Sicherheitskritische Systeme (WS 2015/2016)
Philipp Kochanski
1999848
31/01/2016

Final Project: Creation of an Event-B model for the game “Lightbot”

“Light Bot is a puzzle game that uses game mechanics that are firmly rooted in programming concepts. Light-bot lets players gain a practical understanding of basic control-flow concepts like procedures, loops, and conditionals, just by guiding a robot with commands to light up tiles and solve levels.”

Documentation

As part of the final project, this documentation should serve as a short overview on how the Event-B model was created and what steps were taken at each refinement. Furthermore, in the last section there will be a short explanation in regards to how the model was validated with the goal of finding a solution to a given level.
The project was build using ‘**Rodin Platform x64**’ version: **3.2.0-ecacdcb**.

Modeling

The entire Event-B model for the “**Lightbot**” project is divided into three machines and three contexts which translates into two refinement steps that will be explained further below. In addition, there are three levels, also contexts, that are extending the last extended context “**ctx_2**”.

Initial stage

The initial machine “**mac_0**” and context “**ctx_0**” were created to provide the following functionality:

1. **Movement** – Lightbot should be able to move on platforms in x and y directions (two dimensional movement)
2. **Rotation** – Lightbot should be able to rotate left and right
3. **Lights** – Lightbot should be able to enable or disable lights on a given platform

Missing actions like – jumping and sub-processes will be added in later refinements.

Context: ctx_0

The initial context “**ctx_0**” describes all structures needed for creating a level and performing all basic Lightbot actions. The general idea is to extend this context in order to define a level.
There are 3 sets, 14 constants and 11 axioms used here.

A level is build out of “**platforms**”, where “**platforms**“ are described as a subset of $\mathbb{N}1 \times \mathbb{N}1$, the cartesian product of the set of positive natural numbers with the set of positive natural numbers. For example: $\{(1 \mapsto 1), (2 \mapsto 1)\}$
The first number of each pair is the x coordinate and the second number is the y coordinate of the given platform.

“**Lights**” are a subset of platforms, as “**lights**” can only be placed on “**platforms**” that exist.

At each level, the Lightbot must be placed at a specific starting position “**start_pos**” which is described as a set that contains the x and y coordinate of the starting platform (here a function). For example: $\{(x \mapsto 1), (y \mapsto 1)\}$
As stated, it is obvious that the combination of the x and y coordinates must be an element of “**platforms**”, as the Lightbot can only be placed on a platform that exists. Furthermore, the set “**coord_lables**” and the constants “**x**” and “**y**” were used to improve the readability of coordinates and also to omit the usage of dom/ran operators.

In addition to a starting position, it is also necessary to specify which direction Lightbot is facing. The constant "**start_rot**" stores the direction based on the set "**dir_lables**" which holds the possible directions "**N**" (north), "**E**" (east), "**S**" (south) and "**W**" (west).

The constant "**movement**" stores the rules on how to move based on which direction Lightbot is currently facing. Here each direction ("**N**", "**E**", "**S**", "**W**") is paired with values for x and y that will be added to the current x and y coordinates to perform a move.

Example:
Assuming Lightbot is currently at position $\{(x \mapsto 1), (y \mapsto 1)\}$ and is facing south, performing a movement will result in checking the "**movement**" constant for the pair containing S and adding the related x and y coordinates to the current position.
$S \mapsto \{x \mapsto 0, y \mapsto 1\}$ – which means the new position of Lightbot is $x=1+0$ and $y=1+1$ or simply $\{(x \mapsto 1), (y \mapsto 2)\}$.

Lastly, the constant "**rotations**" stores the rules on how the direction Lightbot is facing will be changed based on the performed rotation. Here for each direction ("**N**", "**E**", "**S**", "**W**") there are two rules, one when rotating right and one when rotating left.

Example:
$(N \mapsto left) \mapsto W$ – which means, when the current direction is north and we are performing a rotation to the left, then the new direction is west.

This concludes context "**ctx_0**".

Machine: mac_0

The initial machine **mac_0** sees a given level (a level extends "**ctx_0**" or later "**ctx_2**") and defines all variables, invariants and events that are needed to create a playable "basic" Lightbot level.
There are 3 variables, 5 invariants and 5 events used here.

First there are 3 variables defined. Throughout the first machine and all following, it is necessary to know:

- the current position of Lightbot, stored in "**lb_pos**"
- the direction Lightbot is currently facing, stored in "**lb_rot**"
- which lights are enabled or disabled, stored in "**lights_stat**"

Based on these 3 variables the following 5 invariants are defined.

Regarding "**lb_pos**" it is obvious that the current position of Lightbot must consist of x and y coordinates that also exist as a platform. Lightbot cannot be placed on a position where no platform exists.

In addition, the current direction Lightbot is facing, "**lb_rot**", must be an element of the directional labels "**N**", "**E**", "**S**" or "**W**".

As for the lights, the goal of Lightbot is to enable all lights in order to finish a level. Therefore, it is necessary to track whether lights are enabled (TRUE) or disabled (FALSE). The variable **lights_stats**, defined as a total function, stores the current status of each light, where each entry is a pair consisting of the light and the related status.
Example **lights_stats** content: $\{((2 \mapsto 2) \mapsto FALSE), ((2 \mapsto 3) \mapsto TRUE)\}$

Lastly there is an invariant labeled with "**validation**" which is used to find a solution for a given level. This invariant, which in general should be commented out, will be further explained in the validation section of this documentation.

During the **INITIALISATION** event, the Lightbot position "**lb_pos**" is initialized with the x and y coordinates from the start position "**start_pos**" defined within the seen level context. Same applies for the initial Lightbot direction which is initialized using "**start_rot**". Lastly all lights defined in the seen level context are set to FALSE and stored as pairs within "**lights_stats**".

The remaining 4 events are used to define the basic actions: movement, rotation to left, rotation to right and light activation/deactivation.

The event "**move**" consists of a guard that checks whether the next combination of x and y coordinates correspond to an element from platforms. Obviously this event should only be executable when the calculated platform exists. A move by itself, is the modification of the current Lightbot position "**lb_pos**" by adding the values stored in the constant "**movement**", based on the direction Lightbot is currently facing. As already explained in the section to context "**ctx_0**", for example when facing south, the next position of Lightbot should be one position below the current one. This means that the x coordinate will stay the same but the y coordinate will be incremented by one.

The next two events, "**rotate_left**" and "**rotate_right**", are used to change the direction Lightbot is currently facing. Both events are using the constant "rotations", which returns the new direction of Lightbot based on the provided rotation (either left or right).

The last event called "**light**" is responsible for enabling or disabling a platform containing a light. The related guards are checking whether the current platform contains a light based on the constant "**lights**" from the seen level context and whether the current platform is an element from the domain of the variable "**lights_stat**". If both guards are true, then the related action is executed. The new value of the light within "**lights_stat**" is calculated based on the current value of the viewed light. There are two possibilities:

a) current value of the light is TRUE then the following expression will be FALSE: bool(TRUE != TRUE)
b) current value of the light is FALSE then the following expression will be TRUE: bool(FALSE != TRUE)

First refinement

The first refinement refines "**mac_0**" into "**mac_1**" and extends "**ctx_0**" with "**ctx_1**"
This refinement provides the following additional functionality:

1. **Jumping** – Lightbot should be able to jump on platforms that are one elevation level below or above the current platform (three dimensional movement)

In general, in this stage an additional height information is added to each platform and a jump event is defined so that Lightbot can navigate to platforms of different heights.

Context: ctx_1

Context "**ctx_1**" extends "**ctx_0**" by adding a new height information to each platform. Therefore, one new constant called "**elevations**" and one new axiom is defined. "**Elevations**" is defined as total function from platforms onto a natural number. For example: `{((1↦1)↦0),((1↦2)↦1)}`
Platform on position `(1↦1)` has the elevation 0 (which is the lowest elevation) and `(1↦2)` has the elevation 1.

This concludes context "**ctx_1**"

Machine: mac_1

For the refinement of "**mac_0**" to "**mac_1**" the existing "**move**" event is modified by adding one additional guard and a new event called "**jump**" is added.

As there is now an elevation value added to each platform we need to make sure that we can only "**move**" to a platform of the same elevation as the current one. Otherwise in order to reach a platform that is higher or lower by one we need to "**jump**". First a new guard to the existing "**move**" event is added. Here it is checked whether the difference between the elevation of the next platform and the current platform equals 0. If that is the case, then a "**move**" is possible, otherwise it is not. The new "**jump**" event extends the existing "**move**" event by adding one

additional guard. Here it is checked whether the difference between the next platform and the current one is either 1 (next platform is higher by one) or -1 (next platform is lower by one). The action stays the same, as Lightbot is simply changing the position to the next platform.

This concludes machine "**mac_1**".

Second refinement

The second refinement refines "**mac_1**" into "**mac_2"** and extends "**ctx_1**" with "**ctx_2**".
This refinement provides the following additional functionality:

1. **Processes** – Lightbot can now be programmed using the three processes **MAIN**, **PROC1** and **PROC2** Having access to a predefined number of commands.

In general, in this stage the missing program-stack is implemented. When the programming mode is enabled, it is possible to add commands to the main process **MAIN** and to the sub-processes **PROC1** and **PROC2**. Once commands are added, the machine can be switched into the play mode in which Lightbot will only execute the commands provided in the **MAIN**, **PROC1** and **PROC2** processes and in the same order as initially defined. Sub-processes can be executed using recursion so that for example **PROC1** will start itself over and over again. It might useful to limit the process stack as in reality the available memory is limited. However, due to simplicity reasons recursions can be executed indefinitely. Lastly, it is possible to define how many commands can be saved into a process via the refined context.

Context: ctx_2

Context "**ctx_2**" extends context "**ctx_1**" by adding 2 sets, 12 constants and 9 axioms.

Every level provides access to a given amount of commands. At most, it is possible to give the player access to 7 commands which consists of: a move command ("**move_c**"), a rotate left command ("**left_c**"), a rotate right command ("**right_c**"), a jump command ("**jump_c**"), a light command ("**light_c**") and two commands for the processes **PROC1** and **PROC2** ("**proc1_c**", "**proc2_c**"). The above commands (constants) are stored using the "**commands**" set. Every level must contain a "**level_commands**" constant which is a subset of "**commands**" and which defines what commands/actions are available in the given level.

The constant "**proc_number**" defines how many processes are available in the given level. Setting "**proc_number**" to 1 means that only the **MAIN** process is available, setting it to 2 means **MAIN** and **PROC1** are available and setting it to 3 means that **MAIN**, **PROC1** and **PROC2** are available. Four additional axioms were created to guarantee that a level is valid when it comes to available processes and commands. First the axiom labeled "**axm16**" makes sure that when setting the "**proc_number**" to 2, "**proc1_c**" will be part of the available "**level_commands**". Providing access to **PROC1** without having the ability to execute **PROC1** can be considered pointless. The axiom labeled "**axm17**" makes sure that when setting the "**proc_number**" to 3, both "**proc1_c**" and "**proc2_c**" will be a subset of the available "**level_commands**". In addition, the other direction must also be satisfied, which is provided by "**axm18**" and "**axm19**". When adding "**proc1_c**" to the available "**level_commands**" it is necessary to ensure that the "**proc_number**" is greater than 1. In other words, it must be possible to add commands to **PROC1**. Same goes for "**proc2_c**". In case "**proc2_c**" is available in "**level_commands**", then the "**proc_number**" must be greater than 2. Creating a level with an invalid "**level_commands**" and "**proc_number**" combination will result in an error during model checking, that no constants can be found to satisfy the given axioms. This is a desired behavior.

In addition, a constant "**proc_length**" is used to define how many commands can be stored in a given process. Here "**proc_length**" is defined as a total function where to each process, based on "**proc_number**", a positive natural number is added.

Furthermore, the set "**stack_lables**" and the constants "**prog**" (program/process) and "**cmd**" (command) were introduced in order to improve the readability of the program-stack within the model checker.

This concludes context "**ctx_2**"

Machine: mac_2

Machine "**mac_2**" refines machine "**mac_1**" by adding 6 variables, 8 invariants and 6 events.
In addition, every existing event from "**mac_1**" is extended and modified by adding several new guards.

First, a new variable called "**proc_list**" is defined which stores the commands added to each process. It consists of ordered pairs, with a process number on the left side and partial function on the right. The partial function serves as a sequence (also a set), where each positive natural number is paired with a given command and later executed in ascending order based on the natural number. As stated before, the **MAIN** process corresponds to the number 1, the **PROC1** process corresponds to the number 2 and the **PROC2** process corresponds to the number 3.
An example "**proc_list**" looks like this: `{(1↦{(1↦proc1_c)}),(2↦{(1↦move_c),(2↦jump_c)})}`
Here the **MAIN** process (1) will execute the command "**proc1_c**" first. **PROC1** (2) however will first execute the "**move_c**" command and after that the "**jump_c**" command.

The variable "**cmd_amt**" stores how many commands are currently stored within a given process.
It is used within the "add_command" event to check whether commands can still be added to a process or whether the command limit was already reached. The same is also checked as an invariant, where it must be ensured that for each process the current "**cmd_amt**" is smaller or equal to the "**proc_length**" defined within the seen level context.

Example:
After initialization all processes should be empty as no command was added yet. This means that "**cmd_amt**" will look like this: `{(1↦0),(2↦0),(3↦0)}` when all three processes are available.

The two variables called "**prog_mode**" and "**play_mode**" are used as an indicator on which mode is currently enabled. Both can be either TRUE or FALSE but they can never be both TRUE or both FALSE at the same time.

Lastly the program-stack is defined using the variable "**proc_stack**". It is build using a partial function where every positive natural number within "**proc_stack**", indicating the stack position, is paired with a set of information. This set provides information about which process is stored at this stack position and which command from this process should be executed next.

Example:
Having "**proc_stack**" set to `{1↦{prog↦1,cmd↦1}}` means, that the **MAIN** process is stored on the program-stack position 1 and the next command that should be executed from **MAIN** (or the "**proc_list**" of **MAIN**) is the command with number 1. Based on this information the command can be found within the "**proc_list**" of **MAIN** and executed accordingly.

Lastly a variable called "**stack_top**" is used to track the current top of stack.

During the extended **INITIALISATION** event, all new variables are initialized with their initial values.
For every available process, based on the constant "**proc_number**", a pair is added to both "**proc_list**" and "**cmd_amt**". At the beginning there are no commands added to any of the processes, so every process is paired with an empty set within "**proc_list**" and 0 within "**cmd_amt**".
Moreover, the machine starts in the programming-mode which means that "**play_mode**" is set to FALSE and "**prog_mode**" is set to TRUE. The program-stack "**proc_stack**" is initially pointing to the first command of the **MAIN** program and "**stack_top**" equals 1.

The first new event is called "**switch_mode**" and is responsible for switching between "**prog_mode**" and "**play_mode**". It consists of two actions that are fairly similar to the logic used within the "**light**" event.
There are two possibilities here:

a) play_mode = TRUE, prog_mode = FALSE – then the two expressions will switch play_mode to FALSE and prog_mode to TRUE

b) play_mode = FALSE, prog_mode = TRUE – then the two expressions will switch play_mode to TRUE and prog_mode to FALSE

The next event is called "**add_command**" and is responsible for adding available commands to each process within the "**proc_list**" variable. It makes use of two parameters called "**command**" and "**proc**", which are used to define the command and process to which the command should be added to. This event is only available when the programming mode is set to TRUE. Additional guards are making sure that the provided parameters are in fact available commands and processes, whether there is space to store the command in the selected process and whether the changes to "**proc_list**" will still satisfy the related invariant. The event itself consists of two actions. First the "**cmd_amt**" counter for the provided process is incremented by one. Second the new command is added to "**proc_list**", while using the current "**cmd_amt**" counter as an indicator what the next number in the "**proc_list**" sequence should be.

Next, all previously defined events (**move**, **jump**, **rotate_left**, **rotate_right** and **light**) are extended and new guards are added. Each event can only be executed when the "**play_mode**" variable is set to TRUE.
For each event it is necessary to check whether the next command specified for the related process at the "**stack_top**" position of the program-stack equals the events command. Which means, the "**jump**" event can only be executed when the current process at "**stack_top**" corresponds to a process where the number stored under "**prog**" shows a "**jump_c**" command within "**proc_list**" of this process.

Example:

Assuming the "**proc_list**" entry for the **MAIN** process is: {(1↦move_c),(2↦jump_c)}

a) if "**proc_stack**" is: {(1↦{(prog|->1),(cmd↦1)} and "**stack_top**" is 1, then the jump event cannot be executed as the next command for process **MAIN** (1) is "**move_c**"

b) if "**proc_stack**" is: {(1↦{(prog↦1),(cmd↦2)} and "**stack_top**" is 1, then the jump event can be executed as the next command for process **MAIN** (1) is "**jump_c**"

Any additional guards are making sure that all well-defindness obligations are satisfied so that all modifications to the program-stack entries are valid. The actions for all 5 events are the same. Next to executing the given command, as defined in the extended machines, it is necessary to modify the program-stack entry to reflect the fact that an action was executed. For this the "**cmd**" entry of the process currently at "**stack_top**" of "**proc_stack**" is incremented by 1.

Example:

- "**proc_stack**" before command/action execution: {(1↦{(prog↦1),(cmd↦**1**)})}
- "**proc_stack**" after command/action execution: {(1↦{(prog↦1),(cmd↦**2**)})}

In addition, it does not matter whether the "**cmd**" value is pointing to an existing entry within "**proc_list**" or not. In case there is no entry with this value for the related process within "**proc_list**", the machine will "terminate" (deadlock) as all possible commands were executed and no further actions are possible.

Lastly, there are 4 new processes that are needed when executing the "**proc1_c**" and "**proc2_c**" commands and when finishing the execution of all commands within **PROC1** and **PROC2**. In general, the events "**start_proc1**", "**start_proc2**", "**end_proc1**" and "**end_proc**", are defined the same way as the "basic" events for commands like "**move_c**", "**jump_c**" etc. The only difference is the modification performed on the program-stack "**proc_stack**" when such an event is executed. Now when executing the "**proc1_c**" or "**proc2_c**" commands, it must be ensured that not only the current "**cmd**" value of the current process at "**stack_top**" in "**proc_stack**" is incremented by 1 but also that a new entry pointing to new sub-process is added to the program-stack and that the "**stack_top**" pointer is also incremented by 1.

Example:

- "**proc_stack**" before execution of "**proc1_c**" command/ "**start_proc1**" event:
  {(1↦{(prog↦1),(cmd↦**1**)})}

- "**proc_stack**" after execution of "**proc1_c**" command / "**start_proc1**" event:
  {(1↦{(prog↦1),(cmd↦**2**)}),(2↦{(prog↦2),(cmd↦1)})}

Once all commands from **PROC1** or **PROC2** are executed and no recursion exists, it is necessary to remove the **PROC1** or **PROC2** entry from the program-stack and to decrement the "**stack_top**" pointer by 1.
This is done by the "**end_proc1**" and "**end_proc2**" events.

Example:

- "**proc_stack**" before execution of the "**end_proc1**" event:

  {(1↦{(prog↦1),(cmd↦2)}),(2↦{(prog↦2),(cmd↦1)})}

- "**proc_stack**" after execution of the "**end_proc1**" event:

  {(1↦{(prog↦1),(cmd↦2)})}

Decrementing the "**stack_top**" by one will ensure that processing will continue with the process that started the sub-process in the first place.

This concludes machine "**mac_2**".

Validation

In order to validate the created "Lightbot" model, an invariant was introduced in **mac_0** called "violation". This invariant, disabled by default, helps in providing a valid solution for a given level.

@validation ∃a,b,c·(a↦b)↦c ∈ lights_stat ∧ c=FALSE

Using an existential quantification, it is said that for some platform containing a light there is at least one light that is disabled. Obviously the goal of Lightbot is to enable all light, which in turn will violate the invariant above.

Please find below the Figures 1 to 3 that show the expected invariant violation when checking for invariant violations using the model checker on Level 3-1. In addition, Figure 4 shows that all proof obligations were proven prior submission.

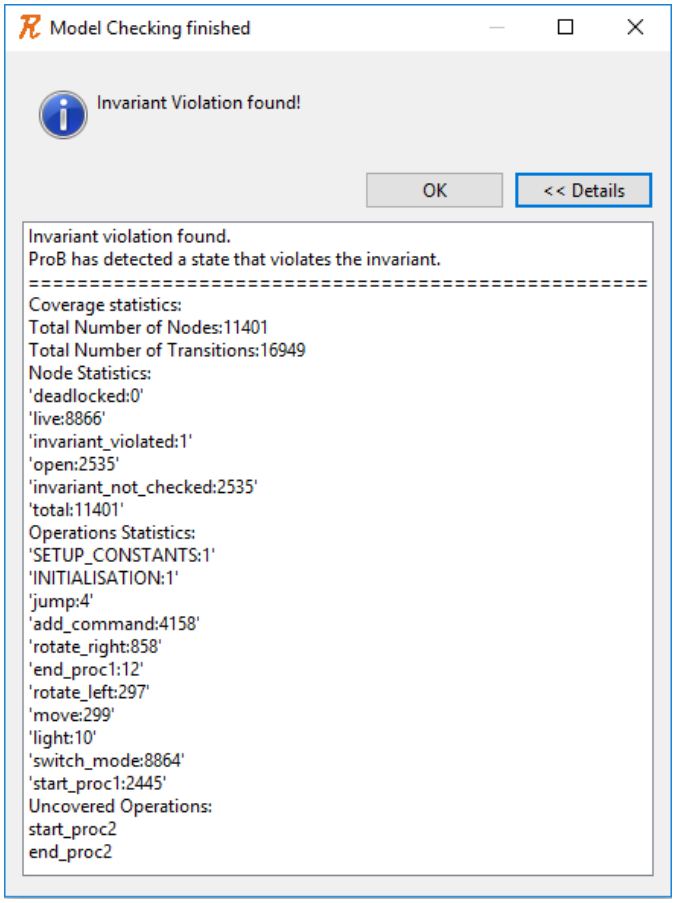

Figure 1 – Invariant Violation found using Model Checker

Figure 2 – Steps to invariant violation (Solution)

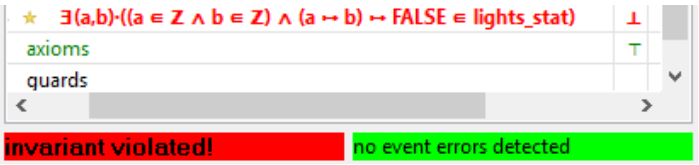

Figure 3 – Violated invariant

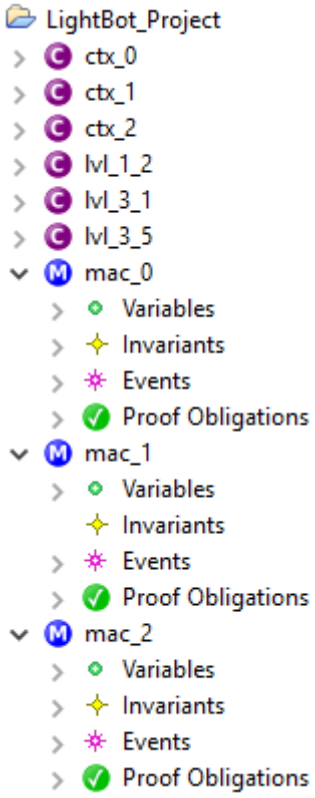

Figure 4 – All Proof Obligations were proven
(mostly by using Pro-B Disprover and Mono-Lemma Prover)