DD2452 Formal Methods
Lab 2: Model Checking of a Device Driver of a Transmitter

Jonas Haglund

August 20, 2019

1 Introduction

In this lab you will:

1. Make models in NuSMV of a transmitter and the device driver controlling the transmitter, given pseudocode descriptions of their behavior.
2. Make a system model of the transmitter and the device driver that composes the two models.
3. Express some properties in CTL and check whether the NuSMV model satisfies them. One example of such a check is to verify that the device driver does not configure the transmitter into an unknown state where the behavior of the transmitter is not specified. If the transmitter enters such a state, it could do anything, e.g. transmitting confidential data in memory.

Read all of this document before you start with the lab. The following sections describe, respectively: The operation of the transmitter, the device driver, how to do the lab, and the grading.

2 Transmitter

The transmitter performs three kinds of operations (which can be thought of as being performed, but not necessarily, in the following order):

1. Resetting itself.
2. Transmiting messages located in RAM.
3. Tearing down transmission to enter an idle state.

To enable the CPU to command the transmitter to perform these operations, the transmitter has the following set of registers (which the CPU can write in order to command the transmitter and read to see the state of the transmitter; the registers are small in order to avoid the state space explosion problem of model checkers):

- **RESET**: 1-bit register with the initial value 0 when the transmitter (computer system) is powered on.
- **TRANSMIT**: 2-bit register that has no specified initial value.
- **TEARDOWN**: 1-bit register with the initial value 0.
- The transmitter has three buffer descriptors (BDs), with indexes 1 through 3, used to describe the location of the buffers containing the messages to transmit. Each BD has five fields:

- **BD_NP[1 ... 3]**: Three 2-bit registers (NP = Next Pointer).
- **BD_BP[1 ... 3]**: Three 2-bit registers (BP = Buffer Pointer).
- **BD_BL[1 ... 3]**: Three 2-bit registers (BL = Buffer Length).
- **BD_OWN[1 ... 3]**: Three 1-bit registers (OWN = Ownership).
- **BD_EOQ[1 ... 3]**: Three 1-bit registers (EOQ = End of Queue).

A device driver is a part of an operating system used to control a specific I/O device. A device driver of the transmitter is intended to use/configure the transmitter by first initializing the transmitter. Then the transmitter can be used to transmit messages. When the transmitter (or the computer) is turned off, the device driver can tear down transmission.

2.1 Initialization

The device driver initializes the transmitter as follows:

a) The device driver resets the transmitter by writing 1 to **RESET**. When the transmitter has reset itself, the transmitter writes 0 to **RESET** to signal to the device driver that the reset is complete.

b) When the transmitter has reset itself, the device driver shall initialize **TRANSMIT** by writing 0b00 to **TRANSMIT**.

A reset operation must not be initiated during an ongoing initialization, transmission or tear down.

2.2 Transmission

The device driver commands the transmitter to transmit a set of messages by first assigning one free BD (not currently in use by the transmitter to transmit a message) to each message. Then, each buffer descriptor is initialized as follows:

a) The start address of the associated message is written to **BD_BP**. For example, if the device driver shall command the transmitter to transmit two messages, using the buffer descriptors with indexes 1 and 2, where the first message starts at byte address 2 and the other message starts at byte address 3, then the device driver writes 0b10 to **BD_BP[1]** and 0b11 to **BD_BP[2]**.

b) The length of the associated message is written to **BD_BL**. For example, if the two messages consist of 2 and 1 bytes, respectively, then the device driver writes 0b10 to **BD_BL[1]** and 0b01 to **BD_BL[2]**.

c) The **ownership** bit is set, to indicate that the transmitter "owns"/uses the buffer descriptor. For the example, 0b1 is written to **BD_OWN[1]** and **BD_OWN[2]**.

d) The **EOQ** bit is cleared (which is set by the transmitter if the transmitter interprets the current buffer descriptor as last in the buffer descriptor queue). For the example, 0b0 is written to **BD_EOQ[1]** and **BD_EOQ[2]**.

e) The **NP** fields of the BDs are written to form a BD queue. Each **NP** field is written with the index of the next BD in the queue, or zero if the BD is last in the BD queue. For the example, if the message identified by the BD with index 1 shall be transmitted first and then the message identified by the BD with index 2, then the device driver writes 0b10 to **BD_NP[1]** and 0b00 to **BD_NP[2]**, resulting in a BD queue starting with the BD with index 1 and ending with the BD with index 2.

After all BDs have been initialized and a queue has been formed, the queue is given to the transmitter for transmission. If the transmitter is currently not transmitting, the device driver writes the index of the first BD of the queue to **TRANSMIT** (for the example, 0b01). If the transmitter is currently transmitting, the device driver appends the new queue to the end of the queue under transmission. For the example, if the transmitter transmits a queue with the tail being the BD with index 3, then the device driver writes 0b01 to **BD_NP[3]**.

During transmission the transmitter operates as follows. When **TRANSMIT** is written **(TRANSMIT must not be written when it is not zero, except during initialization)**, the transmitter starts processing the BD queue starting with the BD having the index just written to **TRANSMIT**. **The BD in the queue must meet the following requirements:**

- **The buffer length field must not be zero.**
- **The buffer to transmit must be completely located in RAM. RAM starts at address 1 and ends at address 2, inclusive.**
- **The buffer must not overflow with respect to unsigned 2^2 arithmetic (the addresses consist of 2 bits; see the description of the BD_BP and BD_BL fields above).**
- **The ownership bit must be set.**
- **The EOQ bit must be cleared.**

After the transmitter has read the BD and identified the location of the message in RAM, the transmitter transmits the message. Then the transmitter writes some fields of the BD being processed. The writes depend on whether the BD is last in the queue or not:

- Last (the **NP** field is zero): The transmitter sets the **EOQ** bit, clears the **ownership** bit (signaling to the device driver that the BD can be used for a new message to transmit), writes 0 to **TRANSMIT**, and enters an idle state.

- Not last (the **NP** field is not zero): The transmitter clears the **ownership** bit, and sets **TRANSMIT** to the value of the **NP** field of the processed BD (which will be the next BD to process). Then the transmitter processes the next BD, in the same way as the transmitter processed the current BD, unless the transmitter has been commanded to perform a tear down. If the transmitter shall perform a tear down, then the transmitter stops processing the queue and performs the tear down operations.

A misqueue condition occurs if a device driver appends a "new" BD queue to the current, "old", queue under transmission after the transmitter has finished the processing of the "old" queue. This means that the device driver has commanded the transmitter to transmit messages, but which the transmitter unintentionally will not transmit. The device driver detects a misqueue condition when the device driver processes the "old" queue (which the device driver does in order to be able to reuse BDs) and reads a BD in the "old" queue with:

- a cleared **ownership** bit,

- a set **EOQ** bit, and

- a non-zero **NP** field (which contains the index of the first BD in the "new" queue that unintentionally will not be processed by the transmitter).

The device driver corrects a misqueue condition by writing the index of the first BD in the "new" queue to the **TRANSMIT** register.

TRANSMIT shall not be written during tear down.

2.3 Tear Down

The device driver can command the transmitter to cancel transmission by writing 1 to **TEARDOWN**. The transmitter reacts to such a write by first finishing the transmission of the current message, and then by performing a number of operations. These latter operations depend on whether the BD whose buffer was just transmitted is last in the queue under transmission:

- Last (**TRANSMIT = 0**): The **TEARDOWN** register is cleared, to signal to the device driver that transmission has been torn down.

- Not last (**TRANSMIT** $\neq$ **0**): The **EOQ** bit is set and the **ownership** bit is cleared of the BD with the index in **TRANSMIT** (the BD following the BD that was just processed), and **TRANSMIT** is cleared. Then the **TEARDOWN** register is cleared.

A tear down shall not be initiated during initialization or tear down.

3 Device Driver

The device driver has three functions that can be invoked by the operating system:

- **open()**: Initializes data structures of the device driver and the transmitter to enable transmission of messages. When **open()** has terminated, the device driver and the transmitter shall be ready to transmit messages. That is, the operating system can now invoke **transmit(address, length)**.
- **transmit(address : word[2], length : word[2])**: Given the start address **address** and the byte length **length** of a message to transmit, **transmit()** configures the transmitter to transmit the message. This means that **transmit()** performs the following operations (not necessarily in the given order):
  - Assigns a new BD to the message, if there are free buffer descriptors (otherwise **transmit()** does nothing and returns).
  - Updates the data structures of the device driver.
  - Initializes the new BD.
  - Gives the new BD to the transmitter. How the new BD is given to the transmitter depends on whether the transmitter is currently transmitting:
    - Not transmitting: The index of the new BD is written to **TRANSMIT**.
    - Transmitting: The index of the new BD is written to the **NP** field of the last BD in the BD queue under transmission, and corrects any potential misqueue condition.
- **stop()**: Configures the transmitter into an idle state and updates data structures of the device driver. When **stop()** has terminated, the transmitter is in an idle state, but is ready to transmit messages.

All three functions must not configure the transmitter into an unknown state, which happens when the device driver configures the transmitter in an erronous way.

At any point in time at most one of these three functions can be executed (no concurrency). Which of the three functions the operating system invokes next depends on which function that was executed most recently. If the computer has just been powered on, then **open()** is invoked first. Otherwise, if the most recently executed function is:

- **open()**: then the operating system may invoke any function.
- **transmit()**: then the operating system may invoke only **transmit()** or **stop()**.
- **stop()**: then the operating system may invoke any function.

4 Tasks

Your main task is to implement a model in NuSMV that describes the operations of and the interaction between the transmitter and the device driver. To your help, pseudocode is given for each of the six operations: initialization, transmission and tear down of the transmitter, and open, transmit and stop of the device driver. The models of the transmitter and the device driver are preferably structured into submodels, where each submodel describes one of the just mentioned operations. To ease the implementation of the combined model in NuSMV, it is recommented to use the "tables" (sets of if-then-else statements) accompanied with the psuedocode that describe how and when the pseudocode modifies each variable. You are encouraged to implements the model according to the following steps.

4.1 Step 1: Study the Formal Transmitter Description

The text file pseudocode_transmitter.txt contains psuedocode with comments that describe by means of three functions each of the initialization, transmission and tear down operations of the transmitter. Think of each function invocation as describing one transition of the corresponding operation. The purpose of variables is documented with the "tables" describing how and when the variables are modified. Familiarize yourself with this formal description and compare it to the informal description in Section 2.

4.2 Step 2: Study the Formal Device Driver Description

The text file pseudocode_driver.txt contains psuedocode, comments and "tables" describing the three operations of the device driver: open(), transmit(address, length) and stop(). Consider the following psuedocode:

```
open()
    RESET := 0b1_1;
```

```
while (RESET = 0b1_1)
    ;
HDP := 0b2_00;
queue_head := 0b2_00;
queue_tail := 0b2_00
```

This sequence of operations must be expressed at an appropriate granularity. If the granularity is too coarse (e.g. all statements are described in a single transition in NuSMV), then the model does not describe all behavior that is desirable to reason about (e.g. the next state would be undefined if the while condition is true, since the loop would not terminate). Therefore, the operations of a function must be described in NuSMV by a sequence of several atomic transitions. More transitions imply more fine grained interleavings of transitions between different parts of a model. For instance, more transitions describing the operations of the part describing the initialization of the transmitter and the part describing open() of the device driver imply more transition interleavings between initialization and open(). A more accurate description is therefore exhibited of the parallel execution of the transmitter and the CPU.

To enable NuSMV to exhibit all relevant transition interleavings for this lab, the operations performed by open() can be described in NuSMV as transitions between three states as follows (where the lines followed by a state name indicate intermediate states in the execution of open()):

```
————————————————open_idle
open()
————————————————open_set_reset
    RESET := 0b1_1;
————————————————open_reset_test
    while (RESET = 0b1_1)
        ;
    HDP := 0b2_00;
    queue_head := 0b2_00;
    queue_tail := 0b2_00;
————————————————open_idle
```

Before open() is invoked, the state is open_idle. When open() is invoked, a transition is made in NuSMV, describing that open() has just been invoked but no operations of open() has been performed, by entering the state open_set_reset. The next transition in NuSMV sets RESET to 1 and enters the state open_reset_test. The next state depends on whether RESET is 1 or 0 (the latter occurs when the transmitter has cleared RESET). If RESET is 1 then the state is still open_set_reset. Otherwise the next state is open_idle and that transition sets HDP, queue_head and queue_tail to 0.

The model could be designed to also have intermediate states between the three last assignments. Since queue_head and queue_tail are local variables of the device driver, the transmitter cannot read them and therefore the model would not exhibit additional behavior by including such states and transitions. For this reason the state open_set_reset is unnecessary, and simply denotes that open() is currently being executed.

Familiarize yourself with the pseudocode in pseudocode_driver.txt and compare it to the informal description in Section 3.

4.3 Step 3: Plan for Combining Models

Decide how to describe the parallel execution of the transmitter and the CPU executing the device driver, and how the model records erronous configurations of the transmitter:

- **Scheduling: When does what part perform an operation?** What part of the model shall make the next transition? Shall it be the transmitter or the device driver, or both? If the transmitter shall perform a transition, shall that transition describe an operation of initialization, transmission or tear down? If the device driver shall perform a transition, shall that transition describe an operation of open(), transmit() or stop()? Recall that the functions cannot be invoked arbitrarily (see the last paragraph in Section 3), and therefore the model must record which function of the device driver that was executed most recently. **Specify these decisions in a list**, which can have the following form:
  - If the transmitter is resetting itself, then the reset part of the transmitter can perform a transition.
  - If the transmitter is transmitting, then the transmission part of the transmitter can perform a transition.
  - ...
  - If a part of the transmitter can perform a transition, then the transmitter can perform a transition.
  - ...
- **Errors: How does the model indicate that the device driver has configured the transmitter erronously?** Section 2 describes erronous configurations. **List the conditions that cause the transmitter to enter an undefined state due to an erronous configuration.** For instance, "The device driver writes a non-zero value to **TRANSMIT** during initialization."

4.4 Step 4: Preparations for Implementing the Combined Model in NuSMV

Specify how the scheduling and error detection can be described in NuSMV:

- Considering your lists from Step 3, Section 4.3, what variables are needed in addition to the variables used in the pseudocode of the device driver and the transmitter? For each of these additional variables, write if-then-else statements to specify **how** and **when** that variable is modified. Regarding scheduling, the if-then-else statements could look like:

  if tx.it.state != idle then TX_ACTIVE := TRUE

  else if tx.tx.state != idle then TX_ACTIVE := TRUE

  else if tx.td.state != idle then TX_ACTIVE := TRUE

  else TX_ACTIVE := FALSE

  An example of a statement for detecting error conditions is:

  if next(TRANSMIT) != 0 & next(TRANSMIT) != TRANSMIT & ... then ...

- For each listed variable, specify whether that variable has an initial value, and if so, what that initial value is.

4.5 Step 5: Implement and Document the Combined Model in NuSMV

Use the psuedocode in pseudocode_driver.txt and pseudocode_transmitter.txt, and the if-then-else statements you wrote in the previous step, to implement a model in NuSMV that describes the operation of and the interaction between the transmitter and the device driver. **Document your NuSMV code with comments describing**:

- What behavior each module describes.
- What each variable is used for.
- How the modules interact and why the modeled interaction makes sense.

4.6 Step 6: Checking Correctness of Implemented Model

Verify some properties of your model to make sure the model describes the desired behavior of the transmitter and the device driver and their interaction. **The following properties must be checked in NuSMV**:

- It is always possible for the model to sooner or later make transitions that describe the operations of open(), transmit(), and stop().

- It is always possible for the model to sooner or later make transitions that describe the initialization, transmission and tear down operations of the transmitter.
- open(), transmit(), stop() cannot be executed simultaneously.
- If the transmitter is resetting itself, then the transmitter does not transmit nor performs a tear down.
- If the transmitter transmits, then the transmitter is not performing a reset.
- If the transmitter transmits, then the transmitter is not performing a tear down or the tear down is waiting for the transmission to finish.
- If the transmitter performs a tear down, then the transmitter is not performing a reset.
- If the transmitter performs a reset, then open is currently executed.
- If the transmitter performs a tear down, then stop is currently executed.

Write the CTL formulas in a readable way (a long formula on a single line is difficult to interpret, but inserting line breaks at appropriate points might make it easier). **Try to figure out three CTL formulas to check the correctness of your model and check them as well.** For instance, if the device driver/transmitter is in a certain state, then the transmitter/device driver is (not) performing a certain operation. **Describe the CTL formulas in natural language and motivate why they are good for checking the correctness of the model.**

4.7 Step 7: Verification of Safe Device Driver

Verify in NuSMV that the transmitter never performs an undefined operation. Motivate why the CTL formula is correct.

Describe in natural language two properties that are relevant for verification. One property shall involve the data structures of the device driver (queue_head and queue_tail), and the other property shall state something about the BD queue of the transmitter. **Formalize the two properties as CTL formulas and motivate why the CTL formulas are true in the model. Check with NuSMV that the two properties hold.**

4.8 Step 8: Verification of Synchronization and No Misqueue Conditions

Verify in NuSMV:

- That all writes to the transmitter registers are performed synchronously. That is, the transmitter and the device driver never writes to the same register simultaneously (**RESET**, **TRANSMIT**, **TEARDOWN**, nor the same field of the same BD; e.g. the transmitter and the device driver never writes **BD_OWN[2]** in the same NuSMV transition).
- That no misqueue condition can occur (see last part of section 2.2).

Write the CTL formulas and motivate why they formalize the desired properties.

5 Grading

The grading is as follows:

E Do steps 1-7 (Sections 4.1 through 4.7) and write a report with answers to all tasks given in those steps. Submit the NuSMV code via Canvas.

D As E with a well-structured report that is easy to read. In addition, discuss and reflect what the meaning of the verification is. You can consider the following aspects:

- What has actually been verified?
- What has not been verified?
- What is the accuracy of the verification?
- How could the verification be made more accurate?
- How reliable is the verification?
- How could the verification be made more reliable?
- What relevant aspects/properties might be desirable to verify or to take into account in the verification (abstraction level of the model) that are not taken into account in the verification (abstracted away)?
- What is of practical importance in construction of models?
- What was most difficult?
- Did you find any bugs in your model or device driver design? In such a case, what was the bug(s)?
- You are encouraged to discuss and reflect over other aspects as well.

C As D but with the addition of step 8, Section 4.8, with the corresponding answers in the report.