Telecom Paris
COMELEC Department

AVATAR Model-Checker

	Document Manager	**Contributors**	**Checked by**
Name	Ludovic APVRILLE	Ludovic APVRILLE	
Contact	ludovic.apvrille@telecom-paris.fr	Alessandro TEMPIA CALVINO	
Date	September 7, 2020		

Contents

1 Preface

1.1 Table of Versions

Version	**Date**	**Description & Rationale of Modifications**	**Sections Modified**
1.0	23/03/2020	First draft	

1.2 Table of References and Applicable Documents

Reference	**Title & Edition**	**Author or Editor**	**Year**

1.3 Acronyms and glossary

Term	**Description**

1.4 Executive Summary

This document describes how the AVATAR model checker of TTool works. This document is divided mainly into two parts. The first one is the user guide on how to use the model-checker. The second one explains the data structure used to represent the models, the algorithm for reachability graph generation, and the properties that can be verified.

2 A First Example

This very first example explains how to verify reachability, liveness, and safety properties using the AVATAR model-checker on a design model.

2.1 Getting the example

Be sure to get the latest version of TTool including the remote loading of models (June 2020 and after). Do: File, Open from TTool repository, and select "PressureController.xml". Then, open the design panel to view the AVATAR design model.

2.2 Understanding the model

The Pressure Controller is built upon a set of blocks representing the system itself, and two blocks representing the environment (the pressure sensor, and the alarm actuator), see Figure 1.

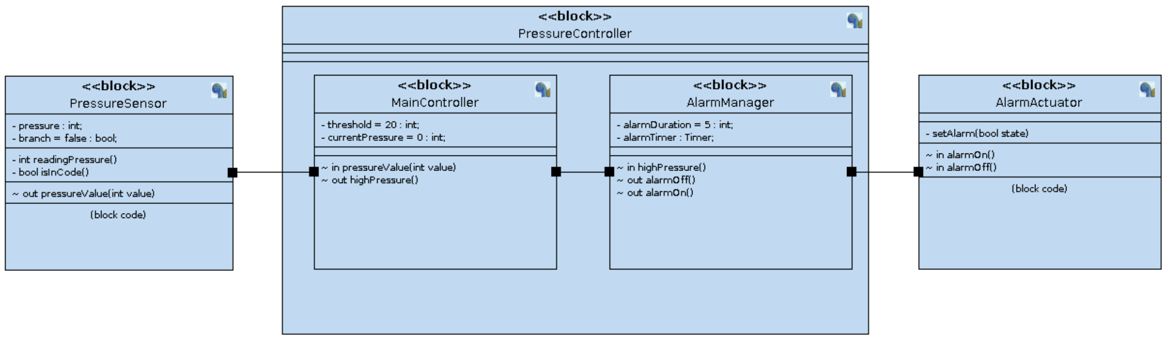


Figure 1: Pressure Controller System: Avatar Design

2.2.1 PressureSensor

The pressure sensor (see Figure 2) monitors the pressure with a period of 20 units of time. In the first branch, "IsInCode" method is **not** executed when the model is considered for functional simulation or formal verification. Indeed, in our case, the "branch" boolean is set to false by default, so the random command is executed (and not the readingPressure() method). The random value, between 18 and 21, is sent every period of time to the main controller.

2.2.2 MainController

The main controller (see Figure 3) receives the pressure value from the pressure sensor. If the value is greater equal than a threshold, set at 20, a high pressure signal is sent to the alarm manager.

2.2.3 AlarmManager

The alarm manager (see Figure 4) activates an alarm when the controller senses a high pressure. The alarm is deactivated after "alarmDuration", controlled by a timer, if no other high pressure signals are sensed in the meanwhile.

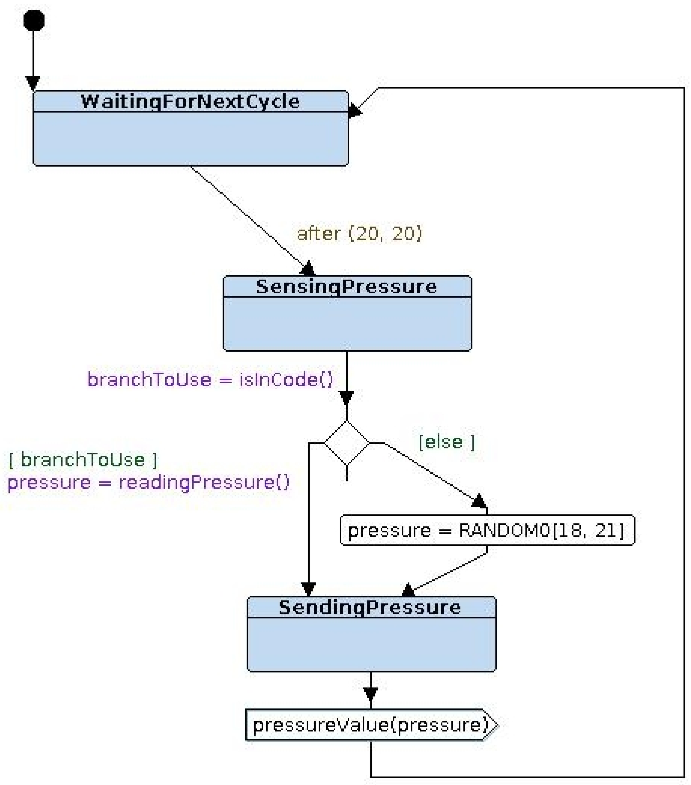


Figure 2: Pressure Sensor

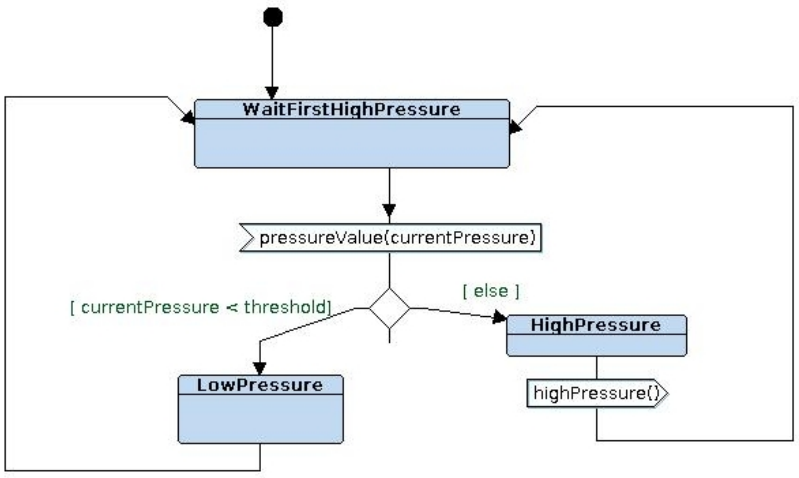


Figure 3: Main Controller

2.2.4 AlarmActuator

The alarm actuator may receive on and off signals activating and deactivating the alarm.

2.3 Reachability and liveness

To activate the reachability and liveness check for specific states, we need to go to the correspondant block state machine, right click on a state or a signal and select "Check for Reachability / Liveness". For instance, let's do that on the *AlarmIsOn* state in the AlarmManager state machine. After this operation, the letters R and L in grey will appear

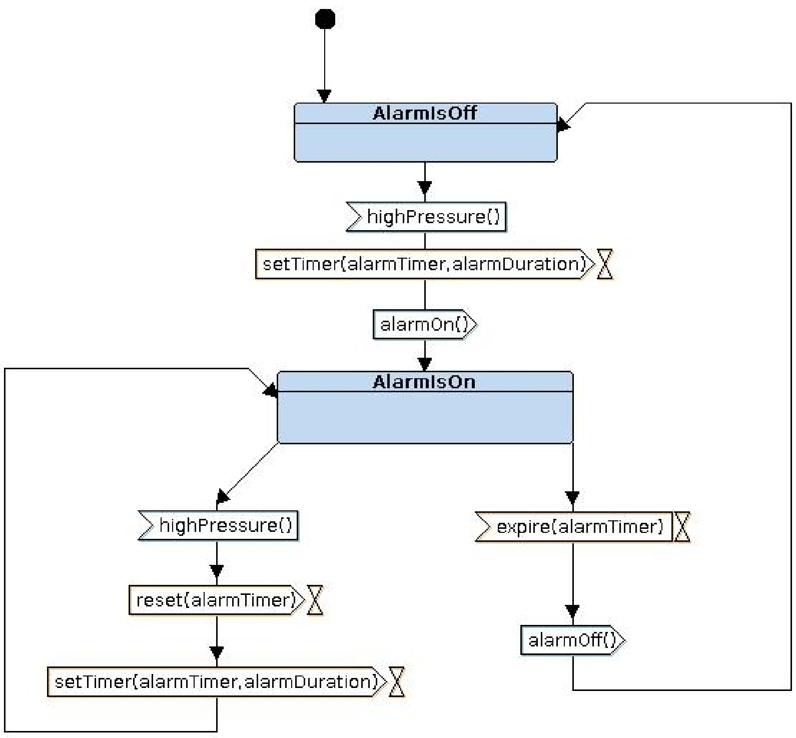


Figure 4: AlarmManager

next to the state (see Figure 5) to confirm that the state is selected for the reachability and liveness check. After that, we may proceed with the verification.

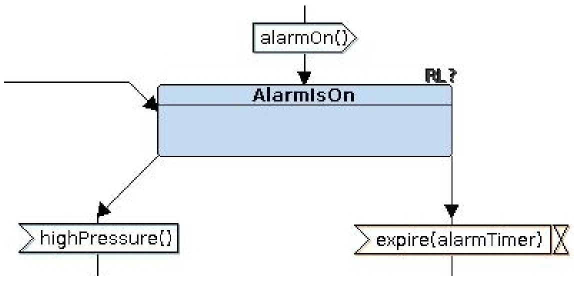
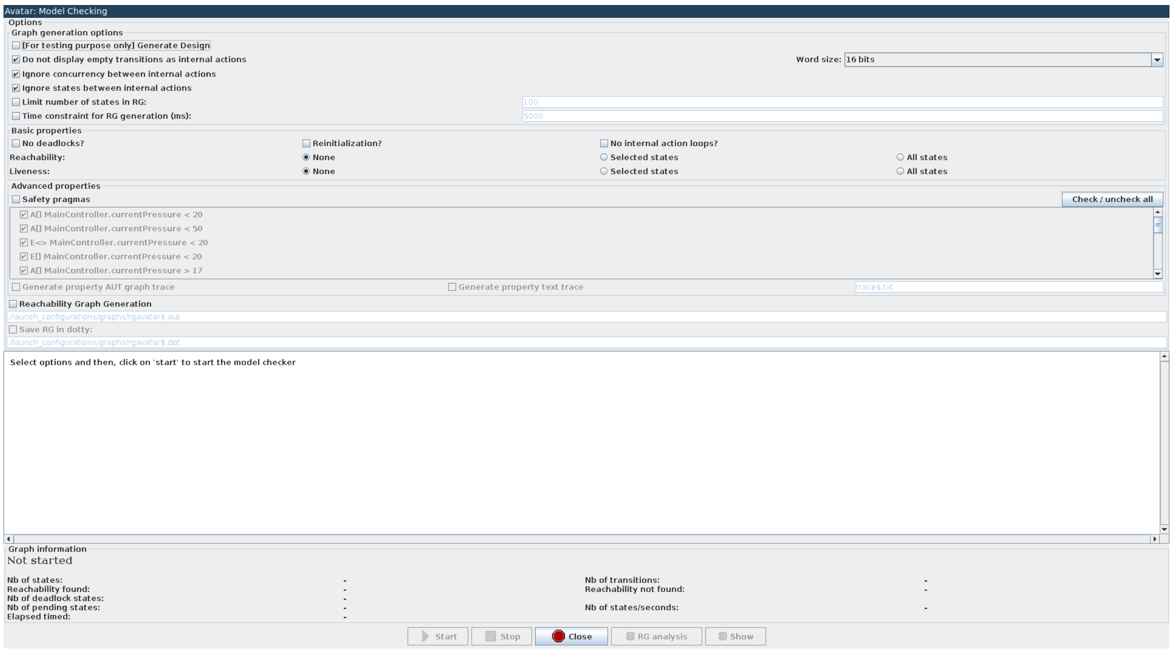


Figure 5: State selected for reachability and liveness check

To open the model-checker window, we must first compile the model. To do that, let's click on the "syntax analysis" icon and then on "check syntax". Then, we click on the "safety verification (internal tool)" icon representing a screwdriver and a wrench crossed with the text "RC". The following window should open (see Figure 6).
To execute the reachability and liveness check on the selected state, let's select "Selected states" in both Reachability and Liveness lines. Then, we can press the start button. The text area will contain the verification report indicating if the properties are satisfied or not. Moreover, if we go back to the AlarmManager state machine, the RL text next to the state is now colored with green for *satisfied* and in red for *not satisfied* (see Figure 7).
In this case, the reachability is satisfied as the pressure sensor may detect a temperature higher than the controller threshold. The liveness, instead, is not satisfied as a temperature higher than the threshold may be never detected.



Avatar: Model Checking

Options

Graph generation options

- ☐ [For testing purpose only] Generate Design
- ☑ Do not display empty transitions as internal actions
- ☑ Ignore concurrency between internal actions
- ☑ Ignore states between internal actions
- ☐ Limit number of states in RG: 100
- ☐ Time constraint for RG generation (ms): 5000

Word size: 16 bits

Basic properties

- ☐ No deadlocks?
- ☐ Reinitialization?
- ☐ No internal action loops?

Reachability:	[✓ None]	Selected states	All states
Liveness:	[✓ None]	Selected states	All states

Advanced properties

- ☐ Safety pragmas (Check / uncheck all)
  - ☑ A[] MainController.currentPressure < 20
  - ☑ A[] MainController.currentPressure < 50
  - ☑ E<> MainController.currentPressure < 20
  - ☑ E[] MainController.currentPressure < 20
  - ☑ A[] MainController.currentPressure > 17
- ☐ Generate property AUT graph trace
- ☐ Generate property text trace: trace$.txt

- ☐ Reachability Graph Generation: ./launch_configurations/graphs/rgavatar$.aut
- ☐ Save RG in dotty: ./launch_configurations/graphs/rgavatar$.dot

```
Select options and then, click on 'start' to start the model checker
```

Graph information

Not started

Nb of states:	-	Nb of transitions:	-
Reachability found:	-	Reachability not found:	-
Nb of deadlock states:	-		
Nb of pending states:	-	Nb of states/seconds:	-
Elapsed timed:	-		

Figure 6: Model-checker window

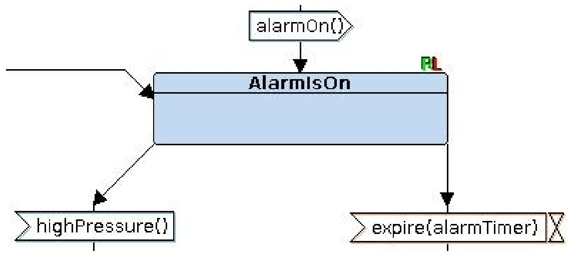


Figure 7: Reachability and Liveness result

If we want to simply verify the reachability and liveness property automatically for all the states, we can select "All states" in the model-checking window.

2.4 Safety properties

We can find, write and modify safety properties in the block diagram tab inside the pink block. The pink block can be created by clicking on the "Safety property button" as shown in figure 8.

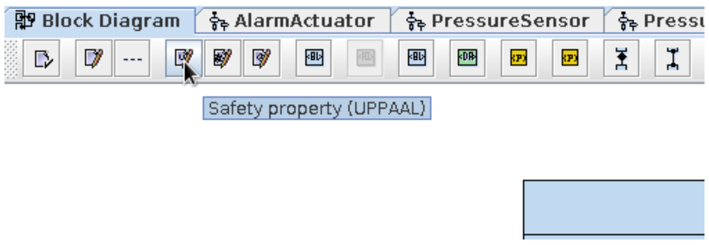
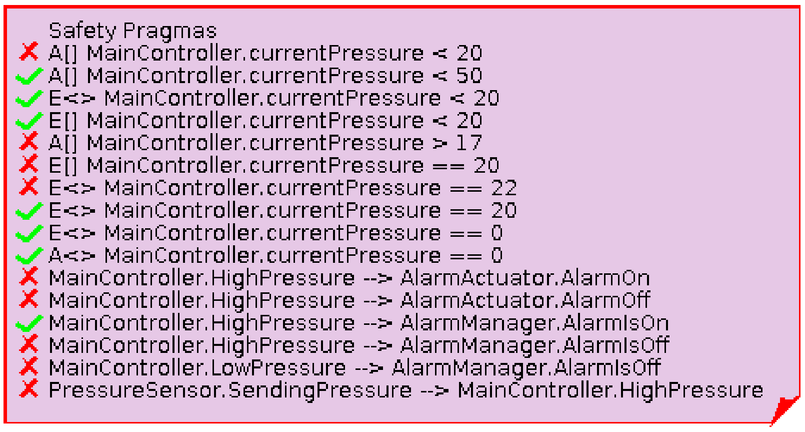


Figure 8: Safety properties creation

If we do a double click on the pink box, we see the safety and liveness pragmas written as a CTL formula using UPPAAL syntax. If we press the question mark in the window, a short illustrated guide on how the queries work appears.

To start the CTL query verification, let's compile again and let's open the model-checker window. As shown in figure 6, we want to select the checkbox "Safety pragmas" and then we select the queries that we want to verify. In this case, let's execute all of them. Let's press the start button to begin the verification. The query results are shown both in the report field and inside the pink box in the block diagram tab (see Picture 9).



```
  Safety Pragmas
✗ A[] MainController.currentPressure < 20
✓ A[] MainController.currentPressure < 50
✓ E<> MainController.currentPressure < 20
✓ E[] MainController.currentPressure < 20
✗ A[] MainController.currentPressure > 17
✗ E[] MainController.currentPressure == 20
✗ E<> MainController.currentPressure == 22
✓ E<> MainController.currentPressure == 20
✓ E<> MainController.currentPressure == 0
✓ A<> MainController.currentPressure == 0
✗ MainController.HighPressure --> AlarmActuator.AlarmOn
✗ MainController.HighPressure --> AlarmActuator.AlarmOff
✓ MainController.HighPressure --> AlarmManager.AlarmIsOn
✗ MainController.HighPressure --> AlarmManager.AlarmIsOff
✗ MainController.LowPressure --> AlarmManager.AlarmIsOff
✗ PressureSensor.SendingPressure --> MainController.HighPressure
```

Figure 9: Safety Result

3 Verification Guide

In this section we will explain all the features and options that are present in the AVATAR model-checker. For a reference to the model-checking window, please refer to figure 6.

The model-checker allows to generate a reachability graph and to prove properties. Starting from the top, the first section contains graph generations options. In particular:

- **Word size**: the words can be represented on 32, 16 or 8 bits.
- **Do not display empty transitions as internal states**: in the reachability graph generation, states are not created for empty transitions (no action, no time, no choice). In practice, empty transition states are collapsed together. This option does not affect the verification. Option on by default.
- **Ignore concurrency between internal actions**: this option allows to ignore a full concurrency study. It is very useful to limit the state explosion of the model due to all the concurrencies among its states. It is so suggested to keep it selected to have a faster verification. This option, however, affects only in the *leadsto* study, and the reachability graph generation. **Important**: basic concurrency it is evaluated also with this optimization selected. Basic concurrency means that concurrency is normally evaluated until states with multiple transitions, or if statement, or signals communication are encounted. In normal usage, you should not need to deselect this option (example in the next section 3.3). Option on by default.
- **Ignore states between internal actions**: This option allows to ignore intermediate states between actions that can be directly executed. this optimization helps decreasing the number of states in the reachability graph generation. If *Ignore concurrency between internal actions* is not selected, this option automatically deactivated. We suggest to keep this option selected for verification as it would not affect the result. Option on by default.
- **Limit the number of states in RG**: it limits the reachability graph generation to a fixed number of states. It is particularly useful for big models. This option is automatically ignored when a verification study is selected.
- **Time constraint for RG generation**: it allows to stop the reachability graph generation after a fixed number of milliseconds. It is particularly useful for big models. This option is automatically ignored when a verification study is selected.

The next section contains options for basic properties verification:

- **No deadlocks**: it checks that the model is deadlock free
- **Reinitialization**: it checks that, for all the paths, the model will eventually restart to the initial state
- **No internal action loops**: it checks that in states machines, there are not possible infinite loops which have no interractions (signal communications) with the environment (other blocks). This check helps to detect if a system stops interracting with the environment during its execution.

- **Reachability**: it checks for states reachability
- **Liveness**: it checks for states liveness

The next section allows us to execute safety and liveness pragmas written as a CTL formula. Each of them can be selected or not for verification. Also, counterexample traces can be generated for each CTL query and deadlock check. The option **Generate counterexample traces** generates a trace in a text file for each pragma with a counterexample. The trace contains all the states and transitions that lead to a counterexample step by step. Selecting **Generate counterexample AUT graphs**, a displayable graph will appear in the reachability graphs section. A detailed example is available in section 3.2.

Then, **Reachability Graph Generation** options is used to generate and save the reachability graph. Moreover, it can can be saved also in "dotty" format.

3.1 CTL query syntax

The query syntax for CTL formulas, for expressions p and q, is the following:

- **A[] p**: property p is always true for each path (other common notation **AG p**)
- **A<> p**: property p will eventually be true for each path (other common notation **AF p**)
- **E[] p**: there exists at least one path in which property p is always true (other common notation **EG p**)
- **E<> p**: there exists a path in which property p will eventually be true (other common notation **EF p**)
- **p –> q**: whenever p is true, then q will eventually be true at some subsequent moment (other common notation **G(p ⇒ Fq)**)

Expression p and q may contain condition on states and/or variables. The format used to reference a variable or a state of a block is *BlockName.AttributeName* or *BlockName.StateName*. Conditions on variables can be combined using classic boolean (||, &&, and, or) and integer (==, >, >=, <, <=, +, -, *, /) operators. Variables can be negated or inverted (not(var), !(var), -var). States are considered as a boolean variable.

Before the pragma, the letter *T* or *F* may be inserted to specify if the expected result of the pragma is true, in the former case, or false, in the latter. The pragma will be considered satisfied if the result is equivalent to the expected one.

Here there are few examples of valid pragmas:

- *E<> Passenger.isInCockpit ==true && DoorAndLockButton.inside==1*: there exists a path in which there is at least one state in which Passenger.isInCockpit is true and DoorAndLockButton.inside is equal to 1
- *F A[] MainController.currentPressure < 25*: MainController.currentPressure is always less than 25. The expected result is false.

- *DoorAndLockButton.IN_EMERGENCY_CALL –> DoorAndLockButton.CLOSED_AND_LOCKED* || *DoorAndLockButton.CLOSED_AND_UNLOCKED*: whenever DoorAndLockButton.IN_EMERGENCY_CALL state is encounted, then or DoorAndLockButton.CLOSED_AND_LOCKED or DoorAndLockButton.CLOSED_AND_UNLOCKED will be encounted at some subsequent moment.

- *T A<> Sensor.iter == Main.dataSize - 1*: all the paths reach the condition where Sensor.iter is equal to Main.dataSize - 1.

3.2 Traces generation

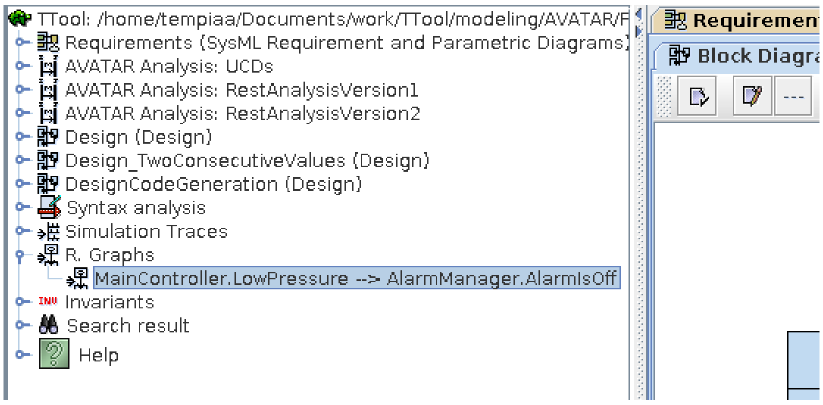
For advanced verification queries and deadlock check, there is the possibility of generating counterexample traces that show the path to prove or disprove a property. In particular:

- **A[] p**: shows the path that disproves the property
- **A<> p**: shows the path that disproves the property
- **E[] p**: shows the path that proves the property
- **E<> p**: shows the path that proves the property
- **p –> q**: shows the path that disproves the property starting from a state with p valid
- **No deadlocks?**: shows a path to a deadlock
- **No internal action loops?**: shows an infinite loop that has no interraction with the environment

The **Generate property AUT graphs trace** option allows us to display the propery traces as a reachability graph, saving it in AUT format. An entry to the graph, with the name of the query, will appear in the *R. Graphs* section in TTool.

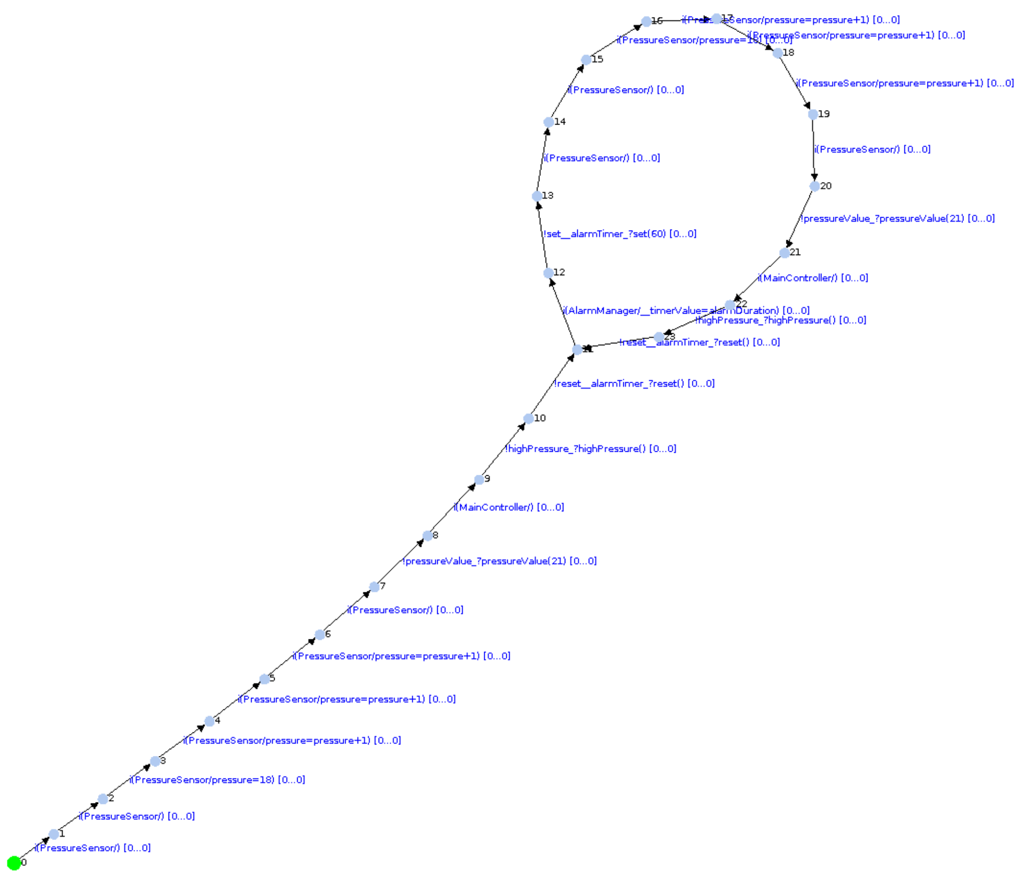
The **Generate property text trace** option writes, in the file indicated in the text field on the right, all the steps to reach a property proof. State lines show the current state in the state machine for each block in the system. Transition lines show details of the transition taken to reach the next state.

Let's do a guidated example to traces generation. Let's open again the "PressureController.xml" model used in the previous section. Let's execute a verification of the only query *MainController.LowPressure –> AlarmManager.AlarmIsOff* activating both the options *Generate property AUT graphs trace* and *Generate property text trace*. Let's press start to execute the verification. If now we open the file *trace*.txt* in which the trace is saved, we can see the name of the trace in the first line, and then states and transitions alternating until the counterexample state that disprove the property. The first state has the state machine pointed to MainController.LowPressure (or to a direct child like MainController.WaitFirstHighPressure). The last state indicates a loop or a state for which the property is proven to be never satisfied. The AUT graph is visible with the name of the query in the Reachability graphs section as shown in Figure 10. Doing right click on the graph and pressing show, we are able to see graphically the trace (see Figure 11).



TTool: /home/tempiaa/Documents/work/TTool/modeling/AVATAR/F
- Requirements (SysML Requirement and Parametric Diagrams)
- AVATAR Analysis: UCDs
- AVATAR Analysis: RestAnalysisVersion1
- AVATAR Analysis: RestAnalysisVersion2
- Design (Design)
- Design_TwoConsecutiveValues (Design)
- DesignCodeGeneration (Design)
- Syntax analysis
- Simulation Traces
- R. Graphs
  - MainController.LowPressure --> AlarmManager.AlarmIsOff
- Invariants
- Search result
- Help

Figure 10: Link to the AUT graph generated on query



0
i(PressureSensor/) [0...0]
1
i(PressureSensor/) [0...0]
2
i(PressureSensor/pressure=18) [0...0]
3
i(PressureSensor/pressure=pressure+1) [0...0]
4
i(PressureSensor/pressure=pressure+1) [0...0]
5
i(PressureSensor/pressure=pressure+1) [0...0]
6
i(PressureSensor/) [0...0]
7
!pressureValue_?pressureValue(21) [0...0]
8
i(MainController/) [0...0]
9
!highPressure_?highPressure() [0...0]
10
!reset__alarmTimer_?reset() [0...0]
11
i(AlarmManager/__timerValue=alarmDuration) [0...0]
12
!set__alarmTimer_?set(60) [0...0]
13
i(PressureSensor/) [0...0]
14
i(PressureSensor/) [0...0]
15
i(PressureSensor/pressure=18) [0...0]
16
i(PressureSensor/pressure=pressure+1) [0...0]
17
i(PressureSensor/pressure=pressure+1) [0...0]
18
i(PressureSensor/pressure=pressure+1) [0...0]
19
i(PressureSensor/) [0...0]
20
!pressureValue_?pressureValue(21) [0...0]
21
i(MainController/) [0...0]
22
!highPressure_?highPressure() [0...0]
23
!reset__alarmTimer_?reset() [0...0]

Figure 11: AUT graph generated on query

3.3 Concurrency behavior

In section 3, we discussed about the option *Ignore concurrency between internal actions* and how it could effect the model-checker behavior. In this section, we will show with a guided example the differences between having this option on or off.

In this simplified example (see Figure 12), we have two blocks: a sensor and a main block that receive the informations. Let's consider the two blocks connected through a synchronous channel. Also with asynchronous there would not be any difference in this

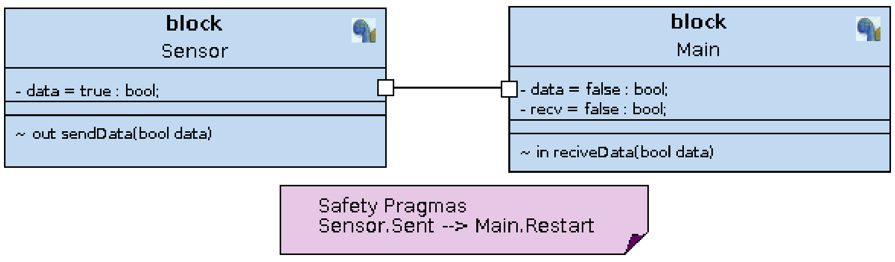
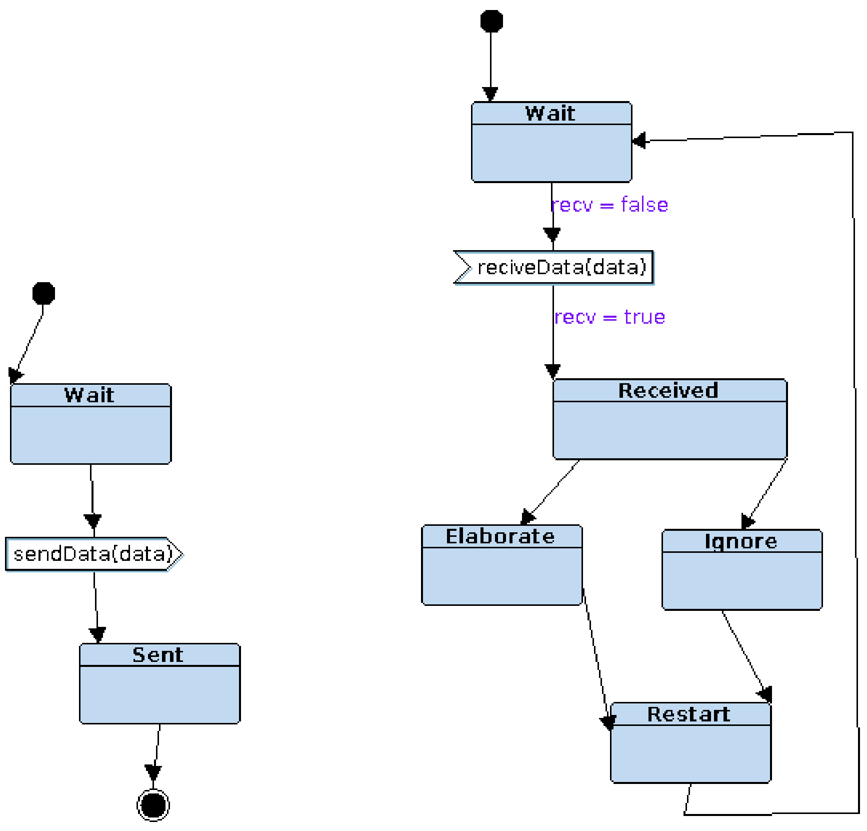


Figure 12: Example on concurrency

analysis. The two states machine are shown in figure 13.



(a) (b)

Figure 13: Sensor (a) and Main (b) state machines

First, we want to prove that after Sensor.Sent, Main will eventually reach the Receive state (*Sensor.Sent –> Main.Receive*). After the data is sent by the sensor and received by the main block, the is a concurrency between executing first the Sent state or the Received state. In this case, both with the option *Ignore concurrency between internal actions* selected or not, the pragma result will be always false. In fact, the Received state may be reached before Sent.
Different is the case for the following pragma: *Sensor.Sent –> Main.Restart*. In this case, after the Received state, there are two possible choices: Elaborate and Ignore. If the

option *Ignore concurrency between internal actions* is selected, the single transitions have priority over the choices. This will lead to Sent being reached before Elaborate or Ignore. The pragma will be true. If the option is not selected instead, a full concurrency study is executed and the pragma will result correctly false. Note that a full concurrency study will considerably slow down the verification performance.

The best way to avoid concurrency problems and to keep this option always selected is to ask concurrency aware queries. If we want to check if something happens after sending the data, the best choice would be to query *Sensor.Wait –> Main.Restart*. This query does not depend by concurrency since it happens before a "synchronization" point.

4 AVATAR Model-Checker

4.1 Introduction

The model-checker is used to generate a reachability graph starting from an AVATAR model. It can be also used to check the reachability and liveness on a list of selected states. The model checker is contained inside the package `avatartranslator.modelchecker`.

4.2 Reachability Graph

The model-checker constructor takes an Avatar specification as input. The main method used for the graph generation is `startModelChecking()`. This method is responsible for preparing the data structure for the main algorithm to be executed. In particular, it runs the following operations:

- Remove else guards, timers, composite states, randoms from the state machine of blocks in the Avatar specification
- Prepare the states
- Prepare the transitions
- Run `startModelChecking(nbThreads)`

The states are prepared inside the method `prepareStates()`. For each block inside the avatar specification, the method extracts all the states definitions (instances of `AvatarStateElement`) from the list of state machine elements `elements` saving them in the array `allStates`.

Transition between states of the state machines, instead, are prepared inside the method `prepareTransitions()`. This method is responsible for storing the type of transaction based on the type of the following state they address. The method is executed over all the blocks of the specification. The transitions are differentiated into the following categories:

- TYPE_RECV_SYNC
- TYPE_SEND_SYNC
- TYPE_ACTION_AND_METHOD
- TYPE_ACTIONONLY
- TYPE_METHODONLY
- TYPE_EMPTY

Then the number of available processors is stored and passed to the next method `startModelChecking(nbThreads)`.

4.3 Main algorithm preparation

The reachability graph generation starts in the method `startModelChecking(nbThreads)`. In the first part, the graph data structures are initialized:

- `states`: map used to store the states of the reachability graph (`SpecificationState`), mapped by the hash of the state
- `statesByID`: map used to store the states of the reachability graph (`SpecificationState`), mapped by the ID (incremented every time a state is created)
- `pendingStates`: list of graph states that can be executed during the current iteration of the algorithm

The initial state of the reachability graph is created. A (`SpecificationState` saves all the current configuration of the blocks and state machines. This is reached wrapping the Avatar blocks into specification blocks. SpecificationBlock adds an integer array saving:

- **State**: it points the current state of the state machine of the wrapped block in `allStates`
- **Clock_min**: minimum value of the current clock used as a lower bound to extract the executable transitions (time domain)
- **Clock_max**: maximum value of the current clock used as a higher bound to extract the executable transitions (time domain)
- **Attributes**: values of block variables

The initial state is initialized storing the specification blocks for each Avatar Block. Specification blocks are initialized with the start state, clock at 0 and with initial variables value.

The method `handleNonEmptyUniqueTransition()` is used to increase the current state of specification block until not empty unique transitions are found. For instance, if from the initial state of the state machine there is only one empty (true guard, no time, no signal, no action) transition to another state, it can be directly executed since it doesn't have any dependency. Executing first these transitions helps to decrease the number of states created in the reachability graph.

Then a hash for the state is created. All the specification block states values (state, clock, values) are stored together in an array called `hash`. For this array, a hash number is calculated. The initial state is then inserted inside the maps `states`, `statesByID` and inside the `pendingStates` list.

Then the method `computeAllStates()` is called to run the main algorithm on multiple threads.

4.4 The Main Algorithm

The main loop of the algorithm is run in parallel by multiple threads in the method `run()`. The main loop executes the following actions:

- Pick-up a state from the pending state queue
- Prepare the transitions from the selected state
- Execute the valid available transitions
- Gather the next possible states
- Create a link in the reachability graph between current and new states
- Insert the new states in the pending list

A state is picked up with the method `pickupState()`. The thread waits for an available state to be processed in the pendent queue.

The main method for the application of the algorithm is `computeAllStatesFrom(SpecificationState)`. First it prepares the transitions from the current state with the method `prepareTransitionsOfState(specificationState)`. For a specification state, it creates an array `transitions` containing all the possible transitions that could be executed from the current specification state, i.e. from the states pointed by the specification blocks for each state machine, wrapped in specification transitions.
The method `handleAvatarTransition` checks if a transition can be executed at the current state. First, it checks if the guard condition is satisfied. Then it wraps the transition under analysis inside a specification transition. A specification transition saves the following information:

- If the start state of the transition (state machine) has multiple transitions
- The transition of the state machine which is represented
- The min and max clock values

The minimum and maximum clock of the transaction are calculated. A transition can happen during a time contained inside an interval[1]. Inside these clock variables we want to store the difference from the current time interval for a transition to occur:

- The minimum clock value would be given by the situation when the past transition occurs as late as possible and the current under analysis happens as soon as possible (assuming that this transition starts after the past one)
- The maximum clock value would be given by the situation when the past transition occurs as soon as possible and the current under analysis happens as late as possible (assuming that this transition starts after the past one)
- When the assumption (the transition starts after the past one) is not valid, the reasoning is exactly the opposite

[1]A trasition may occur after max clock value has elapsed in case the following action is not possible, e.g. waiting for a signal that is not yet available

This interval is important as only transitions within the smallest interval can be executed (exhaustive explanation further on in the documentation).

Then, the main method selects the executable transitions of the specification state base on the following criteria:

- A synchronous transition (Send, Receive) is executable if both the sender and the receiver transitions, which belong to the same signal, are available in `transitions`.
- Only the first available transitions can be executed. The minimum clock depends on the transition with the minimum clock min. The maximum clock, instead, by the first transition with the minimum max clock. For instance, let's imagine that we are at clock (0, 0) and we have three transitions (0, 3), (1, 2), (4, 5). In this case, the min clock will be 0 and max clock 2. The two possible transitions that can be executed are (0, 3) and (1, 2)
- Each transition is limited by the general max clock

The `ignoreConcurrenceBetweenInternalActions` flag controls the possibility of executing as soon as possible empty transitions with no alternatives and no time constraints. For all the transitions that follow these rules, the method `computeAllInternalStatesFrom` is called.

First of all, a new state `newState` is created from a copy of the current one. Then it is updated with new values depending on the transition. The first operation is to update the clock time for each specification block. The time is kept relative to the past transition. So, after a transition is executed, the specification block where the transition occurred will have min and max time at zero. For instance, let's execute a transition A. After transition A, a transition B can happen with a delay between 20 and 30 units. No matter the value of the clock before transition A, the clock is set at 0 to wait for an interval of time between 20 and 30. For specification blocks not in the transitions, instead, the current clock has to be increased and then upper bounded to the max clock of the transition. For instance, let's take a transition A on SP1 with time (10, 20), a transition B on SP2 with time (20, 30), and a global clock at 0. After transition A is executed. The clock in SP2 is increased becoming (10, 20) so that the time is considered advancing during transition A.

The transition is executed in the method `executeTransition`. The next state pointed by the transition is retrieved, the state index of the specification block is updated, and the action or the synchronized signal is executed.

The hash of the newly created state is then used to check if an already existing state with the same configuration already exists. In this case, only a link would be added and the new state copy will be deleted.

For nonempty transitions, in general, the execution procedure is the same as the algorithm's preparation explained in subsection 4.3 which is executed for all the pendent transitions. Continuing the main loop of the algorithm, the global reachability graph can be generated.

5 Reachability of States

The reachability condition of states is checked while creating the reachability graph. The array `reachabilities` contains the states elements to be checked. Every time a transition is executed, from state s_t to state s_{t+1}, s_{t+1} is checked for reachability. When all the states elements in `reachabilities` have been reached or the reachability graph is completed, the reachability study finishes.

5.1 Deadlocks

Deadlocks are searched while building the reachability graph. If a state with no available transitions is encountered, the model-checker is stopped using the flag *propertyDone*. The deadlock study is bypassed if another previous study finds deadlocks.

5.2 Reinitialization

The model-checker contains an option to test if the system always reaches the initial state for each execution trace. This is verified as a liveness of the *State 0* represented by `SpecificationReinit`.

6 Safety and Liveness

6.1 Definitions

In the model also safety and liveness conditions could be proved. In particular, imagining a structure in time of the model, as a tree, the following CTL and LTL formulas are supported:

- **A[] p**: property p is always true for each path (other common notation **AG p**)
- **A<> p**: property p will eventually be true for each path (other common notation **AF p**)
- **E[] p**: there exists at least one path in which property p is always true (other common notation **EG p**)
- **E<> p**: there exists a path in which property p will eventually be true (other common notation **EF p**)
- **p –> q**: whenever p is true, then q will be true at some subsequent moment (other common notation **G(p ⇒ Fq)** called *leadsTo*)

with *p* and *q* properties. **Properties are written as expressions on variables or states of the AVATAR model**.

6.2 Handling of safety and liveness properties by the AVATAR model checker

Safety and liveness properties are represented by the `SafetyProperty` class. The properties are built in the AvatarExpressionSolver class. It creates a syntax tree where each leaf is an AvatarExpressionAttribute or an immediate value and each node is an operator. An

AvatarExpressionAttribute can represent a state or a variable of the model. It contains pointers so that, given a state of the model checker, it extracts the associated value in *O(1).* It can represent integers and boolean values encoded as integers. So, results of properties can be extracted just by visiting the AvatarExpressionSolver tree. AvatarExpressionSolver is also used for guards and actions. The methods `getSolverResult(SpecificationState)` and `getSolverResult(SpecificationState, AvatarStateMachineElement)` can be used to obtain the expressions result.

Safety properties are solved by finding loops or terminal conditions (deadlocks) in the model. For each property type we now explain how the modelchecker behaves:

- **A[] p**: if during the reachability graph creation a new state has *p* false, then the property is false.
- **A<> p**: if the property is true for a state, stop the search on that path. If a false loop or a false deadlock is found, the property is false. If no false loops or false deadlocks are found, the property is true.
- **E[] p**: if the property is false for a state, stop the search on that path. If a true loop or a true deadlock is found , the property is true. If no true loops or false deadlocks are found, the property is false.
- **E<> p**: if during the reachability graph creation, *p* is true in a new state, then the property is true.
- **p –> q**: every time `p` is true, save the SpecificationState in the *safetyLeadStates* array. Then, for each node in *safetyLeadStates* start a liveness check `A<> q` from that node. If the liveness is true for all the nodes in safetyLeadStates or safetyLeadStates is empty, the property is true, else it is false. The model-checking is bounded in order to reduce the exploration space in case of false pragmas. It allows also to prove false properties on infinite graphs. The bound is defined on the number of states matching property `p` in the current execution. When the number of states is greater than a threshold, the model-checker is paused, the current situation is saved, and the verification of *q* starts from the retrieved states. If the property cannot be proved by these states, the model-checker on *p* is resumed and a new grater threshold is defined.

The main methods involved in the properties verification process are:

- `executeSafetyRun`: it executes a safety property and it retrieves the result on the current model. It has options to ignoreConcurrence, emptyTransitions and to use partialHashing in the leadsTo verification (more details in the internal action loops section 7).
- `evaluateSafetyOfProperty`: given the current and old state, this method checks if the property is satisfied, still satisfied or not satisfied. It returns the result which is assigned to the flag *property* of a SpecificationState.
- `actionOnProperty`: it selects an action based on the type of verification, the current result of the property, and if the node exists already (possible loop). If the general result can be concluded, the flag *propertyDone* is set to stop the execution of the model-checker.

- `checkPropertyOnDeadlock`: given the current state with no executable transitions (deadlock), it executes an action depending on the type of CTL query.

6.3 Handling combinatory explosion

The exploration space of states explodes easily in many models. That's why proving properties is hard. The basic idea of liveness is to find a counterexample to prove it wrong. So, we want to detect a cycle (or a path that ends to a deadlock) for which the state we want to check is not live. If the states space explodes, an exploration in breadth will not be effective as the number of nodes increases too much before proving or disproving any property, the space will be too big to be explored.

Moreover, loops are found analyzing a path in depth. So, the check is executed in a depth first search-like manner, instead of a breadth first search implemented for other types of studies, like reachability. It will allow us to search through a path for a property to be true or false having a result within a manageable state-space most of the time. As soon as a property is (un)satisfied the search can continue through another path or stop. In the implementation, each thread follows a path to disprove the liveness. As soon as the property is unsatisfied, the check can stop. As soon as the property is satisfied, the search in that path can stop, and another one is fetched.

To limit the state's explosion, it is advisable to merge action nodes when possible. This happens inside the method *computeAllInternalStatesFrom* where all the possible executable action transitions are executed and merged into one state. During the *leadsTo* verification also *reduceCombinatorialExplosionProperty* is also used to execute and merge transitions keeping a property valid in order to reduce the number of states added in the structure *safetyLeadStates*.

7 Internal action loops

Internal action loops are used to check conditions when the system stops interracting with the environment. This condition may happen when there are action loops, which do not contain any signal communication, in the state machine of the blocks. If a state machine contains this type of loops, we want to verify that an exit condition is always reached (live).

The internal loops are detected statically through a strong connected components search executed by the method `checkStaticInternalLoops` in the AvatarStateMachine class. The method returns a list of all the action loops which are described by a list of transitions.

The verification of internal action loops is then managed in the *SpecificationAction-Loop* class. The check is verified using a *leadsTo* pragma built as follows:

- if no static loops are detected, a block is free from internal action loops
- if the loops contains transitions from a random block, the loop is removed as the exit condition is guaranteed by construction
- all the states contained in the loops are collected

- a *leadsTo* query **p –> q** is built with **p** equal to the logic *OR* of all the states collected and with **q** equal to the not of **p**. This pragma verifies that once entered into a state contained in a possible action loop, in a subsequent moment, a state not contained in a action loop is always reached.
- the pragma is verified by the model-checker. If the result is false, the block contains at least one infinite action loop trace
- these operations are repeated for each block in the system

For instance, if a static loop contains the states *Collect, Elaborate, Fail, Retry* the pragma created would be *Collect* || *Elaborate* || *Fail* || *Retry –> !(Collect* || *Elaborate* || *Fail* || *Retry).*

8 Verification traces

Verification traces are extracted using the class *CounterexampleTrace*. For each state created during the model-checking a *CounterexampleTraceState* is associated with it to save the current state pointer for each block and a pointer to a father. All the "counter" states are inserted in the map *traceStates*. The final verification state is extracted by the method *evaluateSafetyOfProperty* or *checkPropertyOnDeadlock*. That state represents a point of a loop, of a deadlock, or a reachability.

The verification trace is constructed in the *CounterexampleTrace* class using the method *buildTrace*. The start point is the last state *counterexampleState* which ended the verification. If the verification ends with a reachability or a deadlock, so it does not contain a loop, the trace is simply created visiting each father starting from *counterexampleState* until *state 0*. If instead, the verification ends with a loop, we have to eventually manipulate the data since the recorded trace using the fathers may not include a loop. If visiting each father from *counterexampleState* a loop is detected, the trace is already well constructed. If instead, it does not contain a loop, a loop is searched using a deep first search starting from the loop point (the state with the same hash value of *counterexampleState*). Then, the trace visiting each father starting from the loop point is appended to the previously detected cycle.

In the class *AvatarModelChecker* the data structures are allocated in the method *initCounterexample*. The method *resetCounterexample* must be used before starting a new verification to clear the data structures and to prepare the corresponding variables. Finally, the method *generateCounterexample* is used to generate the traces after a verification run if a counterexample has been detected.