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

Jonas Haglund

October 9, 2018

1 Introduction

In this lab you will:

1. Design/model in pseudocode a device driver controlling a transmitter.
2. Model the transmitter in pseudocode.
3. Implement a model in NuSMV that consists of the composition of these two pseudocode models.
4. Express some properties in CTL and check whether the NuSMV model satisfies them. One of these checks are used to verify that the device driver does not configure the transmitter into an undefined/unspecified/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. Section 2 describes the operation of the transmitter. Section 3 describes the interface of the device driver. Section 4 describes the tasks of the lab. Section 5 describes 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. Resets itself.
2. Transmits messages located in RAM.
3. Tear downs (cancels) 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 buffer descriptor 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 piece of software of an operating system controlling 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) shall be 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 buffer descriptor (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 have length 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** field is written to form a buffer descriptor queue. The **NP** fields is written with the index of the next buffer descriptor in the queue, or zero if the buffer descriptor is last in the buffer descriptor queue. For the example, if the message in buffer descriptor 1 shall be transmitted first and then the message in buffer descriptor 2, then the device driver writes 0b10 to **BD_NP[1]** and 0b00 to **BD_NP[2]**, resulting in a buffer descriptor queue starting with buffer descriptor 1 and ending with buffer descriptor 2.

After all buffer descriptors have been initialized and a queue has been formed, the queue is given to the transmitter. If the transmitter is currently not transmitting, the device driver writes the index of the first buffer descriptor 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 buffer descriptor 3, then the device driver writes 0b01 to **BD_NP[3]**.

The actual transmission is done as follows. When **TRANSMIT** is written **(TRANSMIT must not be written when it is not zero, except during initialization)**, the transmitter starts processing the buffer descriptor queue beginning with the buffer descriptor with the index just written to **TRANSMIT**. **The buffer descriptor 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 buffer descriptor and identified the location of the buffer in RAM, the transmitter transmits the message. Then the transmitter performs a number of operations on the buffer descriptor. The performed operations depend on whether the buffer descriptor is last in the queue or not:

- Last (the **NP** field is zero of the buffer descriptor): The transmitter sets the **EOQ** bit, clears the **ownership** bit (signaling to the device driver that the buffer descriptor 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 of the buffer descriptor): The transmitter clears the **ownership** bit, and sets **TRANSMIT** to the value of the **NP** field of the buffer descriptor. Then the transmitter processes the next buffer descriptor, in the same way as the transmitter processed the current buffer descriptor, unless the transmitter has been commanded to perform a tear down. If the transmitter shall tear down transmission, the transmitter stops processing the queue and performs tear down operations.

A misqueue condition occurs if the device driver appends a "new" buffer descriptor queue to the current, "old", queue under transmission immediately after the transmitter has interpreted the "old" queue as being completely processed. 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 buffer descriptors) and reads a buffer descriptor 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 buffer descriptor 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 buffer descriptor 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 finishing the transmission of the current message, and then performs a set of operations. This set of operations depends on whether the buffer descriptor 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 buffer descriptor with the index in **TRANSMIT**, and **TRANSMIT** is cleared. Then the **TEARDOWN** register is cleared to signal to the device driver that the tear down is complete.

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

3 Interface of the 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, invocations of **transmit(address, length)** can now be made.
- **transmit(address : word[2], length : word[2])**: Given the start address **address** and the length **length** in bytes 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 buffer descriptor 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 buffer descriptor appropriately.
  - Gives the new buffer descriptor to the transmitter. How the new buffer descriptor is given to the transmitter depends on whether the transmitter is currently transmitting:
    - Not transmitting: The index of the new buffer descriptor is written to **TRANSMIT**.
    - Transmitting: The index of the new buffer descriptor is written to the **NP** field of the last buffer descriptor in the buffer descriptor queue under transmission, and corrects any 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 undefined state (which is a state that the transmitter enters when the device driver configures the transmitter to perform an undefined operation).

These three functions cannot be executed in parallel. Which of the three functions the operating system invokes next depends on which function that was invoked most recently. If the computer has just been turned on, then the operating system will first invoke **open()**. Otherwise, if the most recently invoked 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 device driver and the transmitter. This model shall be constructed by first modeling the device driver and the transmitter separately in pseudocode. The models of the device driver and the transmitter are preferably structured into submodels, where each submodel describes the operations of **open()**, **transmit()**, **stop()**, or one of the three kinds of the operations of the transmitter, namely initialization/reset, transmission, or tear down. To ease the implementation of the combined model in NuSMV, one "table" (a set of if-then-else statements) is constructed for each variable that is modified in the pseudocode. Each table describes how and when the corresponding variable is modified. These tables are then used to implement in NuSMV the combination of the model of the device driver and the model of the transmitter. The following steps must be followed.

4.1 Step 1: Model open(), transmit() and stop()

Design and write pseudocode of open(), transmit(address, length) and stop() such that they provide the desired functionality without configuring the transmitter into an undefined state. It is important the the device driver is aware of which buffer descriptors that may be in use by the transmitter. Use descriptive variable names. This gives three models, each described in pseudocode, one for each function.

4.2 Step 2: Identify Relevant Control Points/States of open(), transmit() and stop()

Identify the relevant states/control points, of open(), transmit(address, length) and stop() in the pseudocode from Step 1, that must exist in order to model the relevant behavior of and interaction between the device driver and the transmitter. For instance, the transmitter can perform operations simultaneously as the CPU executes **transmit(address, length)**. Say that **open()** is described by the following pseudocode:

```
open()
    RESET := 0b1_1;
    while (RESET = 0b1_1)
        ;
    x := 0;
    y := 0
```

Consider the following three states and their associated meaning to identify the "relevant" control points of this pseudocode description of **open()**:

- **open_idle**: **open()** is currently not executed.
- **open_perform_reset**: **open()** is currently executed and the next operation to be performed by **open()** is to write 1 to the **RESET** register and to loop while **RESET** = 1.
- **open_clear_x_and_y**: **open()** is currently executed and the next operation is to clear x and y.

To easier understand what control point each state denotes, the pseudocode is commented at the control points with the name of the state representing that control point:

```
open()
//open_perform_reset
    RESET := 0b1_1;
    while (RESET = 0b1_1)
        ;
//open_clear_x_and_y
    x := 0;
    y := 0
//open_idle
```

This assignment of control points to states does not make sense. Assume the model includes descriptions of *only* the CPU performing operations during a transition (the model can also include descriptions of the CPU and the transmitter performing operations simultaneously, and also include descriptions of *only* the transmitter performing operations; this gives three possible combinations of what device operations are described by the next transition of the complete model: *only* the CPU, both the CPU and the transmitter, and *only* the transmitter). A "transition" describing *only* CPU operations from a state in which **open** is at the control point denoted by **open_perform_reset**, and ending in a state in which **open** is at the control point denoted by **open_clear_x_and_y**, is not well-defined (such a "transition" does not exist). The reason is that this "transition" describes *only* operations of the CPU, where the CPU sets **RESET** and then "waits" until **RESET** is cleared, which never happens. Hence, the while loop does not terminate and there is no resulting state.

Hence, a state **open_check_reset** must be defined with the meaning that **open()** is currently executed and the next operation is to check the while guard:

```
open()
//open_perform_reset
    RESET := 0b1_1;
//open_check_reset
    while (RESET = 0b1_1)
        ;
//open_clear_x_and_y
    x := 0;
    y := 0
//open_idle
```

With this additional state, the model can describe the behavior of the CPU first setting **RESET**, and the transmitter to clear **RESET** before/after each time the CPU has checked whether **RESET** = 1.

Do you think the state **open_clear_x_and_y** is needed if x and y are variables only used by the device driver? **Why (not)?**

If **open_state** is a variable specifying at which control point/state the CPU is at/in when executing **open()**, then the modifications of **open_state** that reflect the control flow of the pseudocode of **open()**, can be described in a "table" of if-then-else statements as follows (excluding when **open_state** is changed from **open_idle** to **open_perform_reset** which describes when **open()** is invoked):

```
if open_state = open_perform_reset then
    open_state := open_check_reset
```

```
else if open_state = open_check_reset & RESET = 0b1_1 then
    open_state := open_check_reset
else if open_state = open_check_reset & RESET $\neq$ 0b1_1 then
    open_state := open_clear_x_and_y
else if open_state = open_clear_x_and_y then
    open_state := open_idle
```

To get to the point. In this second step, the task is to:

- **Identify all relevant control points of your design of open(), transmit() and stop(). Motivate why the chosen transition granularity makes sense (the set of operations described by each transition).**
- Use descriptive state names (perhaps a name that starts with the name of the function and that describes what the next operation is).
- **In the pseudocode from Step 1, at each control point identified by a state, write a comment with the name of corresponding state at that control point.**

Be careful to not insert too many states since that complicates the implementation of the model in NuSMV (making the model more complicated and less trustworthy/more buggy), and makes the state space explosion a more likely problem.

4.3 Step 3: Model Transmitter

Model the transmitter in pseudocode, with comments. Use descriptive variable names. **Motivate why the chosen transition granularity makes sense. Explain how the model describes the synchronization between the transmission operations and the tear down operations when a tear down shall be performed (after the transmission of the current message).**

4.4 Step 4: Plan for Combining Models

Decide how the following five aspects shall be included in the **combined model**:

- **How does the model record which function of the device driver that was executed most recently?**
- **When does what perform an operation?** When shall what part of the model make a transition? Shall it be the device driver or the transmitter, or both? Shall it be **open()**, **transmit()** or **stop()**?

Shall it be reset, transmission or tear down operations? **The answers to these questions are probably best given in terms of a table (of if-then-else statements) specifying when each part of the model makes a transition.** When answering these questions, consider your answer to the question in the previous bullet.

- **What are the erroneous configurations of the transmitter? (Read, in detail, Section 2, Transmitter.)** How should the model of the transmitter react to an erroneous configuration? **Write a list describing all erroneous configurations, how they are detected in the model, and how the model handles such cases.** For instance, "The device driver writes non-zero to **TRANSMIT** during initialization", is detected when **variable1 = value1 & variable2 $\leq$ value2 & ...**, in which case the model performs the assignment **variable3 := value3**.

- **Operating systems contain bugs.** How are the arbitrary arguments to **transmit(address, length)** modeled?

4.5 Step 5: Preparations for Implementing the Combined Model in NuSMV

Specify the variables to be used in the implementation of the combined model in NuSMV, their initial values, and how and when the variables are modified:

- For each variable in the pseudocode models of the device driver and the transmitter, write one "table" of if-then-else statements specifying **how** and **when** that variable is modified (similar to what is done for the **open_state** variable in the example discussing states that represent control points for **open()** in Section 4.2, Step 2).

- **Considering your answers to the questions in Section 4.4, Step 4, 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 one "table" specifying **how** and **when** that variable is modified.

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

4.6 Step 6: Implement and Document the Combined Model in NuSMV

Use the tables you wrote in the previous step, to implement a model in NuSMV that describes the operation of and the interaction between the device driver and the transmitter. **Document your NuSMV code with comments describing**:

- How the modules are connected and why the chosen structure makes sense.
- What behavior each module describes.
- What each variable is used for.

4.7 Step 7: Checking Correctness of Implemented Model

Verify some properties of your model to make sure the model describes the expected behavior of the device driver and the transmitter and their interaction. **The following properties must be proved by 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 reset, transmission and tear down operations of the transmitter.
- **open()**, **transmit()**, **stop()** cannot be executed simultaneously.
- If the transmitter performs a reset, 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 around 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) doing a certain thing. **Describe the CTL formulas in natural language and motivate why they are good for checking the correctness of the model.**

4.8 Step 8: 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 state something about the data structure(s) of the device driver, and the other property shall state something about the buffer descriptor queue of the transmitter. **Formalize the two properties as CTL formulas and motivate why the CTL formulas are correct.** Check the two properties with NuSMV (NuSMV does not need to prove them, just check them). **If the properties are not proved, what is wrong?**

4.9 Step 9: Verification of Synchronization and No Misqueue Conditions

Model the system accurately in the sense that the model describes the parallel execution of the device driver and the transmitter. **Verify in NuSMV that all writes to the registers of the transmitter are written in a synchronized way.** That is, the device driver and the transmitter never writes to a register simultaneously (**RESET**, **TRANSMIT**, **TEARDOWN**, and the same field of the same buffer descriptor; e.g. the device driver and the transmitter never writes **BD_OWN[2]** in the same transition).

Describe both formally (perhaps by means of if-then-else statements) and informally (in natural language):

- **How the scheduling is modeled (when what part is making an operation; see the second bullet in Section 4.4, Step 4).**
- **The principle of how simultaneous writes are detected.**
- **The CTL formula(s) for checking that all writes to the transmitter registers are performed in a synchronized way. Motivate the correctness of the CTL formulas.**

Verify in NuSMV that no misqueue condition can occur. Specify the CTL formulas and motivate their correctness

5 Grading

The grades are given as follows:

E Do steps 1-8 (Sections 4.1 through 4.8) and write a report with answers to all tasks given in those steps, including pseudocode, tables, design decisions/motivations/explanations and CTL formulas. The NuSMV code shall be submitted through Canvas.

D As E with a well-written 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 in addition do step 9, Section 4.9, and include the corresponding answers in the report.