
RelayModel

Dennis Suermann

Oct 04, 2021

CONTENTS:

1	RelayModel	1
1.1	RelayModel package	1
1.1.1	RelayModel.Communication module	1
1.1.2	RelayModel.ConceptChange module	6
1.1.3	RelayModel.GraphGeneration module	6
1.1.4	RelayModel.KeyGeneration module	7
1.1.5	RelayModel.LinkLayer module	8
1.1.6	RelayModel.ModuleConfig module	10
1.1.7	RelayModel.Node module	12
1.1.8	RelayModel.Relay module	16
1.1.9	RelayModel.RelayId module	19
1.1.10	RelayModel.RelayLayer module	20
1.1.11	RelayModel.RelayLogging module	26
1.1.12	RelayModel.SortedListNode module	26
1.1.13	RelayModel.StateMonitor module	29
1.1.14	RelayModel.Validation module	30
	Python Module Index	33
	Index	35

RELAYMODEL

1.1 RelayModel package

The RelayModel module gives all needed classes and functions to use the RelayModel.

For further information about the usage and structure of this module see the thesis of this module.

The module can be used to simulate distributed systems with the RelayModel. It can also be adapted to implement basic self stabilising protocols. In this module an adaption of the BuildList protocol, which forms a sorted list, is given in the SortedListNode.py file. This protocol got modified to use the RelayModel.

1.1.1 RelayModel.Communication module

class RelayModel.Communication.Action(*action_type: str, parameters: list*)

Bases: object

Defines the main action class for calling actions in a node.

This class holds all information needed to execute an action.

action_type

Holds the action_type or the action name as a string. For a sorted list e.g. “linearize”

Type str

parameters

Holds a list of parameters the action needs for executing

Type list

receiving_relay

Holds a RelayId of the Relay which received this action. This is needed for reversing every receiving relay.

Type *RelayId*

check_has_key_as_parameter(*key*)

Checks the parameters of the action if there is a RelayParameter with the given key.

This is needed for removing keys from a probing message if a relay has this key set in one RelayParameter.

Parameters **key** (*str*) – Holds the key that should be checked

Returns True if there is a RelayParameter with the given key, otherwise False

Return type bool

class RelayModel.Communication.FailureMessage(*msg*)

Bases: object

Holds a message and signaling a failed message transmission

class RelayModel.Communication.Header(*keys, sender_rid, out_id*)

Bases: object

Holds information needed for a message header.

The header is needed to validate a transmission over a specific relay.

keys

Holds a set of keys which can authenticate a transmission

Type Set

sender_rid

Holds the rid from a relay layer which transmitted this message

Type RelayLayerId

out_id

Holds the RelayId of a outgoing relay connection.

Type RelayId

class RelayModel.Communication.InRelayClosedAction(*keys, sender_rid, relay_id*)

Bases: [RelayModel.Communication.Action](#)

Defines the InRelayClosed Action and inherits the functionality of Action.

This action is sent when a relay is closed and a relay layer wants to inform the relay layer the outgoing connection is going to.

keys

Holds a list of keys that a not used anymore and should be corrected in incoming connections

Type list

sender_rid

Holds the RelayLayerId from the RelayLayer that is sending this Action

Type RelayLayerId

relay_id

Holds the RelayId of the Relay that has closing incoming connections

Type RelayId

property keys

Handles the keys stored on the first index in the parameter list

property relay_id

Handles the relay id stored on the third index in the parameter list

property sender_rid

Handles the sender rid stored on the second index in the parameter list

class RelayModel.Communication.LayerMessage(*layer_id: RelayModel.RelayId.RelayLayerId, action: RelayModel.Communication.Action*)

Bases: object

Defines a class that holds information for messages between RelayLayers.

This class is used for communications between two RelayLayers.

layer_id

Holds the RelayLayerId of the RelayLayer the message should be sent to.

Type *RelayLayerId*

action

Holds the Action that should be executed when the relay layer receives the message.

Type *Action*

class RelayModel.Communication.Message(*header: RelayModel.Communication.Header, action: RelayModel.Communication.Action*)

Bases: object

Defines a wrapper class for a message.

The message includes an action and a header.

header

Holds the Header of the message which authenticates it

Type *Header*

action

Holds the Action that should be executed on arrival of this message

Type *Action*

class RelayModel.Communication.NotAuthorizedAction(*keys, out_id*)

Bases: *RelayModel.Communication.Action*

Defines the NotAuthorized Action and inherits the functionality of Action.

The action is used when a transmission over a relay is not authorized with the given keys. It holds the keys that a not authorized and the RelayId of the Relay the transmission was not allowed for.

keys

Holds a list of keys the transmission was not authorized for.

Type list

out_id

Holds a RelayId that defines the Relay which should have received the message.

Type *RelayId*

property keys

Handles the keys attribute stored in the parameter list on first index.

property out_id

Handles the out id attribute stored in the parameter list on second index.

class RelayModel.Communication.OutRelayClosedAction(*relay_id*)

Bases: *RelayModel.Communication.Action*

Defines the OutRelayClosed Action and inherits the functionality of Action.

This action is sent when an Relay is closed and it informs all incoming connections of this relay that it is closed.

relay_id

Holds the RelayId of the Relay that got closed

Type *RelayId*

property relay_id

Handles the relay id that is stored on the first index in the parameter list.

class RelayModel.Communication.Ping(*relay_id, level, sink_rid, key*)

Bases: object

Holds every Attribute needed for a Ping Action. This class is not an Action class itself. This class is rather a helper class for storing information.

relay_id

Holds the RelayId of the relay which incoming connection should be ping checked.

Type *RelayId*

level

Holds the level information that should be checked for the ping.

Type int

sink_rid

Holds the sink RID of the connection that should be checked in the ping.

Type *RelayLayerId*

key

Holds the key of the incoming connection which should be checked.

Type str

class RelayModel.Communication.PingAction(*pings*)

Bases: *RelayModel.Communication.Action*

Defines the Ping Action and inherits the functionality of Action.

This class holds a list of ping objects. This action is sent to validate incoming connections of a relay.

pings

Holds a list of ping objects. This is compressed to one action because it reduces network traffic over the LinkLayer

Type list

property pings

Handles the ping list stored in the parameter list on first index.

class RelayModel.Communication.ProbeAction(*control_keys, key_sequence*)

Bases: *RelayModel.Communication.Action*

Defines the Probe Action and inherits the functionality of Action.

The ProbeAction is used when trying to validate indirect connections.

control_keys

Holds a list of keys that needs to be checked in the probe message.

Type list

key_sequence

Holds a list of key sets that define a route the probe message went.

Type list

property control_keys

Handles the control keys of the Action from the parameters list stored on first index.

property key_sequence

Handles the key sequence list of the Action from the parameters list stored on second index

class RelayModel.Communication.ProbeFailAction(*key, key_sequence*)

Bases: [RelayModel.Communication.Action](#)

Defines the ProbeFail Action and inherits the functionality of Action.

The action is sent when a probe action failed for a specific key. This class hold the key that failed the probe and the key sequence for recreating the route of the probe message.

key

Holds the key that caused the probe to fail.

Type str

key_sequence

Holds a list of key sets that define a route the probe message went.

Type list

property key

Handles the key attribute of the Action stored in the parameter list on first index.

property key_sequence

Handles the key sequence attribute of the Action stored in the parameter list on second index.

class RelayModel.Communication.RelayParameter(*key, relay_id: RelayModel.RelayId.RelayId, level: int, rid: RelayModel.RelayId.RelayLayerId*)

Bases: object

Holds every information needed for a replacement of a relay.

This replacement has information set for another relay layer to establish a connection.

key

Holds the key that is needed for an authorized connection

Type str

relay_id

Holds the relay id defining the connection relay

Type [RelayId](#)

level

Holds the level of the connection

Type int

rid

Holds the RelayLayerId of the sink rid of the connection

Type [RelayLayerId](#)

class RelayModel.Communication.SuccessMessage(*msg*)

Bases: object

Holds a message and signaling a successful message transmission

class RelayModel.Communication.TransmitMessage(*message: RelayModel.Communication.Message*)

Bases: object

Defines a class that holds information for a transmit message.

This class is used for transmitting a message from one node to another.

message

Holds the message that should be transmitted over relays

Type *Message*

1.1.2 RelayModel.ConceptChange module

`RelayModel.ConceptChange.calculate_windows(window_a, window_b)`

Calculates a change rate of two given windows provided as lists.

The two lists should be of same size.

Parameters

- **window_a** (*list*) – Holds the first window
- **window_b** (*list*) – Holds the second window

Returns A change rate of the both windows

Return type `int`

1.1.3 RelayModel.GraphGeneration module

class `RelayModel.GraphGeneration.Blueprint`

Defines a class that is needed to make a initial graph of nodes and relays. It holds outgoing and incoming blueprint to establish a connection to other nodes.

in_blueprints

Holds all incoming blueprints to establish incoming connections of a node

Type `list`

out_blueprints

Holds all outgoing blueprints to establish outgoing connections to other nodes over relays

Type `list`

class `RelayModel.GraphGeneration.InBlueprint(key, rid)`

Holds information to establish an incoming connection with relays.

key

Holds the key that authorizes the incoming connection

Type `str`

rid

Holds the RelayLayerId of the RelayLayer that is allowed to send messages over this incoming connection.

Type *RelayLayerId*

class `RelayModel.GraphGeneration.OutBlueprint(key, sink_rid, direction)`

Holds information to establish an outgoing connection with relays.

key

Holds the key that authorizes the outgoing connection

Type `str`

sink_rid

Holds the RelayLayerId of the RelayLayer that manages the relay of the sink relay of this connection

Type *RelayLayerId*

out_id

Holds the RelayId of the relay the outgoing connection is pointing to

Type *RelayId*

direction

Holds information about a possible sorted list configuration. 0 is nothing is set. If 1 is set the connection is set to the left attribute of the sorted list. If 2 is set the right is set to this relay

Type `int`

`RelayModel.GraphGeneration.make_weakly_connected_sorted_list(node_list)`

Makes a list of Blueprints which information builds a weakly connected directed graph.

It takes a list of RelayLayerIds for a specification which nodes should be present in the graph.

Parameters `node_list (List)` – Holds a list of RelayLayerIds defining Nodes that should be present in the resulting graph

Returns A dictionary where the key is the RelayLayerId of a node and the value is the Blueprint class for this node

Return type `dict`

1.1.4 RelayModel.KeyGeneration module

`RelayModel.KeyGeneration.check_key_origin(key: str, layer_id)`

Checks if the given key contains the given RelayLayerId.

Parameters

- **key (str)** – Holds the key that should be checked.
- **layer_id (RelayLayerId)** – Holds the RelayLayerId of the RelayLayer that should be checked.

Returns True if prefix of key matches given layer id, False otherwise.

Return type `bool`

`RelayModel.KeyGeneration.generate_key(layer_id)`

Generates a key for a given RelayLayerId

Parameters `layer_id (RelayLayerId)` – Holds the RelayLayerId of the RelayLayer that generates the key

Returns The generated key as a string in the format “prefix#uniqueId”

Return type `str`

`RelayModel.KeyGeneration.generate_prefix(layer_id)`

Generates a prefix for a authentication key. It contains the given RelayLayerId to check the origin of a key.

Parameters `layer_id (RelayLayerId)` – The RelayLayerId of the RelayLayer that generates the key

Returns The key prefix as a string

Return type `str`

1.1.5 RelayModel.LinkLayer module

class RelayModel.LinkLayer.**LinkLayer**(*link_layer_buffer, relay_layer_buffer, node_buffer, listen_ip, listen_port, pipe*)

The LinkLayer class provides the functionality to send and receive messages to other nodes.

The LinkLayer watches message buffers from the RelayLayer and Relays and sends every message to the right endpoint. For further information about the functionality see the thesis of this library.

running

Defines the running state of the LinkLayer. If this is False all Thread should be stopped.

Type bool

stopped

Defines the state if the LinkLayer is stopped. It is only set to True if the LinkLayer completely stopped all threads and processes.

Type bool

check_key_in_relay_buffer(*relay_id, check_key*)

Checks the buffer of a given Relay if there is a key in one of the Actions.

First the method checks if the buffer exists for the given relay. If there is a buffer it checks if the buffer is empty. Both conditions return False. After that the message checks all messages in the buffer if there is a key set in one parameter set as a RelayParameter. This method should be called with the method calling pipe.

Parameters

- **relay_id** ([RelayId](#)) – Defines the RelayId of the Relay which buffer should be checked.
- **check_key** (*str*) – Defines the key that should be checked.

Returns True if there is the given key in one RelayParameter in the buffer of the given Relay.
False otherwise

Return type bool

is_buffer_empty(*relay_id*)

Checks if a buffer for a given relay id defining a relay is empty.

The method checks if there is a buffer for the given relay id and checks if it is empty. This method should be called with the method calling pipe.

Parameters **relay_id** ([RelayId](#)) – The RelayId that defines the Relay which buffer should be checked.

Returns True if buffer is not existent or empty. False if the buffer has an entry.

Return type bool

register_relay_layer(*layer_id*)

Registers a RelayLayer and creates a buffer watch process.

The LinkLayer creates the buffer watch process for the relay layer. It only creates the process if it is not set yet.

Parameters **layer_id** ([RelayLayerId](#)) – Defines the RelayLayerId of the RelayLayer which buffer should be watched.

shutdown()

Shutowns the LinkLayer completely.

It first shutdowns all Threads and then all subprocesses. This method should be called over the method call pipe.

stop_relay_watch(*relay_id*)

Stops a watch of a relay buffer.

The method stops the watch from a relay buffer. This method should be called over the method call pipe.

Parameters **relay_id** ([RelayId](#)) – The RelayId of the Relay which buffer watch should be stopped.

`RelayModel.LinkLayer.send_message(context, layer_id, message)`

Used to send a message to a specific link layer.

This message can be used to send a message to another link layer using the lazy pirate pattern. It creates a socket in the given context and tries to send the given message to the given layer id.

Parameters

- **context** (`zmq.Context`) – Defines the context where the socket should created in.
- **layer_id** ([RelayLayerId](#)) – Defines the RID of the LinkLayer that needs to receive the message.
- **message** (`Object`) – The message that should be send. Normally this should be a `TransmitMessage` object or a `LayerMessage`. Otherwise the other LinkLayer will deny the message. But in general every object can be sent over the socket.

Returns

True if the message got accepted or is received by the link layer, False if the Message was denied or could not be sent.

Return type `bool`

`RelayModel.LinkLayer.start_layer_buffer_watch(layer_id, buffer, relay_layer_buffer, running)`

Starts a watch over a RelayLayer buffer.

It watches a given buffer and if there occurs a message it tries to send it to the right endpoint. This method should be called in a seperated process. The function runs as long the running state is equal to 1 and tries to get a message from the buffer.

Parameters

- **layer_id** ([RelayLayerId](#)) – Defines the layer id of the relay layer that registered in this function. This is only needed for creating a logger file.
- **buffer** (`multiprocessing.Queue`) – Defines the buffer that should be watched as a multiprocessing queue.
- **relay_layer_buffer** (`multiprocessing.Queue`) – Defines the message buffer of a RelayLayer. This buffer is needed to send messages to the same RelayLayer as registered.
- **running** (`multiprocessing.Value`) – Defines the running state of the function. If this value is set to 0 the function stops. This is used to stop the function within a process from a parent process.

`RelayModel.LinkLayer.start_link_layer(link_layer_buffer, relay_layer_buffer, node_buffer, listen_ip, listen_port, pipe)`

Creates a link layer and waits until it got stopped.

This function should be called with a new process.

Example

A basic example is given here. It creates a Process and starts it afterwards

```
process = multiprocessing.Process(target=start_link_layer, args=(link_layer_buffer, message_buffer,
node_queue, layer_id.ip, layer_id.port, link_layer_pipe)) process.start()
```

Parameters

- **arguments.** (All arguments are the same as the LinkLayer constructor) –
- **docs.** (For further information see LinkLayer class) –

`RelayModel.LinkLayer.start_listening(listen_ip, listen_port, relay_layer_buffer, running)`

Starts listening on a specific ip and port and can be used to receive messages on this address.

The function creates a listening socket on the given address and tries, as long the running state is 1, to receive messages from it. When a message is received it will get checked if it is a LayerMessage or a TransmitMessage. If the format is correct it sends a *SuccessMessage* back to the sending LinkLayer. Otherwise it sends a *FailureMessage* back. If the message is a valid Message the message will be provided to the RelayLayer of this LinkLayer. This is done by inserting the message in the given RelayLayer message buffer.

Parameters

- **listen_ip** (*str*) – The ip the socket should be listen on. This is only used for the logger file name. The socket normally listens on every interface on a specific port.
- **listen_port** (*int*) – Defines the port the socket should listen on.
- **relay_layer_buffer** (*multiprocessing.Queue*) – Provides the message buffer of a RelayLayer. It is used to send received messages to the RelayLayer so it can process the message.
- **running** (*multiprocessing.Value*) – Defines the running state of the function. If this value is set to 0 the function stops. This is used to stop the function within a process from a parent process.

1.1.6 RelayModel.ModuleConfig module

`RelayModel.ModuleConfig.CHANGE_ALPHA = -10`

Defines the change rate threshold for a dos attack detection.

This value should always be lower than 0. Otherwise falling transmit rates get detected as dos attacks. Normally this is set to -10.

`RelayModel.ModuleConfig.CONSIDER_AS_CLOSED = False`

Stores the flag if the link layer should send closed messages if one link layer is not reachable.

If this is set to False it will only discard the messages after the retries. Otherwise it will send In- and OutRelay-Closed Actions. Normally this is set to False.

Type bool

`RelayModel.ModuleConfig.DOS_DETECTION_ACTIVATED = True`

Stores the flag if the dos detection should be used.

If this is set to True the RelayLayer is watching transmit rates and try to detect dos attacks. Normally this is set to True.

`RelayModel.ModuleConfig.NODE_TIMEOUT_PERIOD = 1`

Defines the timeout period of a node.

Normally this is set to 1 second.

Type float

`RelayModel.ModuleConfig.NO_MONITOR_ACTIONS = ['Probe', 'ProbeFail', 'NotAuthorized', 'Ping', 'InRelayClosed', 'OutRelayClosed']`

Stores the actions that are standard actions from the RelayLayer.

This actions are not monitored on transmission because a node should not have access to this actions. It is stored in a list where each entry has the action name set. Normally this is set to ['Probe', 'ProbeFail', 'NotAuthorized', 'Ping', 'InRelayClosed', 'OutRelayClosed']

`RelayModel.ModuleConfig.POLL_TIMEOUT = 300`

Timeout in milliseconds for polling receiving messages when sending in link layer.

Normally this is set to 300 ms.

Type int

`RelayModel.ModuleConfig.POLL_TRIES = 3`

Tries until the receiving socket is considered as closed.

Normally this is set to 3 tries.

Type int

`RelayModel.ModuleConfig.RELAY_LAYER_TIMEOUT_PERIOD = 1`

Defines the timeout period of a relay layer.

Normally this is set to 1 second.

Type float

`RelayModel.ModuleConfig.RELAY_LOG_LEVEL = 30`

Holds the logging level of all loggers in this module.

Normally this is set to logging.WARNING

Type int

`RelayModel.ModuleConfig.RESULTS_FOLDER = 'results/'`

Defines the folder where the StateMonitor should write result files to.

Normally this is set to the folder named "results/".

`RelayModel.ModuleConfig.STATE_MONITOR_ADDRESS = 'localhost:1999'`

Defines the full address of the StateMonitor including ip and port.

Normally the ip of the address is localhost.

`RelayModel.ModuleConfig.STATE_MONITOR_PORT = 1999`

Defines the port of the StateMonitor

Normally this is set to port 1999.

`RelayModel.ModuleConfig.WINDOW_SIZE = 20`

Defines the window size for the dos detection.

This should be a value that is dividable by 2. Normally this is set to 20.

1.1.7 RelayModel.Node module

class RelayModel.Node.**Node**(*node_id: int, ip: str, port: int, analyse_mode=False*)

The Node class represents a Node in a distributed System.

The Node class is the main class for creating distributed protocols with the relay model. This class implements functionality to work with the relay model and leave a system without harming its connectivity. It implements the protocol provided by Setzer in his phd thesis: (<https://digital.ub.uni-paderborn.de/urn/urn:nbn:de:hbz:466:2-37849>).

buffer

Defines a buffer for the node where actions can be inserted by the RelayLayer. Every action in this buffer gets processed by the node.

Type multiprocessing.Queue

relay_layer

Stores a RelayLayer object for this node.

Type *RelayLayer*

running

Stores the state of the node. If True the node is running if False the node is not running and all threads are stopped.

Type bool

timeout_thread

Stores the timeout thread where timeout action were inserted in the message buffer.

Type threading.Thread

message_thread

Stores the message thread of the node. The thread periodically checks if there is a new Action in the node_buffer and executes the Action.

Type threading.Thread

logger

Stores the Logger class of the Node.

Type logging.Logger

timeout_period

Stores the period of the timeout execution. By default this is set to the value set in the ModuleConfig.

Type float

leaving

Stores the leaving state of the node. When set to True the Node changes its behaviour where it tries to leave the system by the protocol provided by Setzer.

Type bool

in_ref

Stores a relay id to the main sink relay of this node.

Type *RelayId*

N

Stores a list which is a pseudo variable for all relays stored in an underlying protocol.

Type list

D

Stores a list of RelayIds which should be deleted in the protocol.

Type `list`

a_out

Stores the RelayId of the outgoing anchor.

Type `RelayId`

a_in

Stores the RelayId of the incoming anchor.

Type `RelayId`

analyse_mode

Handles the analyse mode of the node. When set to True the Node is executed in analyse mode and sends every timeout a state to a StateMonitor. By default this is set to False and can be activated either in the constructor or by changing the attribute.

Type `bool`

ask_to_reverse(*out*: `RelayModel.RelayId.RelayId`)

Implements the ask_to_reverse protocol method.

For further information see the phd thesis.

Parameters **out** (`RelayId`) – Defines the RelayId that should be reversed.

ask_to_reverse_anchor(*out*: `RelayModel.RelayId.RelayId`, *receiving_relay*: `RelayModel.RelayId.RelayId`)

Implements the ask_to_reverse_anchor protocol method.

For further information see the phd thesis.

Parameters

- **out** (`RelayId`) – Defines a Relay.
- **receiving_relay** (`RelayId`) – Defines the relay that received this action.

call_method(*relay_id*: `RelayModel.RelayId.RelayId`, *method*: `str`, *parameters*: `list`)

Calls a method of another node connected with the given relay.

This method calls the given method with the given parameters in the node that handles the sink relay that is connected with the relay defined by the given relay id. It calls the send method from the RelayLayer to send the method to the node.

Parameters

- **relay_id** (`RelayId`) – Defines the Relay which is used to send the action.
- **method** (`str`) – Defines the name of the method that should be executed in the other node.
- **parameters** (`list`) – Defines a list of parameters used to execute the method.

check_in_original_variables(*relay_id*: `RelayModel.RelayId.RelayId`)

Should be overridden to check if a specific relay is in one of the underlying protocol variables.

By default it checks if the relay is in the attribute N. This should be changed to a better implementation when implementing a underlying protocol.

Example

See *SortedListNode* implementation.

Parameters `relay_id` ([RelayId](#)) – Defines the RelayId that should be checked.

`get_relays_from_original_variables()`

Should be overridden to get all relays from underlying protocol variables.

This normally just returns the N attribute list. This should be changed to a better implementation when implementing a underlying protocol. It should return a list of RelayIds that are stored in any variable define in the protocol.

Example

See *SortedListNode* implementation.

Returns A list of RelayIds containing in variables of the protocol.

Return type list

`notify_anchor()`

Implements the notify_anchor protocol method.

For further information see the phd thesis.

`original_timeout()`

Can be overridden to implement a underlying protocol timeout.

This method is automatically executed on the end of every timeout of this node class. It should be implemented with the timeout of the underlying protocol.

Example

See *SortedListNode*

`remove_from_original_variables`(*relay_id*: [RelayModel.RelayId.RelayId](#))

Should be overridden to remove relays from underlying protocol variables.

By default it removes the given relay id from the N attribute. This should be changed to a better implementation when implementing a underlying protocol.

Example

See *SortedListNode* implementation.

Parameters `relay_id` ([RelayId](#)) – Defines the RelayId that should be removed from the variables.

`replace_action`(*action*)

This method is called whenever the node received a Action.

The method executes the method defined in the received action if the node is staying or reverses the connection if the node is leaving.

Parameters `action` ([Action](#)) – The received action that should be executed.

reversal_of_relay(*relay_id*: RelayModel.RelayId.RelayId)

Should be overridden to reverse a specific relay reference.

The method should just send a node action from the underlying protocol over the given relay with the *in_ref* as parameter.

Example

See *SortedListNode*. Example for the linearize Action

```
self.call_method(relay_id, 'linearize', [self.in_ref])
```

reverse(*out*: RelayModel.RelayId.RelayId)

Implements the reverse protocol method.

For further information see the phd thesis.

Parameters *out* (RelayId) – Defines a Relay.

send_analyse_state()

Should be overridden to send a state to a StateMonitor.

In this method a State object should be created. After that it should be sent with the *send_state_to_monitor* method.

Example

See *SortedListNode* implementation.

send_state_to_monitor(*state*)

Sends the given state to the StateMonitor.

The method sends the given state to the address defined in the ModuleConfig.

Parameters *state* (Object) – Defines the state that should be sent. This should be a object of a state class.

shutdown()

Completely shutdown the node.

The method stops all threads by setting the running state to False. After that it stops the RelayLayer and waits until the RelayLayer deleted all Relays.

start()

Starts the node by starting all threads.

The node gets started by starting the message thread and the timeout thread.

stop()

Stopping the node so that the node should leave the system.

The method sets the leaving state to True and activates the leaving protocol of the node.

timeout()

The Timeout method is called periodically and corrects all values in the node.

The method is correcting every value in the node or if the node is leaving it will prepare the node for leaving. For further information about the protocol see the thesis of this paper.

1.1.8 RelayModel.Relay module

class RelayModel.Relay.InReference(*key="", rid=None, relay=None*)

Holds the information needed for an outgoing connection to another Relay.

key

Stores the key of the incoming connection

Type string

rid

Stores the RelayLayerId defining the RelayLayer which is allowed to send messages over this incoming connection

Type RelayLayerId

relay

Stores the Relay which forwarded the reference. This incoming connection is not validated yet if the relay is set but the rid is not.

Type Relay

check_valid()

Checks if the incoming Reference is filled correctly.

Returns true if key is set and relay is set to a Relay object and rid is None or relay is none and rid is set to a RelayLayerId object.

Returns

True if key is set and relay is set to a Relay object and rid is None or relay is none and rid is set to a RelayLayerId object. False otherwise.

Return type bool

property key

Handles the connection key.

property relay

Handles the relay of the incoming connection.

Sets the value only if the given value is an instance of Relay. Otherwise it is set to None.

property rid

Handles the rid of the incoming connection

When setting this attribute, the value is only set to the given value if it is an instance of RelayLayerId. Otherwise it is set to None.

class RelayModel.Relay.OutReference(*keys=None, out_id=None*)

Holds the information needed for an outgoing connection to another Relay.

keys

Stores all keys in a set used in the outgoing connection

Type set

out_id

Stores the RelayId defining the Relay to which the OutReference is connected to

Type RelayId

add_key(key)

Adds the given key to the set of keys.

Parameters **key** (*str*) – The key that should be added.

property keys

Handles the keys attribute. When setting the value the attribute is only set if it is an object of type set.

property out_id

Handles the outgoing relay id. If setting this value it is only set if the given id is an instance of RelayId or None for a sink relay.

remove_key(*key*)

Removes the given key from the set.

Parameters **key** (*str*) – The key string that should be removed

class RelayModel.Relay.Relay(*relay_id*: [RelayModel.RelayId.RelayId](#))

The Relay class holds every information used to define a Relay for the RelayLayer.

relay_id

Stores the RelayId of the specific relay. Can be set via the constructor.

Type [RelayId](#)

alive

Stores the state of the relay if it is dead or alive. By default this is set to True.

Type bool

out_relay

Stores the outgoing connection of the Relay. This is by default an empty OutReference. Which means the created Relay is a sink relay.

Type [OutReference](#)

level

Stores the level of the Relay which describes the amount of hops needed to arrive at a sink relay. By default is this set to 0.

Type int

sink_rid

Stores the RelayLayerId managing the sink relay of this relay. By default this is set to the RelayLayerId holding this exact relay.

Type [RelayLayerId](#)

in_relays

Stores the incoming connections of the relay as a list of InReference objects. By default this is set to an empty list.

Type list

validated

Stores the validated flag of a relay. Needed for deletion issues in Node class.

Type bool

dos_threshold

Stores the threshold of this relay object for dos detection. By default this is set to the threshold set in the module config.

Type int

add_in_reference(*in_ref*)

Adds a given incoming reference to the list of incoming references.

The given incoming reference gets only added if it is an object of the InReference class.

Parameters **in_ref** (**InReference**) – InReference object that should be added as a incoming connection.

clear_in_relays()

Removes all incoming references from the relay.

get_valid_keys(*sender_rid*)

Gets all keys that are set in incoming references where the rid is set to the given rid.

Parameters **sender_rid** (**RelayLayerId**) – The RelayLayerId for which all keys are fetched.

Returns A list of keys that are set in incoming references.

Return type list

has_incoming()

Return the amount of incoming connections of the relay.

Returns The amount of incoming connections.

Return type int

has_key_in_in_ref(*key*)

Checks if the given key is in one of the incoming references.

Parameters **key** (*str*) – The key that should be checked.

Returns True if the key is set in one incoming reference, False otherwise.

Return type bool

is_dead()

Returns true if the Relay is not alive. Otherwise it returns false.

The relay is not alive if the alive state is false.

Returns True if the alive state is False, False otherwise.

Return type bool

is_direct()

Returns true if level is lower or equal than one. Otherwise it returns false.

Returns True if the relay is a direct relay, False otherwise.

Return type bool

is_sink()

Returns true if the relay level is zero and false otherwise.

Returns True if the relay is a sink relay, False otherwise.

Return type bool

remove_in_reference(*in_ref*)

Removes the given incoming reference from the incoming reference list.

If the given in reference is not in the list nothing happens.

Parameters **in_ref** – The reference that should be removed as InReference object.

remove_in_reference_by_key_and_rid(*key, rid*)

Removes all incoming references of this relay if the InReference key and rid equals to the given key and rid.

Parameters

- **key** (*str*) – The key that should be compared to remove the InReferences
- **rid** (*RelayLayerId*) – The RelayLayerId that should be compared to remove the InReferences

remove_in_reference_by_relay(*relay*)

Removes all incoming references of this relay when the relay of the InReference is set to the given relay.

Parameters **relay** (*Relay*) – The relay that needs to be set for deleting the InReference.

remove_in_reference_by_relay_and_key(*key, relay*)

Removes all incoming reference if the key and relay is set to the given key and relay.

Parameters

- **key** (*str*) – The key that should be compared to remove the InReferences
- **relay** (*Relay*) – The relay that should be compared to remove the InReferences

remove_in_reference_by_rid(*rid*)

Removes all incoming references of this relay when the rid of the InReference is set to the given rid.

Parameters **rid** (*RelayLayerId*) – The relay for which all InReferences should be removed.

replace_relay_in_references(*previous_relay, new_relay*)

Replaces the relay in all in references to the given new relay if the relay is set to the given previous relay.

Parameters

- **previous_relay** (*Relay*) – The relay that should be replaced
- **new_relay** (*Relay*) – The relay that should be in the incoming connections after replacement.

same_target(*other*)

Compares the outgoing connection of this relay to another relay.

Parameters **other** (*Relay*) – The Relay which should be compared.

Returns True if both have same target, False otherwise.

Return type bool

1.1.9 RelayModel.RelayId module

class RelayModel.RelayId.**RelayId**(*layer_id: RelayModel.RelayId.RelayLayerId, relay_id: int*)

Holds the information identifying a specific relay.

layer_id

Holds the reference to a RelayLayerId which defines the RelayLayer holding the Relay defined by this RelayId

Type *RelayLayerId*

relay_id

Holds an identifier for the Relay which defines a specific Relay in a RelayLayer

Type int

class RelayModel.RelayId.**RelayLayerId**(*ip: str, port: int*)

Holds the information identifying a relay layer.

ip

Defines the address on which the relay layer should be reachable.

Type str

port
Defines the port on which the relay layer should be reachable.

Type int

node_id
Defines the id of a node if it is needed to form a topology.

Type int

property node_id
Handles the internal id storing the node id of a node.

1.1.10 RelayModel.RelayLayer module

class RelayModel.RelayLayer.**RelayLayer**(*layer_id, node_queue*)

Represents the implementation of the Relay Layer in the relay model.

The RelayLayer class handles all connections over Relays and tries to stabilize the Relay information by communicating with other RelayLayers. It starts a LinkLayer inside a separated process to send messages over Relays. Furthermore it starts three threads. The first thread sends messages to the LinkLayer. The second Thread is the timeout thread which periodically executes the timeout method of the RelayLayer. The last thread watches the message buffer.

For further information about the RelayLayer see the thesis of this library.

link_layer_buffer

Stores the buffer of the LinkLayer where messages should be inserted. This messages will get inserted in the right buffer and will be processed from the LinkLayer. The objects that are inserted are always a tuple of data. The first entry of the tuple is either a Relay object if the message should be inserted into a relay buffer or a RelayLayerId when the message should be sent to another RelayLayer.

Type multiprocessing.Queue

link_layer_call_pipe

Holds the communication pipe of the LinkLayer. This pipe is used to call methods inside the LinkLayer process. A method can be called with the internal method `_call_link_layer_message`.

Type multiprocessing.Pipe

message_buffer

This buffer holds messages that needs to be processed by this RelayLayer. It gets filled by the LinkLayer if it receives a message.

Type multiprocessing.Queue

link_layer_process

Holds the LinkLayer process executing the LinkLayer.

Type multiprocessing.Process

RID

Holds the id defining this specific relay layer.

Type *RelayLayerId*

relays

Holds a dict of relays. The key is the RelayId of the specific relay and the value the Relay object.

Type dict

last_id

Stores the last id relay id for creation of a new relay. This is counted up when creating a new relay.

Type int

active

Holds the active state of the RelayLayer. If this is set to False the RelayLayer tries to shutdown. By default this is set to True.

Type bool

running

Holds the running state of the RelayLayer. If this is set to False the RelayLayer is stopped and all threads are shutting down.

Type bool

calling_lock

Holds a Lock object to prevent simultaneous access to the link_layer_call_pipe.

Type threading.Lock

logger

Holds the Logger object of this class.

Type logging.Logger

buffer_put_actions

Defines a buffer where every message that needs to be inserted in a buffer gets inserted. The buffer_put_thread is sending this messages to the LinkLayer.

Type deque

buffer_put_thread

The thread removes every message containing in the buffer_put_actions and sends it to the LinkLayer with the link_layer_buffer.

Type threading.Thread

message_handling_thread

Holds the thread that executes every action sent to the RelayLayer. The messages are inside the message_buffer attribute.

Type threading.Thread

timeout_thread

Holds the timeout thread where the RelayLayer periodically executes the timeout method.

Type threading.Thread

relay_windows

Stores the windows of each relay for DoS attack mitigation. The key is the RelayId and the value is the window for the specific Relay as a deque object.

Type dict

transmit_count

Stores the transmit count of a specific relay. The key is the RelayId and the value is the count of transmit messages sent over the specific Relay.

Type dict

add_relay_to_layer(*new_relay*: [RelayModel.Relay.Relay](#))

Adds a Relay object to the RelayLayer.

The method only accepts Relay object and only adds it to the layer if the relay id is not present in the layer.

Parameters `new_relay` ([Relay](#)) – The Relay object that should be added to the layer.

check_dos_attack(*relay*)

Checks if a dos attack is likely.

For further information see thesis of this library.

Parameters `relay` ([Relay](#)) – The Relay for which the rates should be checked.

check_relay_exists(*relay_id*: [RelayModel.RelayId.RelayId](#))

Checks if a Relay is existing in the RelayLayer.

Parameters `relay_id` ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns True if Relay exists in RelayLayer, False otherwise.

Return type bool

delete(*relay_id*: [RelayModel.RelayId.RelayId](#))

Deletes a relay.

For further information see the thesis of the library or the phd thesis of Setzer.

Parameters `relay_id` ([RelayId](#)) – The RelayId of the Relay that should be deleted.

get_key_for_layer()

Gets a authentication key generated by the RelayLayer.

Returns A key used for authenticating a connection.

Return type str

get_level(*relay_id*: [RelayModel.RelayId.RelayId](#))

Gets the connection level of a relay.

Parameters `relay_id` ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns The level of the connection.

Return type int

get_new_relay_object()

Gets a new relay object.

The method creates a new Relay in the RelayLayer but instead of adding it to the RelayLayer it just returns the object of it. This is needed to create connections between Nodes. The Relay can later be added to the RelayLayer by executing the `add_to_layer` method.

Returns A new Relay object.

Return type [Relay](#)

get_relay_by_relay_id(*relay_id*)

Gets the Relay object of defined by the given RelayId.

This should only be used for analysing purposes. Normally one should not have access to the relay object from outside.

Parameters `relay_id` ([RelayId](#)) – The RelayId of the Relay that should be returned.

Returns The Relay object defined by the given RelayId or None if the Relay is not existent in the RelayLayer.

Return type [Relay](#)

get_relays()

Gets all RelayIds of the Relays present in the RelayLayer.

Returns A list of RelayIds present in the RelayLayer.

Return type list

get_sink_node_id(*relay_id*: RelayModel.RelayId.RelayId)

Gets the node id of the node holding the sink relay of the connection defined by the given relay.

Parameters **relay_id** (RelayId) – The RelayId representing the Relay that should be checked.

Returns The node id of the sink relay.

Return type int

get_validated_relays()

Gets all RelayIds of the validated Relays present in the RelayLayer.

Returns A list of RelayIds which Relay is validated present in the RelayLayer.

Return type list

handle_in_relay_closed(*keys*: list, *sender_rid*: RelayModel.RelayId.RelayLayerId, *relay_id*: RelayModel.RelayId.RelayId)

The method is called when a InRelayClosed message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters

- **keys** (list) – The list of keys that are used in the closed connection.
- **sender_rid** (RelayLayerId) – The RelayLayerId of the RelayLayer that sent the message.
- **relay_id** (RelayId) – The relay id that is closed.

handle_not_authorized(*keys*, *out_id*)

The method is called when a NotAuthorized message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters

- **keys** (list) – List of keys.
- **out_id** (RelayId) – The RelayId that is not authorized.

handle_out_relay_closed(*relay_id*: RelayModel.RelayId.RelayId)

The method is called when a OutRelayClosed message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters **relay_id** (RelayId) – The RelayId of the Relay that is closed.

handle_ping(*relay_id*: RelayModel.RelayId.RelayId, *level*: int, *sink_rid*: RelayModel.RelayId.RelayLayerId, *key*: str)

The method is called when a Ping message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters

- **relay_id** (RelayId) – The relay id there should be a connection to.
- **level** (list) – The level of the connection.

- **sink_rid** ([RelayLayerId](#)) – The sink rid of the connection.
- **key** (*str*) – The key of the connection

handle_probe_fail(*key*, *key_sequence*)

The method is called when a ProbeFail message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters

- **key** (*str*) – Key that failed the Probe message.
- **key_sequence** (*list*) – List of key lists that represent the route the Probe message went.

handle_transmit(*message*)

The method is called when a transmit message is received by the RelayLayer.

For further information see the thesis of this library.

Parameters **message** ([Message](#)) – The message that got transmitted.

has_incoming(*relay_id*: [RelayModel.RelayId.RelayId](#))

Returns the amount of incoming connections of a given relay.

Parameters **relay_id** ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns The amount of incoming connections.

Return type `int`

is_dead(*relay_id*: [RelayModel.RelayId.RelayId](#))

Checks a Relay if it is dead.

Parameters **relay_id** ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns True if the specific relay is dead, False otherwise.

Return type `bool`

is_direct(*relay_id*: [RelayModel.RelayId.RelayId](#))

Checks a Relay if it is a direct relay.

Parameters **relay_id** ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns True if the specific relay is a direct relay, False otherwise.

Return type `bool`

is_sink(*relay_id*: [RelayModel.RelayId.RelayId](#))

Checks a Relay if it is a sink relay.

Parameters **relay_id** ([RelayId](#)) – The RelayId representing the Relay that should be checked.

Returns True if the specific relay is a sink relay, False otherwise.

Return type `bool`

merge(*relays*)

Merge method of the RelayLayer.

For further information see the thesis of the library or the phd thesis of Setzer.

Parameters **relays** (*list*) – A list of relay ids that should be merged to one relay.

Returns The RelayId of the merged Relay or None if merge was not successful.

Return type [RelayId](#) or None

monitor_timeout_transmit_rate(*relay*)

Monitors transmit rates.

For further information see thesis of this library.

Parameters **relay** ([Relay](#)) – The Relay for which the rate should be inserted.

monitor_transmit(*message*, *relay*)

Monitors transmits.

For further information see thesis of this library.

Parameters

- **message** ([Message](#)) – The message that is transmitted.
- **relay** ([Relay](#)) – The Relay that transmitted the message.

new_relay()

Gets a new relay reference.

The RelayLayer creates a new Relay object and provides the RelayId of it.

Returns of the new Relay object.

Return type [RelayId](#)

same_target(*relay_id_1*: [RelayModel.RelayId.RelayId](#), *relay_id_2*: [RelayModel.RelayId.RelayId](#))

Checks if two relays have the same outgoing connection.

Parameters

- **relay_id_1** ([RelayId](#)) – The first relay that should be checked.
- **relay_id_2** ([RelayId](#)) – The second relay that should be checked.

Returns True if the specific relays have the same outgoing connection, False otherwise.

Return type bool

send(*send_relay_id*: [RelayModel.RelayId.RelayId](#), *action*: [RelayModel.Communication.Action](#))

Send a action over a specific relay.

For further information see the thesis of this library.

Parameters

- **send_relay_id** ([RelayId](#)) – The RelayId representing the Relay which should transmit the action.
- **action** ([Action](#)) – The action that should be transmitted.

shutdown()

Shutdown the RelayLayer completely without removing Relay connections.

stop()

Stops the Relay Layer.

The method stops by deleting all relays and setting the active state to False.

timeout()

The timeout method of the RelayLayer.

For further information see the thesis of the library or the phd thesis of Setzer.

validate_relay(*relay_id*: [RelayModel.RelayId.RelayId](#))

Validates a given Relay.

The method validates the Relay by setting its validated flag to True.

Parameters `relay_id` ([RelayId](#)) – The RelayId of the Relay that should be validated.

1.1.11 RelayModel.RelayLogging module

`RelayModel.RelayLogging.get_logger(log_level, file_name)`

Creates a logger object with the given log level and the output file name.

Note: The file_name should be unique to the specific purpose. Otherwise it can

Parameters

- **log_level** (*int*) – Defines the log level of the Logger. Normally this should be set to the module config log level.
- **file_name** (*str*) – Defines the file the logging informations should be written to.

Returns The logger object with the specific log level and filename.

Return type `logging.Logger`

`RelayModel.RelayLogging.relay_list_to_string(relays)`

Makes a well formed string from a relay list.

Outputs a string for the given relay list where every relay is shown with its relay id and its connection to.

Parameters `relays` (*list*) – A list of relays that should be converted to a string.

Returns A well formed str of a list full of relays.

Return type `str`

1.1.12 RelayModel.SortedListNode module

class `RelayModel.SortedListNode.SortedListNode(node_id, ip: str, port: int, analyse_mode=False)`

Bases: [RelayModel.Node.Node](#)

The implementation of a sorted list node.

This class is an implementation of a sorted list protokoll. It is adjusted to work with the relay modell.

node_id

Holds the node id of the node to form a list topology.

Type `int`

left

Holds the RelayId of the left neighbor Relay connection.

Type [RelayId](#)

right

Holds the RelayId of the right neighbor Relay connection.

Type [RelayId](#)

check_in_original_variables(*relay_id*: RelayModel.RelayId.RelayId)

Overrides the method from the node class.

It checks if the given RelayId is present in the left or right neighbor variable.

Parameters **relay_id** (RelayId) – Defines the RelayId that should be checked.

Returns True if the given RelayId is in one of the variables, False otherwise.

Return type bool

get_relays_from_original_variables()

Overrides the method from the node class.

It forms a list with the left and right neighbor and returns it.

Returns A list containing the left and right neighbor RelayIds. Or an empty list if none is set at the moment.

Return type list

property left

Handles the left neighbor attribute.

When setting the left neighbor it only accepts None or a RelayId as a value. When setting the neighbor the Relay gets validated after setting.

linearize(*v*: RelayModel.RelayId.RelayId)

Impelements the linearize method from the BuildList protocol.

For further information see the thesis of this module.

Parameters **v** (RelayId) – The connection that should be linearized.

original_timeout()

Overrides the method from the node class to implement a BuildList protocol.

The method corrects left and right neighbors and introduces itself to them. For further information see the thesis of this module.

remove_from_original_variables(*relay_id*: RelayModel.RelayId.RelayId)

Overrides the method from the node class.

It removes the given RelayId from the left or right neighbor if they are set to this RelayId.

Parameters **relay_id** (RelayId) – The RelayId that should be deleted from the variables.

reversal_of_relay(*relay_id*: RelayModel.RelayId.RelayId)

Overrides the method from the node class.

The reversal of a relay is given by sending a linearize action to the given relay with the in_ref as parameter.

Parameters **relay_id** (RelayId) – Defines the Relay connection that should be reversed.

property right

Handles the right neighbor attribute.

When setting the right neighbor it only accepts None or a RelayId as a value. When setting the neighbor the Relay gets validated after setting.

send_analyse_state()

Overrides method from the node class.

Creates a SortedListNodeState with all information needed and sends it to the StateMonitor.

Returns Returns the system state. True if system is valid, False otherwise.

Return type bool

class RelayModel.SortedListNode.SortedListNodeState(*node_id, left, right, relays, running, leaving*)

Bases: object

Holds all information needed to pass a state of the SortedListNode.

This class is needed to analyse a system of sorted list nodes. It gets send to the StateMonitor while simulating.

node_id

Holds the node id of the node.

Type int

left

Holds the relay which defines the connection of the left neighbor.

Type *StateRelay*

right

Holds the relay which defines the connection of the right neighbor.

Type *StateRelay*

relays

Holds a list of StateRelays which containing relays that are present in the underlying RelayLayer.

Type list

running

Holds the running status of the node.

Type bool

leaving

Holds the leaving status of the node.

Type bool

class RelayModel.SortedListNode.StateRelay(*relay*)

Bases: object

Holds every information needed to analyse Relays.

This class is used to send Relay information to the StateMonitor. It should only be used in a State object.

relay_id

Holds the RelayId of the Relay which information should be stored.

Type *RelayId*

alive

Holds the alive state of the Relay.

Type bool

sink_rid

Holds the rid of the sink relay of the relay connection.

Type *RelayLayerId*

direct

Holds the direct information of the Relay.

Type bool

1.1.13 RelayModel.StateMonitor module

class RelayModel.StateMonitor.StateMonitor(*node_count*)

The StateMonitor watches the state of a simulated system, checks for validation and monitors timeouts.

The StateMonitor is a class that can watch a system from outside and monitors the simulated nodes. It stores the states of the nodes and analyse it until the system gets valid. It also stores the timeout counts of each node so we can later analyse it. The in depth description of the StateMonitor is given in the thesis of this framework.

node_count

Stores the amount of nodes present in the system.

Type int

node_timeout_count

Stores the timeout counts of each node in a dict. The key of the dict is the node id and the value is a integer counting the timeouts executed by this node.

Type dict

nodes

Stores the node states of each node in a dict. The key of this dict is the node id of the node and the value is a node state storing the state of this specific node.

Type dict

valid

Stores the state of the system. False if not valid. True if system is valid.

Type bool

notify_nodes

Stores a set of node ids to check if a node should be notified after system got valid. The StateMonitor waits after the system got valid until every node got notified about this.

Type set

logger

Holds the logger object for this class.

Type logging.Logger

listening_thread

Holds the Thread for listening for node states.

Type threading.Thread

analyse_thread

Holds the Thread for analysing the system.

Type threading.Thread

stop()

Stops the StateMonitor.

The StateMonitor gets stopped by this function by setting the internal running flag to False. After that all threads gets stopped.

1.1.14 RelayModel.Validation module

`RelayModel.Validation.BFS(visited_nodes, adjacency)`

Breadth first search algorithm.

The function takes a list of visiting information where the index defines a node and the value defines the visited information. Zero stands for not visited. One stands for queued but not visited yet. Two stands for visited. It also takes a adjacency matrix to go along edges. The method writes the results directly in the visited nodes variable given in the parameters.

Parameters

- **visited_nodes** (*list*) – A list of visited information for each node.
- **adjacency** (*list*) – A adjacency matrix.

`RelayModel.Validation.check_sorted_list_node_is_valid(node:`

`RelayModel.SortedListNode.SortedListNode,`
`node_ids, should_be_connected=False)`

Checks if a given SortedListNode is valid.

The function checks if the given Node is valid. That means they have to be connected if they are staying and should not be connected to other nodes if they are leaving. Furthermore it checks if the connections are direct and if the connections are to the right node.

Parameters

- **node** (*SortedListNode*) – The node that should be checked as a SortedListNode object.
- **node_ids** (*list*) – Defines a list of all staying node ids in the system. This list should be sorted.
- **should_be_connected** (*bool, optional*) – Defines if a node should be connected or not. By default this is set to False. This should be set to True if a staying node is checked and False otherwise.

Returns True if the given node is a valid SortedListNode, False otherwise.

Return type bool

`RelayModel.Validation.check_weak_connectivity(nodes)`

Checks a system of nodes for weakly connectivity.

The function takes a dict of nodes where the key is the node id and the value is the node itself. Then it creates a adjacency matrix with the relays in the node and checks the weakly connectivity with the BFS function.

Parameters **nodes** (*dict*) – The nodes dict where the key is the node id and the value the node object.

Returns True if the nodes are weakly connected. False otherwise.

Return type bool

`RelayModel.Validation.system_has_valid_state(nodes, logger)`

Checks a system of nodes if the system is in a valid state.

The method takes a dictionary of nodes. Every value of the dict should be a Node object. It checks every node if they are stopped when they are leaving and if they are running if they stay. Also it checks if every node is a valid sorted list node with the `check_sorted_list_node_is_valid` function.

If one condition fails it returns False. Only if all nodes are valid it checks if all staying nodes form a weakly connected graph. If that is given it returns True otherwise False.

To log the validation process a logger can be passed to the function.

Parameters

- **nodes** (*dict*) – The node dict where every value is a Node object.
- **logger** (*logging.Logger*) – A logger object to log some results of the validation process.

Returns True if system of nodes is valid False otherwise.

Return type bool

PYTHON MODULE INDEX

r

- [RelayModel](#), [1](#)
- [RelayModel.Communication](#), [1](#)
- [RelayModel.ConceptChange](#), [6](#)
- [RelayModel.GraphGeneration](#), [6](#)
- [RelayModel.KeyGeneration](#), [7](#)
- [RelayModel.LinkLayer](#), [8](#)
- [RelayModel.ModuleConfig](#), [10](#)
- [RelayModel.Node](#), [12](#)
- [RelayModel.Relay](#), [16](#)
- [RelayModel.RelayId](#), [19](#)
- [RelayModel.RelayLayer](#), [20](#)
- [RelayModel.RelayLogging](#), [26](#)
- [RelayModel.SortedListNode](#), [26](#)
- [RelayModel.StateMonitor](#), [29](#)
- [RelayModel.Validation](#), [30](#)

A

a_out (RelayModel.Node.Node attribute), 13
 Action (class in RelayModel.Communication), 1
 action (RelayModel.Communication.LayerMessage attribute), 3
 action (RelayModel.Communication.Message attribute), 3
 action_type (RelayModel.Communication.Action attribute), 1
 active (RelayModel.RelayLayer.RelayLayer attribute), 21
 add_in_reference() (RelayModel.Relay.Relay method), 17
 add_key() (RelayModel.Relay.OutReference method), 16
 add_relay_to_layer() (RelayModel.RelayLayer.RelayLayer method), 21
 alive (RelayModel.Relay.Relay attribute), 17
 alive (RelayModel.SortedListNode.StateRelay attribute), 28
 analyse_mode (RelayModel.Node.Node attribute), 13
 analyse_thread (RelayModel.StateMonitor.StateMonitor attribute), 29
 ask_to_reverse() (RelayModel.Node.Node method), 13
 ask_to_reverse_anchor() (RelayModel.Node.Node method), 13

B

BFS() (in module RelayModel.Validation), 30
 Blueprint (class in RelayModel.GraphGeneration), 6
 buffer (RelayModel.Node.Node attribute), 12
 buffer_put_actions (RelayModel.RelayLayer.RelayLayer attribute), 21
 buffer_put_thread (RelayModel.RelayLayer.RelayLayer attribute), 21

C

calculate_windows() (in module RelayModel.ConceptChange), 6
 call_method() (RelayModel.Node.Node method), 13
 calling_lock (RelayModel.RelayLayer.RelayLayer attribute), 21
 CHANGE_ALPHA (in module RelayModel.ModuleConfig), 10
 check_dos_attack() (RelayModel.RelayLayer.RelayLayer method), 22
 check_has_key_as_parameter() (RelayModel.Communication.Action method), 1
 check_in_original_variables() (RelayModel.Node.Node method), 13
 check_in_original_variables() (RelayModel.SortedListNode.SortedListNode method), 26
 check_key_in_relay_buffer() (RelayModel.LinkLayer.LinkLayer method), 8
 check_key_origin() (in module RelayModel.KeyGeneration), 7
 check_relay_exists() (RelayModel.RelayLayer.RelayLayer method), 22
 check_sorted_list_node_is_valid() (in module RelayModel.Validation), 30
 check_valid() (RelayModel.Relay.InReference method), 16
 check_weak_connectivity() (in module RelayModel.Validation), 30
 clear_in_relays() (RelayModel.Relay.Relay method), 18
 CONSIDER_AS_CLOSED (in module RelayModel.ModuleConfig), 10
 control_keys (RelayModel.Communication.ProbeAction attribute), 4
 control_keys (RelayModel.Communication.ProbeAction property), 4

D

D (*RelayModel.Node.Node* attribute), 12

delete() (*RelayModel.RelayLayer.RelayLayer* method), 22

direct (*RelayModel.SortedListNode.StateRelay* attribute), 28

direction (*RelayModel.GraphGeneration.OutBlueprint* attribute), 7

DOS_DETECTION_ACTIVATED (in module *RelayModel.ModuleConfig*), 10

dos_threshold (*RelayModel.Relay.Relay* attribute), 17

F

FailureMessage (class in *RelayModel.Communication*), 1

G

generate_key() (in module *RelayModel.KeyGeneration*), 7

generate_prefix() (in module *RelayModel.KeyGeneration*), 7

get_key_for_layer() (*RelayModel.RelayLayer.RelayLayer* method), 22

get_level() (*RelayModel.RelayLayer.RelayLayer* method), 22

get_logger() (in module *RelayModel.RelayLogging*), 26

get_new_relay_object() (*RelayModel.RelayLayer.RelayLayer* method), 22

get_relay_by_relay_id() (*RelayModel.RelayLayer.RelayLayer* method), 22

get_relays() (*RelayModel.RelayLayer.RelayLayer* method), 22

get_relays_from_original_variables() (*RelayModel.Node.Node* method), 14

get_relays_from_original_variables() (*RelayModel.SortedListNode.SortedListNode* method), 27

get_sink_node_id() (*RelayModel.RelayLayer.RelayLayer* method), 23

get_valid_keys() (*RelayModel.Relay.Relay* method), 18

get_validated_relays() (*RelayModel.RelayLayer.RelayLayer* method), 23

H

handle_in_relay_closed() (*RelayModel.RelayLayer.RelayLayer* method), 23

handle_not_authorized() (*RelayModel.RelayLayer.RelayLayer* method), 23

handle_out_relay_closed() (*RelayModel.RelayLayer.RelayLayer* method), 23

handle_ping() (*RelayModel.RelayLayer.RelayLayer* method), 23

handle_probe_fail() (*RelayModel.RelayLayer.RelayLayer* method), 24

handle_transmit() (*RelayModel.RelayLayer.RelayLayer* method), 24

has_incoming() (*RelayModel.Relay.Relay* method), 18

has_incoming() (*RelayModel.RelayLayer.RelayLayer* method), 24

has_key_in_in_ref() (*RelayModel.Relay.Relay* method), 18

Header (class in *RelayModel.Communication*), 2

header (*RelayModel.Communication.Message* attribute), 3

I

in_blueprints (*RelayModel.GraphGeneration.Blueprint* attribute), 6

in_ref (*RelayModel.Node.Node* attribute), 12

in_relays (*RelayModel.Relay.Relay* attribute), 17

InBlueprint (class in *RelayModel.GraphGeneration*), 6

InReference (class in *RelayModel.Relay*), 16

InRelayClosedAction (class in *RelayModel.Communication*), 2

ip (*RelayModel.RelayId.RelayLayerId* attribute), 19

is_buffer_empty() (*RelayModel.LinkLayer.LinkLayer* method), 8

is_dead() (*RelayModel.Relay.Relay* method), 18

is_dead() (*RelayModel.RelayLayer.RelayLayer* method), 24

is_direct() (*RelayModel.Relay.Relay* method), 18

is_direct() (*RelayModel.RelayLayer.RelayLayer* method), 24

is_sink() (*RelayModel.Relay.Relay* method), 18

is_sink() (*RelayModel.RelayLayer.RelayLayer* method), 24

K

key (*RelayModel.Communication.Ping* attribute), 4

key (*RelayModel.Communication.ProbeFailAction* attribute), 5

key (*RelayModel.Communication.ProbeFailAction* property), 5

key (*RelayModel.Communication.RelayParameter* attribute), 5

- key (*RelayModel.GraphGeneration.InBlueprint attribute*), 6
- key (*RelayModel.GraphGeneration.OutBlueprint attribute*), 6
- key (*RelayModel.Relay.InReference attribute*), 16
- key (*RelayModel.Relay.InReference property*), 16
- key_sequence (*Relay-Model.Communication.ProbeAction attribute*), 4
- key_sequence (*Relay-Model.Communication.ProbeAction property*), 4
- key_sequence (*Relay-Model.Communication.ProbeFailAction attribute*), 5
- key_sequence (*Relay-Model.Communication.ProbeFailAction property*), 5
- keys (*RelayModel.Communication.Header attribute*), 2
- keys (*RelayModel.Communication.InRelayClosedAction attribute*), 2
- keys (*RelayModel.Communication.InRelayClosedAction property*), 2
- keys (*RelayModel.Communication.NotAuthorizedAction attribute*), 3
- keys (*RelayModel.Communication.NotAuthorizedAction property*), 3
- keys (*RelayModel.Relay.OutReference attribute*), 16
- keys (*RelayModel.Relay.OutReference property*), 17
- ## L
- last_id (*RelayModel.RelayLayer.RelayLayer attribute*), 20
- layer_id (*RelayModel.Communication.LayerMessage attribute*), 2
- layer_id (*RelayModel.RelayId.RelayId attribute*), 19
- LayerMessage (*class in RelayModel.Communication*), 2
- leaving (*RelayModel.Node.Node attribute*), 12
- leaving (*RelayModel.SortedListNode.SortedListNodeState attribute*), 28
- left (*RelayModel.SortedListNode.SortedListNode attribute*), 26
- left (*RelayModel.SortedListNode.SortedListNode property*), 27
- left (*RelayModel.SortedListNode.SortedListNodeState attribute*), 28
- level (*RelayModel.Communication.Ping attribute*), 4
- level (*RelayModel.Communication.RelayParameter attribute*), 5
- level (*RelayModel.Relay.Relay attribute*), 17
- linearize() (*RelayModel.SortedListNode.SortedListNode method*), 27
- link_layer_buffer (*Relay-Model.RelayLayer.RelayLayer attribute*), 20
- link_layer_call_pipe (*Relay-Model.RelayLayer.RelayLayer attribute*), 20
- link_layer_process (*Relay-Model.RelayLayer.RelayLayer attribute*), 20
- LinkLayer (*class in RelayModel.LinkLayer*), 8
- listening_thread (*Relay-Model.StateMonitor.StateMonitor attribute*), 29
- logger (*RelayModel.Node.Node attribute*), 12
- logger (*RelayModel.RelayLayer.RelayLayer attribute*), 21
- logger (*RelayModel.StateMonitor.StateMonitor attribute*), 29
- ## M
- make_weakly_connected_sorted_list() (*in module RelayModel.GraphGeneration*), 7
- merge() (*RelayModel.RelayLayer.RelayLayer method*), 24
- Message (*class in RelayModel.Communication*), 3
- message (*RelayModel.Communication.TransmitMessage attribute*), 5
- message_buffer (*RelayModel.RelayLayer.RelayLayer attribute*), 20
- message_handling_thread (*Relay-Model.RelayLayer.RelayLayer attribute*), 21
- message_thread (*RelayModel.Node.Node attribute*), 12
- module
- RelayModel, 1
 - RelayModel.Communication, 1
 - RelayModel.ConceptChange, 6
 - RelayModel.GraphGeneration, 6
 - RelayModel.KeyGeneration, 7
 - RelayModel.LinkLayer, 8
 - RelayModel.ModuleConfig, 10
 - RelayModel.Node, 12
 - RelayModel.Relay, 16
 - RelayModel.RelayId, 19
 - RelayModel.RelayLayer, 20
 - RelayModel.RelayLogging, 26
 - RelayModel.SortedListNode, 26
 - RelayModel.StateMonitor, 29
 - RelayModel.Validation, 30
- monitor_timeout_transmit_rate() (*Relay-Model.RelayLayer.RelayLayer method*), 24
- monitor_transmit() (*Relay-Model.RelayLayer.RelayLayer method*), 25

N

`N` (*RelayModel.Node.Node* attribute), 12

`new_relay()` (*RelayModel.RelayLayer.RelayLayer* method), 25

`NO_MONITOR_ACTIONS` (in module *RelayModel.ModuleConfig*), 11

`Node` (class in *RelayModel.Node*), 12

`node_count` (*RelayModel.StateMonitor.StateMonitor* attribute), 29

`node_id` (*RelayModel.RelayId.RelayLayerId* attribute), 20

`node_id` (*RelayModel.RelayId.RelayLayerId* property), 20

`node_id` (*RelayModel.SortedListNode.SortedListNode* attribute), 26

`node_id` (*RelayModel.SortedListNode.SortedListNodeState* attribute), 28

`node_timeout_count` (*RelayModel.StateMonitor.StateMonitor* attribute), 29

`NODE_TIMEOUT_PERIOD` (in module *RelayModel.ModuleConfig*), 10

`nodes` (*RelayModel.StateMonitor.StateMonitor* attribute), 29

`NotAuthorizedAction` (class in *RelayModel.Communication*), 3

`notify_anchor()` (*RelayModel.Node.Node* method), 14

`notify_nodes` (*RelayModel.StateMonitor.StateMonitor* attribute), 29

O

`original_timeout()` (*RelayModel.Node.Node* method), 14

`original_timeout()` (*RelayModel.SortedListNode.SortedListNode* method), 27

`out_blueprints` (*RelayModel.GraphGeneration.Blueprint* attribute), 6

`out_id` (*RelayModel.Communication.Header* attribute), 2

`out_id` (*RelayModel.Communication.NotAuthorizedAction* attribute), 3

`out_id` (*RelayModel.Communication.NotAuthorizedAction* property), 3

`out_id` (*RelayModel.GraphGeneration.OutBlueprint* attribute), 7

`out_id` (*RelayModel.Relay.OutReference* attribute), 16

`out_id` (*RelayModel.Relay.OutReference* property), 17

`out_relay` (*RelayModel.Relay.Relay* attribute), 17

`OutBlueprint` (class in *RelayModel.GraphGeneration*), 6

`OutReference` (class in *RelayModel.Relay*), 16

`OutRelayClosedAction` (class in *RelayModel.Communication*), 3

P

`parameters` (*RelayModel.Communication.Action* attribute), 1

`Ping` (class in *RelayModel.Communication*), 4

`PingAction` (class in *RelayModel.Communication*), 4

`pings` (*RelayModel.Communication.PingAction* attribute), 4

`pings` (*RelayModel.Communication.PingAction* property), 4

`POLL_TIMEOUT` (in module *RelayModel.ModuleConfig*), 11

`POLL_TRIES` (in module *RelayModel.ModuleConfig*), 11

`port` (*RelayModel.RelayId.RelayLayerId* attribute), 20

`ProbeAction` (class in *RelayModel.Communication*), 4

`ProbeFailAction` (class in *RelayModel.Communication*), 5

R

`receiving_relay` (*RelayModel.Communication.Action* attribute), 1

`register_relay_layer()` (*RelayModel.LinkLayer.LinkLayer* method), 8

`Relay` (class in *RelayModel.Relay*), 17

`relay` (*RelayModel.Relay.InReference* attribute), 16

`relay` (*RelayModel.Relay.InReference* property), 16

`relay_id` (*RelayModel.Communication.InRelayClosedAction* attribute), 2

`relay_id` (*RelayModel.Communication.InRelayClosedAction* property), 2

`relay_id` (*RelayModel.Communication.OutRelayClosedAction* attribute), 3

`relay_id` (*RelayModel.Communication.OutRelayClosedAction* property), 3

`relay_id` (*RelayModel.Communication.Ping* attribute), 4

`relay_id` (*RelayModel.Communication.RelayParameter* attribute), 5

`relay_id` (*RelayModel.Relay.Relay* attribute), 17

`relay_id` (*RelayModel.RelayId.RelayId* attribute), 19

`relay_id` (*RelayModel.SortedListNode.StateRelay* attribute), 28

`relay_layer` (*RelayModel.Node.Node* attribute), 12

`RELAY_LAYER_TIMEOUT_PERIOD` (in module *RelayModel.ModuleConfig*), 11

`relay_list_to_string()` (in module *RelayModel.RelayLogging*), 26

`RELAY_LOG_LEVEL` (in module *RelayModel.ModuleConfig*), 11

`relay_windows` (*RelayModel.RelayLayer.RelayLayer* attribute), 21

`RelayId` (class in *RelayModel.RelayId*), 19

- RelayLayer (class in RelayModel.RelayLayer), 20
- RelayLayerId (class in RelayModel.RelayId), 19
- RelayModel
- module, 1
- RelayModel.Communication
- module, 1
- RelayModel.ConceptChange
- module, 6
- RelayModel.GraphGeneration
- module, 6
- RelayModel.KeyGeneration
- module, 7
- RelayModel.LinkLayer
- module, 8
- RelayModel.ModuleConfig
- module, 10
- RelayModel.Node
- module, 12
- RelayModel.Relay
- module, 16
- RelayModel.RelayId
- module, 19
- RelayModel.RelayLayer
- module, 20
- RelayModel.RelayLogging
- module, 26
- RelayModel.SortedListNode
- module, 26
- RelayModel.StateMonitor
- module, 29
- RelayModel.Validation
- module, 30
- RelayParameter (class in RelayModel.Communication), 5
- relays (RelayModel.RelayLayer.RelayLayer attribute), 20
- relays (RelayModel.SortedListNode.SortedListNodeState attribute), 28
- remove_from_original_variables() (RelayModel.Node.Node method), 14
- remove_from_original_variables() (RelayModel.SortedListNode.SortedListNode method), 27
- remove_in_reference() (RelayModel.Relay.Relay method), 18
- remove_in_reference_by_key_and_rid() (RelayModel.Relay.Relay method), 18
- remove_in_reference_by_relay() (RelayModel.Relay.Relay method), 19
- remove_in_reference_by_relay_and_key() (RelayModel.Relay.Relay method), 19
- remove_in_reference_by_rid() (RelayModel.Relay.Relay method), 19
- remove_key() (RelayModel.Relay.OutReference method), 17
- replace_action() (RelayModel.Node.Node method), 14
- replace_relay_in_references() (RelayModel.Relay.Relay method), 19
- RESULTS_FOLDER (in module RelayModel.ModuleConfig), 11
- reversal_of_relay() (RelayModel.Node.Node method), 14
- reversal_of_relay() (RelayModel.SortedListNode.SortedListNode method), 27
- reverse() (RelayModel.Node.Node method), 15
- rid (RelayModel.Communication.RelayParameter attribute), 5
- rid (RelayModel.GraphGeneration.InBlueprint attribute), 6
- rid (RelayModel.Relay.InReference attribute), 16
- rid (RelayModel.Relay.InReference property), 16
- RID (RelayModel.RelayLayer.RelayLayer attribute), 20
- right (RelayModel.SortedListNode.SortedListNode attribute), 26
- right (RelayModel.SortedListNode.SortedListNode property), 27
- right (RelayModel.SortedListNode.SortedListNodeState attribute), 28
- running (RelayModel.LinkLayer.LinkLayer attribute), 8
- running (RelayModel.Node.Node attribute), 12
- running (RelayModel.RelayLayer.RelayLayer attribute), 21
- running (RelayModel.SortedListNode.SortedListNodeState attribute), 28
- ## S
- same_target() (RelayModel.Relay.Relay method), 19
- same_target() (RelayModel.RelayLayer.RelayLayer method), 25
- send() (RelayModel.RelayLayer.RelayLayer method), 25
- send_analyse_state() (RelayModel.Node.Node method), 15
- send_analyse_state() (RelayModel.SortedListNode.SortedListNode method), 27
- send_message() (in module RelayModel.LinkLayer), 9
- send_state_to_monitor() (RelayModel.Node.Node method), 15
- sender_rid (RelayModel.Communication.Header attribute), 2
- sender_rid (RelayModel.Communication.InRelayClosedAction attribute), 2
- sender_rid (RelayModel.Communication.InRelayClosedAction property), 2

shutdown() (*RelayModel.LinkLayer.LinkLayer method*),
 8
 shutdown() (*RelayModel.Node.Node method*), 15
 shutdown() (*RelayModel.RelayLayer.RelayLayer
 method*), 25
 sink_rid (*RelayModel.Communication.Ping attribute*),
 4
 sink_rid (*RelayModel.GraphGeneration.OutBlueprint
 attribute*), 6
 sink_rid (*RelayModel.Relay.Relay attribute*), 17
 sink_rid (*RelayModel.SortedListNode.StateRelay
 attribute*), 28
 SortedListNode (*class in RelayModel.SortedListNode*),
 26
 SortedListNodeState (*class in Relay-
 Model.SortedListNode*), 28
 start() (*RelayModel.Node.Node method*), 15
 start_layer_buffer_watch() (*in module Relay-
 Model.LinkLayer*), 9
 start_link_layer() (*in module Relay-
 Model.LinkLayer*), 9
 start_listening() (*in module Relay-
 Model.LinkLayer*), 10
 STATE_MONITOR_ADDRESS (*in module Relay-
 Model.ModuleConfig*), 11
 STATE_MONITOR_PORT (*in module Relay-
 Model.ModuleConfig*), 11
 StateMonitor (*class in RelayModel.StateMonitor*), 29
 StateRelay (*class in RelayModel.SortedListNode*), 28
 stop() (*RelayModel.Node.Node method*), 15
 stop() (*RelayModel.RelayLayer.RelayLayer method*), 25
 stop() (*RelayModel.StateMonitor.StateMonitor
 method*), 29
 stop_relay_watch() (*Relay-
 Model.LinkLayer.LinkLayer method*), 9
 stopped (*RelayModel.LinkLayer.LinkLayer attribute*), 8
 SuccessMessage (*class in Relay-
 Model.Communication*), 5
 system_has_valid_state() (*in module Relay-
 Model.Validation*), 30

T

timeout() (*RelayModel.Node.Node method*), 15
 timeout() (*RelayModel.RelayLayer.RelayLayer
 method*), 25
 timeout_period (*RelayModel.Node.Node attribute*), 12
 timeout_thread (*RelayModel.Node.Node attribute*), 12
 timeout_thread (*RelayModel.RelayLayer.RelayLayer
 attribute*), 21
 transmit_count (*RelayModel.RelayLayer.RelayLayer
 attribute*), 21
 TransmitMessage (*class in Relay-
 Model.Communication*), 5

V

valid (*RelayModel.StateMonitor.StateMonitor at-
 tribute*), 29
 validate_relay() (*Relay-
 Model.RelayLayer.RelayLayer method*),
 25
 validated (*RelayModel.Relay.Relay attribute*), 17

W

WINDOW_SIZE (*in module RelayModel.ModuleConfig*), 11