Student information

*Ivo Willemsen*

*851926289*

**System Verification and Testing / Report Assignment 1 + 2**

|  |  |
| --- | --- |
|  | Task 2.1 - Verification I have provided different UPPAAL xml definition files for several properties to be tested. Reason for this being, that some properties asked for more rigorous testing than just three floors and two elevators. It was also pointed out during the lecture that the scenario of 11 floors and 3 elevators would be tested to check whether the floor-to-elevator assignment algorithm works correctly. Also, the property which tests some time constrains requires more elaborate settings of constants in order to test the cases thoroughly.  Instead of describing or commenting on the formalization in the comment section of UPPAAL, I prefer to do it here as it is easier to read for the examiner. Property 1 File: elevator\_1236.xml  The elevator control unit is deadlock-free  A[] not deadlock  No comments Property 2 File: elevator\_1236.xml  The elevator never travels with its door open  A[]  (  not  (  (engine(0).moving\_up or engine(0).moving\_down)  and  (cabin\_door(0).opened or protecting\_door(0).opened)  )  )  This is clearly a safety property which should hold always globally. It could have been written by using an implication:  A[]  (  (engine(0).moving\_up or engine(0).moving\_down)  imply  (cabin\_door(0).closed and protecting\_door(0).closed)  )  But the first formalization matches the specification better. Property 3 File: elevator\_1236.xml  The amount of requests can reach the threshold value (in the description of Task 1 the value is 5) when the elevator is on the ground floor This is clearly an “EF” formula that needs to be applied, as it should hold possibly (can) eventually. The “when” could be misleading, but I interpreted the “when” as a “while”, i.e. a temporal indicator and not a causal indicator: *While* the elevator is on the floor, the amount of requests can reach the threshold value. E<>  (  (dgc\_request\_handler(0).len == (MAX\_SIZE)  and  (main\_control(0).current\_floor == 0))  ) Property 4 File: elevator\_4.xml  Each floor (except the ground floor) can be reached, and only by the correct elevator The default floor number generator is based on non-deterministic behavior by mean of a “select” UPPAAL statement. So, there can never be complete certainty regarding the visit of an elevator to a certain floor. Therefore, the default behavior of the button was changed and replaced by a deterministic rotating schema, where every floor number is generated at least once. In that case, the following formula suffices:  A[]  (  (main\_control(0).dest\_reached imply  (main\_control(0).current\_floor == 0 or  (main\_control(0).current\_floor >= 1 and  main\_control(0).current\_floor <= 3 and  button.generated[main\_control(0).current\_floor] == true))) and  (main\_control(1).dest\_reached imply  (main\_control(1).current\_floor == 0 or  (main\_control(1).current\_floor >= 4 and  main\_control(1).current\_floor <= 7 and  button.generated[main\_control(1).current\_floor] == true))) and  (main\_control(2).dest\_reached imply  (main\_control(2).current\_floor == 0 or  (main\_control(2).current\_floor >= 8 and  main\_control(2).current\_floor <= 10 and  button.generated[main\_control(2).current\_floor] == true)))  )  The red-highlighted lines are not necessary, because the deterministic rotating schema ensures monotonically increasing floor number, without skipping. But for completeness reasons, an array of Booleans was created, with the index being the floor number. The respective index is made true every time the floor number is generated and checked in the formula. Also note that trips to floor 0 are ignored.  Although an “AG” formula is used, the semantics are the same. The above stated, formally checks that always and globally, **IF** the destination is reached, the correct floor is involved. Property 5 File: elevator\_5.xml  All floor requests are eventually served, i.e., the elevator reaches its destination.  This formula on the next page checks the liveness property in question: When a request is made, then it needs to be served. Basically, it checks that no requests are lost in the process.  In the setup, there are 4 floors and 2 elevators. Floors 0 and 1 are served by elevator 1 and floors 2 and 3 by elevator 2. First, just as with property 4, the non-deterministic floor number generator (button template) was replaced by a monotonically increasing, deterministic floor number generating. But for another reason. The reason here is that two subsequent requests are detected by the main controller and the latter is skipped. Another option would have been changing the main controller template.  (  ( dgc\_request\_handler(0).request\_received and  (  dgc\_request\_handler(0).req\_floor == 0 or  dgc\_request\_handler(0).req\_floor == 1  )  ) or  (  dgc\_request\_handler(1).request\_received and  (  dgc\_request\_handler(1).req\_floor == 2 or  dgc\_request\_handler(1).req\_floor == 3  )  )  )  -->  (  main\_control(0).dest\_reached and  (main\_control(0).current\_floor == 0 or  main\_control(0).current\_floor == 1)  or  main\_control(1).dest\_reached and  (main\_control(1).current\_floor == 2 or  main\_control(1).current\_floor == 3)  ) Property 6 File: elevator\_1236.xml  Whenever a request is served, the passengers will be able to leave the cabin  At the moment when the request is served, i.e. when the request handler has received the *served?* action from the main controller, at that moment, all doors should still be open. Still, because the doors have already been opened just before the main controller send the served! action to the request handler. The following formula checks the model that is configured with 3 floors and 2 elevators:  A[]  (  (dgc\_request\_handler(0).finished imply  (protecting\_door(0).opened and cabin\_door(0).opened))  or  (dgc\_request\_handler(1).finished imply  (protecting\_door(1).opened and cabin\_door(1).opened))  or  (dgc\_request\_handler(2).finished imply  (protecting\_door(2).opened and cabin\_door(2).opened))  ) Property 7 File: elevator\_7.xml  The time needed to serve a request is bounded by the product of the distance in number of floors by the time needed to travel the distance of one floor plus twice the time it takes to open or close the door. More formally the property is defined as follows. Let p be the time to open or close the door. Let t be the time needed to cover the distance between two floors. Let s be the service time. Let d be the distance in number of floors. Then, the property is expressed as:  s <= d \* t + 2 \* p  The setup in the .xml file takes care of a system configured with 25 floors, 1 elevator (not relevant), a door time of 4 and a floor time of 3.  A[]  (  main\_control(0).handle\_served  imply  main\_control(0).s <=  (main\_control(0).floor\_distance \* FLOOR\_TIME) + (2 \* DOOR\_TIME)  )  In order to test the formula, the model had to be adjusted in the main controller:   * A variable was introduced that holds the *floor\_distance* to be travelled * A timer called *s* was created, this variable is reset after a *go* is received * A function was created, called *set\_floor\_distance* that sets the *floor\_distance*   When the state *handle\_served* is reached, it is implied that the formula *s <= d \* t + 2 \* p* holds Task 2.2 - Validation Vaandrager’s article mentions several aspects that are characteristics of a good model. They will be covered in sequential order. Please use the .xml file *elevator\_1236.xml* as a reference. Due to the layout of this report, a bit more than a page is used to express my ideas.  **Object of Modeling**  This boils down to naming your entities, attributes and functions in a succinct and non-ambiguous manner, with the goal of conveying the object to be modelled. Care has been taken in my model to give proper names to locations and to use composite, meaningful functions in the transitions, instead of tuple of separate attributes mutations or checks. Crucial parts of the model should be documented if needed. For example, the algorithm that assigs a floor to an elevator can be written in just a few lines, but a dry-test in comments can give the reader a better understanding. I think that the UPPAAL model alone is not enough in order to explain the object of modeling. The description text in the workbook is also necessary to get a good view of what the object is. The most important template of the model is the main controller. An interested reader is already attracted to the word “main”, which implies that it is the most important or entry point of the model. Glancing through the locations and transitions of the main controller gives the reader a good idea about the object of modeling.  **Traceability**  The structural elements of the model are: locations, transitions, constants, variables, functions, modules, templates and data types. Only data types, constants and variables are primitives, but the rest of the elements link at least one element with another element. Especially functions and transitions should be documented, if needed. I’ve documented both functions (extensively) and transitions (where needed) in order to promote the traceability in the model. Vaandrager mentions that fact that some aspects of the model are considered *common knowledge*, or *domain knowledge* of the environment, which is assumed to be implicitly known to the reader. For example, the function of an engine is to move the elevator. This is not explicitly described but assumed implicit knowledge.  **Truth**  The UPPAAL model that is presented should adhere to the truth: It should implement the specification as intended without errors. The model-checking part is performed through formalization of specification rules in the verifier section of the tool. Special attention is needed with regards to the simplification that I performed related to the protecting doors. The specification stated that each floor has a separate protecting door (for each elevator). But due to performance reasons (state-space reduction), it is better to model it in a different way, so that there is just one protecting door per elevator. It is not possible for two protecting doors to be accessed by the same elevator at the same time. This reasoning justifies the simplification that I performed. I’ve documented this in the cabin door template.  **Simplicity**  Special attention has been given to increasing the simplicity. Increasing simplicity leads to a minimal set of locations (and transitions). For example, after an elevator takes off, it should return to the base floor. This has been modeled in such a way that the extra final trip is not part of the main controller, but by adding an extra request to the queue. This way, the number of locations is limited a the expensive of just an increase of a call to a function. Another example of simplification that was performed was the shortcut from *request\_received* to *handle\_served* in the main controller, in case two subsequent request are to the same floor: It would be waste of cycles and state to have the elevator make the trip for nothing. It’s not really a simplification, because it resembles the reality.  Too much simplification can also have adversary effects: In the main controller it is tempted to make the stopped location the initial one and thus saving two locations (*idle* and *initial\_door\_opening*). But in order to make it work, an extra variable and transition needs to be created. This was tested and the actual state-space was increased. So, it was decided to not perform simplification.  **Extensibility and usability**  The designed model is mostly parametrizable: Constants in the general declaration sections and parametrized templates have been used in order to be flexible with regards to dimensional changes.  **Interoperability and Semantics Sharing**  In my opinion, this aspect is a bit vague and it’s the least useful. I’ve not only tried to implement the model with UPPAAL, but also gave it a try with implementing the model in NuSMV. Of course, when implementing a model, an extensive search to already existing models is first performed, in order to collect good ideas, look for design patterns and best practices. But when implementing a model in a different setting, concessions need to be made because of restrictions in environment and or programming language. Notwithstanding, the original gist needs to be respected. |
|  | Task 2.3 – Scaling It needs to be noted that I performed the scaling while going through the modeling exercise 1 and making the adjustments in exercise 2. So, the version that I had when getting to task 3 of exercise 2 was already the most performant model. So, I will reverse the question. The first paragraph will present my most performant model, with all performance iterations performed. The second paragraph I will give the statistics for a version that I created, where I removed all performance upgrades that I have performed throughout the exercises.  **Most optimal configuration**  The following properties where used when reporting on scaling:   * Property 2: The elevator never travels with its door open * Property 5: All floor requests are eventually served, i.e., the elevator reaches its destination.   Note that property 2 only performs the check for the first elevator, whereas *property 5 performs the check for the complete powerset of elevators and floors*. This should be taken into considerations when comparing performance results.  The original configuration of the following attributes was respected:   * const int MAX\_SIZE = 5 * const int DOOR\_TIME = 1 * const int FLOOR\_TIME = 1 * const int MAX\_WAIT = 5 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Config** |  | **Property 2** | | **Property 5** | |
| 2 E\* / 3 F\* | File | Time (s) | MPWLL\* | Time (s) | MPWLL\* |
| opt\_2E\_2F.xml | 0.2 | N/A\* | 0.4 | N/A\* |
| 4 E / 4 F | File | Time | MPWLL | Time | MPWLL |
| opt\_4E\_4E.xml | 30 | 21K | 71 | 25K |
| 3 E / 11 F | File | Time | MPWLL | Time | MPWLL |
| opt\_3E\_11F.xml | 790 | 490K |  |  |

\*F = Floors

\*E = Elevators

\*MPWLL = Max post-waiting list load

N/A = Not reported by UPPAAL, possible because it was too low