Model Checking for State Machines with mbeddr and NuSMV

1 Abstract

State machines are a powerful tool for modelling software. Particularly in the field of embedded software development where parts of a system can be abstracted as state machine.

Temporal logic languages can be used to formulate desired behaviour of a state machine. NuSMV allows to automatically proof whether a state machine complies with properties given as temporal logic formulas.

mbedder is an integrated development environment for the C programming language. It enhances C with a special syntax for state machines. Furthermore, it can automatically translate the code into the input language for NuSMV. Thus, it is possible to make use of state-of-the-art mathematical proofing technologies without the need of error prone and time consuming manual translation.

This paper gives an introduction to the subject of model checking state machines and how it can be done with mbeddr. It starts with an explanation of temporal logic languages. Afterwards, some features of mbeddr regarding state machines and their verification are discussed, followed by a short description of how NuSMV works.

Author:	Christoph Rosenberger
Supervising Tutor:	Peter Sommerlad
Lecture:	Seminar Program Analysis and Transformation
Term:	Spring 2013
School:	HSR, Hochschule für Technik Rapperswil

2 Introduction

Model checking

In the words of Cavada et al.: „The main purpose of a model checker is to verify that a model satisfies a set of desired properties specified by the user." [1]

mbeddr

As Ratiu et al. state in their paper "Language Engineering as an Enabler for Incrementally Defined Formal Analyses" [2], the semantic gap between general purpose programming languages and input languages for formal verification tools is too big. This prevents developers from using these verification tools. Their approach is the use of language engineering techniques. More abstract domain specific languages should be developed and used. To make automatic verification feasible, they should be limited to an automatically analysable subset of the general purpose language. All this should be possible in one tool to reduce time consuming and error prone processes and manual transformation.

mbeddr [3] is an IDE built on top of the JetBrains Meta-Programming System [4] as well as a set of domain specific language fragments which are integrated into the C language. Some main features are:

- A C extension which allows a very convenient and good readable implementation of state machines. Via a translation for NuSMV the state machines can be analysed and some behaviour can be proofed mathematically.
- Another C extension for decision tables making the code more readable than nested if statements. This decision tables can be checked for completeness and consistency. [5]
- Requirements traces can be used to annotate program elements to link them with the requirements.
- One can build own extensions to the C language.

An overview over the mbeddr design can be seen in Figure 1.

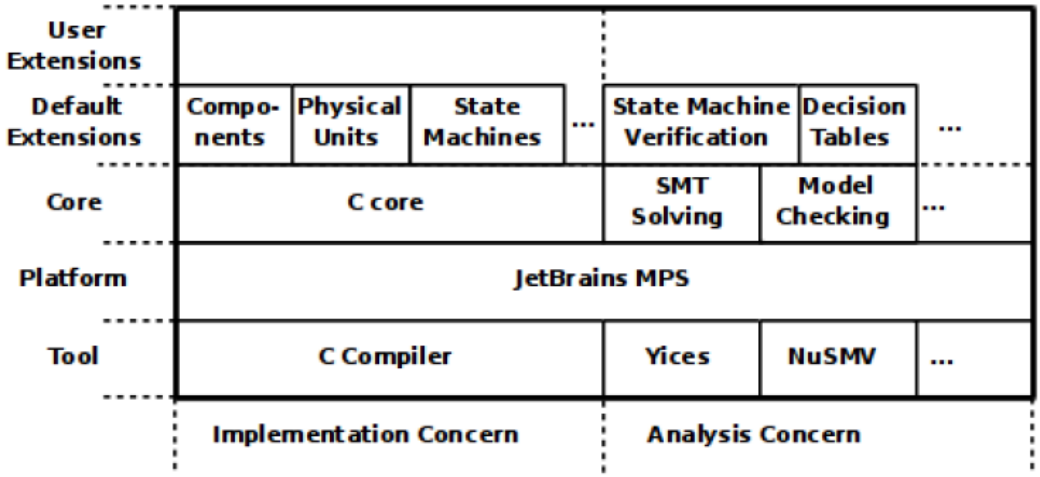

Figure 1 mbeddr at a glance [6]

NuSMV

NuSMV [7] is used by mbeddr to analyse state machines. Its developers Cimatti et al. describe it as follows:
"NuSMV is a symbolic model checker originated from the reengineering, reimplementation and extension of SMV, the original BDD-based model checker developed at CMU. The NuSMV project aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer projects: it is a well-structured, open, flexible and documented platform for model checking, and is robust and close to industrial systems standards." [8]

Algorithms

NuSMV uses the algorithm presented in [9] as the basis for fair CTL model checking and the algorithm presented in [10] is used to support LTL model checking [11].

In [10] E. Clarke et al. "show how LTL model checking can be reduced to CTL model checking with fairness constraints."

In [9] J. R. Burch et al. „describe a general method that represents the state space symbolically instead of explicitly." The relations and formulas are represented as Binary decision Diagrams BDDs. By representing the state space symbolically instead of explicitly, they found an effective technique for combatting the state explosion problem. "Often systems with a large number of components have a regular structure that would suggest a corresponding regularity in the state graph. Consequently, it may be possible to find more sophisticated representations of the state space that exploit this regularity in a way that a simple table of states cannot. One good candidate for such a symbolic representation is the binary decision diagram (BDD) (Bryant, 1986), which is widely used in various tools for the design and analysis of digital circuits. BDDs do not prevent a state explosion in all cases, but they allow many practical systems with extremely large state spaces to be verified - systems that would be impossible to handle with explicit state enumeration methods."

A short introduction to BDDs is given in the appendix. A further discussion of the algorithms is beyond the scope of this paper.

3 The Mathematical Basis

Model checking is about proofing. Therefore, a minimal mathematical background is needed to be able to handle the subject. This chapter is limited to the languages needed to formulate the desired properties of a model. This chapter is an aggregation of a lot of sources: [1] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22].

3.1 Temporal Logic

Predicate logic

Predicate logic allows statements like "All lights are turned on" or "The light is dimmed down or the light is turned off". These are general statements about a system without any reference to time. The well-known operators are [22]:

- existential quantifier ∃
- universal quantifier ∀
- negation ¬
- conjunction ∧
- disjunction ∨
- implication →
- biconditional ↔

Temporal logic

Temporal logic is an extension to the predicate logic. It allows to refer to a system in different states at different times. One could make statements like "The light is turned *on* until the light switch is toggled".

3.2 Linear Temporal Logic LTL

With linear temporal logic LTL properties of a system can be described. In LTL a system is understood as a linear sequence of states the system can be in. Different system runs produce a different sequence of occupied states; in LTL a statement always refers to all possible sequences.

Operators

LTL provides the following operators:

- **X**: Refers to the **next** state. E.g., X 'the light is off' states that in the state following to the actual state, the light will be off.
- **G**: Implies a **global** statement. E.g., G 'the light is off' denote that the light will be *off* in all upcoming states.
- **F**: Tells that a statement will **finally** be true. It doesn't tell when this will happen. E.g., F 'the light bulb is broken' tells that somewhere in the future the light bulb will be broken.
- **U**: **Until** is a binary operator stating that a statement holds at least until another statement holds. E.g., 'the light doesn't shine' U 'the light switch is turned on'. In other words: It is sure that the light doesn't shine until the light switch is turned on. After the light switch is turned on, the light can either shine or, if the light bulb is broken, it will not shine.

- **R**: The **Release** operator states that a statement holds until and including a state in which another statement will hold. Under the assumption that while mounting a light bulb, the light switch is turned off, the following example could be made: 'the light doesn't shine' R 'the light bulb is mounted'. This means that there isn't light until the light bulb is mounted. In the first state after the mounting there isn't any light, because there is no power. This operator also supports the possibility that the second state never occurs. E.g., the light bulb will never be mounted and thus the light will never shine.

  The essential difference between p U q and p R q is at the transition between p and q: U tells, that p holds until but not necessarily including the first state where q holds. R requires that there is at least one state where both p and q hold.

3.3 Computation Tree Logic CTL

CTL is a branching time logic. The future states of a system are not determined yet and thus different evolutions are possible. This can be thought off as a branching tree where each branch stands for an alternative state transition. In CTL you can make statements that indicate a statement for at least one further development or for all further developments.

Quantifiers over paths

Two additional quantifiers exist:

- **A**: Is used for a statement that holds in **all** paths.
- **E**: States that at least one path **exists** with a given property.

There is a difference between ∀ and A, respectively between ∃ and E. The quantifiers of the predicate logic refer to the objects at one time. The path quantifiers of the CTL refer to states of the following paths. E.g., ∃ 'a light that doesn't shine' states, that there is at least one light that does not shine at the moment. There is no connotation to the future or the past. EG 'the light doesn't shine' states that there is at least one possible evolution where the light will never shine. So this is only a statement of the further development of the system.

Path-specific quantifiers

To build a statement the same operators as in LTL are used. (Except the R operator. But this is not a restriction: For two expressions p and q, R can be simulated with "¬ (¬p U ¬q)").

Legal statements

In CTL a legal statement is always composed of a quantifier over paths and a path-specific quantifier. Legal examples are:

- AX 'the light doesn't shine': All possible next steps lead to a state where the light doesn't shine.
- EX 'the light doesn't shine': At least one next step leads to a state where the light doesn't shine.

- EG ‘the light doesn’t shine’: There is at least one possible evolution where the light will never shine. (Could be true if the light bulb is broken.)
- AG ‘the light doesn’t shine’ U ‘the light switch is turned on’: In all possible evolutions the light will not shine until the light switch is turned on.
- EF ‘the light bulb is broken’: At least one immediate transition exists after which the light bulb will be broken.

3.4 CTL*

CTL*

CTL and LTL are not identical although the expression power has some common possibilities. Neither is CTL a subset of LTL nor is it a superset. Thus, CTL* is defined which is a combination of CTL and LTL.

Difference between CTL and LTL

The most obvious difference between CTL and LTL originates in the different rules using the A and E quantifiers: While in CTL every expression must have one, LTL doesn’t even know these symbols. LTL formulas always refer to all possible system runs. But the approach of translating an LTL formula to a CTL formula by adding an A quantifier in front of each clause will fail. This will be shown later on. The actual difference between these two languages lies in the semantic and the way a system run is thought of.

A LTL formula makes a statement about all possible paths the system could run through. One single path is an infinite linear sequence of states the system could run through. Figure 2 shows a visualization of this mind model.

A CTL formula makes a statement about all possible states a system could be in. From the most states the system has different branches of possible evolutions. This mind model is visualized in Figure 3. While looking at one single state in the system run, a CTL formula makes a statement about all possible further developments. While the A quantifier requires a property to hold in all branches, the E quantifier only requires the property to hold in one of them.

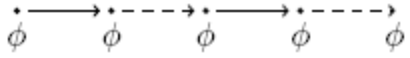

Figure 2 Visualization of a system run in LTL [17]

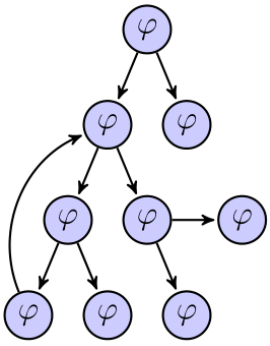

Figure 3 Visualization of a system run in CTL [20]

Example statement only expressible in LTL

The following example refers to the state diagram shown in Figure 4. The system has three states: *a*, *b* and *c*. From *a* it can either loop back to *a* or change over to *b*. From *b* it can only go further to *c*. And from *c* it can only loop back to *c*. Basically there are two distinct system evolutions. Either the system loops forever in state *a*. Or it eventually changes over to *b* and afterwards to *c*, where it will loop forever.

The LTL formula
FG("the system is in state *a*" ∨ "the system is in state *c*")
states that in all possible system runs, the system will eventually reach a point after which it loops forever in state *a* or in state *c*. In other words (adapted from [12]): Every path has a finite prefix after which the property "the system is in state *a*" ∨ "the system is in state *c*" is always true.

This statement can't be translated to CTL. The approach of adding an A quantifier in front of each clause fails. The formula
AF(AG("the system is in state *a*" ∨ "the system is in state *c*"))
would be too strict. It states that eventually it will hold that in all further branches the system can't reach the state *b* anymore. This statement is false. In the system run which loops in *a* forever, there is always a branch where the system could change over to *b*.

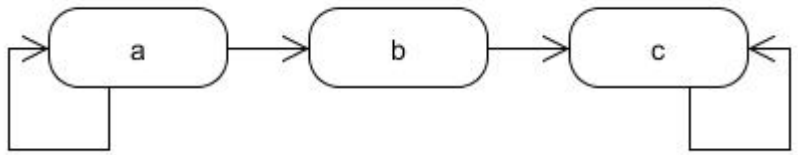

Figure 4 Example state diagram

Example statement only expressible in CTL

This example also refers to the state diagram shown in Figure 4.
The CTL formula
AG("the system is in state *a*" → (EF "the system is in state *c*"))
states that "On all branches it holds globally that: After being in state *a*, there is a possible further evolution which eventually reaches state *c*."

One naïve approach to translate this formula to LTL would be to omit the path quantifiers A and E. This would lead to the formula
G("the system is in state *a*" → (F "the system is in state *c*")).
This statement is false. In the system run where the system loops in state *a* forever, the system will never reach state *c*. LTL can only make statements which have to hold in every system run. Neither is it possible to make a statement which only holds in some system runs, nor is it possible to make a statement about possible alternative developments.

3.5 CTL* in mbeddr

The correct usage of CTL* expressions can be quite cumbersome. To facilitate writing correct checks, mbeddr provides a set of specification patterns. This patterns are based on CTL* expressions. Their description can be found on a website that is a "home of an online repository for information about property specification for finite-state verification" [23]. This is further discussed in chapter 4.5.2.

4 State Machines in mbeddr

In this chapter the handling of state machines in mbeddr is discussed on the basis of the LightSwitch example. First different representations of the LightSwitch are discussed, afterwards the possibilities to ensure the correctness of the implementation are shown: Unit testing and symbolic model checking. And last some further state machine modelling options provided by mbeddr are demonstrated.

LightSwitch

For this paper a simple model of a light switch was developed. It will be used throughout this paper. There are three states:

- *off*: In the State *off* the light is turned off. The LightSwitch doesn't react to dimUp and dimDown events. On toggle it will go to the *on* state except the SERVICESTATE_THRESHOLD is reached, then it goes into the *servicestate* state.
- *on*: In the State *on* the light is on. The brightness can be regulated with dimUp and dimDown. If the light is dimmed to dark, the LightSwitch turns off. If the light is too bright, it can't be turned any brighter.
- *servicestate*: If the LightSwitch is in the servicestate it doesn't react to any further inputs.

4.1 Graphical Representation of a State Model

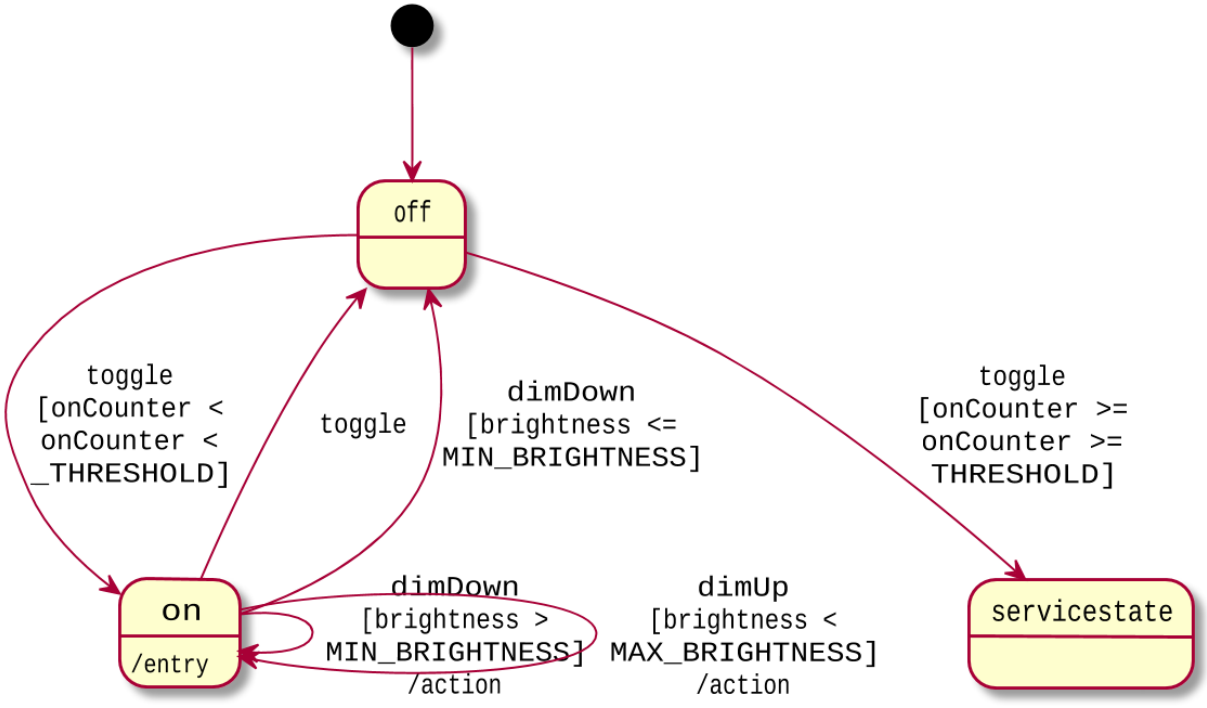

Figure 5 Graphical representation of the LightSwitch State Model

Graphical representation

mbeddr is able to generate graphical representations of a state machine. The guards for the transitions can be shown as in the example above, or there is the choice to not show them. Unfortunately, only very short variable names can be handled correctly. Only a fragment will be shown if the variable name is too long. (The "_THRESHOLD" variable is actually called SERVICESTATE_THRESHOLD).

4.2 State Model DSL in mbeddr

In mbeddr there is a C extension for state machines. This allows to specify a state machine in a very convenient and readable way. Some of the language is explained in this chapter. Listing 1 shows the implementation of the LightSwitch example.

```
1.  #define SERVICESTATE_THRESHOLD = 255;
2.  #define MIN_BRIGHTNESS = 1;
3.  #define MAX_BRIGHTNESS = 8;
4.  #define BRIGHTNESS_START = 5;
5.
6.  [verifiable]
7.  statemachine LightSwitch initial = off {
8.    var bounded_int[0..SERVICESTATE_THRESHOLD] onCounter = 0
9.    readable var bounded_int[MIN_BRIGHTNESS..MAX_BRIGHTNESS] brightness = BRIGHTNESS_START
10.
11.   in toggle() <no binding>
12.   in dimUp() <no binding>
13.   in dimDown() <no binding>
14.
15.   state off {
16.     on toggle [onCounter < SERVICESTATE_THRESHOLD] -> on
17.     on toggle [onCounter >= SERVICESTATE_THRESHOLD] -> servicestate
18.   } state off
19.
20.   state on {
21.     entry { ++onCounter; }
22.     on toggle [ ] -> off
23.     on dimDown [brightness <= MIN_BRIGHTNESS] -> off
24.     on dimDown [brightness > MIN_BRIGHTNESS] -> on { --brightness; }
25.     on dimUp [brightness < MAX_BRIGHTNESS] -> on { ++brightness; }
26.   } state on
27.
28.   state servicestate {
29.
30.   } state servicestate
31. }
```

Listing 1 LightSwitch state machine

[verifiable] (line 6)

The keyword [verifiable] declares this state machine to be verifiable with the integrated model checker. Only a subset of the possibilities provided by the state machine construct can be used. Some restrictions are:

- "Data types: all local variables, arguments of input or output events should have as type one of the following types: enumeration, boolean, int8, int16, int32 and bounded_int. In particular, we do not support floats or structs." [24] (Page 270)

- "No access to global state: accessing global variables or calling global functions is not allowed. Mapping out-events to arbitrary functions is legal, though." [24] (Page 270)
- "Single assignment actions: in each effective action executed as a consequence of an event (i.e., exit action of the current state + transition action + entry action of the target state), a variable can be assigned only once." [24] (Page 270)
- No composite states are allowed. (For an example of a composite state see chapter 4.6)

bounded_int (line 8)

In the LightSwitch example it would be possible to model all eight dim-levels as separate states. Variables are a convenient way to make scalable models. For the model checker the way of modelling these states doesn't change the fact, that all states have to be checked. So limiting the possible values of a variable has a big impact on the time needed for the verification.

Furthermore, by declaring the exact domain of definition, the checking algorithm has much more information to verify the model.

readable var (line 9)

With the readable keyword a variable can be read from outside the state machine. Otherwise the value of variable is encapsulated inside the state machine.

in (line 11)

The events that the state machine can receive are declared with the in keyword.

state (line 15)

The state keyword marks the beginning of a state definition.

entry (line 21)

The commands that will be executed when entering a state are listed after the entry keyword.

on (line 16)

The desired transitions after an event are stated after the on keyword. Guards to restrict the transition can be specified in brackets.

4.3 Translation to C-Code

This chapter discusses the mapping from the mbeddr representation of the state machine to C-Code. For better readability some function and variable names were modified. The complete and unmodified Code can be found in the appendix. The mapping of the state machines states and input as well as the type for keeping the current state are shown in Listing 2 and Listing 3. Listing 4 and Listing 5 show the code for initializing and running the state machine.

Data structure

The current state and the values of the variables are administrated in a struct as seen in Listing 2.

```
1.  struct sm_data_LightSwitch {
2.    sm_states_LightSwitch __currentState;
3.    uint8_t onCounter;
4.    uint8_t brightness;
5.  };
```

Listing 2 struct LightSwitch

Events and states

Enumerations are generated to label the possible events and the states. This enumerations are shown in Listing 3.

```
1.  typedef enum sm_events_LightSwitch{
2.    event_toggle,
3.    event_dimUp,
4.    event_dimDown
5.  } sm_events_LightSwitch;
6.
7.  typedef enum sm_states_LightSwitch{
8.    off__state,
9.    on__state,
10.   servicestate__state
11. } sm_states_LightSwitch;
```

Listing 3 Enumerations for events and states

Initialisation

Listing 4 shows how the init method sets the initial state and values.

```
1.  void sm_init_LightSwitch(
                          struct sm_data_LightSwitch* instance)
2.  {
3.    instance->__currentState = off__state;
4.    instance->onCounter = 0;
5.    instance->brightness = MAIN_BRIGHTNESS_START;
6.  }
```

Listing 4 LightSwitch initialisation

Transition logic

The sm_execute_LightSwitch method shown in Listing 5 handles all the logic of the state machine. The transition which has to be made is determined by two nested switch statements for the current state and the event, followed by if-statements to check the guard conditions. In the following block follows the actual transition logic: The new state is assigned and the entry actions for the new state are executed.

```
1.  void sm_execute_LightSwitch(
                          struct sm_data_LightSwitch* instance,
                          sm_events_LightSwitch event,
                          void** arguments)
2.  {
3.    switch (instance->__currentState)
4.    {
5.      case off__state: {
6.        switch (event)
7.        {
8.          case event_toggle: {
9.            if (instance->onCounter < SERVICESTATE_THRESHOLD)
10.           {
11.             // switch state
12.             instance->__currentState = on__state;
13.             // entry actions
14.             ++instance->onCounter;
15.             return ;
16.           }
17.           if (instance->onCounter >= SERVICESTATE_THRESHOLD)
18.           {
19.             // removed
20.           }
21.           break;
22.         }
23.       }
24.       break;
25.     }
26.     case on__state:
27.     {
28.       // removed
29.     }
30.     case servicestate__state:
31.     {
32.       // removed
33.     }
34.   }
35. }
```

Listing 5 Transition logic

Usage in code

While the generated code can be read and understood quite easy, the use of the somewhat long function, typedef and enum names could be quite cumbersome. To facilitate the use of state machines, mbeddr provides specialised syntax. Thereby, all the implementation details remain behind the scene. The handling of a state machine is discussed in the next chapter.

4.4 Unit Testing a State Model

mbeddr extends the C-language with unit tests. This facilitates writing unit tests. For this chapter two unit tests were written to show the special syntax provided for unit tests as well as to show how to address a state machine in C code.

Code listing

```
1.  exported int32 main(int32 argc, string[] argv) {
2.    return test [ testToggle
3.                  testDimDown ];
4.  } main (function)
5.
6.  exported test case testToggle {
7.    LightSwitch lightswitch;
8.    sminit(lightswitch);
9.    test statemachine lightswitch {
10.     toggle ▯ on
11.     toggle ▯ off
12.   }
13. } testToggle(test case)
14.
15. exported test case testDimDown {
16.   LightSwitch lightswitch;
17.   sminit(lightswitch);
18.   smtrigger(lightswitch, toggle);
19.   assert(0) smIsInState(lightswitch, on);
20.   assert(1) lightswitch.brightness == BRIGHTNESS_START;
21.   for (i ++ in [1..BRIGHTNESS_START - MIN_BRIGHTNESS + 1[) {
22.     smtrigger(lightswitch, dimDown);
23.     assert(2) lightswitch.brightness == BRIGHTNESS_START - i;
24.   } for
25.   smtrigger(lightswitch, dimDown);
26.   assert(3) smIsInState(lightswitch, off);
27. } testDimDown(test case)
```

Listing 6 Unit tests for LightSwitch

testToggle (line 6)

testToggle sends a toggle event to the lightSwitch and checks whether the reached state is on. Then it sends another toggle event and checks whether the reached state is off.

testDimDown (line 15)

The second test case is a bit large and tests different behaviours in the context of dimDown. At line 20 the initial value of lightSwitch.brightness is checked. The loop starting at line 21 dims the lightSwitch several times so that the minimal brightness is reached. (Note that a while loop checking whether lightSwitch.brightness != MIN_BRIGHTNESS would not be a good choice: You wouldn't test how many times you have to dimDown until the minimal brightness is reached.) At line 25 a final dimDown event is sent and at line 26 it is asserted that this leads to the *off* state.

General unit testing syntax

mbeddr provides several syntax elements to support unit tests.

- At line 2 and 3 the test cases are called with the test command. The test cases well-arranged in big brackets.
- For declaring a test case the first level language element "test case" (line 6) is provided. This brands a function clearly as a unit test function.
- The assert statement (line 19) is to formulate the checks. The number in the brackets can be used to refer to a assert statement within a test. If one check fails it eases to find the failing test.

Unit testing state machines

A special syntax element for unit testing a state machine exists: test state machine (line 9 – 12). With this statement a sequence of events can be listed together with the expected following states. More complex sequences or tests which access variables of the

state machine can be written with conventional code as in testDimDown.

Handling a state machine

mbeddr provides three functions to handle a state machine. This functions are not specific to unit testing but can also be used in normal C code.

- sminit (line 17): Initialises the state machine in the initial state and the default values for the variables.
- smtrigger (line 18): Sends an event to the state machine.
- smIsInState (line 19): Checks weather the state machine is in the desired state.

The readable variables of the state machine can be accessed like a member of an object in most object oriented languages (line 20).

4.5 Symbolic Model Checking

Besides the many strong points, unit tests have one intrinsic down side: The only check particular cases. This is the strength of symbolic model checking: It provides a mathematical proof that a property holds in all possible program runs.

mbeddr not only allows to specify specific checks for a state machine, but also brings default checks suitable for all state machines.

4.5.1 Default Checks

mbeddr provides four different default checks which don't have to be implemented by the user [24]. Figure 6 shows the check results for the LightSwitch. It can be seen that the variable onCounter can possibly be out of range after 257 events. Figure 7 shows the last few events of the trace triggering the bug.

Property	Status	Trace Size
State 'off' is reachable	SUCCESS	
State 'on' is reachable	SUCCESS	
State 'servicestate' is reachable	SUCCESS	
Variable 'onCounter' is possibly out of range	FAIL	257
Variable 'brightness' is always between its defined bounds	SUCCESS	
State 'off' has deterministic transitions	SUCCESS	
State 'on' has deterministic transitions	SUCCESS	
State 'servicestate' has deterministic transitions	SUCCESS	
Transition 0 of state 'off' is not dead	SUCCESS	
Transition 1 of state 'off' is not dead	SUCCESS	
Transition 0 of state 'on' is not dead	SUCCESS	
Transition 1 of state 'on' is not dead	SUCCESS	
Transition 2 of state 'on' is not dead	SUCCESS	
Transition 3 of state 'on' is not dead	SUCCESS	

Figure 6 Result of default checks

Unreachable States

It is checked for all states if they can be reached. If a state is unreachable under any circumstances, it would be dead code and could be removed. The question is whether the state is not needed or if there is a bug in the state model preventing the state to be reached.

Variable bounds

For the variables with bounds it is checked whether its bounds hold or if there is a way to bring its values out of its domain. In the LightSwitch example the bound for the variable onCounter is violated. The reason for this will be discussed further down.

Nondeterministic Transitions

All transitions should be deterministic. I.e., if a state can have several successors for the same event, the guards should unambiguously define which transition is to take.

If this rule is not obeyed, the state machine is still deterministic in mbeddr, because it implements guards with consecutive if statements. The first transition with a fulfilled guard is taken.

Not-fireable Transitions

It is analysed whether all transitions can be fired or not. A transition that can't ever be fired is dead code and the user should check if it can be removed or if there is a bug preventing the transition from being fired.

Counter example

If a check fails, mbeddr provides an example how a condition can be violated. The trace shows as one block the state the system under test is in, the values of all variables and the 'in'-event that will fire next. E.g., on the first four lines of Figure 7 one can see that the LightSwitch is in state *on*, the onCounter has the value 251, brightness is 1 and the next 'in'-event will be dimUp.

State on	
in_event: dimUp	dimUp()
onCounter	251
brightness	1
State on	
in_event: dimDown	dimDown()
onCounter	252
brightness	2
State on	
in_event: dimUp	dimUp()
onCounter	253
brightness	1
State on	
in_event: dimDown	dimDown()
onCounter	254
brightness	2
State on	
in_event: dimUp	dimUp()
onCounter	255
brightness	1
State on	
in_event: toggle	toggle()
onCounter	256
brightness	2

Figure 7 Counter example for range check of variable 'onCounter'

As seen in Figure 6 the onCounter of the LightSwitch can get out of its defined range. The trace size for the counter example is 257 events long. For the understanding why the onCounter can get out of range the last few iterations are enough. One can see that each time dimUp or dimDown is fired, the onCounter is incremented. But the check whether the onCounter has reached the SERVICESTATE_THRESHOLD is implemented at the exit of the *off* state. The semantic of the variable name 'onCounter' indicates that it should only be increased when turning the light on. So this

is clearly a bug. An improved LightSwitch where this bug is eliminated can be seen in chapter 4.6.

4.5.2 Manual Analyses

Further analyses can be formulated manually. The available patterns to formulate the desired properties are listed in Figure 8. The specification patterns mbeddr relies on [24], can be found on a website that is a "home of an online repository for information about property specification for finite-state verification" [23]. CTL as well as LTL based expressions are supported.

In this chapter two patterns are exemplified. Listing 7 shows how this two patterns are applied to check the LightSwitch.

```
P is false After Q                  (BinaryVerificationPattern in c.m.a.nusmv.statemachine)
P is false After Q Until R          (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
P is false Before R                 (BinaryVerificationPattern in c.m.a.nusmv.statemachine)
P is false Between Q and R          (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
P is false Globally                  (UnaryVerificationPattern in c.m.a.nusmv.statemachine)
P is true After Q                   (BinaryVerificationPattern in c.m.a.nusmv.statemachine)
P is true After Q Until R           (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
P is true Before R                  (BinaryVerificationPattern in c.m.a.nusmv.statemachine)
P is true Between Q and R           (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
P is true Globally                   (UnaryVerificationPattern in c.m.a.nusmv.statemachine)
S Responds to P After Q Until R (QuaternaryVerificationPattern in c.m.a.nusmv.statemachine)
S Responds to P After R             (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
S Responds to P Before R            (TernaryVerificationPattern in c.m.a.nusmv.statemachine)
S Responds to P Between Q and R  (QuaternaryVerificationPattern in c.m.a.nusmv.statemachine)
S Responds to P Globally             (BinaryVerificationPattern in c.m.a.nusmv.statemachine)
always eventually reachable      (AbstractVerificationCondition in c.m.a.nusmv.statemachine)
```

Figure 8 Available expressions for defining checks

always eventually reachable	This pattern allows to check, if a state remains reachable in any cases or if a sequence of events exists after which a state isn't reachable anymore. E.g., the *on* state is not live, so the check 'always eventually reachable *on*' will fail. Once fallen into the *servicestate*, the *on* state can't be reached any more. On the other hand, the *servicestate* is live. From every possible state the state machine can be in, the *servicestate* remains reachable.
P is false Before R	This pattern checks whether a condition P can be true before R happens. Line 4 in Listing 7 says that state *on* can't be reached without ever toggling the light switch.

```
1. verification conditions
2.   always eventually reachable on
3.   always eventually reachable servicestate
4.   P is false Before R   P: on R: toggle
```

Listing 7 Example checks for LightSwitch

4.6 Further Possibilities

This chapter shows some further possibilities that one has for modelling state machines in mbeddr. Therefore, the LightSwitch example is enhanced to a second version: The bug found in chapter 4.5.1 is corrected by using a composite state, the dim functions can now handle a parameter and C functions are called which could address a device driver. The graphical representation of the LightSwicht2 can be seen in Figure 9 and the actual implementation is listed in Listing 8.

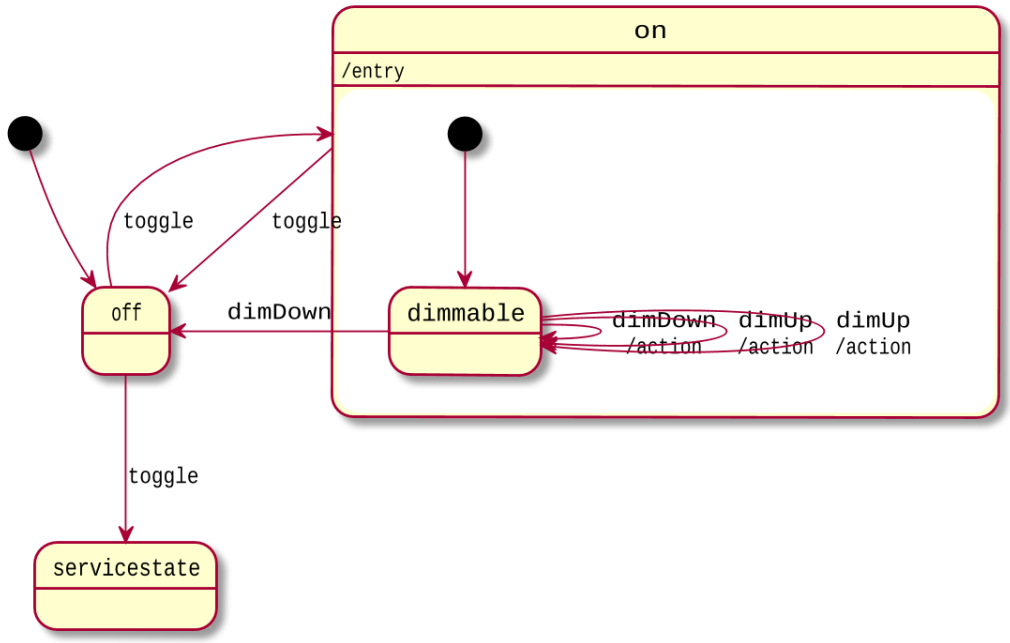

Figure 9 Graphical representation of the LightSwitch2 State Model

composite state (line 18)

With a composite state a hierarchical state machine can be built. A composite state has sub states. In the example the *on* state has the sub state *dimmable* which is used to handle the dimUp and dimDown events without leaving and re-entering the *on* state. Therefore, the bug found in the original LightSwitch is mended. This construct unfortunately prevents the state machine from being model checked.

events with parameters (line 6 & 24)

Events can have parameters. LightSwitch2 can change the brightness several steps with one call dimUp or dimDown. The value of the parameter can be used in the guard condition as well as in the action. This can be seen on line 24 where the delta first is used to determine whether this transition should fire and if so how to calculate the new brightness.

out events (line 9, 22 & 35)

When developing for embedded systems, a state machine is usually an abstraction of a real world device. Thus, the state machine doesn't only need to manage the state and the transitions but also has to provoke the calls to the right functions of the device drivers.

On line 9 and 10 the out events are declared and bound to C functions outside of the state machine. The dependency of the state machine to other code is limited to one place. And furthermore, this still allows to model check the state machine.

On line 22 the send statement is used to call the C function. And of course there has to be a corresponding function that could make a call to a device driver, as seen on line 35.

```
1.  statemachine LightSwitch2 initial = off {
2.    var bounded_int[0..SERVICESTATE_THRESHOLD] onCounter = 0
3.    readable var bounded_int[MIN_BRIGHTNESS..MAX_BRIGHTNESS] brightness = BRIGHTNESS_START
4.
5.    in toggle() <no binding>
6.    in dimUp(bounded_int[1..MAX_BRIGHTNESS] delta) <no binding>
7.    in dimDown(bounded_int[1..MAX_BRIGHTNESS] delta) <no binding>
8.
9.    out turnOn(bounded_int[MIN_BRIGHTNESS..MAX_BRIGHTNESS] brightness) => letThereBeLight
10.   out turnOff() => letThereBeDarkness
11.
12.   state off {
13.     entry { send turnOff(); }
14.     on toggle [onCounter < SERVICESTATE_THRESHOLD] -> on
15.     on toggle [onCounter >= SERVICESTATE_THRESHOLD] -> servicestate
16.   } state off
17.
18.   composite state on initial = dimmable {
19.     entry { ++onCounter; }
20.     on toggle [ ] -> off
21.     state dimmable (on.dimmable) {
22.       entry { send turnOn(brightness); }
23.       on dimDown [brightness - delta < MIN_BRIGHTNESS] -> off
24.       on dimDown [brightness - delta >= MIN_BRIGHTNESS] -> dimmable { brightness -= delta; }
25.       on dimUp [brightness + delta > MAX_BRIGHTNESS] -> dimmable { brightness = MAX_BRIGHTNESS; }
26.       on dimUp [brightness + delta <= MAX_BRIGHTNESS] -> dimmable { brightness += delta; }
27.     } state dimmable
28.   } state on
29.
30.   state servicestate {
31.     entry { send turnOff(); }
32.   } state servicestate
33. }
34.
35. void letThereBeLight(int8 brightness) {
36.   //  call to device driver
37. } letThereBeLight (function)
38.
39. void letThereBeDarkness() {
40.   //  call to device driver
41. } letThereBeDarkness (function)
```

Listing 8 Code for the LightSwitch2 state machine

5 NuSMV

mbeddr doesn't do the model checking itself but delegates this to the NuSMV model checker. The collaboration works by generating an input file for NuSMV and after the checks parsing the output. This chapter discusses the input to NuSMV that mbeddr generates to analyse the LightSwitch state machine with the checks shown in chapter 4.5. For better readability, the variable names are shorted in this chapter. The complete original code can be found in the appendix.

5.1 LightSwitch Definition

This chapter discusses how the implementation of the actual model is generated by mbeddr. Listing 9 shows the code. As said before, for better readability the variable names are shorted.

Variable declaration (lines 1 – 4)

The state machine variables onCounter and brightness are declared on lines 2 and 3. The range for the NuSMV model is extended by one at the lower bound as well as at the upper bound. This is essential for the range checks.

For managing the current state a variable _current_state is defined at line 4: This is an enumeration with all states of the LightSwitch.

State transitions (lines 7 – 16)

The next section handles the state transitions. On line 7 the initial state is set. The lines 8 to 15 define the transition conditions including the guards. The pattern is to first check the current state, then check the fired event and last check if the guards let us through. The last transition on line 15 doesn't change the current state: It assures that there is always a transition to take. Thus, it takes all the conditions checked above and negates them.

onCounter (lines 18 – 24)

Each variable needs a separate block for keeping track of its value changes. First the initial value for the onCounter is set on line 18. For the actual calculation of the next value pretty much the same pattern is used as for the state transitions: First the current state is checked, then the fired event and last the guards are calculated.

In contrast to the states, the new value for the onCounter must be calculated and the bounds have to be checked before the new value can be assigned. This makes the second part of the lines 20 – 23. First the range of the value has to be assured: Thus, it is checked if the new value violates the lower or the upper bound. If the new value is ok it is assigned to the variable. The onCounter has a declared range that begins at -1, one under the lower bound, and ends at 256, one above the upper bound. These extra values are used if the onCounter violates its bounds. Hence, it will later be easy to formulate the bound checks.

brightness (lines 26 – 31)

For the brightness the same algorithm is used as for the onCounter.

```
1.  VAR
2.    v_onCounter:-1..256;
3.    v_brightness:0..9;
4.    _current_state:{off,on,servicestate};
5.
6.  ASSIGN
7.    init (_current_state) := off;
8.    next (_current_state) := case
9.      (_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255) : on;
10.     (_current_state = off) & (in_toggle = TRUE) & (v_onCounter >= 255) : servicestate;
11.     (_current_state = on) & (in_toggle = TRUE) : off;
12.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness <= 1) : off;
13.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1) : on;
14.     (_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8) : on;
15.     !(((_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255)))
          & !(((_current_state = off) & (in_toggle = TRUE) & (v_onCounter >= 255)))
          & !(((_current_state = on) & (in_toggle = TRUE))) & !(((_current_state = on)
          & (in_dimDown = TRUE) & (v_brightness <= 1))) & !(((_current_state = on)
          & (in_dimDown = TRUE) & (v_brightness > 1))) & !(((_current_state = on)
          & (in_dimUp = TRUE) & (v_brightness < 8))) : _current_state;
16.   esac;
17.
18.   init (v_onCounter) := 0;
19.   next (v_onCounter) := case
20.     (_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255) :
          v_onCounter + 1 < 0 ? -1 : v_onCounter + 1 > 255 ? 256 : v_onCounter + 1;
21.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1) :
          v_onCounter + 1 < 0 ? -1 : v_onCounter + 1 > 255 ? 256 : v_onCounter + 1;
22.     (_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8) :
          v_onCounter + 1 < 0 ? -1 : v_onCounter + 1 > 255 ? 256 : v_onCounter + 1;
23.     !(((_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255)))
          & !(((_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1)))
          & !(((_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8))) :
          v_onCounter < 0 ? -1 : v_onCounter > 255 ? 256 : v_onCounter;
24.   esac;
25.
26.   init (v_brightness) := 5;
27.   next (v_brightness) := case
28.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1) :
          v_brightness - 1 < 1 ? 0 : v_brightness - 1 > 8 ? 9 : v_brightness - 1;
29.     (_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8) :
          v_brightness + 1 < 1 ? 0 : v_brightness + 1 > 8 ? 9 : v_brightness + 1;
30.     !(((_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1)))
          & !(((_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8))) :
          v_brightness < 1 ? 0 : v_brightness > 8 ? 9 : v_brightness;
31.   esac;
```

Listing 9 LightSwitch implementation for NuSMV (generated by mbeddr)

5.2 Automatic Checks

In this chapter the automatically provable assertions are discussed.

Reachability checks

The reachability checks seen in Listing 10 assert that all states are reachable, i.e., there are no dead states. For the reachability check a CTL formula is used. Line 1 says “In all possible system runs it is true in every state that the current state is not the *off* state”. Therefore, if this NuSMV check fails we have the success scenario of the mbeddr reachability check. On line 2 the text is defined which will be used in mbeddr to describe the outcome of the check.

```
1.  SPEC  AG _current_state != off
2.  --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      State 'off' is reachable | State 'off' is unreachable
3.
4.  SPEC  AG _current_state != on
5.  --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      State 'on' is reachable | State 'on' is unreachable
6.
7.  SPEC  AG _current_state != servicestate
8.  --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      State 'servicestate' is reachable | State 'servicestate' is unreachable
```

Listing 10 NuSMV reachability checks (generated by mbeddr)

Range checks

The range checks make sure that variables will stick to their bounds. Line 1 in Listing 11 says "In all possible evolutions and in all states onCounter will be bigger or equal to 0 and smaller or equal to 255". 0 and 255 are de bounds defined for the variable onCounter.

```
1.  SPEC  AG (0 <= v_onCounter & v_onCounter <= 255)
2.  --Variable 'onCounter' is always between its defined bounds
      | Variable 'onCounter' is possibly out of range
3.
4.  SPEC  AG (1 <= v_brightness & v_brightness <= 8)
5.  --Variable 'brightness' is always between its defined bounds
      | Variable 'brightness' is possibly out of range
```

Listing 11 NuSMV range checks (generated by mbeddr)

Determinism checks

The nondeterminism checks assert that all transitions are deterministic, i.e., the state machine doesn't allow two transitions to fire at the same time. Listing 12 shows how the model for the NuSMV state machine is enhanced with an additional variable and its transitions.

The variable on line 2 is an enumeration with four possible values:

- no_nondeterminism states that no nondeterminism was found
- nd_detected_inoff states that in state *off* a nondeterministic transition is found
- nd_detected_inon states that in state *on* a nondeterministic transition is found
- nd_detected_inservicestate state that in the servicestate a nondeterministic transition is found.

At line 5 the transitions for the nondeterminism checks begin with the initialisation of the nondeterminism detector variable. Afterwards, for each state that can react to an event with different transitions it is checked, whether the guards are distinct or not. This is achieved by making checks where the guards are checked in a pairwise conjunction. E.g., the state *off* can react to the event toggle by going into the *on* state if the onCounter is smaller than 255 or by going into the servicestate if the onCounter is 255 or bigger. Thus, these to conditions are put into a conjunction (together with the prerequisite that the state machine actually is in the *off* state and the toggle event is fired). If this conjunction can

be true, a non-deterministic transition would exist (what obviously isn't the case in this scenario).

For three different transitions from one state to one event, six check cases would be formulated, etc. As a conjunction is commutative, this are more checks than actually needed. But it generates the desired output.

This groundwork provided the actual check conditions can be formulated in trivial CTL statements, like on line 14: In all possible evolutions on all states the nondeterminism detector variable never is in the state which indicates a nondeterministic transition.

```
1.  VAR
2.    _nondeterminism_detector:{no_nondeterminism,nd_detected_inoff,
                                nd_detected_inon,nd_detected_inservicestate};
3.
4.  ASSIGN
5.      init (_nondeterminism_detector) := no_nondeterminism;
6.    next (_nondeterminism_detector) := case
7.      (_current_state = off) & (in_toggle = TRUE) & (v_onCounter >= 255)
         & (v_onCounter < 255) : nd_detected_inoff;
8.      (_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255)
         & (v_onCounter >= 255) : nd_detected_inoff;
9.      (_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1)
         & (v_brightness <= 1) : nd_detected_inon;
10.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness <= 1)
         & (v_brightness > 1) : nd_detected_inon;
11.     TRUE : no_nondeterminism;
12.   esac;
13.
14. SPEC  AG _nondeterminism_detector != nd_detected_inoff
15. --State 'off' has deterministic transitions
      | State 'off' contains nondeterministic transitions
16.
17. SPEC  AG _nondeterminism_detector != nd_detected_inon
18. --State 'on' has deterministic transitions
      | State 'on' contains nondeterministic transitions
19.
20. SPEC  AG _nondeterminism_detector != nd_detected_inservicestate
21. --State 'servicestate' has deterministic transitions
      | State 'servicestate' contains nondeterministic transitions
```

Listing 12 NuSMV determinism checks (generated by mbeddr)

Dead transition checks

The dead transition checks, listed in Listing 13, assert that all transitions can be fired. For these checks the model for the NuSMV state machine is enhanced with an additional variable and its transitions.

The variable _dead_transition is an enumeration. Beside an initial value, every value can indicate whether the according transition is dead or not.

The names of the values are a bit misleading: If the variable _dead_transition can have e.g. the value state_off_transition_0_is_dead this indicates that the first transition of the *off* state is not dead.

For every transition in the LightSwitch state machine one according transition for the _dead_transition variable is defined. As in all other checks the first to parts of the conjunction is to check the state-event combination. In the third argument the guard is checked. E.g., on line 7 it is checked whether the transition from the *off* state to the *on* state fired on the toggle event can actually be fired, or if the guard always is false.

The groundwork for this check is done in the transition logic. So the CTL check conditions are quite trivial. It is checked that in all possible evolutions in all states the _dead_transition variable never has another value than dead_trans_init. Therefore, on lines 16 to 32 for every other value of the enumeration a check case is specified which asserts that the _dead_transition variable can't ever have that value.

```
1.  VAR
2.    _dead_transition:{dead_trans_init,state_off_transition_0_is_dead,
                       state_off_transition_1_is_dead,state_on_transition_0_is_dead,
                       state_on_transition_1_is_dead,state_on_transition_2_is_dead,
                       state_on_transition_3_is_dead};
3.
4.  ASSIGN
5.    init (_dead_transition) := dead_trans_init;
6.    next (_dead_transition) := case
7.      (_current_state = off) & (in_toggle = TRUE) & (v_onCounter < 255) :
         state_off_transition_0_is_dead;
8.      (_current_state = off) & (in_toggle = TRUE) & (v_onCounter >= 255) :
         state_off_transition_1_is_dead;
9.      (_current_state = on) & (in_toggle = TRUE) & (TRUE) :
         state_on_transition_0_is_dead;
10.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness <= 1) :
         state_on_transition_1_is_dead;
11.     (_current_state = on) & (in_dimDown = TRUE) & (v_brightness > 1) :
         state_on_transition_2_is_dead;
12.     (_current_state = on) & (in_dimUp = TRUE) & (v_brightness < 8) :
         state_on_transition_3_is_dead;
13.     TRUE : dead_trans_init;
14.   esac;
15.
16. SPEC  AG _dead_transition != state_off_transition_0_is_dead
17. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 0 of state 'off' is not dead | Transition 0 of state 'off' is dead
18.
19. SPEC  AG _dead_transition != state_off_transition_1_is_dead
20. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 1 of state 'off' is not dead | Transition 1 of state 'off' is dead
21.
22. SPEC  AG _dead_transition != state_on_transition_0_is_dead
23. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 0 of state 'on' is not dead | Transition 0 of state 'on' is dead
24.
25. SPEC  AG _dead_transition != state_on_transition_1_is_dead
26. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 1 of state 'on' is not dead | Transition 1 of state 'on' is dead
27.
28. SPEC  AG _dead_transition != state_on_transition_2_is_dead
29. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 2 of state 'on' is not dead | Transition 2 of state 'on' is dead
30.
```

```
31. SPEC  AG _dead_transition != state_on_transition_3_is_dead
32. --SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###
      Transition 3 of state 'on' is not dead | Transition 3 of state 'on' is dead
```

Listing 13 NuSMV dead transition checks (generated by mbeddr)

5.3 Manual Checks

In this chapter the translation of the manually formulated checks to NuSMV input is explained.

Liveness checks

To assure that a state is live, i.e., in every evolution of the state machine it remains reachable, the following CTL statement is checked: In all evolutions in every state it is true that: In at least one further evolution finally the desired state can be reached. The corresponding NuSMV code is listed in Listing 14.

```
1.  SPEC  AG (EF _current_state = on)
2.  --State 'on' is live | State 'on' is not live!
3.
4.  SPEC  AG (EF _current_state = servicestate)
5.  --State 'servicestate' is live | State 'servicestate' is not live!
```

Listing 14 NuSMV liveness checks (generated by mbeddr)

P is false before R

As seen in Listing 7, it is checked if the state *on* is not reached before toggle is fired. This check is translated into the LTL formula shown in Listing 15. The left part of the implication is true iff the event toggle will eventually be fired in all possible evolutions. As nothing can prevent an event from being fired, this of course is true. Therefore, the result of this specific check depends only on the right site of the implication. There it is assured that the current state is not *on* until the event toggle is fired. So this is straight forward the originally formulated condition.

```
1.  LTLSPEC ((F (in_toggle)) -> ((!(_current_state = on)) U (in_toggle)))
2.  --Condition 'on' is not true before 'toggle'
      | Condition 'on' can be true before 'toggle'
```

Listing 15 NuSMV P is false before R checks (generated by mbeddr)

6 Conclusion

This paper gives a rough walkthrough how state machines can be modelled and checked with mbeddr. It shows the mathematical languages CTL and LTL and how they are used in mbeddr and NuSMV. The modelling possibilities for state machines in mbeddr are discussed including the unit test support and the translation to C code. Furthermore, the possibilities provided by mbeddr of formulating desired properties for a state model are shown. Finally it is revealed how mbeddr uses NuSMV to do the check work.

It is apparent that supporting higher level abstractions like state machines as a first level language concept brings many benefits. Not only an IDE can support a good readable syntax but also can generate valuable outputs, e.g., graphical representations or translations for other tools like NuSMV. Furthermore, automatisms can be supported like default check cases. Eliminating the manual work not only saves time but also increases quality.

Unit tests are an important tool for software development. They not only help avoiding bugs, they can also help to improve system design and architecture. Furthermore, good unit tests can show how a unit can be accessed. But there is one intrinsic down side: They only check particular cases. A fully coverage of all possibilities a system can run through is hardly possible. This is where model checking comes into play. Symbolic model checking allows to proof that a given property holds in every possible system run. This power comes with a price: The properties have to be written in a mathematical language. This language is not necessarily intuitive and may be difficult to use correctly. It is probably not the language of a software engineer used to write imperative code.

None the less, the certainty of a mathematical proof can be of high value. Especially for safety critical systems. Today different technologies are available. This paper discussed how state machines can be analysed using mbeddr together with NuSMV relying on a BDD based proofing technique. Other approaches exist. E.g., NuSMV was extended with a SAT based bounded model checker [25]. mbeddr includes an alternative to NuSMV: it supports the CBMC bounded model checker [26].

7 References

[1] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore and Marco Roveri, "NuSMV 2.5 Tutorial," [Online]. Available: http://nusmv.fbk.eu/NuSMV/tutorial/index.html. [Accessed 10 March 2013].

[2] Daniel Ratiu, Markus Voelter, Bernhard Schaetz, Bernd Kolb, "Language Engineering as an Enabler for Incrementally Defined Formal Analyses," in *FORMSERA 2012 Workshop*, 2012.

[3] Bernd Kolb, Markus Völter, Daniel Ratiu, Domenik Pavletic, "mbeddr.com | engineering the future of embedded software.," itemis, [Online]. Available: http://mbeddr.wordpress.com/. [Accessed 30 May 2013].

[4] "JetBrains :: Meta Programming System - Language Oriented Programming environment and DSL creation tool," JetBrains, [Online]. Available: http://www.jetbrains.com/mps/. [Accessed 30 May 2013].

[5] M. Steiner, "Model Checking decision tables with mbeddr and yices," Rapperswil, 2013.

[6] Daniel Ratiu, Markus Voelter, Zaur Molotnikov, Bernhard Schaetz, "Implementing Modular Domain Specific Languages and Analyses," in *Modevva*, 2012.

[7] "NuSMV home page," open source, [Online]. Available: http://nusmv.fbk.eu/. [Accessed 20 May 2013].

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *CAV 2002*, Copenhagen, 2002.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^20 States and beyond*," Information and Computation , 98(2):142–170, 1992.

[10] E. Clarke, O. Grumberg and K. Hamaguchi, "Another Look at LTL Model Checking," 1994.

[11] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri, "NUSMV: a new symbolic model checker," *STTT International Journal on Software Tools for Technology Transfer,* 1999.

[12] E. Axelsson, "Functional Discoveries: Difference between LTL and CTL," [Online]. Available: http://fun-discoveries.blogspot.ch/2012/04/difference-between-ltl-and-ctl.html. [Accessed 02 June 2013].

[13] "lo.logic - What is the difference between LTL and CTL? - Theoretical Computer Science Stack Exchange," [Online]. Available: http://cstheory.stackexchange.com/questions/6735/what-is-the-difference-between-ltl-and-ctl. [Accessed 02 June 2013].

[14] E. A. Emerson, “Temporal And Modal Logic*,” 14 March 1995. [Online]. Available: http://samy.informatik.hu-berlin.de/top/lehre/WS04-05/sem_spezi_verifi/literatur/emerson_tempNmodalLogik.ps. [Accessed 02 June 2013].

[15] “Temporal logic,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Temporal_logic. [Accessed 13 May 2013].

[16] “Temporale Logik,” Wikipedia, [Online]. Available: http://de.wikipedia.org/wiki/Temporale_Logik. [Accessed 13 May 2013].

[17] “Linear temporal logic,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Linear_temporal_logic. [Accessed 13 May 2013].

[18] “Lineare temporale Logik,” Wikipedia, [Online]. Available: http://de.wikipedia.org/wiki/Lineare_temporale_Logik. [Accessed 13 May 2013].

[19] “Computation tree logic,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Computation_tree_logic. [Accessed 13 May 2013].

[20] “Computation Tree Logic,” Wikipedia, [Online]. Available: http://de.wikipedia.org/wiki/Computation_Tree_Logic. [Accessed 13 May 2013].

[21] “CTL*,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/CTL*. [Accessed 13 May 2013].

[22] “First-order logic,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/First-order_logic. [Accessed 18 May 2013].

[23] Hamid Alavi, George Avrunin, James Corbett, Laura Dillon, Matt Dwyer, Corina Pasareanu , “About Specification Patterns,” SAnTos laboratory, [Online]. Available: http://patterns.projects.cis.ksu.edu/. [Accessed 30 May 2013].

[24] “mbeddr c userguide,” [Online]. Available: http://mbeddr.com. [Accessed 29 March 2013].

[25] A. Cimatti, E. Giunchiglia, M. Roveri, M. Pistore, R. Sebastiani and A. Tacchella, “Integrating BDD-based and SAT-based Symbolic Model Checking,” in *4th International Workshop on Frontiers of Combining Systems*, Santa Margherita Ligure, 2002.

[26] “The CBMC Homepage,” [Online]. Available: http://www.cprover.org/cbmc/. [Accessed 03 June 2013].

Appendix

The appendix consists of four parts: The documentation of a bug found in mbeddr, a very short introduction to BDDs, the complete, unmodified C code for the LightSwitch and the complete, unmodified input for NuSMV.

8 Bug Report

While working on this paper, I found and reported a bug. Fiddling around with some checks in mbeddr, I realized that the check outcome depends on the existence of other checks. Actually the results seemed quite random. Asking the developers of mbeddr, they confirmed the bug and provided the following workaround: The check conditions based on CTL must be stated in front of the LTL conditions.

9 Binary Decision Diagrams BDDs

This chapter gives a very short introduction to Binary Decision Diagrams BDDs. A BDD is a directed acyclic graph. It represents a Boolean formula. Figure 10 shows the BDD for the formula "(a ∧ b) ∨ (c ∧ d)". Given a specific assignment for the variables, one can determine the result of the formula by traversing the graph starting at its root. The assignment a = 0, b = 1, c = 1, d = 0 leads to a leaf with value 0. Therefore, the result of the formula is false. This can be seen in Figure 11.

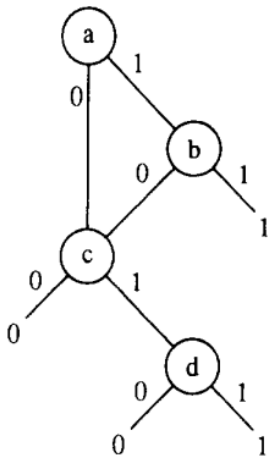

Figure 10 BDD for the formula (a ∧ b) ∨ (c ∧ d) [9]

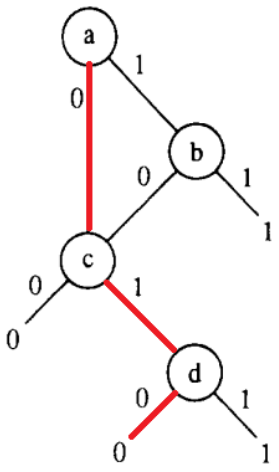

Figure 11 BDD evaluation of the formula (a ∧ b) ∨ (c ∧ d) for a = 0, b = 1, c = 1, d = 0

10 LightSwitch C Code

Subsequent is the C code for the LightSwitch. This code is automatically generated by mbeddr. This is the original version with original variable names.

```
1.  #define MAIN_SERVICESTATE_THRESHOLD (255)
2.  #define MAIN_MIN_BRIGHTNESS (1)
3.  #define MAIN_MAX_BRIGHTNESS (8)
4.  #define MAIN_BRIGHTNESS_START (5)
5.
6.
7.  typedef enum ___main_sm_events_LightSwitch{
8.    main_sm_events_LightSwitch__LightSwitch__event_toggle,
9.    main_sm_events_LightSwitch__LightSwitch__event_dimUp,
10.   main_sm_events_LightSwitch__LightSwitch__event_dimDown
11. } main_sm_events_LightSwitch;
12.
13. typedef enum ___main_sm_states_LightSwitch{
14.   main_sm_states_LightSwitch__LightSwitch_off__state,
15.   main_sm_states_LightSwitch__LightSwitch_on__state,
16.   main_sm_states_LightSwitch__LightSwitch_servicestate__state
17. } main_sm_states_LightSwitch;
18.
19. struct main_sm_data_LightSwitch {
20.   main_sm_states_LightSwitch __currentState;
21.   uint8_t onCounter;
22.   uint8_t brightness;
23. };
24.
25. void main_sm_init_LightSwitch(struct main_sm_data_LightSwitch* instance)
26. {
27.   instance->__currentState = main_sm_states_LightSwitch__LightSwitch_off__state;
28.   instance->onCounter = 0;
29.   instance->brightness = MAIN_BRIGHTNESS_START;
30. }
31.
32. void main_sm_execute_LightSwitch(struct main_sm_data_LightSwitch* instance, main_sm_
    events_LightSwitch event, void** arguments)
33. {
34.   switch (instance->__currentState)
35.   {
36.     case main_sm_states_LightSwitch__LightSwitch_off__state: {
37.       switch (event)
38.       {
39.         case main_sm_events_LightSwitch__LightSwitch__event_toggle: {
40.           if ( instance->onCounter < MAIN_SERVICESTATE_THRESHOLD )
41.           {
42.             // switch state
43.             instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_on__state;
44.             // entry actions
45.             ++instance->onCounter;
46.             return ;
47.           }
48.
49.           if ( instance->onCounter >= MAIN_SERVICESTATE_THRESHOLD )
50.           {
51.             // switch state
52.             instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_servicestate__state;
53.             return ;
54.           }
55.
```

```
56.           break;
57.         }
58.       }
59.
60.       break;
61.     }
62.     case main_sm_states_LightSwitch__LightSwitch_on__state: {
63.       switch (event)
64.       {
65.         case main_sm_events_LightSwitch__LightSwitch__event_toggle: {
66.           if ( 1 )
67.           {
68.             // switch state
69.             instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_off__state;
70.             return ;
71.           }
72.
73.           break;
74.         }
75.         case main_sm_events_LightSwitch__LightSwitch__event_dimDown: {
76.           if ( instance->brightness <= MAIN_MIN_BRIGHTNESS )
77.           {
78.             // switch state
79.             instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_off__state;
80.             return ;
81.           }
82.
83.           if ( instance->brightness > MAIN_MIN_BRIGHTNESS )
84.           {
85.             // transition actions
86.             --instance->brightness;
87.             // switch state
88.             instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_on__state;
89.             // entry actions
90.             ++instance->onCounter;
91.             return ;
92.           }
93.
94.           break;
95.         }
96.         case main_sm_events_LightSwitch__LightSwitch__event_dimUp: {
97.           if ( instance->brightness < MAIN_MAX_BRIGHTNESS )
98.           {
99.             // transition actions
100.            ++instance->brightness;
101.            // switch state
102.            instance-
    >__currentState = main_sm_states_LightSwitch__LightSwitch_on__state;
103.            // entry actions
104.            ++instance->onCounter;
105.            return ;
106.          }
107.
108.          break;
109.        }
110.      }
111.
112.      break;
113.    }
114.    case main_sm_states_LightSwitch__LightSwitch_servicestate__state: {
115.      switch (event)
116.      {
117.      }
```

```
118.
119.       break;
120.     }
121.   }
122.
123.}
```

Listing 16 Generated C code for LightSwitch

11 LightSwitch NuSMV Code

Subsequent is the input for NuSMV. This code is automatically generated by mbeddr. This is the original version with original variable names.

```
1.  --r:e9b6775d-aa85-4773-8f3a-876caff40de4(LightSwitch.Main.main)
2.
3.  MODULE statemachine( in__toggle__present_ID_2288752385279320977, in__dimUp__present_
    ID_2288752385279321016, in__dimDown__present_ID_2288752385279321017)
4.
5.  VAR
6.      lv__onCounter_ID_2288752385279320971:-1..256;
7.      lv__brightness_ID_2288752385279321005:0..9;
8.      _current_state:{off_ID_2288752385279320980,on_ID_2288752385279320992,servicestat
    e_ID_2288752385279321001};
9.      _nondeterminism_detector:{no_nondeterminism,nd_detected_inoff,nd_detected_inon,n
    d_detected_inservicestate};
10.     _dead_transition:{dead_trans_init,state_off_transition_0_is_dead,state_off_trans
    ition_1_is_dead,state_on_transition_0_is_dead,state_on_transition_1_is_dead,state_on
    _transition_2_is_dead,state_on_transition_3_is_dead};
11.
12.
13. ASSIGN
14.     init (_current_state) := off_ID_2288752385279320980;
15.     next (_current_state) := case
16.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255) : on_ID_22887
    52385279320992;
17.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 >= 255) : servicesta
    te_ID_2288752385279321001;
18.         (_current_state = on_ID_2288752385279320992) & (in__toggle__present_ID_22887
    52385279320977 = TRUE) : off_ID_2288752385279320980;
19.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 <= 1) : off_ID_2288
    752385279320980;
20.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1) : on_ID_228875
    2385279320992;
21.         (_current_state = on_ID_2288752385279320992) & (in__dimUp__present_ID_228875
    2385279321016 = TRUE) & (lv__brightness_ID_2288752385279321005 < 8) : on_ID_22887523
    85279320992;
22.         !(((_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2
    288752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255))) & !(((_c
    urrent_state = off_ID_2288752385279320980) & (in__toggle__present_ID_228875238527932
    0977 = TRUE) & (lv__onCounter_ID_2288752385279320971 >= 255))) & !(((_current_state
    = on_ID_2288752385279320992) & (in__toggle__present_ID_2288752385279320977 = TRUE)))
    & !(((_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288752
    385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 <= 1))) & !(((_current
    _state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288752385279321017 =
    TRUE) & (lv__brightness_ID_2288752385279321005 > 1))) & !(((_current_state = on_ID_2
```

```
    288752385279320992) & (in__dimUp__present_ID_2288752385279321016 = TRUE) & (lv__brig
    htness_ID_2288752385279321005 < 8))) : _current_state;
23.     esac;
24.
25.     init (_nondeterminism_detector) := no_nondeterminism;
26.     next (_nondeterminism_detector) := case
27.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 >= 255) & (lv__onCou
    nter_ID_2288752385279320971 < 255) : nd_detected_inoff;
28.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255) & (lv__onCoun
    ter_ID_2288752385279320971 >= 255) : nd_detected_inoff;
29.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1) & (lv__brightn
    ess_ID_2288752385279321005 <= 1) : nd_detected_inon;
30.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 <= 1) & (lv__bright
    ness_ID_2288752385279321005 > 1) : nd_detected_inon;
31.         TRUE : no_nondeterminism;
32.     esac;
33.
34.     init (_dead_transition) := dead_trans_init;
35.     next (_dead_transition) := case
36.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255) : state_off_t
    ransition_0_is_dead;
37.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 >= 255) : state_off_
    transition_1_is_dead;
38.         (_current_state = on_ID_2288752385279320992) & (in__toggle__present_ID_22887
    52385279320977 = TRUE) & (TRUE) : state_on_transition_0_is_dead;
39.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 <= 1) : state_on_tr
    ansition_1_is_dead;
40.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1) : state_on_tra
    nsition_2_is_dead;
41.         (_current_state = on_ID_2288752385279320992) & (in__dimUp__present_ID_228875
    2385279321016 = TRUE) & (lv__brightness_ID_2288752385279321005 < 8) : state_on_trans
    ition_3_is_dead;
42.         TRUE : dead_trans_init;
43.     esac;
44.
45.     init (lv__onCounter_ID_2288752385279320971) := 0;
46.     init (lv__brightness_ID_2288752385279321005) := 5;
47.     next (lv__onCounter_ID_2288752385279320971) := case
48.         (_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2288
    752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255) : lv__onCount
    er_ID_2288752385279320971 + 1 < 0 ? -
    1 : lv__onCounter_ID_2288752385279320971 + 1 > 255 ? 256 : lv__onCounter_ID_22887523
    85279320971 + 1;
49.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1) : lv__onCounte
    r_ID_2288752385279320971 + 1 < 0 ? -
    1 : lv__onCounter_ID_2288752385279320971 + 1 > 255 ? 256 : lv__onCounter_ID_22887523
    85279320971 + 1;
50.         (_current_state = on_ID_2288752385279320992) & (in__dimUp__present_ID_228875
    2385279321016 = TRUE) & (lv__brightness_ID_2288752385279321005 < 8) : lv__onCounter_
    ID_2288752385279320971 + 1 < 0 ? -
    1 : lv__onCounter_ID_2288752385279320971 + 1 > 255 ? 256 : lv__onCounter_ID_22887523
    85279320971 + 1;
51.         !(((_current_state = off_ID_2288752385279320980) & (in__toggle__present_ID_2
    288752385279320977 = TRUE) & (lv__onCounter_ID_2288752385279320971 < 255))) & !(((_c
    urrent_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_228875238527932
    1017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1))) & !(((_current_state =
    on_ID_2288752385279320992) & (in__dimUp__present_ID_2288752385279321016 = TRUE) & (l
```

```
    v__brightness_ID_2288752385279321005 < 8))) : lv__onCounter_ID_2288752385279320971 <
    0 ? -
    1 : lv__onCounter_ID_2288752385279320971 > 255 ? 256 : lv__onCounter_ID_228875238527
    9320971;
52.     esac;
53.
54.     next (lv__brightness_ID_2288752385279321005) := case
55.         (_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2288
    752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1) : lv__brightne
    ss_ID_2288752385279321005 - 1 < 1 ? 0 : lv__brightness_ID_2288752385279321005 - 1 >
    8 ? 9 : lv__brightness_ID_2288752385279321005 - 1;
56.         (_current_state = on_ID_2288752385279320992) & (in__dimUp__present_ID_228875
    2385279321016 = TRUE) & (lv__brightness_ID_2288752385279321005 < 8) : lv__brightness
    _ID_2288752385279321005 + 1 < 1 ? 0 : lv__brightness_ID_2288752385279321005 + 1 > 8
    ? 9 : lv__brightness_ID_2288752385279321005 + 1;
57.         !(((_current_state = on_ID_2288752385279320992) & (in__dimDown__present_ID_2
    288752385279321017 = TRUE) & (lv__brightness_ID_2288752385279321005 > 1))) & !(((_cu
    rrent_state = on_ID_2288752385279320992) & (in__dimUp__present_ID_228875238527932101
    6 = TRUE) & (lv__brightness_ID_2288752385279321005 < 8))) : lv__brightness_ID_228875
    2385279321005 < 1 ? 0 : lv__brightness_ID_2288752385279321005 > 8 ? 9 : lv__brightne
    ss_ID_2288752385279321005;
58.     esac;
59.
60.
61. SPEC    AG _current_state != off_ID_2288752385279320980
62. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###State 'off' is reachable | State 'off' is unre
    achable
63.
64. SPEC    AG _current_state != on_ID_2288752385279320992
65. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###State 'on' is reachable | State 'on' is unreac
    hable
66.
67. SPEC    AG _current_state != servicestate_ID_2288752385279321001
68. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###State 'servicestate' is reachable | State 'ser
    vicestate' is unreachable
69.
70. SPEC    AG (0 <= lv__onCounter_ID_2288752385279320971 & lv__onCounter_ID_22887523852
    79320971 <= 255)
71. --
    Variable 'onCounter' is always between its defined bounds | Variable 'onCounter' is
    possibly out of range
72.
73. SPEC    AG (1 <= lv__brightness_ID_2288752385279321005 & lv__brightness_ID_228875238
    5279321005 <= 8)
74. --
    Variable 'brightness' is always between its defined bounds | Variable 'brightness' i
    s possibly out of range
75.
76. SPEC    AG _nondeterminism_detector != nd_detected_inoff
77. --
    State 'off' has deterministic transitions | State 'off' contains nondeterministic tr
    ansitions
78.
79. SPEC    AG _nondeterminism_detector != nd_detected_inon
80. --
    State 'on' has deterministic transitions | State 'on' contains nondeterministic tran
    sitions
81.
82. SPEC    AG _nondeterminism_detector != nd_detected_inservicestate
83. --
    State 'servicestate' has deterministic transitions | State 'servicestate' contains n
    ondeterministic transitions
84.
```

```
85. SPEC    AG _dead_transition != state_off_transition_0_is_dead
86. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 0 of state 'off' is not dead | Tran
    sition 0 of state 'off' is dead
87.
88. SPEC    AG _dead_transition != state_off_transition_1_is_dead
89. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 1 of state 'off' is not dead | Tran
    sition 1 of state 'off' is dead
90.
91. SPEC    AG _dead_transition != state_on_transition_0_is_dead
92. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 0 of state 'on' is not dead | Trans
    ition 0 of state 'on' is dead
93.
94. SPEC    AG _dead_transition != state_on_transition_1_is_dead
95. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 1 of state 'on' is not dead | Trans
    ition 1 of state 'on' is dead
96.
97. SPEC    AG _dead_transition != state_on_transition_2_is_dead
98. --
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 2 of state 'on' is not dead | Trans
    ition 2 of state 'on' is dead
99.
100.SPEC    AG _dead_transition != state_on_transition_3_is_dead
101.--
    SUCCESS_WHEN_PROPERTY_DOES_NOT_HOLD###Transition 3 of state 'on' is not dead | Trans
    ition 3 of state 'on' is dead
102.
103.SPEC    AG (EF _current_state = on_ID_2288752385279320992)
104.--State 'on' is live | State 'on' is not live!
105.
106.SPEC    AG (EF _current_state = servicestate_ID_2288752385279321001)
107.--State 'servicestate' is live | State 'servicestate' is not live!
108.
109.LTLSPEC ((F (in__toggle__present_ID_2288752385279320977)) -
    > ((((!(_current_state = on_ID_2288752385279320992))) U (in__toggle__present_ID_2288
    752385279320977))))
110.--
    Condition 'on' is not true before 'toggle' | Condition 'on' can be true before 'togg
    le'
111.
112.
113.MODULE main
114.
115.VAR
116.    in__toggle__present_ID_2288752385279320977:boolean;
117.    in__dimUp__present_ID_2288752385279321016:boolean;
118.    in__dimDown__present_ID_2288752385279321017:boolean;
119.    sm:statemachine(in__toggle__present_ID_2288752385279320977,in__dimUp__present_ID
    _2288752385279321016,in__dimDown__present_ID_2288752385279321017);
120.
121.INVAR   (TRUE & !((in__dimDown__present_ID_2288752385279321017 & in__dimUp__present_
    ID_2288752385279321016)) & !((in__dimDown__present_ID_2288752385279321017 & in__togg
    le__present_ID_2288752385279320977)) & !((in__dimUp__present_ID_2288752385279321016
    & in__toggle__present_ID_2288752385279320977)) & !((in__toggle__present_ID_228875238
    5279320977 & in__dimUp__present_ID_2288752385279321016))) & (FALSE | in__dimDown__pr
    esent_ID_2288752385279321017 | in__dimUp__present_ID_2288752385279321016 | in__toggl
    e__present_ID_2288752385279320977)
```

Listing 17 Generated NuSMV input for LightSwitch