with the flexibility of conventional development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications. We demonstrate the practicality of Termite-2 by synthesizing drivers for a number of I/O devices representative of a typical embedded platform.

1 Introduction

Device driver synthesis has been proposed as a radical alternative to traditional driver development that offers the promise of creating drivers faster and with far fewer defects [24]. The idea is to automatically generate the driver code responsible for controlling device operations from a behavioral model of the device and a specification of the driver-OS interface.

The primary motivation for device driver synthesis is the fact that device drivers are hard and tedious to write, and they are notorious for being unreliable [8, 13]. Drivers generally take a long time to bring to production—given the speed at which new devices can be brought to market today, it is not uncommon for a device release to be delayed by driver rather than silicon issues [33].

Automatic driver synthesis was proposed in our earlier work on the Termite-1 project [24], where we formulated the key principles behind the approach and demonstrated its feasibility by synthesizing drivers for several real-world devices. The next logical step is to develop driver synthesis into a practical methodology, capable of replacing the conventional driver development process. To this end we have to address the key problems left open by Termite-1. The most important one is the quality of synthesized drivers. While functionally correct, Termite-1 drivers were bloated and poorly structured. This made it impossible for a programmer to maintain and improve the generated code and prevented synthesized drivers from being adopted by Linux and other major OSs. Furthermore, it was impossible to enforce non-functional properties such as CPU and power efficiency.

Another critical limitation of Termite-1 was the limited scalability of its synthesis algorithm, which made synthesis of drivers for real-world devices intractable. Termite-1 got around the problem by using carefully crafted simplified device specifications, which is acceptable in a proof-of-concept prototype, but not in a practical tool.

In the present project we set out to address these limitations. After several years of research we achieved significant improvement to all components of the synthesis technology: the specification language, the synthesis algorithm and the code generator.

Despite these improvements, we had come to the conclusion that the approach taken was initially *critically flawed*. The fundamental problem, in our view, was that the synthesis was viewed as a "push-button" technology that generated a specification-compliant implementation without any user involvement. As a result, the user had to rely on the synthesis tool to produce a good implementation. Unfortunately, even the most intelligent algorithm cannot fully capture the user-perceived notion of high-quality code. While in theory one might be able to enforce some of the desired properties by adding appropriate constraints to the input specification, in our experience

[*]This research is supported by a grant from Intel Corporation.
[†]NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

creating such specifications is extremely hard and seldom yields satisfactory results.

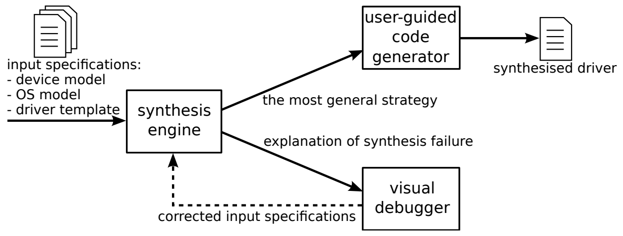
A radically different approach was needed—one that combines the power of automation with the flexibility of conventional development, and that involves the developer from the start, guiding the generation of the driver. In many ways, synthesis and conventional development are conflicting. Hence, a key challenge was to conceive of a way that allowed the two to be combined so that the developer could do their job more efficiently and with fewer errors without having the synthesis tool get in the way.

The primary contribution of this paper is a novel *user-guided* approach to driver synthesis implemented in our new tool called Termite-2 (further referred to as Termite). In Termite, the user has full control over the synthesis process, while the tool acts as an assistant that suggests, but does not enforce, implementation options and ensures correctness of the resulting code. At any point during synthesis the user can modify or extend previously synthesized code. The tool automatically analyses user-provided code and, on user's request, suggests possible ways to extend it to a complete implementation. If such an extension is not possible due to an error in the user code, the tool generates an explanation of the failure that helps the user to identify and correct the error.

In an extreme scenario, Termite can be used to synthesize the complete implementation fully automatically. At the other extreme, the user can build the complete implementation by hand, in which case Termite acts as a static verifier for the driver. In practice, we found the intermediate approach, where most of the code is auto-generated, but manual involvement is used when needed to improve the implementation, to be the most practical.

From the developer's perspective, user-guided synthesis appears as an enhancement of the conventional development process with very powerful autocomplete functionality, rather than a completely new development methodology. This vision is implemented in all aspects of the design of Termite. In particular, input specifications for driver synthesis are written as imperative programs that model the behavior of the device and the OS. The driver itself is modelled as a source code template where parts to be synthesized are omitted. This approach enables the use of familiar programming techniques in building input specifications. In contrast, previous synthesis tools, including Termite-1, require specifications to be written in formal languages based on state machines and temporal logic, which proved difficult and error-prone to use even for formal methods experts, not to mention software development practitioners.

Most previous research on automatic synthesis, including Termite-1, considered input specifications to be "correct by definition". In contrast, we recognise that input specifications produced by human developers are likely to contain defects, which can prevent the synthesis algorithm from finding a correct driver implementation. Therefore Termite incorporates powerful debugging tools that help the developer identify and fix specification defects through well-defined steps, similar to how conventional debuggers help troubleshoot implementation errors.



input specifications:
- device model
- OS model
- driver template
synthesis engine
user-guided code generator
synthesised driver
the most general strategy
explanation of synthesis failure
visual debugger
corrected input specifications

Figure 1: Termite synthesis workflow.

Another important contribution of this project is a new scalable synthesis algorithm, which mitigates the computational bottleneck in driver synthesis. Following the approach proposed in Termite-1, we treat the driver synthesis problem as a two-player game between the driver and its environment, comprised of the device and the OS. In this work, we develop this approach into the first precise mathematical formulation of the driver synthesis problem based on game theory. This enables us to apply theoretical results and algorithmic techniques from game theory to driver synthesis.

Our game-based synthesis algorithm relies on abstraction and symbolic reasoning to achieve orders of magnitude speed up compared to the current state-of-the-art synthesis techniques. The main idea of the algorithm is described in Section 4, but we refer the reader to a detailed description in an accompanying publication [30].

We evaluate Termite by synthesizing drivers for several I/O devices. Our experience demonstrates that our methodology meets our design goals, and indeed makes automatic driver synthesis practical.

Overview of Termite Figure 1 gives an overview of the driver synthesis process, described in detail in the rest of the paper. Termite takes three specifications as its inputs: a device model that simulates software-visible device behavior, an OS model that specifies the software interface between the driver and the OS, and a driver template that contains driver entry point declarations and, optionally, their partial implementation to be completed by Termite.

Given these specifications, driver synthesis proceeds in two steps. The first step is carried out fully automatically by the Termite game-based synthesis engine, which computes *the most general strategy* for the driver—a data

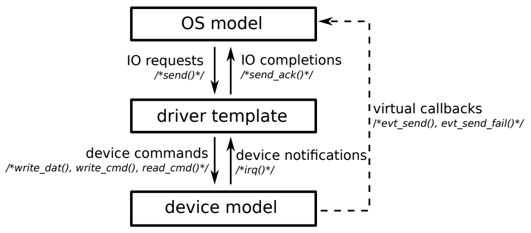


Figure 2: Input specifications for driver synthesis. Labels in italics show interfaces from the running example (Figure 3).

structure that compactly represents all possible correct driver implementations. This step encapsulates the computationally expensive part of synthesis. At the second step, the most general strategy is used by the Termite code generator to construct one specific driver implementation in C with the help of interactive input from the user.

The synthesis engine may establish that, due to a defect in one of the input specifications, there does not exist a specification-compliant driver implementation. In this case, it produces an explanation of the failure, which can be analysed with the help of the Termite debugger tool in order identify and correct the defect.

Limitations of Termite The device driver synthesis technology is still in its early days and, as such, has several important limitations. Most notably, Termite does not currently support synthesis or verification of code for managing direct memory access (DMA) queues. This code must be written manually and is treated by Termite as an external API invoked by the driver. As another example, in certain situations, explained in Section 3, Termite is unable to produce correct code without user assistance; however it is able to verify the correctness of user-provided code. We discuss limitations of Termite in more detail in Section 6.

2 Developing specifications

Input to Termite consists of the three specifications, which model the complete system consisting of the driver, the device, and the OS, shown in Figure 2. The OS and device models simulate the execution environment of the driver and specify constraints on correct driver behavior. The device model simulates software-visible device behavior. The OS model serves as a workload generator that issues I/O requests to the driver and accepts request completions in a way consistent with real OS behavior.

The virtual interface between the device and the OS, shown with the dashed arrow in Figure 2, is used by the device model to notify the OS model about important hardware events, such as completion of I/O transactions and error conditions. Methods of the virtual interface do not represent real runtime interactions between the device and the OS, but are used by the OS model to specify correctness constraints for the driver (see Section 2.3).

Finally, the driver template contains a partial driver implementation to be completed by Termite. A minimal template consists of a list of driver entrypoints without implementation. At the other extreme, it can provide a complete implementation, in which case Termite acts as a static verifier for the driver.

All specifications are written using the Termite Specification Language (TSL). In line with our goal of making synthesis as close to the conventional driver development workflow as possible, TSL is designed as a dialect of C with additional constructs for use in synthesis. We introduce relevant features of TSL throughout this section.

We minimize the amount of work needed to develop specifications for every synthesized driver by maximizing the reuse of specifications. In particular, Termite allows the use of existing device specifications developed by hardware designers in driver synthesis. Furthermore, the OS specification for the driver can be derived from a generic specification for a class of similar devices (e.g., network or storage). Thus we expect that additional per-driver effort will consist of: (1) inserting device-class callbacks in appropriate locations of the device model and (2) extending the OS specification to support device-specific features missing in the generic OS specification.

2.1 Device model

The device model simulates the device operation at a level of detail sufficient to synthesize a correct driver for it. To this end, it must accurately model external device behavior visible to software. At the same time, it is not required to precisely capture internal device operation and timing, as these aspects are opaque to the driver.

Such device models are routinely developed by hardware designers for the purposes of design exploration, simulation, and testing. They are widely used by hardware manufacturers in-house [14] and are available commercially from major silicon IP vendors [28]. These models are known as *transaction-level models* (TLMs) (in contrast to the detailed register-transfer-level models used in gate-level synthesis) [4]. A TLM focuses on software-visible events, or *transactions*, such as a write to a device register or a network packet transmission.

Existing TLMs created by hardware designers can be used with minor modifications (explained in Section 2.3) for driver synthesis. Model reuse dramatically reduces the effort involved in synthesizing a driver and is therefore crucial to practical success of driver synthesis. By reusing an existing model, we also reuse the effort invested by hardware designers into testing and debugging the model throughout the hardware design cycle, thus making driver

```
1template dev /* Device model */
2  uint8 reg_dat, reg_cmd, reg_status = 0;
3  /* device commands */
4  controllable void write_dat(uint8 v)
5  { reg_dat = v; };
6  controllable void write_cmd(uint8 v)
7  { reg_cmd = v; };
8  controllable uint8 read_cmd()
9  { return reg_cmd; };
10 controllable uint8 read_status()
11 { return reg_status; };
12 /* internal behavior */
13 process ptx {
14   forever {
15     wait (reg_cmd == 1);
16     choice {
17       { os.evt_send(reg_dat);
18         reg_status=0; };
19       { os.evt_send_fail(reg_dat);
20         reg_status=1; };
21     };
22     reg_cmd = 0;
23     /*drv.irq(); (see Section 4*/
24   };
25 };
26endtemplate
27
28template os /* OS model */
29 uint8 dat;
30 bool inprogress, acked, success;
31 /* driver workload generator */
32 process psend {
33   forever {
34     dat = *; /*randomise dat*/
35     inprogress = true;
36     acked = false;
37     drv.send(dat);
38     wait(acked);
39   };
40 };
41 /* I/O completions */
42 controllable void send_ack(bool status) {
43   assert (!inprogress && !acked &&
44           status == success);
45   acked = true;
46 };
47 /* virtual callbacks */
48 void evt_send(uint8 v) {
49   assert (inprogress && v==dat);
50   inprogress = false;
51   success = true;
52 };
53 void evt_send_fail(uint8 v) {
54   assert (inprogress && v==dat);
55   inprogress = false;
56   success = false;
57 };
58 goal idle_goal = acked;
59endtemplate
60
61template drv /* Driver template */
62 void send(uint8 v){...;};
63 /*void irq(){...;}; (see Section 4)*/
64endtemplate
```

Figure 3: Trivial serial controller driver specifications.

synthesis less susceptible to specification bugs. Finally, since TLMs are created early in the hardware design cycle, TLM-based driver synthesis can be carried out early as well, thus removing driver development from the critical path to product delivery.

TLMs are written in high-level hardware description languages like SystemC and DML. In order to use these models in driver synthesis, we need to convert them to TSL. This translation can be performed automatically, and we are currently working on a DML-to-TSL compiler. Since this work is not yet complete, device models used in the experimental section of this paper are either manually translated from existing TLMs or written from scratch using TLM modeling style guidelines [31].

The top part of Figure 3 shows a fragment of a model of a trivial serial controller device used as a running example. The fragment specifies the send logic of the controller, which allows software to send data characters over the serial line. The model is implemented as a TSL *template*. The template encapsulates data and code that manipulates the data, similar to a class in OOP.

The software interface of the device consists of data, command, and status registers declared in line 2. The registers can be accessed from software via the `write_dat`, `write_cmd`, `read_cmd`, and `read_status` methods (lines 4–11). The `controllable` qualifier denotes a method that is available to the driver and can be invoked from synthesized code.

The transmitter logic is modelled in lines 13–25. It is implemented as a TSL *process*. A TSL specification can contain multiple processes. The choice of the process to run is made non-deterministically by the scheduler. The process executes atomically until reaching a `wait` statement or a controllable placeholder (see below).

In line 15, the transmitter waits for a command, issued by the driver by writing value 1 to the command register. Upon receiving the command, it sends the value in the data register over the serial line. The transmission may fail, e.g., due to a serial link problem. The device signals transmission status to software by setting the status register to 0 or 1. Finally, it clears the command register, thus notifying the driver the request has completed.

Internally, the transmitter circuit consists of a shift register and a baud rate generator used to output data on the serial line. These details are not visible to software and are abstracted away in the model. We use the non-deterministic `choice` construct to choose between successful transmission and failure, without modelling the details of serial link operation. Successful and failed transmissions are modelled using `evt_send` and `evt_send_fail` events, explained in Section 2.2.

2.2 OS model

The OS model specifies the API mandated by the OS for all drivers of the given type. For example, any Ethernet driver must implement the interface for sending and receiving Ethernet packets. A separate specification is needed for each supported OS, as different OSs define different interfaces for device drivers.

Additionally, each particular device can support non-standard features, e.g., device-specific configuration options or transfer modes. These features must be added as extensions to the generic OS specification in order to synthesize support for them in the driver. TSL supports such extensions in a systematic way via the template inheritance mechanism. We do not describe this in detail due to limited space.

The OS model is written in the form of a test harness that simulates all possible sequences of driver invocations issued by the OS. The `os` template in Figure 3 shows the OS model for our running example. The main part of the model is the `psend` process. At every iteration of the loop, it non-deterministically chooses an 8-bit value (line 34) and calls the `send` method of the driver, passing this value as an argument. It then waits for the driver to acknowledge the transmission of the byte (line 38) before issuing another request. The driver acknowledges the transmission via the `send_ack` callback (line 42). The callback sets the `acked` flag, which unblocks the `psend` process.

We keep the specification concise by modeling the state of the driver-OS interface, as opposed to the internal OS state and behavior. For example, the `acked` variable (line 30) serves to model the flow of data between the OS and the driver and is not necessarily present in the OS implementation.

2.3 Connecting device and OS models

In addition to simulating I/O requests to the driver, the OS model also specifies the semantics of each request in terms of device-internal events that must occur in order to complete the requested I/O operation. In our running example, after the OS invokes the `send` method of the driver and before the driver acknowledges completion of the request, the device must attempt to send the requested data over the serial line. This requirement establishes a connection between the device and OS models and must be specified explicitly in order to enable Termite to generate a driver implementation that correctly handles the OS request. Note that we only need to specify *which* hardware events must occur, but not *how* the driver generates them.

In order to develop such specifications, we need a way to refer to relevant state and behavior of the device from the OS model. At the same time, in order to maximize specification reuse, we would like to keep the OS specification device-independent. To reconcile these conflicting requirements, we introduce a *virtual interface* between the device and OS model. This interface consists of callbacks used by the device model to notify the OS model about important hardware events. The virtual interface does not represent real runtime interactions between the device and the OS, but serves as part of the correctness specification.

We define a virtual interface for each class of devices. Such *device-class* interfaces are both device and OS-independent. The device-class interface can be extended with additional device-specific callbacks as required to specify a driver for a particular device.

In our example, we define a device-class interface consisting of two virtual callbacks: `evt_send` and `ev_send_failed`, invoked respectively when the device successfully transmits and fails to transmit a byte. These callbacks are invoked in lines 17 and 19 of the device model. The `evt_send` handler is shown in line 48 of the OS model. The assertion in line 49 specifies that the send event is only allowed to occur if there is an outstanding send request in progress and the value being sent is the same as the one requested by the OS. We reset the `inprogress` flag to false in line 50, thus marking the current request as completed; line 51 sets the `success` flag to true, thus indicating that the transfer completed without an error. The `evt_send_fail` handler is identical, except that it sets the `success` flag to false. The flags are checked by the `send_ack` method, which asserts that the driver is only allowed to acknowledge a completed request (`!inprogress`) that has not been acknowledged yet (`!acked`) and that the completion status reported by the driver must match the one recorded in the `success` flag.

In this example we use C-style assertions to rule out invalid system behaviors. Assertions alone do not fully capture requirements for a correct driver behavior. For example, a driver that remains idle does not violate any assertions. Hence, we need to specify requirements for the driver to make forward progress. We introduce such requirements into the model in the form of *goal conditions*, that must hold *infinitely often* in any run of the system. For example, a goal may require that the driver is infinitely often in an idle state with no outstanding requests from the OS. The OS can force the driver out of the goal by issuing a new I/O request. To satisfy the goal condition, the driver must return to the goal state by completing the request. Line 58 in Figure 3 defines such a goal condition that holds whenever the `acked` flag is set, i.e., the driver has no unacknowledged send requests.

2.4 Driver template

The bottom part of Figure 3 shows the driver template for the running example consisting of a single `send` entry point invoked by the OS. The ellipsis in line 62 represent a location for inserting synthesized code and are part of TSL syntax. We refer to such locations as *controllable placeholders*.

3 User-guided code generation

The set of input TSL specifications is fed into the Termite synthesis engine, which then automatically computes the most general strategy for the driver. Given a state of the system, the most general strategy determines the set of all valid driver actions in this state. The most general strategy is used by the Termite code generator to produce a driver implementation in C in a user-guide fashion.

The Termite code generator GUI is similar to a traditional integrated development environment with two additional built-in tools: the *generator* and the *verifier*. The generator works as advanced auto-complete that helps the user to fill the controllable placeholders inside the driver template with code. At any point, the user can invoke the generator to synthesize a single statement or a complete block of code inside a controllable placeholder via a mouse click on the target code location. The user can arbitrarily modify and amend the generated code. However, the generator never modifies user code. Instead it tries to extend it to a complete implementation, which is always possible provided that the existing code is consistent with the most general strategy. The generator currently only allows synthesizing statements after the last control location within a branch. However this restriction is not a conceptual one and will be lifted by ongoing development.

The verifier automatically and on the fly checks that the driver implementation, comprised of a mix of generated and manually written code, is consistent with the most general strategy, thus maintaining strong correctness guarantees that one would expect in automatically synthesized code. The verifier symbolically simulates execution of the system, following the partial driver implementation created so far, and signals the user whenever it encounters a transition that violates the most general strategy.

In the first approximation, the generator algorithm is quite simple: given a source code location, it determines the set of possible system states in this location, picks an action for each state from the most general strategy and translates this action into a code statement. In practice the algorithm uses a number of heuristics to produce compact and human-readable code. In particular, whenever there exists a common action in all possible states in the given location, the algorithm produces straight-line code without branching. For example, when running the generator on the specification in Figure 3, it automatically generates the following code for the `send` function (line 62):

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);
    wait(dev.reg_cmd==0);
    if (os.success) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

This implementation correctly starts the data transfer by writing the value to be sent to the data register and setting the command register to 1. It then waits for the transfer to complete, which is signalled by the device by resetting the command register to 0. Finally, it acknowledges the completion of the transfer to the OS.

Note that the generated code refers to the `dev.reg_cmd` and `os.success` variables. These variables model internal device and OS state respectively and cannot be directly accessed by the driver. This example illustrates an important limitation of Termite—it assumes a white-box model of the system, where every state variable is visible to the driver. Ideally, we would like to synthesize an implementation that automatically infers the values of important unobservable variables. In this case, the value of the command register can be obtained by the driver by executing the `read_cmd` action. Furthermore, the value of the `os.success` variable is correlated with the completion status of the last transfer, which can be obtained by reading the device status register.

While Termite currently cannot produce such an implementation automatically, it implements a pragmatic tradeoff that helps the user build and validate a correct implementation with modest manual effort. The code generator warns the user that the auto-generated code accesses private variables of the device and OS templates. This prompts the user to provide a functionally equivalent valid implementation, replacing the `wait` statement with a polling loop and using the `read_status` method to check transfer status:

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);
    while(dev.read_cmd()==1);
    if (dev.read_status()) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

The verifier automatically checks the resulting implementation and confirms that it satisfies the input specification.

Note that in this example we have synthesized code that correctly handles device errors. This was possible, as our input device specification correctly captures device failure modes (namely, transmission failure) and our OS specification describes how the driver must report errors to the OS (via the `status` argument of the completion callback).

In principle, it is also possible to synthesize a driver implementation that handles device and OS failures *not* captured in the specifications: since the synthesis tool knows all possible valid environment behaviors, it can easily detect invalid behaviors and handle them gracefully. Automatic synthesis of such *hardened* device drivers is a promising direction of future research.

The final step of the code generation process translates the synthesized driver implementation to C. This is a trivial line-by-line translation. We expect this translation to become unnecessary in the future as our ongoing work on the TSL syntax aims to make the synthesized subset of TSL a strict subset of C.

Maintaining synthesized code Device driver development is not a one-off task: following the initial implementation, drivers are routinely modified to implement additional functionality, adapt to the changing OS interface or support new device features.

The user-guided code generation method naturally supports such incremental maintenance. A typical maintenance task proceeds in three steps. First, the developer amends device and OS models to reflect the new or changed functionality. Second, they add new methods to the previously synthesized driver, if necessary, and replace existing driver code that is expected to change with a controllable placeholder. Finally, the user runs Termite to synthesize code for all controllable placeholders. Termite treats all existing driver code as part of the uncontrollable environment. Hence, if some of the old code is incorrect in the context of the new specifications, this will lead to a synthesis failure, and counterexample-based debugging is used to identify the faulty code, as described in Section 5.

As an example, we synthesize a new version of the driver for our running example assuming a more advanced version of the serial controller device that uses interrupts to notify the driver on completion of a data transfer. The new device model is obtained by uncommenting line 23 of the device model in Figure 3, which invokes the interrupt handler method of the driver after each transfer. The driver template is extended with the `irq` method (line 63). We use the previously synthesized implementation of the `send` method, but manually remove the last two lines, which implement polling, as we want the new implementation to use interrupts instead:

```
void send(uint8 v){
    dev.write_dat(v);
    dev.write_cmd(1);}
```

Finally, we run Termite on the resulting specifications and use the generator to automatically produce the following implementation of the new `irq` method:

```
void irq(){
    if (os.success) {
        os.send_ack(true);
    } else {
        os.send_ack(false);
    };}
```

As before, we manually replace the if-condition in the first line with

```
if (dev.read_status())
```

This example illustrates how Termite supports incremental changes to the driver by reusing previously synthesized code, while maintaining strong correctness guarantees.

Instrumenting synthesized code Termite does not automatically instrument synthesized code for debugging, logging, accounting, etc. However, the user can add such instrumentation manually. Termite interprets such code as no-ops and, as with any manual code, never makes any modifications to it.

4 Synthesis

In this section we give a high-level overview of the Termite synthesis algorithm. We refer the reader to the accompanying publication [30] for a detailed description.

4.1 Driver synthesis as a game

We formalize the driver synthesis problem as a *two-player game* [29] between the driver and its environment. The game is played over a finite automaton that represents all possible states and behaviors of the system. Transitions of the automaton are classified into *controllable* transitions triggered by the driver and *uncontrollable* transitions triggered by the device or OS. A winning strategy for the driver in the game corresponds to a correct driver implementation. If, on the other hand, a winning strategy does not exist, this means that there exists no specification-conforming driver implementation.

Two-player games naturally capture the essence of the driver synthesis problem: the driver must enforce a certain subset of system behaviors while having only partial control over the system.

Figure 4 illustrates the concept using a trivial game automaton that models the core of our running example. Controllable and uncontrollable transitions of the automaton are shown with solid and dashed arrows respectively.

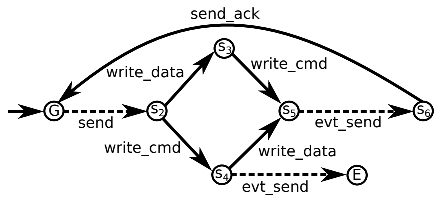


Figure 4: A simple two-player game.

The goal of the driver in the game is to infinitely often visit the initial state, labelled `G`, which represents the situation when the driver does not have any outstanding requests. After getting a `send` request from the OS, the driver must write data and command registers to start the data transfer. Writing the command register first may trigger a hardware send event before the driver has a chance to write the data register. As a result, wrong data value gets sent, taking the game into an error state `E`. Hence, state s_4 is losing for the driver. To avoid this state, the correct strategy for the driver is to play `write_data` in state s_2, followed by `write_cmd`. In s_5 the driver must remain idle until the environment executes the `evt_send` transition.

Games and strategies Formally, a two-player game $G = \langle S, I, L_c, L_u, \delta_c, \delta_u, \Phi\rangle$ consists of a set of states S, a subset of initial states $I \subseteq S$, sets of controllable and uncontrollable actions L_c and L_u, controllable transition relation $\delta_c \subseteq S \times L_c \times S$, uncontrollable transition relation $\delta_u \subseteq S \times L_u \times S$, and a game objective $\Phi \subseteq S^\omega$ (where S^ω represents the set of infinite sequences of states in S).

The game proceeds in rounds, starting from an initial state. In each round, in state s, both players select actions l_c and l_u available to them in s, and the game transitions non-deterministically to one of the states in $\delta_c(s, l_c) \cup \delta_u(s, l_u)$. Intuitively, the system scheduler chooses the player to make a move at each round. The scheduler can be thought of as part of the uncontrollable environment. Note that this is different from turn-based games like chess, where players strictly alternate in making moves. In the example in Figure 4, the driver can avoid the error state by choosing the `write_dat` action in state s_4; however the environment can override this choice by playing `evt_send`.

The infinite sequence of states $(s_0, s_1, ...) \in S^\omega$ visited by the game is called a *run*. A *strategy* for the driver player is a function $\pi : S \rightarrow 2^{L_c}$ that maps each state of the game into a set of actions to play in this state. The strategy determines a set $Outcomes(I, \pi) \subseteq S^\omega$ of all possible runs generated by the driver choosing one of the actions in $\pi(s)$ in every state s in the run.

Given a state s and a strategy π such that $Outcomes(\{s\}, \pi) \subseteq \Phi$, we say that s is a *winning state* for the driver, π is a winning strategy in s, and actions in $\pi(s)$ are *winning moves* in s. The game G is winning for the driver if all states in I are winning. The *most general winning strategy* maps every winning state s to a set of all winning moves in s, and all other states to an empty set.

In Termite we use game objectives of a particular form, called *generalised reactivity-1* (GR-1) objectives [22]. Such an objective consists of a finite set $\{B_1, ...B_n\}$, $B_i \subseteq S$ of *goal sets* and a finite set $\{F_1, ...F_k\}$, $F_i \subseteq S$ of *fair sets*. A winning strategy for the driver must make sure that the game infinitely often visits each of the goal sets, provided that the environment guarantees that the game does not get stuck in a fair set forever.

Intuitively, a goal set represents a constraint on the driver behavior, requiring the driver to force the game into the goal infinitely often, while a fair set represents a constraint on the environment, preventing it from staying in certain states forever. The game in Figure 4 has a single goal set $B_1 = \{g\}$ and a single fair set $F_1 = \{s_4, s_5\}$, i.e., the driver must acknowledge each send request from the OS, provided that the environment eventually performs the `evt_send` action after it has been enabled.

4.2 TSL compiler

In order to compute the most general driver strategy as a solution of a two-player game, we must first convert input TSL specifications into a game automaton. This conversion is performed by the TSL compiler.

Real driver specifications have large state spaces, which cannot be feasibly represented by explicitly enumerating states, as in Figure 2. Therefore, in Termite we represent games symbolically. The state space of the game is defined in terms of a finite set of state variables X, with each state $s \in S$ representing a valuation of variables in X. The TSL compiler introduces a state variable for each TSL variable declared in one of the input templates. In addition, auxiliary state variables are introduced to model the current control location of each TSL process.

We model controllable and uncontrollable actions as valuations of action variables Y_c and Y_u. Transition relations δ_c and δ_u are represented symbolically as formulas over state variables X, action variables Y_c and Y_u, and next-state variables X'.

The TSL compiler splits the input specification into controllable and uncontrollable parts and translates them into controllable and uncontrollable transition relations respectively. The controllable part is comprised of controllable methods that can be invoked by the driver. The controllable transition relation δ_c is computed by rewriting controllable methods in the *variable update form*. Consider, for example, variable `reg_dat` declared in line 2 in Figure 3. This variable is only modified by the `write_dat` method in line 4. The corresponding fragment of the controllable transition relation in the variable update form is $\texttt{reg\_dat}' := (\texttt{tag} = \texttt{write\_dat})\ ?\ \texttt{v} : \texttt{reg\_dat}$, where $\texttt{reg\_dat}'$ is the next-state variable representing the value of `reg_dat` after the transition, and `tag` and `v` are controllable action variables, where `tag` models the method being invoked, and `v` is the argument of the method.

The uncontrollable part of the specification is comprised of TSL processes, which model device and OS behavior. We syntactically decompose each process into atomic transitions. Recall that a process executes atomically until reaching a `wait` statement or a controllable placeholder. Consider the `ptx` process in line 13 in Figure 3. The process is initially paused in the wait statement. It is scheduled to run when the wait condition holds. It executes the statements in lines 16–22 atomically and stops again in line 15. As part of this atomic transition, the process sets the `reg_cmd` variable to 0 (line 22). This is the only uncontrollable transition that modifies this variable, hence the uncontrollable update function for this variable is defined as follows: $\texttt{reg\_cmd}' := (\texttt{reg\_cmd} = \texttt{1} \land \texttt{pid} = \texttt{ptx})\ ?\ \texttt{0} : \texttt{reg\_cmd}$, where `pid` is an uncontrollable action variable that models the scheduler's choice of a process to run, and the $\texttt{reg\_cmd} = \texttt{1}$ conjunct corresponds to the wait condition in line 15.

Finally, we need to generate the game objective Φ. In a symbolic representation of the game, goal and fair sets are specified as conditions over state variables that hold for each state in the set. The TSL compiler outputs a goal set B_i for each goal declared in the input specification and a fair set F_i for each `wait` statement. The latter guarantees that every runnable process gets scheduled eventually.

In addition to goal conditions, a TSL specification also contains assertions, which must never be violated. We model assertions using an auxiliary boolean state variable ε, which is set to true whenever an assertion is violated and remains true forever after. We add an extra constraint $\varepsilon = false$ to each accepting set B_i. An assertion violation permanently takes the game out of B_i, and therefore can not occur in any winning run of the game.

4.3 Solving the game

The Termite game solver takes a game automaton produced by the TSL compiler, determines whether all initial states of the system are winning and, if so, computes the most general winning strategy for the game. A successful approach to solving two-player games with GR-1 objectives was proposed by Piterman et al. [22]. We give an overview of their algorithm and briefly explain how we extend it to address the scalability bottleneck.

Algorithm 1 Computing the set of winning states

```
function REACH($B$)
    $Y \leftarrow \varnothing$
    loop
        $Y' \leftarrow CPre(Y \cup B)$
        if $Y' = Y$ return $Y$
        $Y \leftarrow Y'$

function WINNINGSET($\{B_1, ..., B_n\}$)
    $Z \leftarrow S$
    loop
        $Z' \leftarrow \bigcap_{i=1..n} \text{REACH}(Z \cap B_i)$
        if $Z' = Z$ return $Z$
        $Z \leftarrow Z'$
```

The algorithm is based on exhaustive exploration of the state space of the game. Given a goal set B, we first determine the set of states from which the driver can force the game into B in one step, called the *controllable predecessor* of B. The controllable predecessor consists of all states s that satisfy both of the following conditions:

1. All uncontrollable transitions available in s lead to some state in B. Hence, if the scheduler chooses to execute an uncontrollable transition, it is guaranteed to take the game to B.
2. There exists at least one winning controllable transition from s to B **or** s belongs to a fair region. In the former case, the driver must perform the winning transition; in the latter case it must remain idle waiting for an uncontrollable transition, which is guaranteed to occur due to fairness.

Having computed the controllable predecessor of B, we apply the controllable predecessor operator again to the resulting set, thus obtaining the set of states from which the driver can force the game into the goal within two steps. We repeat until no new states can be discovered, at which point we have found all states from which the driver can force the game into the goal in a finite number of rounds. This computation is performed by the REACH function shown in Algorithm 1.

Recall that a GR-1 game can have multiple goal regions, and in order to win the game the driver must visit each goal region B_i infinitely often. Using the REACH function, we compute the set $Z = \bigcap_i \text{REACH}(B_i)$, from which any of the goals can be reached at least once. Next, we compute $Z' = \bigcap_i \text{REACH}(Z \cap B_i)$. It is easy to see that Z' contains all states from which any of the goals can be reached twice. Furthermore, by construction, $Z' \subseteq Z$. By continuing the last computation until a fixed point is reached, we obtain all winning states of the game, as shown in function WINNINGSET (Algorithm 1).

The algorithm presented above is polynomial in the size of the game automaton. We have developed a highly optimized implementation of the algorithm, which uses symbolic data structures [3] to compactly represent large sets of states and transitions. Nevertheless, when applying it to games arising in driver synthesis, we hit a computational bottleneck due to a state explosion.

We overcome this bottleneck by using abstraction to reduce the dimensionality of the problem. The particular form of abstraction used by Termite is *predicate abstraction* [12], where concrete state variables of the game are replaced with boolean predicates over the original variables. Abstraction is adaptively refined by introducing new predicates that capture important relations among concrete variables. The predicate-based abstraction-refinement algorithm for games is one of the key technical contributions of Termite. It is described in detail in an accompanying paper [30].

4.4 Verification as a special case of synthesis

Consider the situation where not only the OS and the device, but also the driver behavior is fully specified, so that the synthesizer does not have any freedom to pick driver actions. If the resulting game is winning for the driver, i.e., every possible run of the game satisfies the objective, then the provided driver implementation is correct. Thus, verification can be seen as a special case of the synthesis problem where all transitions in the system are uncontrollable. Hence, our game solving algorithm doubles as a driver verification algorithm. Termite also supports hybrid scenarios: given a partially implemented driver with placeholders for synthesized code, it determines whether the given partial implementation can be extended to a complete one and, if so, fills out the placeholders in the user-guided fashion.

5 Debugging with counterexamples

An important practical issue in game-based synthesis is the complexity of diagnosing synthesis failures due to defects in the input specifications. In the event that Termite fails to solve the game, the user needs to trace the failure back to the specification defect. However, the failure does not carry any information about the defect, which makes the problem harder to resolve.

In Termite we propose a new approach to troubleshooting synthesis failures based on the use of *counterexample strategies*. A counterexample strategy is a strategy on behalf of the environment that prevents the driver from winning the game. It is obtained by solving the *dual game*, where, in order to win, the environment must permanently force the game out of one of the goal regions. A winning strategy in the dual game is guaranteed to exist whenever solving of the primary game fails.

In order to detect and fix the defect in an input specification, the driver developer relies on their understanding of the OS and device logic. The role of the counterexample strategy is to guide the developer towards the defect. To automate this process, we developed a powerful visual debugging tool that allows the user to interactively simulate intended driver behavior and observe environment responses to it. The user plays the game on behalf of the driver, while the tool responds on behalf of the environment, according to the counterexample strategy.

In a typical debugging session, the debugger, following the counterexample strategy, generates a sequence of requests that are guaranteed to win against the driver. The user plays against these requests by specifying device commands that, they believe, represent a correct way to handle the request. Since this sequence of requests *cannot* be handled correctly given the current input specification, at some point in the game the user runs into an unexpected behavior of one of the players, e.g., one of user-provided commands does not change the state of the device as expected or the environment performs an uncontrollable transition that violates an assertion. Based on this information, the user can revise the faulty specification.

At every step of the interactive debugging session, the debugger either chooses a spoiling uncontrollable action based on the counterexample strategy or, if the system is inside a controllable placeholder, allows the user to choose a controllable action to execute on behalf of the driver. In the former case the spoiling uncontrollable action corresponds to a transition in one of the TSL processes. The user can explore this transition by stepping through it, exactly as they would in a conventional debugger. In the latter case, the user provides the action that they would like to perform by typing and executing corresponding code statements.

The tool supports a number of features aimed to make the debugging process as simple as possible for the user. We mention two of them here. First, the debugger interactively prompts actions available to the driver at each step. Second, the debugger keeps the entire history of the game and allows the user to go back to one of previously explored states and try a different behavior from there.

6 Limitations of Termite

In Section 3, we described one limitation of Termite, namely the lack of support for grey-box synthesis. In this section we discuss other limitations, which, we hope, will help define the agenda for continuing research in driver synthesis.

Most importantly, Termite does not currently support automatic synthesis of direct memory access (DMA) management code. Many modern devices transfer data directly to and from main memory, where it is buffered in data structures such as circular buffers and linked lists. These data structures can have very large or infinite state spaces and cannot be easily modeled within the finite state machine-based framework of Termite. Efficient synthesis for DMA requires enhancing the synthesis algorithm to use more compact representation of DMA data structures, which is the focus of our ongoing research. At this time, code for manipulating DMA data structures must be written manually. This code is not interpreted or verified by Termite. For example, we use this approach to synthesize a DMA-capable IDE disk driver (Section 7).

Device drivers in modern OSs contain a significant amount of boilerplate code that is not directly related to the task of controlling the device. This includes binding the driver to I/O resources (memory mapped regions, interrupts, timers), registering the driver with various OS subsystems, allocating DMA memory regions, creating sysfs entries, etc. While much of this functionality could be synthesized within the game-based framework, we do not believe that this is the correct approach. Previous research has demonstrated that this boilerplate code can be generated in a principled way from declarative specifications of the driver's requirements and capabilities [26]. This technique has lower computational complexity than game solving and better captures the essence of the task. A practical driver synthesis tool can combine game-based synthesis of the core driver logic responsible for controlling the device with declarative synthesis of boilerplate code. As a result, the current version of Termite assumes this boilerplate code is written manually as a wrapper around the synthesized driver.

Drivers execute in a concurrent OS environment and must handle invocations from multiple threads, as well as asynchronous hardware interrupts. We separate synthesis for concurrency into a separate step. Drivers synthesized by Termite are correct assuming a sequential environment, where driver entry points are invoked atomically. The resulting sequential driver is then processed by a separate tool that performs a sequence of transformations of the driver source code, which preserve the driver's sequential behavior, while making the driver thread-safe. Such transformations include adding locks around critical code sections, inserting memory barriers, and reordering instructions to avoid race conditions. Concurrency synthesis is still work in progress and is beyond the scope of this paper. Our preliminary results are published in [5, 6].

Termite does not explicitly support specification and synthesis of timed behaviors. Instead, it uses a pragmatic approach that allows it to synthesize time-sensitive behavior without having to explicitly reason about time. To this end, Termite conservatively approximates timed operations by fairness constraints: it ignores the exact duration of each device operation, but keeps the knowledge that the operation will complete *eventually*, and synthesizes a driver that waits for the completion. Termite is also able to handle time-out conditions, modeled as external events. However, at this time it is not capable of generating device drivers for hard real-time systems, where the driver must guarantee completion of I/O operations by a certain deadline.

7 Implementation and evaluation

The version of Termite presented here consists of 30,000 lines of Haskell code. The estimated overall project effort is 10 person years. Termite is available in source and binary form from the project webpage[1].

We evaluate Termite by synthesizing drivers for eight I/O devices. Specifically, we synthesized drivers for a UVC-compliant USB webcam, the 16550 UART serial controller, the DS12887 real-time clock, and the IDE disk controller for Linux, as well as seL4 [16] drivers for I2C, SPI, and UART controllers on the Samsung exynos 5 chipset[2] and SPI controller on the STM32F10 chipset. With the exception of the IDE disk, these devices are representative of peripherals found in a typical embedded platform, such as a smartphone. Our synthesized drivers implement data transfer, configuration and error handling. The main barrier to synthesizing drivers for more advanced devices, e.g., high-performance network controllers, is the current lack of support for synthesis of DMA code in the current version of Termite.

Modelling complexity Models of UART and DS12887 devices were developed based on existing publicly available device models [32, 20]. Models of other devices were derived from their vendor-provided documentation, following standard TLM modeling guidelines [31]. OS models for the relevant device classes were created based on Linux kernel documentation and source code.

Table 1 summarises the size, in lines of code, of device and OS models in our case studies. Developing a complete set of specifications for each driver took approximately one week, of which only one to three days were spent building the models and the rest of the time was spent studying device and OS documentation. This efficiency can be attributed to the choice of the right level of

[1]`http://termite2.org`

[2]At the time of writing, the exynos drivers have not yet been tested due to hardware availability issues; however we confirmed via manual inspection that they implement the same device control sequences as existing manually developed drivers.

	input spec		driver	
	OS	device	synthesized	native
webcam	102	385	113	307
16450 UART	122	167	74	261
exynos UART	128	252	37	166
STM SPI	73	244	24	64
exynos SPI	88	239	40	183
exynos I2C	146	180	79	211
RT clock	118	252	84	183
IDE	188	480	94[a]	474

[a]Excluding 36 lines of manually written code that manipulates the DMA descriptor table.

Table 1: Size (in lines of code) of input specifications and of synthesized and equivalent manually written drivers.

	vars(bits)	refine-ments	predi-cates	synt. time (s)	verif. time (s)
webcam	128 (125565)	47	192	215	794
16450 UART	81 (407)	65	128	210	464
exynos UART	80 (1185)	54	111	645	82
STM SPI	68 (389)	29	63	67	31
exynos SPI	83 (933)	31	72	25	44
exynos I2C	65 (303)	21	56	45	96
RT clock	92 (810)	25	74	56	127
IDE	114 (1333)	42	105	285	778

Table 2: Performance of the Termite game solver.

abstraction and modeling language. In particular, the use of transaction-level device modeling abstracts away complicated internal device machinery by focusing on high-level events relevant to driver synthesis, while the TSL language allows modeling the driver environment using standard programming techniques, as illustrated by our running example.

Interestingly, we found the most error-prone step in developing specifications for driver synthesis to be defining correct relative ordering of OS-level and device-level events with the help of the virtual interface (Section 2.3). Naïve specifications tend to be either too restrictive, leading to synthesis failures, or too liberal, leading to incorrect synthesized drivers. As we gained more experience synthesizing different types of drivers, we identified common modeling patterns that help avoid errors in virtual interface specifications.

As a common example, most virtual interfaces contain callbacks that signal a change to one of device configuration parameters, e.g., transfer speed, parity, etc. A naïve OS model may only allow such a callback to be triggered when the OS has requested a change to the corresponding device setting. However, many devices only allow setting multiple configuration parameters simultaneously, so that setting any individual parameter triggers multiple callbacks, thus making the specification non-synthesizable. The problem can be rectified by changing the device specification to only trigger callbacks if the new value of the parameter is different from the old one; however this bloats the device model due to the extra checks. A better solution, used in all our models, is to design the OS specification to allow configuration callbacks to be triggered at any time, provided that the new value of the parameter is equal to the last value requested by the OS.

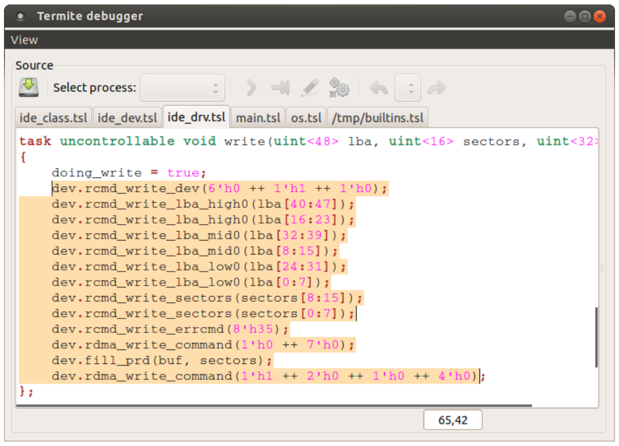
Synthesis time Table 2 summarises the performance of the Termite game solver in our case studies. The second column of the table characterises the complexity of the two-player game constructed by the TSL compiler from the input specifications in terms of the number of states variables and the total number of bits in these variables. The third column shows the number of iterations of the abstraction refinement loop required to solve the game. The next column shows the size of the abstract game at the final iteration, in terms of the number of predicates in the abstract state space of the game. These results demonstrate the dramatic reduction of the problem dimension achieved by our abstraction refinement method. The second-last column shows that the Termite game solver was able to find the most general winning strategy within a few minutes in all case studies.

We compared the performance of the Termite game solver against a state-of-the-art abstraction refinement algorithm for games [10] as well as against the standard symbolic algorithm for solving games without abstraction [22]. In all case studies, the Termite solver was the only one to find a winning strategy within a two-hour limit. We refer the reader to [30] for a more detailed performance analysis of the Termite synthesis algorithm.

The final column of Table 2 shows the time that it took Termite to verify a complete driver. Recall that the Termite synthesis algorithm doubles as a verification algorithm and can be used to verify drivers written in TSL. We used complete synthesized drivers, containing a combination of manual and automatically generated code, as inputs to Termite. We have been able to successfully verify all of our drivers. We also experimented with introducing faults to synthesized drivers. Termite was able to detect these faults and produce correct counterexample strategies. In most cases verification took longer than synthesis. The reason for this is that Termite has not yet been optimized for verification workloads. This is one area for future improvement.

User-guided code generation and debugging We evaluate the key contribution of this paper, namely the user-guided debugging and code generation technique. Each line of code in a Termite-generated driver originates from one of three sources: it can be (1) synthesized automatically by the tool, (2) developed offline and given to Termite as part of the driver template, or (3) added or modified by the user during an interactive code generation session. A perfect synthesis tool, capable of generating a complete driver fully automatically while producing code that meets all non-functional requirements, would eliminate the need for manual code altogether. We do not believe that such a tool is feasible in the near future. We therefore explore the tradeoffs that arise when using our current, imperfect, tool. In particular, we would like to empirically characterize situations when the user can rely on the synthesizer to automatically produce near-optimal code, and when they are better off completely or partially implementing certain functionality manually. These tradeoffs are likely to change as the tool improves.

Based on our experience so far, automatic synthesis is most helpful in generating code that performs device configuration or starts a data transfer. This code may involve a long sequence of commands to the device, which must be issued in the right order and with correct arguments. The synthesis algorithm of Termite proved more effective at doing this than human developers, producing correct code that only requires minimal cosmetic changes in most cases. For example, Figure 5 shows a screenshot of Termite with a synthesized implementation of the IDE driver `write()` function, which starts a data transfer to the device. The function writes request parameters into appropriate device data registers and sets bit fields in command registers to prepare the device for data transfer. One deficiency in this auto-generated implementation is that it uses absolute values instead of symbolic constants for bit fields.



```
task uncontrollable void write(uint<48> lba, uint<16> sectors, uint<32
{
    doing_write = true;
    dev.rcmd_write_dev(6'h0 ++ 1'h1 ++ 1'h0);
    dev.rcmd_write_lba_high0(lba[40:47]);
    dev.rcmd_write_lba_high0(lba[16:23]);
    dev.rcmd_write_lba_mid0(lba[32:39]);
    dev.rcmd_write_lba_mid0(lba[8:15]);
    dev.rcmd_write_lba_low0(lba[24:31]);
    dev.rcmd_write_lba_low0(lba[0:7]);
    dev.rcmd_write_sectors(sectors[8:15]);
    dev.rcmd_write_sectors(sectors[0:7]);
    dev.rcmd_write_errcmd(8'h35);
    dev.rdma_write_command(1'h0 ++ 7'h0);
    dev.fill_prd(buf, sectors);
    dev.rdma_write_command(1'h1 ++ 2'h0 ++ 1'h0 ++ 4'h0);
};
```

Figure 5: Screenshot of Termite with a synthesized implementation of the IDE driver. Automatically generated code is highlighted.

As another example of suboptimal synthesized code, consider the following synthesized fragment

```
void packet_received() {
  if (((packet_data[9:9] == 1) &&
       (packet_data[14:14] == 1))) {
    os.ack_packet(1,1,packet_data[16:32]);
  } else if ((dev.packet_data[9:9] == 1)) {
    os.ack_packet(1,0,packet_data);
  } else if ((dev.packet_data[14:14] == 1)) {
    os.ack_packet(0,1,packet_data[16:32]);
  } else {
    os.ack_packet(0,0,packet_data[16:32]);
  };
};
```

which can be replaced by an equivalent one-liner

```
os.ack_packet(packet_data[9:9],
    packet_data[14:14],packet_data[16:32]);
```

While both issues can, and will, be addressed by an improved code generation algorithm, our experience shows that unaccounted corner cases will arise occasionally. Therefore, the ability to manually modify synthesized code without sacrificing correctness is crucial for a practical synthesis tool.

Limitations of Termite are most noticeable in synthesizing interrupt handler code responsible for processing I/O completions. This involves querying device state to determine which operations completed and with what status, reporting results to the OS, and clearing interrupt status registers. Since Termite does not support grey-box synthesis, it can not generate this code automatically and instead produces code that directly accesses device-internal state (see Section 3). Termite correctly reports such situations and allows the user to mitigate them by manually editing synthesized code. In practice, however, we found it easier to develop most of the interrupt handler logic offline, as part of the driver template, and rely on Termite to (a) establish correctness of this code and (b) extend it to a complete implementation.

In our case studies, 60% to 90% of the code was generated fully automatically, with the rest of the code produced in a user-guided fashion. Once an initial version of device and OS specifications was ready, it took us several hours to generate the driver implementation for each of our case studies. Three quarters of this time was spent debugging the input specifications, with the rest of it spent generating driver source code with the help of the user-guided code generation GUI.

We found counterexample-driven debugging to be crucial to the productivity of synthesis-based development. Before the debugger was available, we had to rely on code inspection to identify defects in the input specifications, which proved to be a frustrating and unpredictably long process. The Termite debugger streamlines this process, giving us the confidence that any failure can be localised by following well-defined steps. A typical debugging session takes a few minutes and involves entering only a few commands manually before the defect is localised.

Size of synthesized code The last two columns of Table 1 compare the size of synthesized drivers to existing manually developed drivers. Synthesised drivers are significantly more compact than conventional drivers for two main reasons. First, as explained in Section 6, we only synthesize the driver logic directly responsible for controlling the device. Conventional drivers typically contain a large amount of boilerplate code managing various OS resources. We believe that this code can and should be synthesized using complementary techniques. At the moment we implement this functionality manually as a wrapper around the synthesized driver.

Second, conventional device drivers are often designed to support multiple similar devices with slightly different interfaces and capabilities. This leads to code bloat, as the driver must implement multiple versions of various operations, as well as logic to dynamically discover device capabilities and choose the right implementation to use. In contrast, every Termite driver supports one specific device model with a fixed set of features. Drivers for similar devices can share common specification code, but are synthesized as separate source code modules. This approach leads to simpler code and is preferable for platforms with a fixed set of peripheral devices, such as smartphones, where shipping drivers that support only the required devices enables smaller system image.

Specification reuse Our specification methodology ensures mutual independence of device and OS specifications, and thus facilitates their reuse. We have not yet carried out a substantial evaluation of such reuse; however we report our limited experience based on synthesizing two SPI drivers for the seL4 OS. The corresponding OS specification was initially developed during the work on the SPI driver for the exynos chipset. It was later used to synthesize a driver for the STM32F10 chipset. We were able to reuse most of the original specification. Minor changes (8 lines of code) were required in the part of the specification describing configuration functionality of the driver, since the STM SPI controller supports a number of ad hoc transfer modes. We expect to observe similar pattern for other devices and operating systems: generic OS specifications can be reused with localized, device-specific changes required to support non-standard device features.

Performance of synthesized drivers Our synthesized drivers implement effectively identical device control logic to their conventional counterparts and therefore have similar performance. We benchmarked the USB webcam driver, which is the most performance-critical one among our case studies. We measured CPU load and data throughput generated by the conventional and synthesized drivers for varying bitrates. We obtained identical results, modulo measurement errors, for both drivers in all cases.

8 Related work

Device driver reliability has been an active area of research for a number of years. Some of the techniques for dealing with buggy drivers include runtime isolation [27, 17], virtualisation [18], static verification [2, 9, 21], symbolic execution [7], language-based protection [34, 23], domain-specific languages [11, 19], hardware-software co-verification [25], etc.

This research has demonstrated the effectiveness of formal techniques in improving driver reliability. Interestingly, formal approaches to driver correctness fall into methods that verify existing drivers and methods that combine verification with an improved driver architecture. The latter rely on language and architectural support to eliminate entire families of driver bugs *by design*. Recent examples include the P programming language [11] and the active driver framework [1], which facilitate the development and automatic verification of asynchronous event-driven code. Our work can be seen as taking this correctness-by-construction approach to the extreme by generating drivers in an automated fashion.

9 Conclusion and future work

We presented the design and implementation of the Termite driver synthesis tool. Termite is the first tool to marry automatic game-based synthesis with conventional manual development. It is also the first practical synthesis tool based on abstraction refinement. Finally, it is the first synthesis tool to support automated debugging of input specifications.

Based on our experimental results, we consider Termite to be an important step towards truly practical device driver synthesis. In particular, our synthesis algorithm is able to efficiently handle real-world device specifications, while the user-guided approach reliably leads to high-quality code.

Our ongoing research focuses on solving the key remaining problems described in Section 6, primarily the DMA problem, which poses the main obstacle to synthesis of more complex drivers, and the grey-box synthesis problem, which limits the degree of automation achieved by Termite. Next, we will explore ways to improve the quality of automatically generated code and thus further reduce the need for user involvement. This includes performance- and power-aware synthesis. Finally, we plan to investigate automatic synthesis of hardened device drivers, i.e., drivers that gracefully handle misbehaving devices [15].

References

[1] S. Amani, P. Chubb, A. Donaldson, A. Legg, K. C. Ong, L. Ryzhyk, and Y. Zhu. Automatic verification of active device drivers. *ACM Operating Systems Review*, 48(1), May 2014.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *1st EuroSys Conference*, pages 73–85, Leuven, Belgium, Apr. 2006.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[4] L. Cai and D. Gajski. Transaction level modeling: an overview. In *"1st International Conference on Hardware/Software Codesign and System Synthesis"*, pages 19–24, Newport Beach, CA, USA, 2003.

[5] P. Cerny, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, Saint Petersburg, Russia, July 2013.

[6] P. Cerny, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free synthesis for concurrency. In *CAV*, Vienna, Austria, July 2014.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1):2:1–2:49, Feb. 2012.

[8] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, Oct. 2001.

[9] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[10] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *18th International Conference on Concurrency Theory*, pages 74–89, Lisboa, Portugal, Sept. 2007.

[11] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 321–332, Seattle, Washington, USA, 2013.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 191–202, Portland, Oregon, 2002.

[13] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th USENIX Large Installation System Administration Conference*, pages 101–111, Washington, DC, USA, 2006.

[14] Intel Corporation. Cofluent technolofy. `http://www.intel.com/content/www/us/en/cofluent/cofluent-difference.html`.

[15] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, 2009.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct. 2009.

[17] B. Leslie, P. Chubb, N. FitzRoy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, USA, Dec. 2004.

[19] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, CA, USA, Oct. 2000.

[20] 16550 UART core. `http://opencores.org/project,a_vhd_16550_uart`.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *16th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 305–318, Newport Beach, CA, USA, Mar. 2011.

[22] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380, Jan. 2006.

[23] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a moderm language. In *USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009.

[24] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct. 2009.

[25] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser. Improved device driver reliability through hardware verification reuse. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, Mar. 2011.

[26] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: device drivers as self-describing artifacts. In *1st EuroSys Conference*, pages 45–57, Leuven, Belgium, 2006.

[27] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA, Oct. 2003.

[28] Synopsys. Virtual prototyping models. `http://www.synopsys.com/Systems/VirtualPrototyping/VPModels`.

[29] W. Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.

[30] A. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. In *FMCAD*, Lausanne, Switzerland, Oct. 2014.

[31] Wind River. Wind River Simics Model Builder user guide. version 4.4, Sept. 2010.

[32] WindRiver Simics DS12887 Model. `http://www.windriver.com/products/simics`.

[33] R. Yavatkar. Era of SoCs, presentation at the Intel Workshop on Device Driver Reliability, Modeling and Synthesis, Mar. 2012.

[34] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th USENIX Symposium on Operating Systems Design and Implementation*, pages 45–60, Seattle, WA, USA, Nov. 2006.