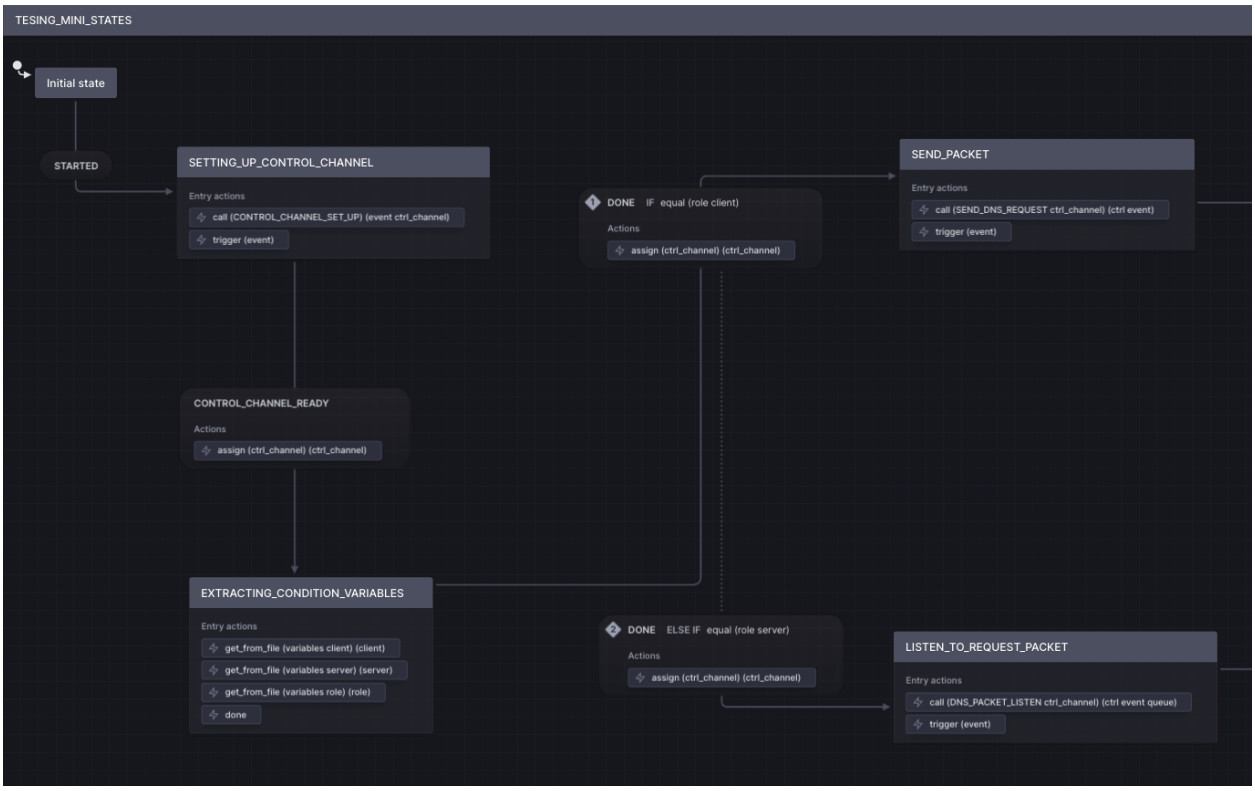
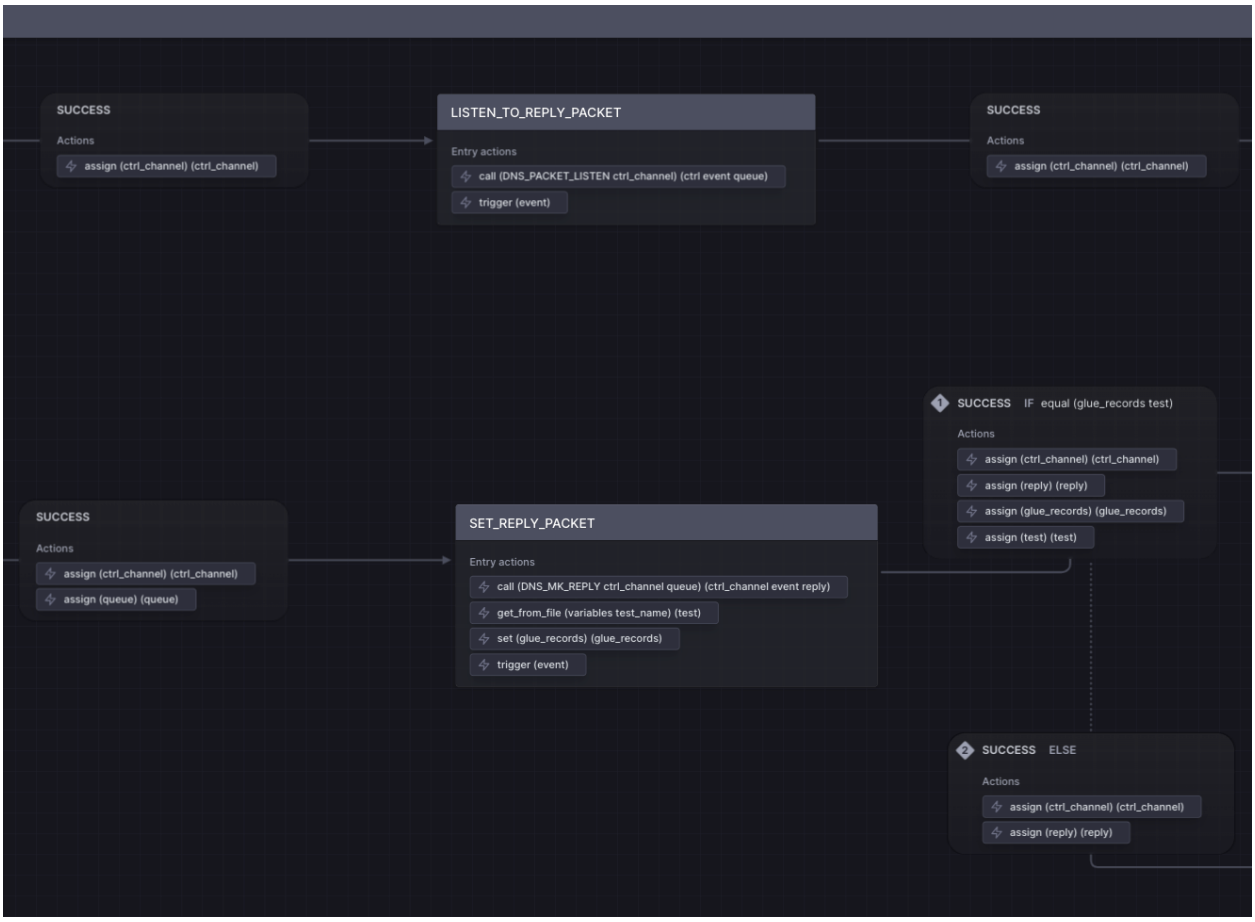
Guide to Using the “mini” State Machine in a Main scenario

This report is a detailed explanation of using the state machine diagrams applauded in this GitHub repository in main test scenario. The scenario below represents the test case of a client sending a request message to the resolver. The resolver forwards this request to the server, which crafts a reply. Subsequently, the server sends the reply to the client through both the data link and the control channel.

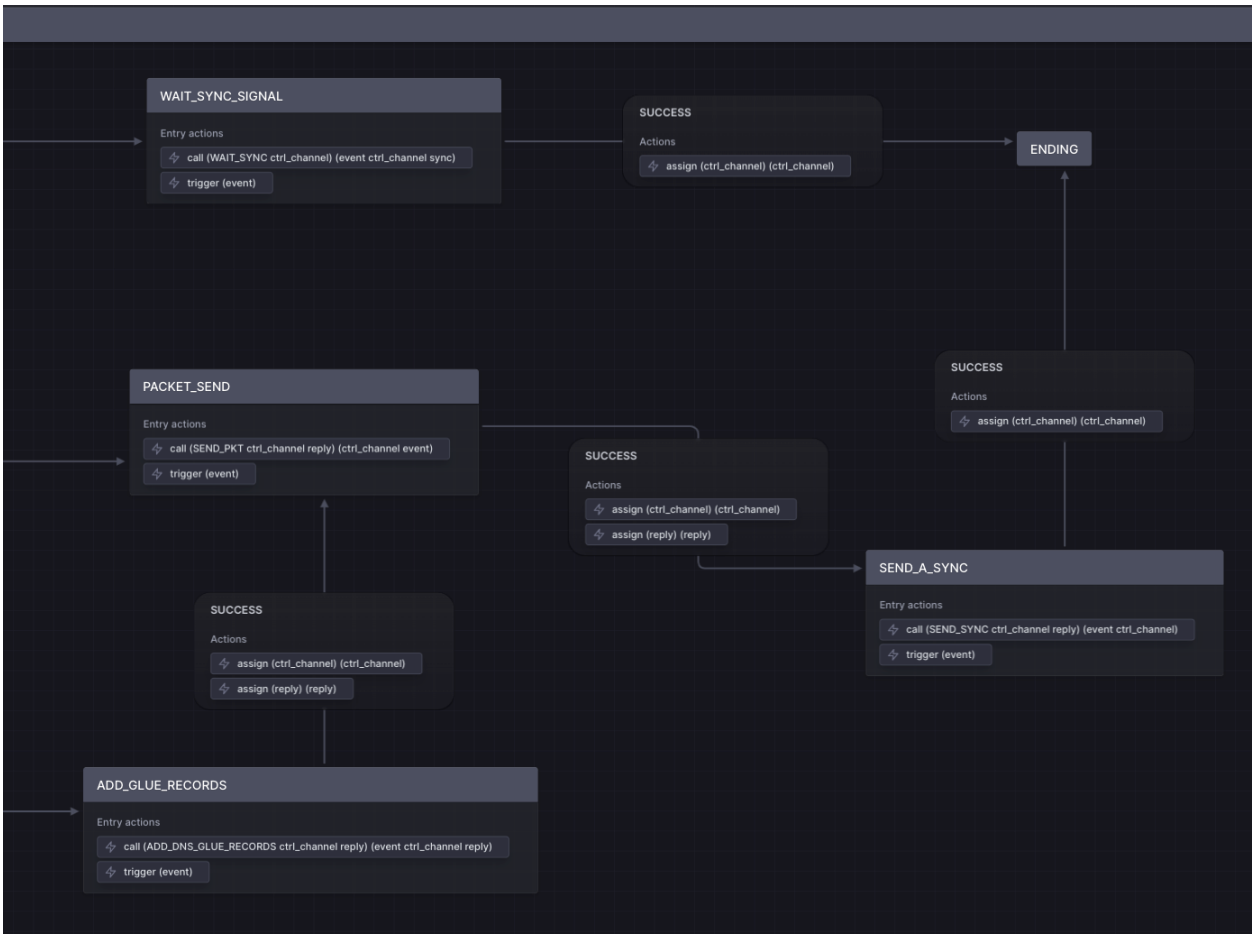
MAIN SCENARIO (PART1)



MAIN SCENARIO (PART2)

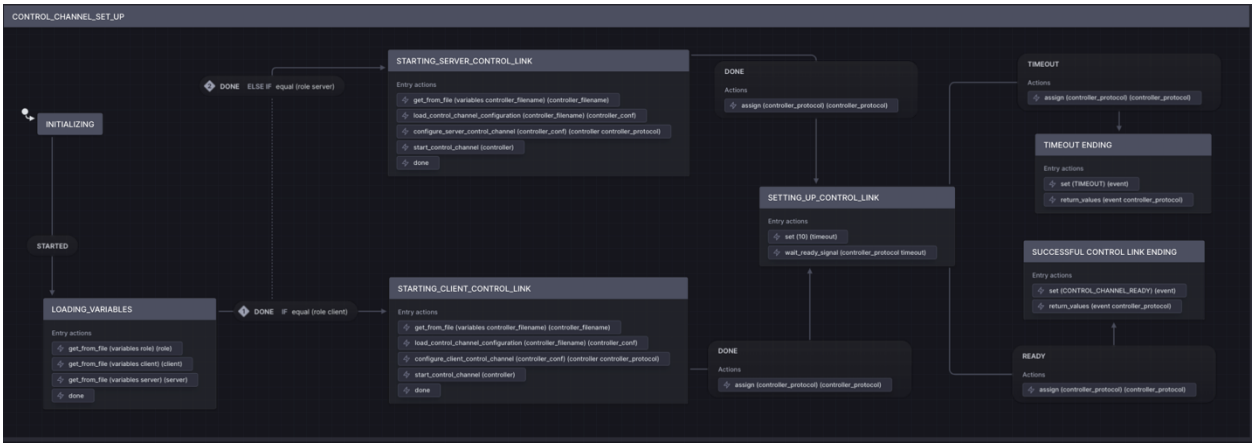


MAIN SCENARIO (PART3)



The state machine diagram scenario starts with the initial state triggering the event “STARTED” and moves to the “SETTING_UP_CONTROL_CHANNEL” event. There, the CONTROL_CHANNEL_SET_UP state machine is called using the primitive CALL.

CONTROL_CHANNEL_SET_UP



In that state machine, the variable role is extracted from a file named variables.json, which includes all the variables needed in the scenario (and mini scenarios). This file is stored on both the client and the server. The file is almost the same at both ends, with the only differences being the “role” value,the packet filter, and the controller filename are also different. The role value corresponds to the IP address of the client or server, and is used as a conditional variable to lead the client and server to their specific path in the scenario.

variables.json on the client endpoint:

```
{
    "role": "192.168.123.31",
    "client": "192.168.123.31",
    "server": "192.168.123.11",
    "bind9" : "192.168.123.22",
    "type" : "A",
    "glue_type": "A",
    "value" : "10.10.10.10",
    "glue_value" : "11.11.11.11",
    "domain_name" : "host.zone-nps.iliesbenhabbour.com",
    "glue_domain" : "host.nopasaran.com",
    "packet_filter": "udp and dst host 192.168.123.31",
    "controller_filename" : "control_client.json"

}
```

variables.json on the server endpoint:

```
{
    "role": "192.168.123.11",
    "client": "192.168.123.31",
    "server": "192.168.123.11",
    "bind9" : "192.168.123.22",
    "type" : "A",
    "glue_type": "A",
    "value" : "10.10.10.10",
    "glue_value" : "11.11.11.11",
    "domain_name" : "host.zone-nps.iliesbenhabbour.com",
    "glue_domain" : "host.nopasaran.com",
    "packet_filter": "udp and dst host 192.168.123.11",
    "controller_filename" : "control_server.json"

}
```

Following that, the control channel configuration primitives create the connection between the two endpoints, and at the end of the CONTROL_CHANNEL_SET_UP scenario, it returns the value “ctrl_channel”, which will continue to be assigned for the rest of the state machine s to maintain the connection. Finally, it then triggers the event “CONTROL_CHANNEL_READY”.

To avoid explaining the primitive assign for the rest of this report, it is important to note that it is used whenever a variables is moved from one state to another through an event.

It is important to note that the control channel connection is prone to end if one of the endpoints loses connection by finishing the application first, so it is important to keep this in mind when designing state machine diagrams.

Moving forward, we reach the state "EXTRACTING_CONDITION_VARIABLES." Here, the primitive "get_from_file" is used to fetch the variables client, server, and role from the file variables.json. As discussed earlier, the "role" value varies between the client and server.

After that, we perform a conditional check using the "equal" primitive. We compare the role value with the IP addresses of both the client and server. This helps to lead the client and server to their specific path, letting them each do their part in the scenario.

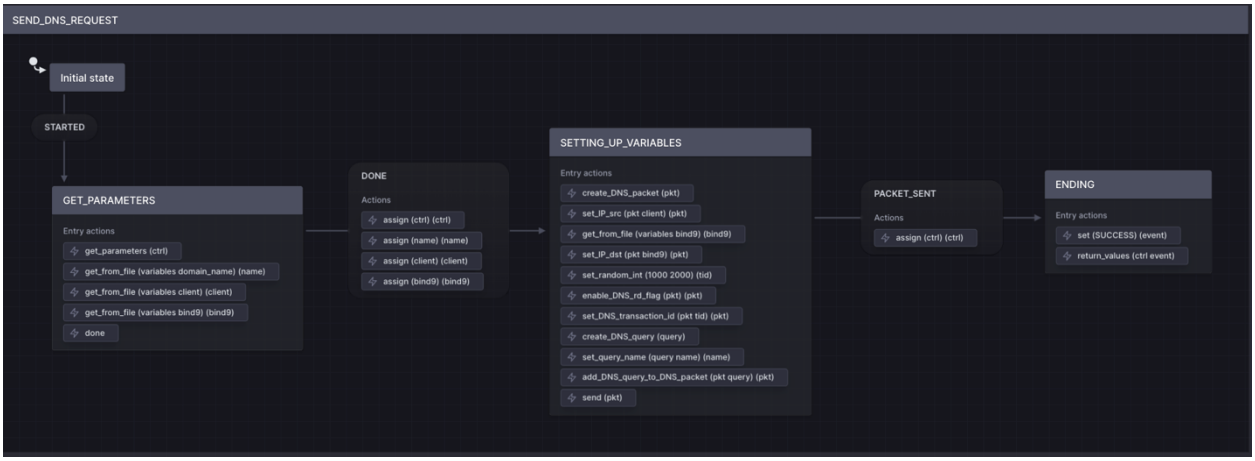
Note that this scenario is executed on both the server end and client at the same time. This means that it's crucial to design it in a way that ensures both endpoints can smoothly carry out their roles within the scenario.

From this point onwards, the explanation will split into two distinct sections, one for the server and the other for the client. This division is necessary because the path now diverges for both entities, the server and the client:

Client:

The client then calls the state machine “SEND_DNS_REQUEST”, which uses NoPASARAN’s DNS primitives to craft a DNS packet and sent it to the resolver, triggering the event PACKET_SENT, and returning the control channel and the event SUCCESS to the main scenario. Note that the event that is returned from the nested state machine is then triggered in the main test scenario.

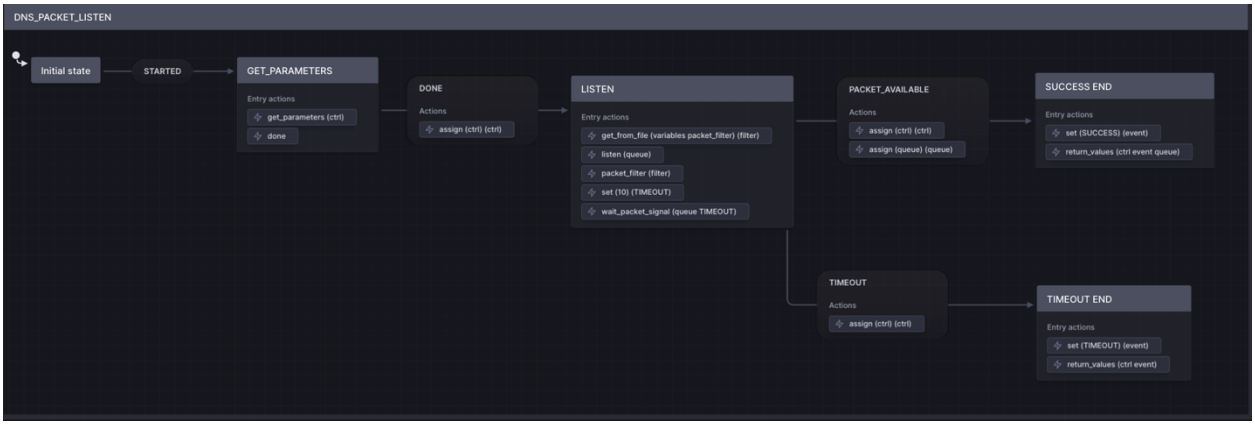
SEND_DNS_REQUEST:



Server:

The server calls the state machine “DNS_PACKET_LISTEN”, which takes the packet filter from the variables.json file and uses NoPASARAN’s data channel primitives to listen to the packet sent by the client.

DNS_PACKET_LISTEN



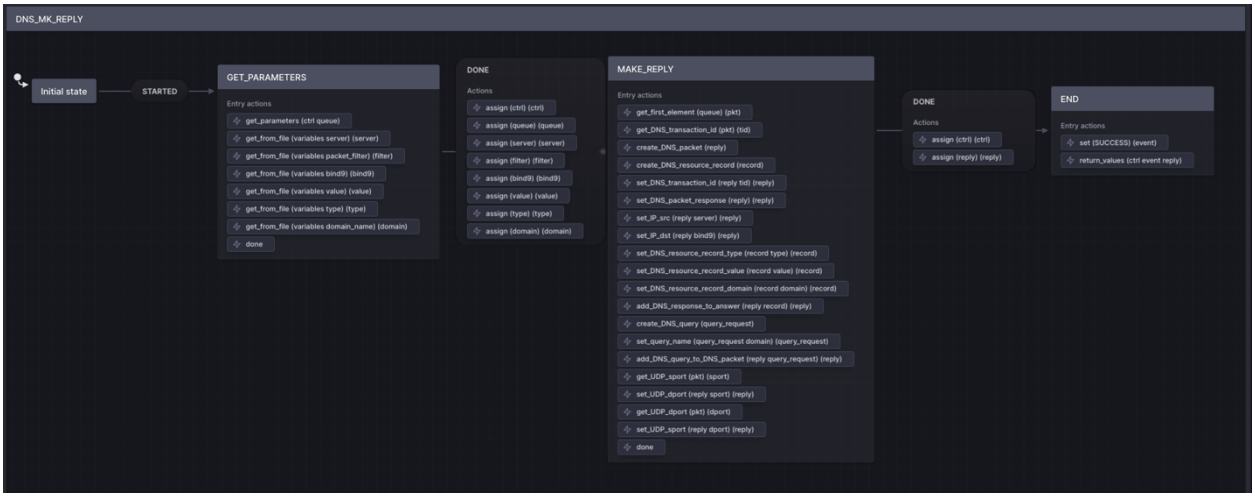
It is important to use the correct packet filter depending on the scenario, you want to avoid the server from listening to all packets on the packet sniffer.

(The parts in orange are for a proposed initial suggestion on how to make the scenario generic)

Server:

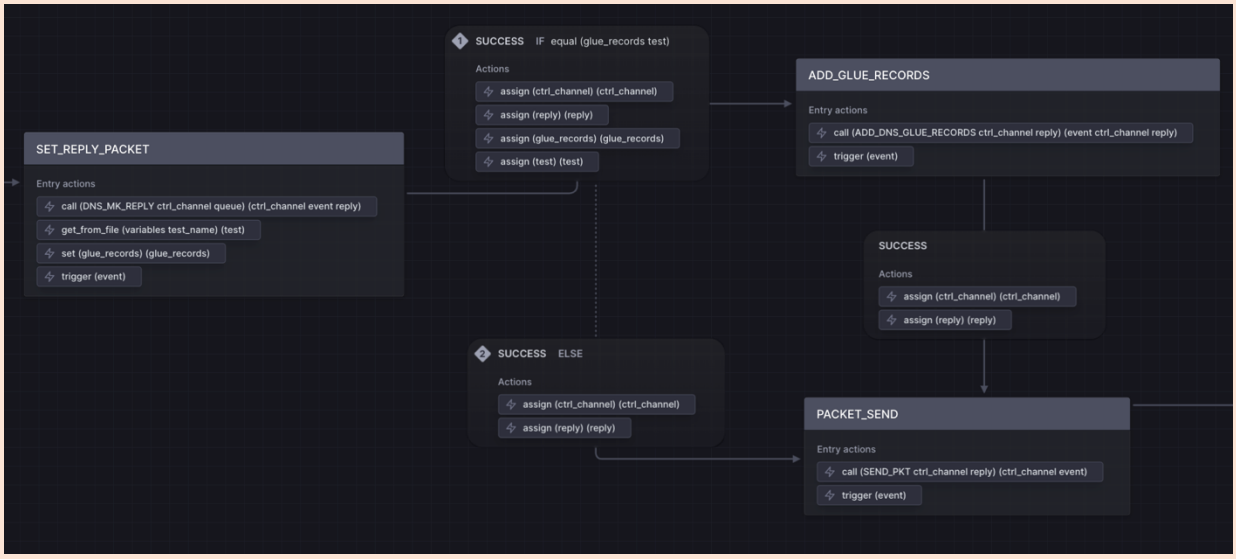
The next state for the server is “SET_REPLY_PACKET”, there the server calls the state machine “DNS_MK_REPKY”, which will use many of NoPASARAN’s DNS primitives to get the necessary packet information from the received packet, create the reply packet, and add an answer as a resource record in the packet. It then returns the control channel, the event SUCCESS, and the reply.

DNS_MK_REPKY:



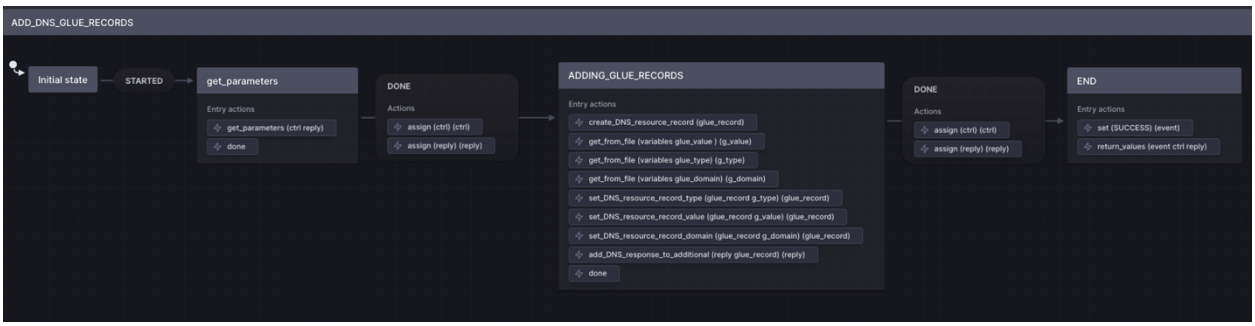
Following that is a proposed suggestion on how to make the scenario more generic .This proposal aims to enable the use of the same scenario for various tests. It's important to clarify that each time the scenario is executed, it performs a single specific test. Therefore, when referring to using the same scenario for multiple tests, it implies that each run performs one particular test out of several possibilities.

The primitive "get_from_file" gets a variable named "test_name" from the file variables.json. To run a specific test, you have to manually change this variable. (Assume that the value of test_name in this case is glue_records.) Next, using the primitive set, the variable glue_records is set to the value “glue_records”, which is equal to the value of the variable “test_name”. This primitive selection could use more enhancement as more tests could use the coming state which is “ADD_GLUE_RECORDS”. The equal primitives checks if “test_name” and “glue_records” are equal, which in this case, they are indeed equal. Thus, leading to the state “ADD_GLUE_RECORDS”. Note that more guards like the concept in this example can be used throughout the main state machine to make it generic to fit all DNS tests.



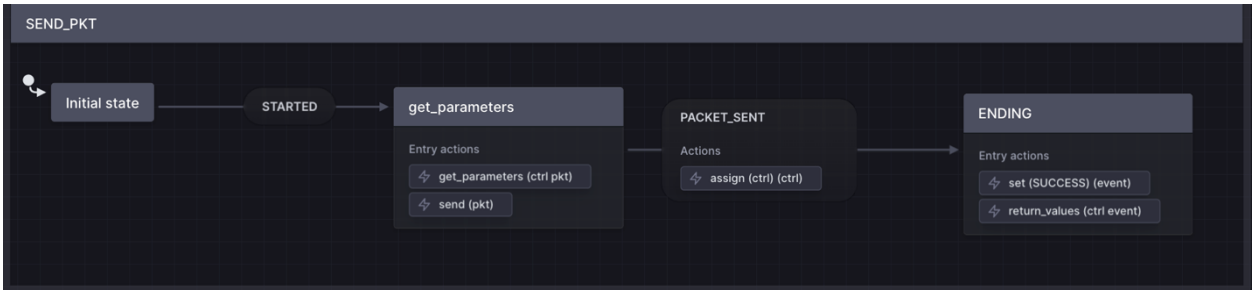
The next state is “ADD_GLUE_RECORDS”, which calls the state machine “ADD_DNS_GLUE_RECORDS”. The state machine then takes the reply, and adds another resource records in the additional answer section of the packet. It then returns the modified reply packet, control channel, and the event SUCCESS.

ADD_DNS_GLUE_RECORDS



Following that, the server goes to the state PACKET_SEND, which calls the state machine “SEND_PKT” to send the packet, triggering the event “PACKET_SENT”, and returning the event SUCCESS.

SEND_PKT



Client:

The next state in the client’s path is “LISTEN_TO_REPLY_PACKET”, it calls the state machine “DNS_PACKET_LISTEN” which takes the packet filter from the variables.json file and uses NoPASARAN’s data channel primitives to listen to the reply packet sent by the server.

Server:

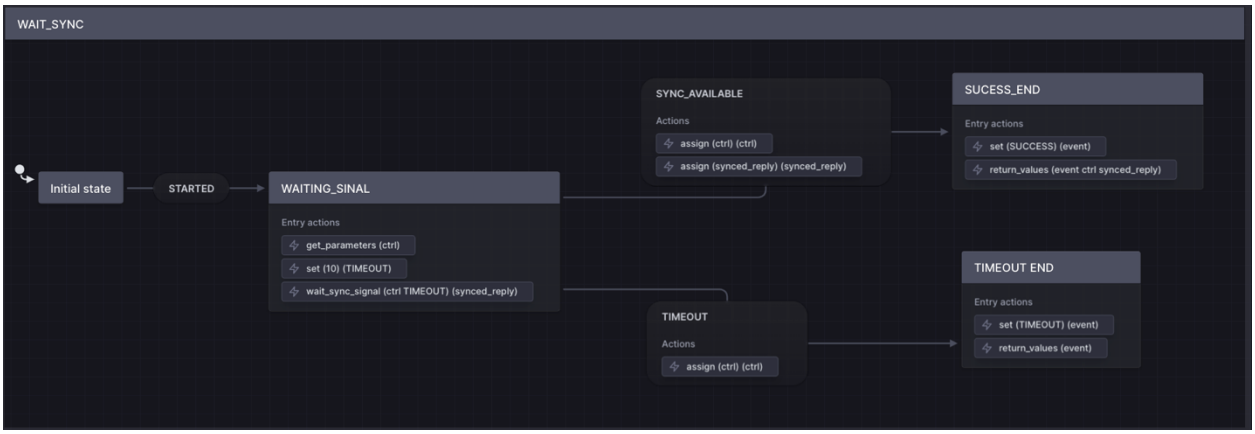
The server goes to the state “SEND_A_SYNC” upon sending the reply packet to the client via the data channel, only this time, it sends the packet through the control channel. The primitive CALL calls the state machine “SEND_SYNC”, there the packet is sent using NoPASARAN’s control channel primitive sync (), triggering the event “SYNC_SENT”, and returning the event SUCCESS, and the control channel to the main state machine.

SEND_SYNC

Client:

On the client end, the next state is “WAIT_SYNC_SIGNAL” where it calls the state machine “WAIT_SYNC”. There, the packet is received from the server through control channel using NoPASARAN’s control channel primitive wait_sync_signal(), triggering the event “SYNC_AVAILABLE”, and returning event SUCCESS, control channel, and the synced message.

WAIT_SYNC



The client has now received two messages: one from the data channel and another from the control channel. This opens up a wide range of possibilities going forward. Numerous packet components can be compared and analyzed to understand the resolver's behavior. These attributes may include the domain name, source port, and more.

Finally, the client and the server both reach the ending state, ending the control channel connection, and finishing the scenario.