Termite Tutorial

Adam Walker Leonid Ryzhyk

September 2, 2014

Contents

1 Introduction

Before reading this tutorial, it is recommended that you read [1] to understand the main concepts of the Termite tool. Additionally, the reader is referred to [2] to further understand the driver synthesis algorithm.

In this tutorial we will generate a driver for a hypothetical one bit GPIO device. The operating system may request that the driver set and clear the bit. The device exposes a simple interface consisting of a single 8 bit register. One of these bits is connected to the external GPIO output; the rest are ignored. The driver that will be automatically synthesized writes the appropriate value to this register to fulfill the OS request.

The thoroughly commented source code of this example is available in the `documentation/GPIO` directory in the Termite repository at `http://github.com/termite2/Termite`.

2 Specifications

The inputs to the Termite specification tool are:

1. the device specification
2. the operating system specification
3. the empty driver template
4. the class specification that declares the interfaces of the above 3 components
5. the main file that connects everything together

The specifications are connected as shown in figure 1. We will describe each of these in turn.

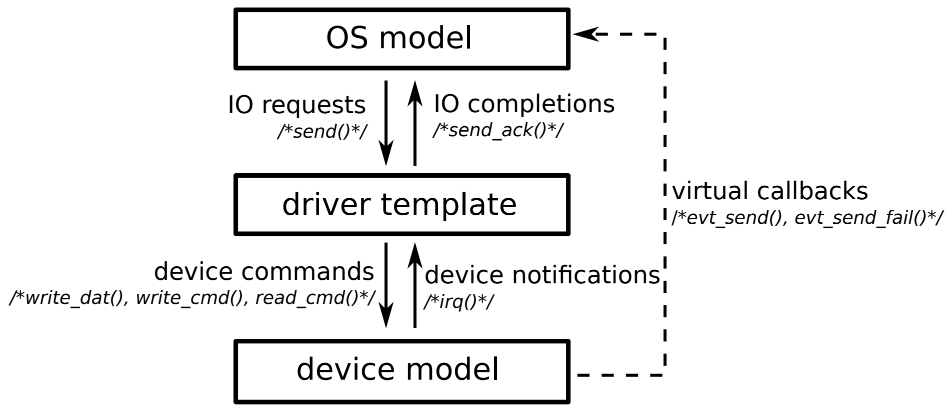


Figure 1: Input specifications for driver synthesis

2.1 Device specification

The device we are specifying is a simple 1 bit GPIO device whos output bit can be accessed by writing to a register. We declare a 1 bit variable named output to keep track of the current state of the GPIO output. The GPIO device has two interface methods - `get_value()` for querying the current state of the GPIO pin, which will be used by the OS specification later, and `write8` which the driver can use to write to the register connected to the GPIO pin.

`get_value()` operates as expected - it simply returns the value that is currently being output. Note that `get_value` is a virtual function, it does not correspond to any run time behavior and is only used by the OS specification to enforce correct behavior.

`write8` sets the GPIO output to bit 0 of the value written, and, it additionally notifies the OS that the value was set using `os.value_set()`. This is also a virtual function. As we will see later, this is used to prevent the driver from changing the value of the GPIO when it was not requested to.

```
import <class.tsl>

template gpio_dev_inst(gpio_os os)

derive gpio_dev;

uint<1> output;

task controllable void write8(uint<8> val){
    output = val[0:0];
    os.value_set(output);
};

function uint<1> get_value(){
    return output;
};

endtemplate
```

2.2 OS specification

In this example, the operating system specification is the most complex specification. Unlike the device, the OS contains an active process. The role of this process is to generate driver requests, dispatch them, and ensure that the driver completed its job. It is similar to a test harness for the driver.

The process `pos` loops forever (the `forever` block), nondeterministically choosing one of two driver requests to make. The forever block must contain a `pause` statement to that its body cannot complete instantaneously. The `choice` block non-deterministically decides to either set or clear the GPIO bit. We will only consider the `set_bit()` function as `clr_bit()` is very similar.

The set_bit task ensures that the making_request flag is true whenever execution is in the driver function. This is to ensure that the driver does not make changes to the GPIO bit when not requested to as explained below. It then calls the driver method and checks that the driver performed the required action by calling the device's get_value() function and checking that it is the expected value.

All that remains is the value_set() virtual callback function. This is called by the device whenever the value of the GPIO pin is changed. It asserts that we are currently making a request using the making_request flag. This prevents the driver from changing the GPIO output when not requested to. It does not, however, prevent the driver from changing the value to a temporarily wrong value during the driver call.

```
import <class.tsl>

template gpio_os_inst(gpio_drv drv, gpio_dev dev)

derive gpio_os;

process pos {
    forever{
        choice{
            set_bit();
            clr_bit();
            {};
        };
        pause;
    };
};

bool making_request = false;

task void set_bit(){
    making_request = true;
    drv.set_bit(); // call the driver
    making_request = false;

    assert(dev.get_value() == 1'h1);
};

task void clr_bit(){
    making_request = true;
    drv.clr_bit(); // call the driver
    making_request = false;

    assert(dev.get_value() == 1'h0);
};

procedure void value_set(uint<1> value){
    // Only allow the GPIO pin to change its state when there is a request in progress
    assert(making_request);
};

endtemplate
```

2.3 Driver template

The driver template is what the Termite tool will interactively fill in to create the final driver. The driver consists of two methods: set_bit() and clr_bit() as declared in the class specification. They neither return a result or take any arguments. You will notice that the body of each driver method contains only an ellipsis. This is referred to as a magic block. It is Termite's job to replace all magic blocks with driver code. When we run the tool in section 3 we will interactively ask Termite to fill these in.

```
import <class.tsl>
import <gpio.tsl>

template gpio_drv_inst(gpio_dev_inst dev, gpio_os os)

derive gpio_drv;

task uncontrollable void set_bit(){
    ...;
};

task uncontrollable void clr_bit(){
    ...;
};

endtemplate
```

2.4 Class specification

The class specification declares three abstract interfaces, one each for the driver, device and operating system components.

```
template gpio_drv

task uncontrollable void set_bit();
task uncontrollable void clr_bit();

endtemplate

template gpio_dev

function uint<1> get_value();

endtemplate

template gpio_os

procedure void value_set(uint<1> value);

endtemplate
```

2.5 Main file

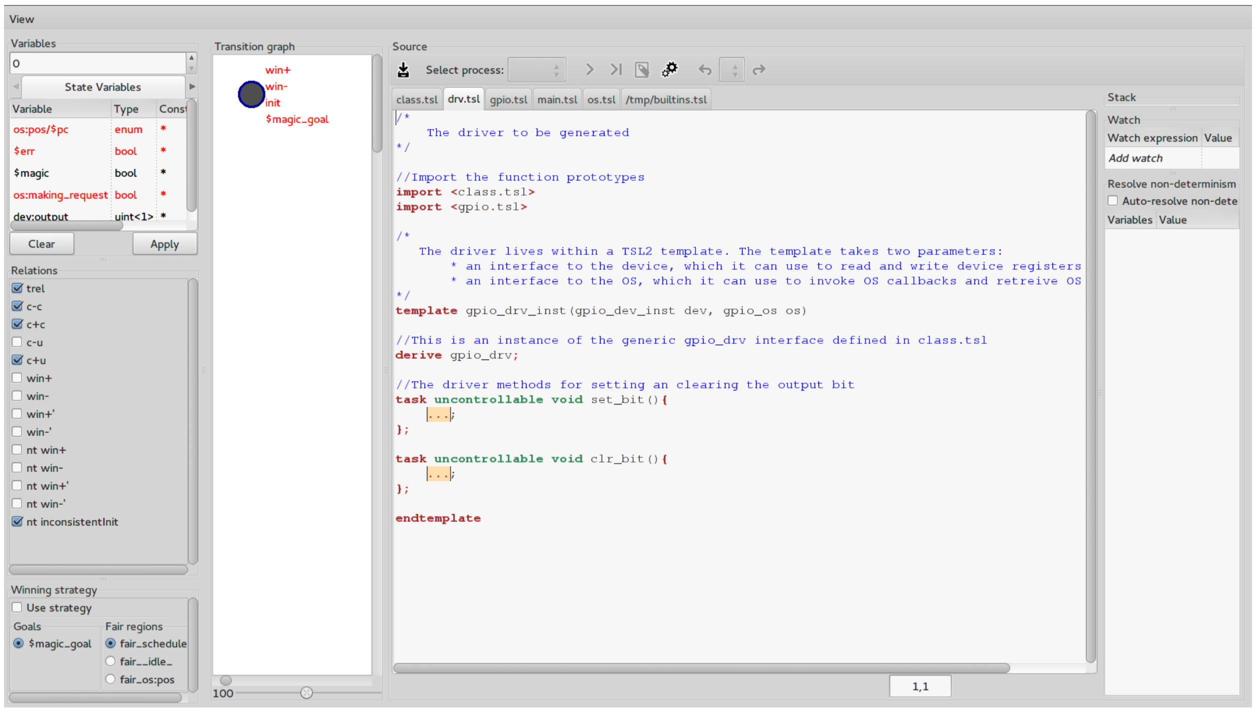
The main file simply creates an instance of each of the components described earlier and ties them together.

```
import <os.tsl>
import <gpio.tsl>
import <drv.tsl>

template main

instance gpio_os_inst  os  (drv, dev);
instance gpio_dev_inst dev (os);
instance gpio_drv_inst drv (dev, os);

endtemplate
```

3 Running the tool

Enter the directory of the specifications and run the tool with:

```
termite -i main.tsl -s
```



Variable	Type	Const
os:pos/$pc	enum	*
$err	bool	*
$magic	bool	*
os:making_request	bool	*
dev:output	uint<1>	*

- [✓] trel
- [✓] c-c
- [✓] c+c
- [ ] c-u
- [✓] c+u
- [ ] win+
- [ ] win-
- [ ] win+'
- [ ] win-'
- [ ] nt win+
- [ ] nt win-
- [ ] nt win+'
- [ ] nt win-'
- [✓] nt inconsistentInit

Goals: ● $magic_goal

Fair regions: ● fair_schedule ○ fair__idle_ ○ fair_os:pos

```
/*
    The driver to be generated
*/

//Import the function prototypes
import <class.tsl>
import <gpio.tsl>

/*
   The driver lives within a TSL2 template. The template takes two parameters:
       * an interface to the device, which it can use to read and write device registers
       * an interface to the OS, which it can use to invoke OS callbacks and retreive OS
*/
template gpio_drv_inst(gpio_dev_inst dev, gpio_os os)

//This is an instance of the generic gpio_drv interface defined in class.tsl
derive gpio_drv;

//The driver methods for setting an clearing the output bit
task uncontrollable void set_bit(){
    ...;
};

task uncontrollable void clr_bit(){
    ...;
};

endtemplate
```

Figure 2: The debugger after switching to the `drv.tsl` tab

`-i` specifies the input file and `-s` tells Termite to perform synthesis.

It should only take a few seconds to synthesize and then launch a graphical debugging console. We will now ask Termite to fill in the magic blocks.

4 Generating the driver

Once the graphical debugger has been launched, switch to the `drv.tsl` tab in the rightmost pane. The window should look like figure 2. Notice the yellow highlighted magic blocks. We are now going to generate the code that goes in these magic blocks. Click on the first magic block and then click on the button with the gears icon at the top of the page. A line of code should be automatically generated above the magic block. Click the gears one more time. This time the magic block should be replaced by {}, indicating that there is nothing more to do in the function.

Repeat the above steps for the second magic block. The `drv.tsl` file should appear as in figure 3. You should convince yourself that the generated code is correct. Congratulations, you have generated your first Termite driver!

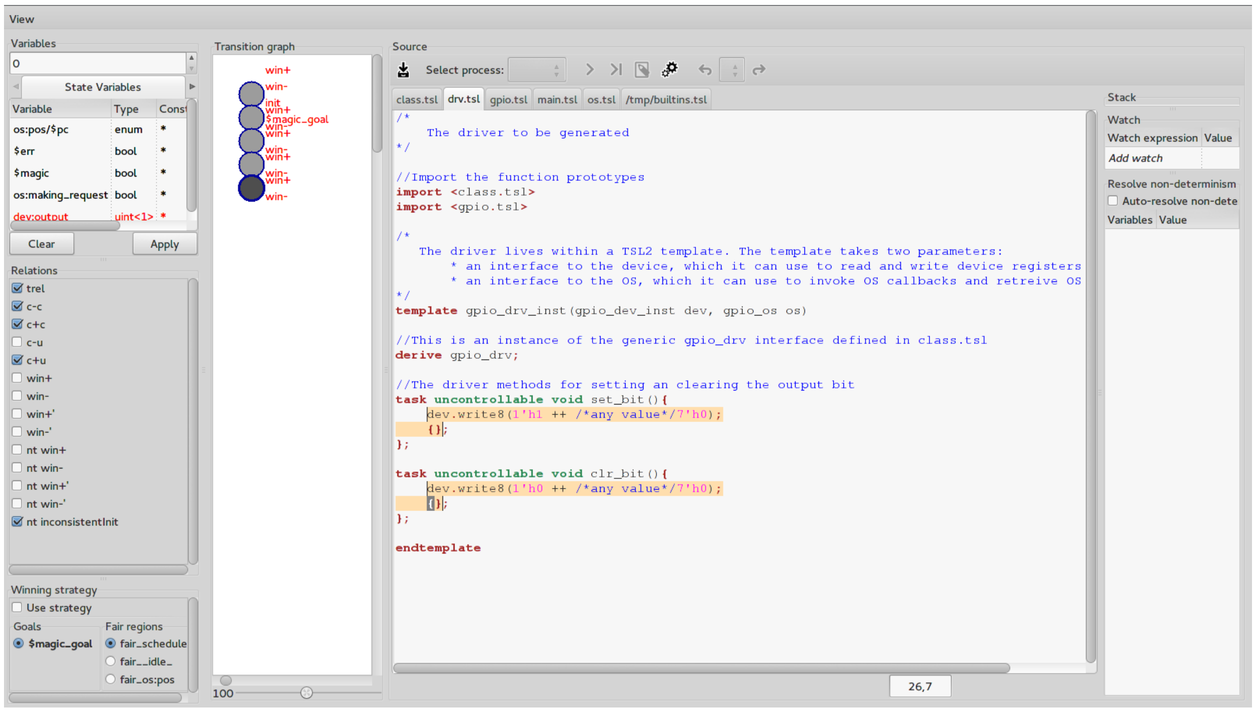
You may save `drv.tsl` and then rerun Termite. As there are no magic blocks, Termite is now acting as a verifier. It should report that synthesis succeeded.

5 Further reading

An extensively commented I2C driver example is available at `https://github.com/termite2/Termite/tree/master/documentation/i2c`. It demonstrates more advanced usage of Termite on a real device.

References

[1] L. Ryzhyk, A. C. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)*,



Variable	Type	Const
os:pos/$pc	enum	*
$err	bool	*
$magic	bool	*
os:making_request	bool	*
dev:output	uint<1>	*

- [✓] trel
- [✓] c-c
- [✓] c+c
- [ ] c-u
- [✓] c+u
- [ ] win+
- [ ] win-
- [ ] win+'
- [ ] win-'
- [ ] nt win+
- [ ] nt win-
- [ ] nt win+'
- [ ] nt win-'
- [✓] nt inconsistentInit

- [ ] Use strategy

Goals: [● $magic_goal]

Fair regions: [● fair_schedule] ○ fair__idle_ ○ fair_os:pos

```
/*
    The driver to be generated
*/

//Import the function prototypes
import <class.tsl>
import <gpio.tsl>

/*
   The driver lives within a TSL2 template. The template takes two parameters:
       * an interface to the device, which it can use to read and write device registers
       * an interface to the OS, which it can use to invoke OS callbacks and retreive OS
*/
template gpio_drv_inst(gpio_dev_inst dev, gpio_os os)

//This is an instance of the generic gpio_drv interface defined in class.tsl
derive gpio_drv;

//The driver methods for setting an clearing the output bit
task uncontrollable void set_bit(){
    dev.write8(1'h1 ++ /*any value*/7'h0);
    {};
};

task uncontrollable void clr_bit(){
    dev.write8(1'h0 ++ /*any value*/7'h0);
    {};
};

endtemplate
```

Figure 3: The debugger after generating code

Broomfield, CO, USA, oct 2014.

[2] A. C. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, Lausanne, Switzerland, oct 2014.