MAJAN Guideline Document

Author: Akbar Kazimov

Acronyms

AJAN: Accessible Java Agent Nucleus

AKB: Agent Knowledge Base

BT: Behavior Tree

CMD: Command Prompt

CSGP: Coalition Structure Generation Problem

EKB: Execution Knowledge Base

IRI: International Resource Identifier

LAR: Local Agents Repository

MAC: Multiagent Coordination

MAJAN: Multi AJAN Agent Coordination

Prefixes

ajan: <http://www.ajan.de/ajan-ns#>
mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
xsd: <http://www.w3.org/2001/XMLSchema#>
domain: <http://localhost:8090/rdf4j/repositories/domain_specific_ontology#>

Document Structure

This document provides the necessary information to use MAJAN. It is structured sequentially starting from the basics, to the creation, execution, and finally evaluation of multiagent coordination use-cases into groups in the AJAN agent engineering tool.

MAJAN Overview

MAJAN is an extension of the agent engineering tool AJAN, which provides features to realize and evaluate SPARQL-BT-based distributed coordination of AJAN agents into groups. In case you are not familiar with AJAN, please refer to the respective wiki sections of GitHub repositories for AJAN Service and AJAN Editor.

AJAN Service is JAVA based execution engine to run intelligent *SPARQL-BT-based* AJAN agents. AJAN Editor is a web application that provides a user interface for functionalities such as designing behaviors of agents as SPARQL-BTs, creating, executing agents, and more.

Since AJAN hasn't supported multiagent coordination, **MAJAN** extends and adopts it appropriately in a way that AJAN users can develop **multiagent coordination use cases**, execute them and analyze the results of use cases easily by using provided features.

View a **video tutorial** of MAJAN features in this link: https://youtu.be/xN8KtZVryLU

MAJAN provides the features listed below:

1. **Template generic coordination BTs** to design multiagent coordination use-cases. These BTs cover all the required steps from the beginning until the end of coordination (into groups). Moreover, they (i.e. BTs) are *use-case independent*, and thus, users can customize them depending on domain-specific use-cases they have at hand.

2. **Postman collections** to **create** multiple agents and **execute** coordination use-cases easily with just one click. These collections cover creating agents, populating their agent repository, and starting coordination processes.

3. **Monitoring multiagent coordination activities** in AJAN Editor. AJAN provides logs in Netbeans or CMD depending on where AJAN Service is executed. However, agent names are not logged in Netbeans or CMD and there are huge amounts of logs since all agents log every single BT node execution, unlike MAJAN Monitoring feature.

4. **Evaluating grouping results of multiagent coordination** use-cases. Since template coordination BTs coordinate agents into groups, it is necessary to be

able to analyze the result in a visual and user-friendly format rather than RDF triples in RDF4J repositories. Moreover, MAJAN supports executing centrally running grouping algorithms and visualizing the results. Finally, comparing the grouping solutions of Multi-Agent Coordination and Centrally Running Algorithms is also supported in MAJAN.

5. Finally, MAJAN provides **tips**, complex **SPARQL queries** for MAC, and **more support** in the *MAJAN Extra* section.

MAJAN Installation

MAJAN consists of MAJAN Plugin and MAJAN Web extensions respectively for extending AJAN Service and AJAN Editor. The following two sections give the instructions to install MAJAN.

MAJAN Plugin

MAJAN Plugin provides BT nodes that are required for multiagent coordination. This plugin is integrated into AJAN Service via its Plugin System. To install AJAN Service with MAJAN Plugin, please pull MAJAN GitHub repository and find AJAN Service in the *AJAN_w_MAJAN* folder. To install AJAN Service, the required instructions are given in the AJAN Service GitHub repository.

MAJAN Web

MAJAN Web provides features to monitor and evaluate multiagent coordination processes and results. It is integrated directly to AJAN Editor and to install it (i.e. AJAN Editor with MAJAN Web), please pull MAJAN GitHub repository and find AJAN Editor in the *AJAN_Editor_w_MAJAN* folder.

Using MAJAN

MAJAN Ontology

This section provides information about the ontology that is used in MAJAN. It describes RDF IRIs that are used as **rdf:type**, their respective predicates, and values. Below, each IRI is described in a table and its properties are described in the table that comes right after it.

IRI	Description
mac:MACProblemInstance	*Every multiagent coordination problem in MAJAN should be described with this type. This type contains all the necessary information about a multiagent coordination problem, starting from its runtime to its solution. Each MACProblemInstance should have a unique ID such that they can be differentiated.*

Property (RDF)	Description	Data type	Example
mac:hasUseCase	*Name of a use case*	xsd:string	“Example multiagent coordination use case”
mac:hasParticipants	*ID/Name of participant agents*	xsd:string	“Agent1”, “Agent2”
mac:hasNotificationNecessary	*Whether it is necessary to return a notification*	xsd:string	“true” or “false”
mac:hasTimeout	*Expiration time of the coordination process*	xsd:dateTime	“2022-04-18T14:58:00”
mac:hasQuorum	*Amount of agents that is required to actively*	xsd:integer	4

	participate (i.e. receive, respond to messages) in coordination process		
mac:hasId	*A unique ID of coordination process*	xsd:string	“4894131654as16as798”
mac:hasNumberOfAgents	*Number of agents that participate in coordination process*	xsd:integer	5
mac:hasStartTime	*The start time of coordination process*	xsd:dateTime	“2022-04-12T14:58:00”
mac:hasMinPoints	*Minimum points parameter that is required for HDBSCAN algorithm*	xsd:integer	2
mac:hasMinClusterSize	*Minimum cluster size parameter that is required for HDBSCAN algorithm*	xsd:integer	2
mac:hasStatus	*Status of coordination process*	xsd:string	“running” or “completed” or “failed”
mac:hasSolution	*IRI of a grouping solution*	xsd:anyURI	mac:4afd565464as6d546CS2
mac:hasSolver	*Name of solver algorithm that is used to group agents*	xsd:string	“BOSS”
mac:hasFeasibleCoalition	*IRI of a valid coalition for solution of coordination process*	xsd:anyURI	_:coalition654a6ds4f1365

IRI	Description
mac:Conversation	*In each MACProblemInstance, there can be multiple conversations among participant agents, and these conversations are described with this type (i.e.* ***mac:Conversation****). Each conversation should have a unique ID such that it can be differentiated*

Property (RDF)	Description	Data type	Example
mac:hasMacProblemId	*ID of MAC Problem, this conversation belongs to*	xsd:string	“4894131654as16as798”
mac:hasInitiator	*ID/Name of initiator agent*	xsd:string	“Agent1”
mac:hasContent	*Describes the domain specific content to be sent to other agent(s).*	xsd:anyURI	_:agentProfileInfo465saa654dsf4612, _:solution65431d6f54
mac:hasReceiver	*ID/Name of agents who should receive the message*	xsd:string	“Agent1”, “Agent2”
mac:hasReceiverCapability	*Capability of agents to be used when sending a message to receiver agents*	xsd:string	“clusteringDistanceScores”
mac:hasAgreement	*Describes whether agent agrees to or refuses the received request*	xsd:string	“true” or “false”

IRI	Description
mac:RequestResponse	*Once a conversation starts, everything that agents exchange should be in the type of mac:RequestResponse. According to the FIPA Request protocol, agents can either agree, refuse or send a result for a request and thus, one of* ***mac:RequestResult****,* ***mac:RequestRefusal****, or* ***mac:RequestAgreement*** *should also be used together with RequestResponse*
mac:RequestAgreement	*This is a subtype of* **RequestResponse** *and is used to represent that the message is the agreement of the request*
mac:RequestRefusal	*This is a subtype of* **RequestResponse** *and is used to represent that the message is the refusal of the request*
mac:RequestResult	*This is a subtype of* **RequestResponse** *and is used to represent that the message is the result of the request*

Property (RDF)	Description	Data type	Example
mac:hasConversationId	*Unique ID of conversation, this request belongs to*	xsd:string	“conversation646a5s43564sa3as54”

IRI	Description
mac:AgentProfileInfo	*During multiagent coordination (into groups), agents might need to exchange or use profile information (e.g. gender, nationality, etc.) of users they represent, with other agents. For this purpose, agents use AgentProfileInfo type in MAJAN*

Property (RDF)	Description	Data type	Example
mac:belongsTo	*ID/Name of agent, this info belongs to*	xsd:string	“Agent1”
domain:hasGender	*Gender of agent*	xsd:string	“Male” or “Female” or “Other”
domain:hasNationality	*Nationality of agent*	xsd:string	“Greek”

IRI	Description
mac:AgentPreferences	*During multiagent coordination (into groups), agents might need to exchange or use preferences (e.g. gender preference, nationality preference, etc.) of users they represent, with other agents. For this purpose, agents use AgentPreferences type in MAJAN*

Property (RDF)	Description	Data type	Example
domain:hasGenderPreference	*Gender Preference of agent*	xsd:string	“Same” or “Mixed” or “Don’t mind”

domain:hasNationPreference	*Nationality Preference of agent*	xsd:string	“Same” or “Mixed” or “Don’t mind”
domain:hasGenderPrefWeight	*Importance weight of gender preference. 0 ≤ gender_weight ≤ 1, gender_weight + nation_weight = 1*	xsd:float	0.7
domain:hasNationPrefWeight	*Importance weight of nationality preference. 0 ≤ nation_weight ≤ 1, gender_weight + nation_weight = 1*	xsd:float	0.3

IRI	Description
mac:CSGP-CoalitionStructure	*This type is used to describe a Coalition Structure which is found as a solution for a CSGP by a CSGP solver. A Coalition Structure is basically a grouping, and it consists of coalitions*

Property (RDF)	Description	Data type	Example
mac:hasValue	*Value of Coalition Structure*	xsd:float	15
mac:hasRank	*Rank of Coalition Structure*	xsd:integer	2
mac:hasSolutionOf	*ID of MAC problem instance, this solution belongs to*	xsd:string	“asf6431asdf43afsd654”
mac:hasMembers	*IRIs of coalitions*	xsd:anyURI	_:coalitionas61654fd3

IRI	Description
mac:CSGP-Coalition	*This type is used to describe coalitions, which are basically groups, and consist of agents*

Property (RDF)	Description	Data type	Example
mac:hasValue	*Value of Coalition*	xsd:float	15
mac:hasMembers	*ID/Name of member agents*	xsd:string	“Agent1”, “Agent2”

IRI	Description
mac:UtilityValue	*Coalition Structures have values that are usually the sum of values of their Coalitions. And a coalition value is usually the sum of the Utility values of its member agents, and these utility values are described with the UtilityValue type*

Property (RDF)	Description	Data type	Example
mac:hasValue	*Utility value computed for specific coalition*	xsd:float	1.5
mac:isComputedBy	*ID/Name of agent who computes this Utility value*	xsd:string	“Agent1”
mac:isComputedAgainst	*ID/Name of an agent who this score is computed against*	xsd:string	“Agent2”
mac:isComputed	*ID of MAC Problem*	xsd:string	“asf6a43sd1f35”

For	*instance this score belongs to*		

IRI	Description
mac:Clustering	*This type is used to describe clustering which is found as a solution for a Clustering problem by a clustering problem solver. Clustering is basically a grouping, and it consists of clusters*

Property (RDF)	Description	Data type	Example
mac:hasSolutionOf	*ID of MAC problem instance, this solution belongs to*	xsd:string	“asf6431asdf43afsd654”
mac:hasMembers	*IRIs of clusters*	xsd:anyURI	_:clusteras61654fd3

IRI	Description
mac:Cluster	*This type is used to describe clusters which are basically groups, and they consist of agents*

Property (RDF)	Description	Data type	Example
mac:isClusterOf	*IRI of a clustering this cluster belongs to*	xsd:anyURI	_:Clustering46as133543

IRI	Description
mac:DistanceScore	*Each clustering algorithm requires distance scores between data points (i.e. agents in this case). This type is used to describe distance scores*

Property (RDF)	Description	Data type	Example
mac:hasValue	*Distance value computed by an agent*	xsd:float	1.5
mac:isComputedBy	*ID/Name of agent who computes this value*	xsd:string	“Agent1”
mac:isComputedAgainst	*ID/Name of an agent who this value is computed against*	xsd:string	“Agent2”
mac:isComputedFor	*ID of MAC Problem instance this value belongs to*	xsd:string	“asf6a43sd1f35”

IRI	Description
mac:ReciprocalScore	*Since agents compute distance scores from their own perspective, there are two (most likely different) distance scores for two agents. However, clustering algorithms expect only one distance score between two agents. Therefore, it is necessary to compute one score out of two and this score is described with the ReciprocalScore type. To compute it, the Harmonic Mean* (Prabhakar, 2017) *of two distance scores are found.*

Property (RDF)	Description	Data type	Example
mac:hasValue	*Reciprocal value*	xsd:float	1.5
mac:isComputedBy	*ID/Name of agent who computes this value*	xsd:string	“Agent1”
mac:isComputedAgainst	*ID/Name of an agent who this value is computed against*	xsd:string	“Agent2”
mac:isComputedFor	*ID of MAC Problem instance, this value belongs to*	xsd:string	“asf6a43sd1f35”

IRI	Description
mac:Log	*In order to be able to monitor the activities of coordinating agents, MAJAN provides a monitoring panel where agents send their logs as HTTP messages. This type is used to describe the log message content to be sent to the monitoring panel, such that there is a common message structure that can be understood by agents and MAJAN*

Property (RDF)	Description	Data type	Example
mac:hasAgentId	*ID/Name of agent who sends this log*	xsd:string	“Agent1”
mac:hasActivity	*Plain text that describes the activity of log*	xsd:string	“Started Coordination…”
mac:hasActivityStatus	*Status of activity*	xsd:string	“Success” or “Failure”

MAJAN Agent Communication

Message (*in figure 1*) and **Broadcast** (*in figure 2*) BT nodes are provided to achieve communication among agents in multiagent coordination. While the **Message** node supports sending a message to **only one receiver**, the **Broadcast** node supports sending a message to **multiple receivers**.

Figure 1. Message BT node in AJAN Editor

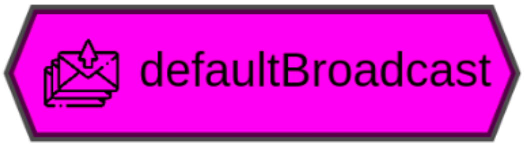


Figure 2. Broadcast BT node in AJAN Editor

In order to let agents understand each other when they communicate, messages should be in a certain structure. Moreover, the communication is designed based on [FIPA Request Protocol]() as shown in *figure 3*.

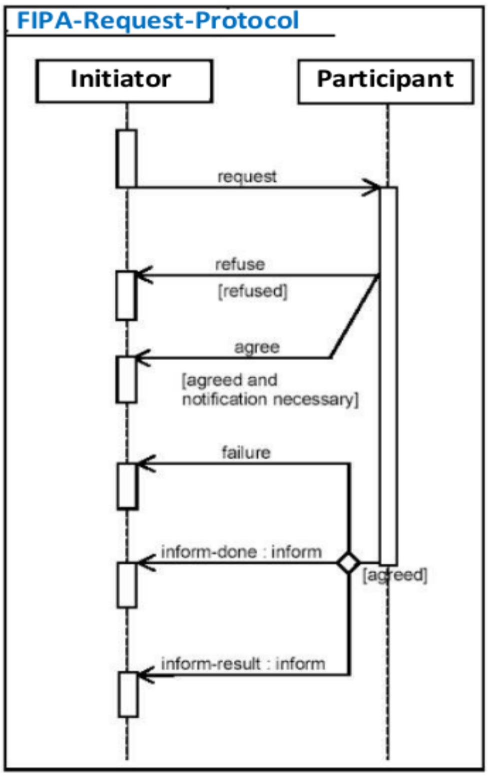


Figure 3. FIPA Request Protocol

FIPA Request protocol steps:

Step 1. Initiator agent sends a request to participant agents

Step 2. Participant agents send either an agreement or refusal back to the initiator agent.

Step 3. Initiator agent receives all agreements and refusals.

Step 4. Participant agent compute a result and send it back to initiator agent.

Step 5 (final). Initiator agent receives results.

The steps of Request protocol is implemented in MAJAN as described below:

1. **Sending a Request to agents**

MessageToParticipants (*figure 4*) is a **predefined** node by MAJAN. This node is used when an agent sends a request to other agent(s). In other words, whenever an agent wants to **start a conversation**, this message structure and template node are used. Moreover, this node expects a **subject IRI** in the type of **mac:Conversation** and this subject IRI should include predicates and respective values as listed below:

   a. Should be sent in message payload: **mac:hasInitiator, mac:hasId, mac:hasNotificationNecessary, mac:hasTimeout, mac:hasUseCase, mac:hasMacProblemId, mac:hasContent**

      i. SPARQL query to construct **message payload** is described in *figure 5*.

   b. Necessary to identify receivers of message**: mac**:**hasReceiver, mac**:**hasReceiverCapability**

      i. SPARQL query to select respective **receiver URIs** is described in *figure 6*.

   c. Refer to the **MAJAN Ontology** section for more detailed information about the definition of predicates.

Figure 4. MessageToParticipants predefined broadcast node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {
	?bnode rdf:type	mac:Conversation ;
			mac:hasInitiator	?thisAgentId ;
			mac:hasId	?conversationId ;
			mac:hasNotificationNecessary	?notifNecessary ;
			mac:hasTimeout	?timeout ;
			mac:hasUseCase	?useCaseTitle ;
			mac:hasMacProblemId	?macId ;
			mac:hasContent	?requestContent .

	?requestContent	?predicate	?object .
}
WHERE {
	?bnode rdf:type	mac:Conversation ;
			mac:hasId	?conversationId ;
			mac:hasUseCase	?useCaseTitle ;
			mac:hasNotificationNecessary	?notifNecessary ;
			mac:hasTimeout	?timeout ;
			mac:hasMacProblemId	?macId .

	OPTIONAL {
		?bnode	mac:hasContent	?requestContent .
		?requestContent	?predicate	?object .
	}

	?thisAgent	rdf:type	ajan:Agent, ajan:ThisAgent ;
			ajan:agentId ?thisAgentId .
}
```

Figure 5. SPARQL query to construct MessageToParticipants node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
```

```
SELECT DISTINCT ?requestURI
WHERE {
  ?bnode  rdf:type  mac:Conversation ;
          mac:hasParticipants ?participantId ;
          mac:hasReceiverCapability ?capability .

  ?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
              ajan:agentId  ?thisAgentId .

  FILTER(?participantId != ?thisAgentId)
  ?partAgentUri ajan:agentId  ?participantId ;
                ajan:hasAddress ?address .
  BIND(CONCAT(?address, "?capability=", ?capability) AS ?requestURI)
}
```
Figure 6. SPARQL query to select URIs of receiver agents in MessageToParticipants node

Once a conversation has been started with **MessageToParticipants** node, participant agents should respond to this request accordingly. For this purpose, agents use **mac:RequestResponse** and its subtypes.

2. **Sending an Agreement as the response of Request**

In order to **agree** to the request, MAJAN provides **AgreedMessageToInitiator** predefined message node (*figure 7*). This node sends a response back to the initiator agent telling that the **participant agrees** to the respective request. In order to build the payload of this message, SPARQL query in *figure 8* is used. To identify the request URI for **AgreedMessageToInitiator** message node, agents use the SPARQL query in *figure 9*.

Figure 7. AgreedMessageToInitiator predefined message node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
  ?newNode  rdf:type mac:RequestAgreement, mac:RequestResponse ;
            mac:hasUseCase  ?useCase ;
            mac:hasId ?conversationId ;
            mac:hasMacProblemId   ?macId ;
            mac:hasParticipants   ?thisAgentId .
}
WHERE {
  ?bnode  rdf:type  mac:Conversation ;
          mac:hasUseCase  ?useCase ;
          mac:hasMacProblemId   ?macId ;
          mac:hasId ?conversationId .

  ?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
              ajan:agentId  ?thisAgentId .
  BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
  BIND( IRI(CONCAT(STR(mac:RequestAgreement), STR(?uniqueId))) AS ?newNode )
}
```
Figure 8. SPARQL query to construct AgreedMessageToInitiator node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

SELECT DISTINCT ?requestURI
WHERE {
  ?bnode  rdf:type  mac:Conversation ;
          mac:hasInitiator  ?initiatorId ;
          mac:hasReceiverCapability ?initiatorCapability.

  ?initAgentUri ajan:agentId  ?initAgentId ;
                ajan:hasAddress ?address .
  BIND(CONCAT(?address, "?capability=", ?initiatorCapability) AS ?requestURI)
}
```
Figure 9. SPARQL query to select URI of receiver agent in AgreedMessageToInitiator and RefusedMessageToInitiator nodes

3. **Sending a Refusal as the response of Request**

In order to refuse the request, MAJAN provides **RefusedMessageToInitiator** predefined message node (*figure 10*). This node sends a response back to the initiator agent telling that the **participant refuses** the respective request. In order to build the payload of this message, SPARQL query in *figure 11* is used. To identify the request URI for **RefusedMessageToInitiator** message node, agents use the SPARQL query in *figure 9*.

Figure 10. RefusedMessageToInitiator predefined message node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
?newNode  rdf:type mac:RequestRefusal, mac:RequestResponse ;
          mac:hasConversationId ?conversationId ;
          mac:hasUseCase  ?useCase ;
          mac:hasMacProblemId   ?macId ;
          mac:hasParticipants   ?thisAgentId .
}
WHERE {
?bnode  rdf:type  mac:Conversation ;
        mac:hasUseCase  ?useCase ;
        mac:hasMacProblemId   ?macId ;
        mac:hasId ?conversationId .

?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
            ajan:agentId  ?thisAgentId .

BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
BIND( IRI(CONCAT(STR(mac:RequestRefusal), STR(?uniqueId))) AS ?newNode )
}
```

Figure 11. SPARQL query to construct RefusedMessageToInitiator node payload

4. **Sending a Result as the response of Request**

In order to send a **result** to the request, MAJAN provides **SendResultMessage** predefined broadcast node (*figure 12*). This node sends a **result back** to the initiator agent. In order to build the **payload** of this message, SPARQL query in *figure 13* is used. To identify the **request URI** for SendResultMessage broadcast node, agents use the SPARQL query in *figure 14*.

Since it is necessary to design this node in a generic, use-case-independent way, domain-specific results should be attached to the **mac:hasContent** predicate. This way, there will be no need to modify predefined communication nodes. As can be seen from *figure 13*, it is enough to attach the subject IRI and the query will retrieve **all related predicates and objects automatically** and send them as the result of the request.

Figure 12. SendResultMessage predefined broadcast node in AJAN Editor

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
?newNode  rdf:type  mac:RequestResult, mac:RequestResponse ;
          mac:hasId ?convId ;
          mac:hasParticipants   ?thisAgentId ;
          mac:hasMacProblemId   ?macId ;
          mac:hasUseCase  ?useCase ;
          mac:hasContent  ?resultContent .
?resultContent  ?predicate  ?object .
}
WHERE {
?bnode  rdf:type  mac:Conversation ;
        mac:hasMacProblemId   ?macId ;
        mac:hasUseCase  ?useCase ;
        mac:hasId ?convId ;
        mac:hasContent  ?resultContent .

?resultContent  ?predicate  ?object .

?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
            ajan:agentId  ?thisAgentId .
{
  BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
  BIND( IRI(CONCAT(STR(mac:RequestResult), STR(?uniqueId))) AS ?newNode )
```

```
}
}
```

Figure 13. SPARQL query to construct SendResultMessage node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

SELECT DISTINCT ?requestURI
WHERE {
?bnode  rdf:type  mac:Conversation ;
        mac:hasReceiver ?receiverId ;
        mac:hasReceiverCapability ?receiverCapability.

?receiverAgentIRI ajan:agentId  ?receiverId ;
                  ajan:hasAddress ?address .
BIND(CONCAT(?address, "?capability=", ?receiverCapability) AS ?requestURI)
}
```

Figure 14. SPARQL query to select URI of receiver agent in SendResultMessage node

MAJAN Coordination Nodes

In order to support **coordinating** AJAN agents **into groups**, MAJAN provides **five BT nodes** as listed below:

1. **Broadcast node**

   This node (*figure 15*) allows agents to **send the same message to multiple agents**, unlike the existing Message node.

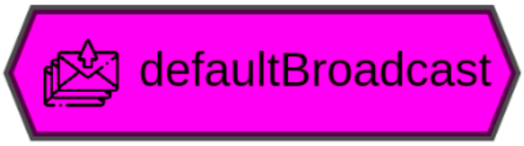


Figure 15. Broadcast BT node in AJAN Editor

2. **Coalition Generator node**

   This node (*in figure 16*) is used to **generate coalitions to solve CSGP** with a selected solver algorithm. In order to make this node generic, it expects input as described in the SPARQL query *in figure 17*.

Figure 16. Coalition Generator node in AJAN Editor.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

Construct {
?macInstance  rdf:type  mac:MACProblemInstance ;
              mac:hasId ?macId ;
              mac:hasParticipants ?participantId ;
              mac:hasNumberOfAgents ?numOfAgents ;
              mac:hasMinCoalitionSize "1" ;
              mac:hasMaxCoalitionSize ?numOfAgents ;
              mac:hasCannotLinkConnections  ?cannotLinkBnode ;
              mac:hasMustLinkConnections  ?mustLinkBnode .

?cannotLinkBnode  mac:hasCannotConnect  ?clAgentId, ?clAgentId2 .
?mustLinkBnode  mac:hasMustConnect  ?mlAgentId, ?mlAgentId2 .
}
Where{
```

```
?macInstance  rdf:type  mac:MACProblemInstance ;
              mac:hasId   ?macId ;
              mac:hasParticipants   ?participantId ;
              mac:hasNumberOfAgents   ?numOfAgents .
OPTIONAL {
  ?macInstance  mac:hasCannotLinkConnections  ?cannotLinkBnode .
  ?cannotLinkBnode  mac:hasCannotConnect  ?clAgentId, ?clAgentId2 .
}
OPTIONAL {
  ?bnode  mac:hasMustLinkConnections  ?mustLinkBnode .
  ?mustLinkBnode  mac:hasMustConnect  ?mlAgentId, ?mlAgentId2 .
}}
```

Figure 17. SPARQL query to construct input for Coalition Generator node

As the output, this node produces coalitions in the type of **mac:CSGP-Coalition** and attaches them to the given **mac:MACProblemInstance** subject IRI.

3. BOSS node

This node (*in figure 18*) is used to solve **CSGP** with the **BOSS** algorithm (Changder & Aknine, 2021), which finds an exact solution. This node expects generic input as described *in figure 19*. As the output, it attaches solutions (i.e. Coalition Structures) to the given MACProblemInstance’s **mac:hasSolution** predicate.

Figure 18. BOSS node in AJAN Editor.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

Construct {
  ?macInstance  rdf:type  mac:MACProblemInstance ;
                mac:hasNumberOfAgents ?numberOfAgents ;
                mac:hasParticipants ?participantId ;
                mac:hasId ?macId ;
                mac:hasFeasibleCoalitions ?feasibleCoalition .

# FeasibleCoalitions only include the coalitions which are feasible for this
use-case (because of constraints). Therefore, feasibleCoalitions don't include
all possible coalitions which is required by BOSS algorithm. Therefore, we pass
```

```
a value as "NonExistentCoalitionValue" which will be assigned to the missing
coalitions. By default, this value is very small since infeasibleCoalitions are
infeasible and we don't want them to be part of the solution.
  ?macInstance  mac:hasNonExistentCoalitionValue  -1000000 .

  ?feasibleCoalition  rdf:type  mac:CSGP-Coalition ;
                      mac:hasMembers  ?memberAgentId ;
                      mac:hasValue ?coalitionValue .
}
Where{
?macInstance  rdf:type  mac:MACProblemInstance, mac:CurrentMACProblemInstance ;
              mac:hasNumberOfAgents   ?numberOfAgents ;
              mac:hasParticipants   ?participantId ;
              mac:hasId   ?macId ;
              mac:hasFeasibleCoalitions   ?feasibleCoalition .

?feasibleCoalition  rdf:type  mac:CSGP-Coalition ;
                    mac:hasMembers  ?memberAgentId ;
                    mac:hasValue ?coalitionValue .
}
```

Figure 19. SPARQL query to construct input for BOSS node

4. HDBSCAN node

This node (*figure 20*) is used to solve **Clustering** Problem with the **HDBSCAN** algorithm (Campello & Moulavi, 2015). It expects generic input as described in *figure 21*. As the output, it attaches solutions (i.e. clustering results) to the given MACProblemInstance's **mac:hasSolution** predicate.

Figure 20. HDBSCAN node in AJAN Editor

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

CONSTRUCT {
?macInstance  rdf:type  mac:MACProblemInstance;
              mac:hasId   ?macId;
              mac:hasNumberOfAgents ?numberOfAgents;
              mac:hasParticipants   ?participantId1, ?participantId2 ;
```

```
              mac:hasPerfectMatchScore ?perfectMatchScore ;
              mac:hasCannotLinkConnections  ?cannotConnection ;
              mac:hasMustLinkConnections  ?mustConnection .

?cannotConnection   ?clPred   ?clObj .
?mustConnection   ?mlPred   ?mlObj .

# HDBSCAN Parameters: min Points and min Cluster Size
?macInstance  mac:hasMinPoints  ?boundMinPoints ;
              mac:hasMinClusterSize ?boundMinClSize .

?rrsIri   rdf:type  mac:DistanceScore ;
          mac:isComputedBy  ?participantId1 ;
          mac:isComputedAgainst   ?participantId2 ;
          mac:isComputedFor   ?macId ;
          mac:hasValue  ?reciprocalDistance .}
WHERE {
?macInstance  rdf:type  mac:MACProblemInstance, mac:CurrentMACProblemInstance ;
              mac:hasId   ?macId ;
              mac:hasNumberOfAgents   ?numberOfAgents ;
              mac:hasParticipants   ?participantId1, ?participantId2 .

OPTIONAL{
  ?macInstance  mac:hasCannotLinkConnections  ?cannotConnection .
  ?cannotConnection   ?clPred   ?clObj . }

OPTIONAL{
  ?macInstance  mac:hasMustLinkConnections  ?mustConnection .
  ?mustConnection   ?mlPred   ?mlObj . }

?macInstance  mac:hasReciprocalScore  ?rrsIri .
?rrsIri   rdf:type  mac:ReciprocalScore ;
          mac:isComputedBy  ?participantId1 ;
          mac:isComputedAgainst   ?participantId2 ;
          mac:isComputedFor   ?macId ;
          mac:hasValue  ?reciprocalDistance .

OPTIONAL {
  ?macInstance  mac:hasMinPoints    ?minPoints ;
                mac:hasMinClusterSize   ?minClSize .
  BIND(IF(BOUND(?minPoints), ?minPoints, 1) AS ?boundMinPoints)
  BIND(IF(BOUND(?minClSize), ?minClSize, 2) AS ?boundMinClSize) }

  BIND(0 AS ?perfectMatchScore)
}
```

Figure 21. SPARQL query to construct input for HDBSCAN node

5. **Insert node**

The goal of this node (*in figure 22*) is to **add given triples** (just like the Write node) to the specified repository. Unlike **Write** node, **Insert** node **adds** the given *Values* of **RDF** triples to the **existing Properties** (i.e. predicates) in the repository. Write node, on the other hand, **replaces** the given Values of the existing predicates. The example below explains the difference best.

Let's say the first triple exist in AKB, and we want to write the second triple to AKB.

```
Exists in AKB -> ajan:SubjectIRI ajan:hasPredicate "object" .
New Triple to be added -> ajan:SubjectIRI ajan:hasPredicate "object2" .
```

Below are the final triple(s) in AKB if we were to use Write or Insert nodes.

```
Result of Write node in AKB:
ajan:SubjectIRI ajan:hasPredicate "object2" .
Result of Insert node in AKB:
ajan:SubjectIRI ajan:hasPredicate "object", "object2" .
```

Figure 22. Insert node in AJAN Editor

MAJAN Coordination Behavior Trees

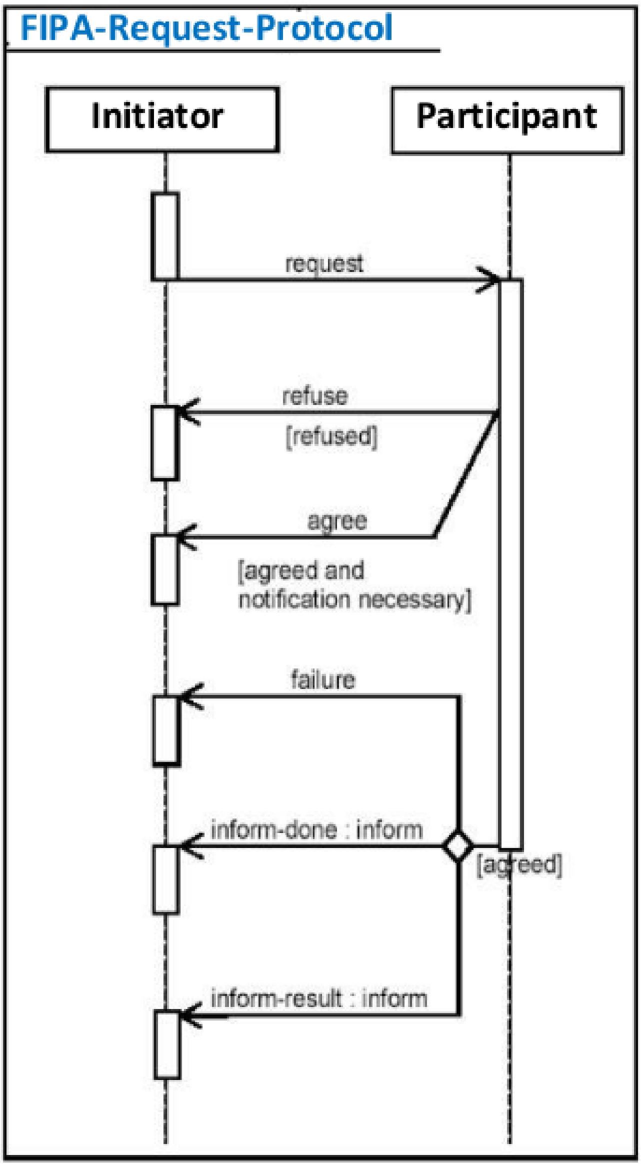
MAJAN provides **template coordination BTs** for users to be able to design **MAC use-cases** easily. There are three different sets of BTs for **three coordination protocols**. **The first** one is designed based on the FIPA Request protocol. In this protocol, agents coordinate with each other and exchange basic information. **The second** protocol is designed to solve **CSGP** to create groups of agents. This protocol is developed by extending the first protocol. **The last** protocol is designed to solve **Clustering** problems, and it is also developed based on the first one. All of these protocols are **generic, use-case independent**, such that users can customize them based on the use case at hand. Moreover, in these protocols, CSGP and Clustering are solved with *BOSS and HDBSCAN* algorithms, respectively. However, it is also possible to use other CSGP and Clustering solver algorithms by only replacing the one within the respective BTs. These protocols are explained in the following subsections.

ALL COORDINATION BTs ARE PROVIDED IN *SPARQL-BTs-for-MAC* FOLDER of MAJAN GITHUB REPOSITORY.

Request-Coordination-Protocol BTs

FIPA Request Protocol

Request Coordination protocol is designed completely based on [FIPA Request Protocol]() as described below.



Actions of agents in FIPA Request protocol are listed below:

Step 1. Initiator agent sends a request message to participant agents.

Step 2. Participant agents receive the request message and send back either a Refused or Agreed message.

Step 3. Initiator agent receives the message (either Refused or Agreed).

Step 4. Participant agents compute a result and send either a Result or Failure back to Initiator agent.

Step 5 (final). Initiator agent waits until the specified Quorum or Timeout is reached and then continues the execution of its individual behaviors.

In Request-Coordination-Protocol, one agent sends a request message to request *profile information* from other agents and waits until either **quorum** or **timeout** is **reached.** Just like in FIPA Request protocol, participant agents return an **Agreement** or **Refusal**, and then they return a **Result** or **Failure** in **Request-Coordination-Protocol** of MAJAN. In the provided BTs, agents return simple profile information for the sake of achieving successful coordination.

Explanation of BTs

There are **three BTs** that are designed to implement this protocol, and they are listed below:

1. Name: **SendCoordRequestTempBT**, Label: "*Send Coord. Request Temp. BT*" *in figure 23*

In this BT, once the agent handles the respective event, it updates the **mac:MACProblemInstance** to make sure that it has a **unique** subject IRI and ID such that it is **not being mixed** with any other MACProblemInstance that already exists in AKB. Then agent **logs** this activity to MAJAN Monitoring panel (*in figure 24*).

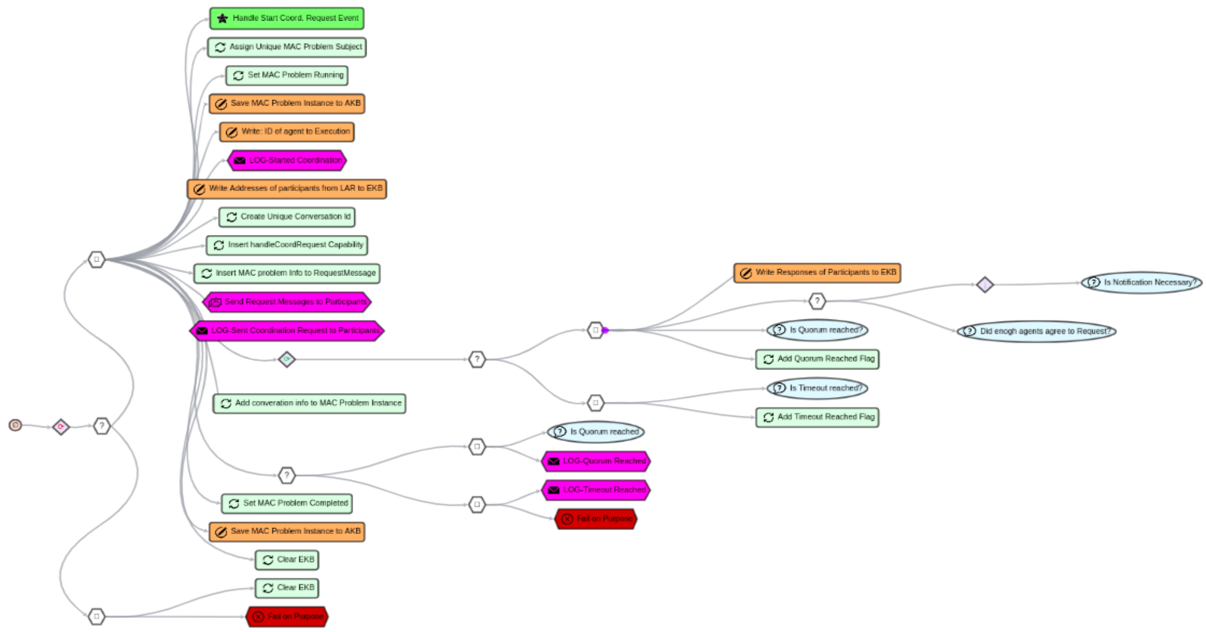


Figure 23. Send Coord. Request Temp. BT in AJAN Editor.

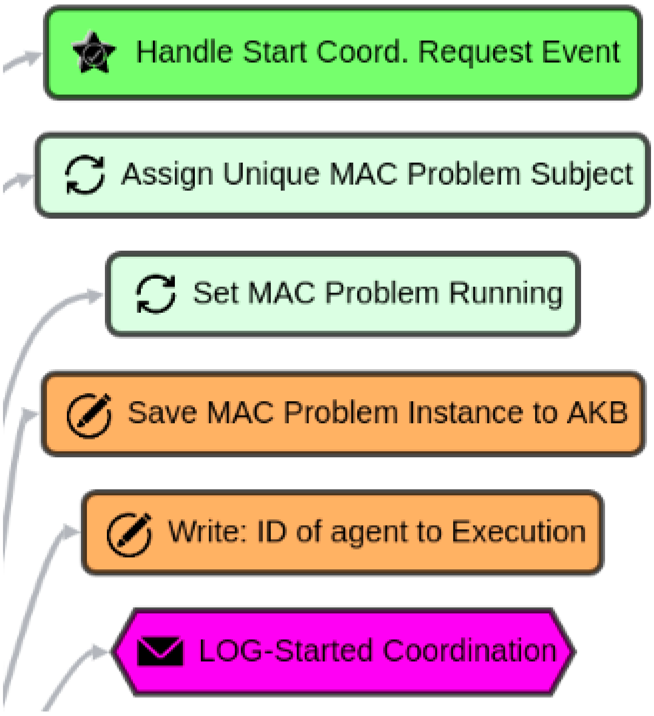


Figure 24. PART 1 of Send Coord. Request Temp. BT

In the **second part** of this BT, the agent creates a **unique conversation** and sends the request messages to participant agents (*figure 25*).

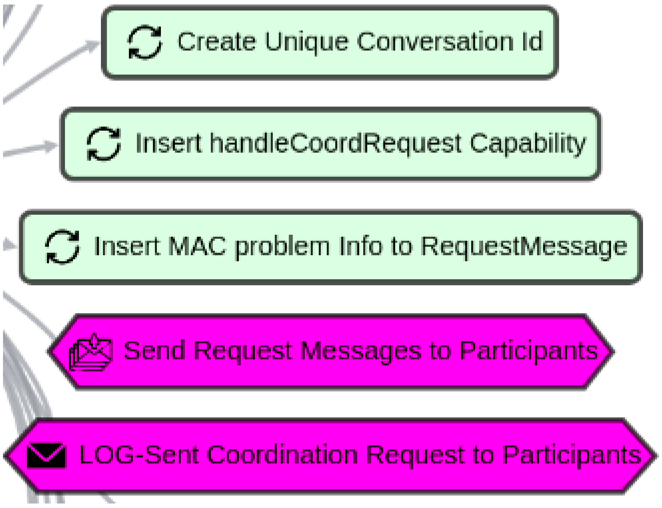


Figure 25. PART 2 of Send Coord. Request Temp. BT

In the **third part** of this BT, the agent waits until **quorum or timeout reached** (*figure 26*).

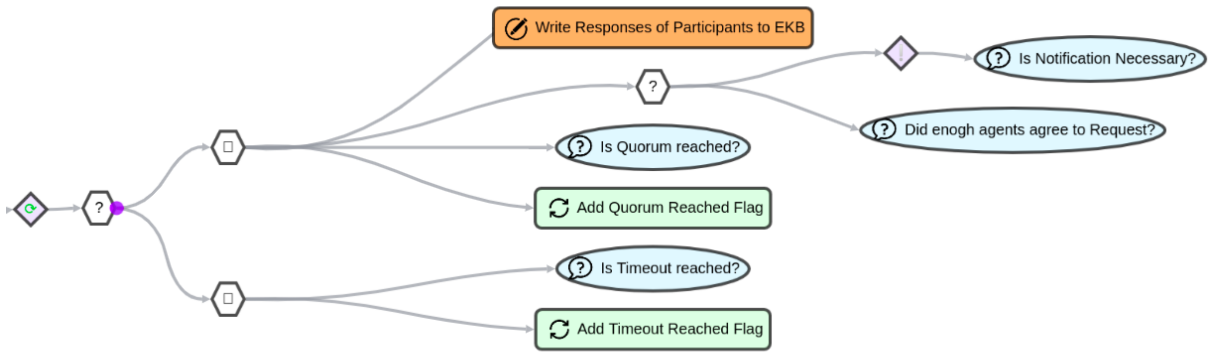


Figure 26. PART 3 of Send Coord. Request Temp. BT

Finally, (in 4th part), the agent logs whether **quorum or timeout reached** the Monitoring panel and **saves** the **MACProblemInstance** to AKB (*figure 27*).

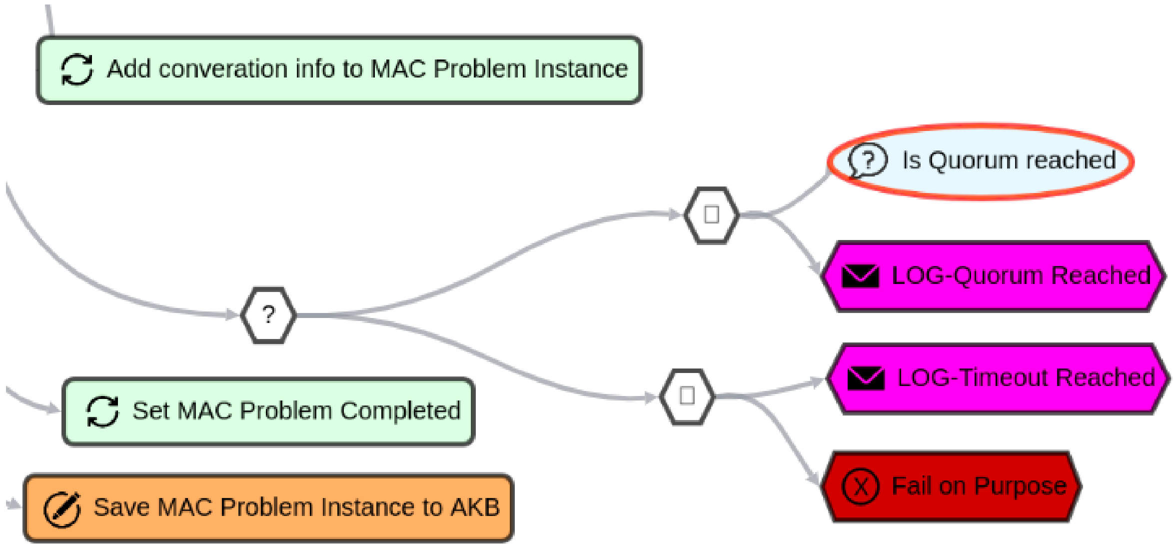


Figure 27. PART 4 of Send Coord. Request Temp. BT

2. Name: **HandleCoordRequestTempBT**, Label: *“Handle Coord. Request Templ. BT”*

In this BT, once the agent **handles** the respective **event**, it **logs** this activity to the MAJAN Monitoring panel. Then it **waits** for three seconds, which is a **placeholder** where users can add anything they want. Once waiting is over, it needs to **send** an **agreement or refusal** as a response. After replying appropriately, it needs to **send a result** as a response if the timeout is not reached yet to finalize handling this request. To compute the necessary results, users can **replace** the Wait node as they like, since it is a placeholder. (*figure 28*).

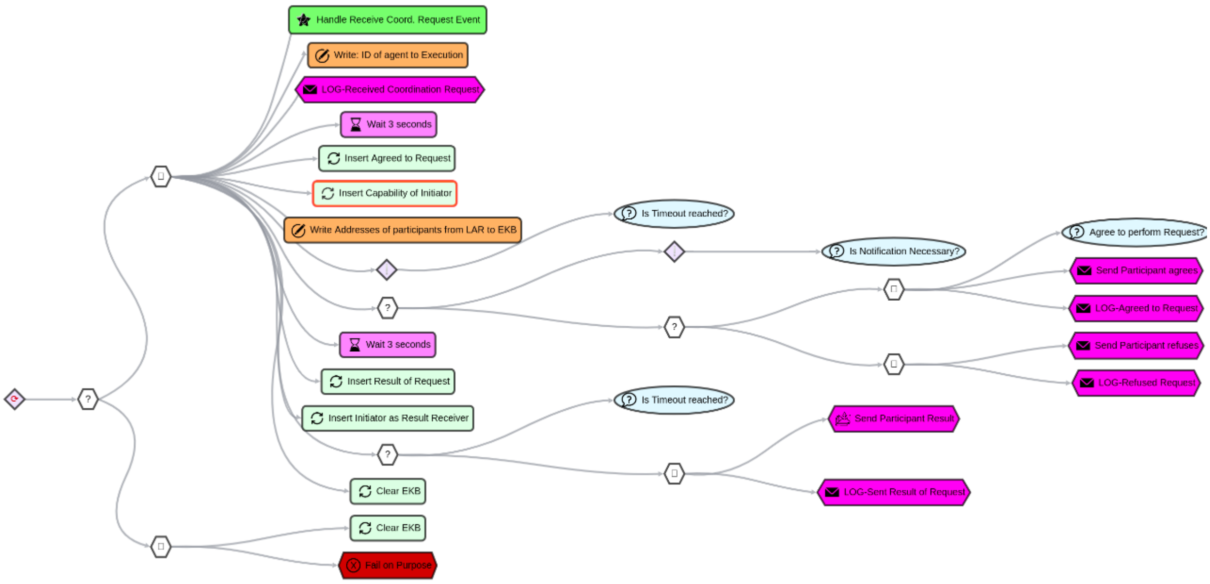


Figure 28. Handle Coord. Request Templ. BT

3. Name: **ReceiveCoordRequestResponseTempBT**, Label: "*Receive Coord. Request Response Temp. BT*"

This BT (*in figure 29*) is used to **receive a mac:RequestResponse** and **save** it to **AKB** such that the responses of participant agents can be **used** than in other **coordination BTs**.

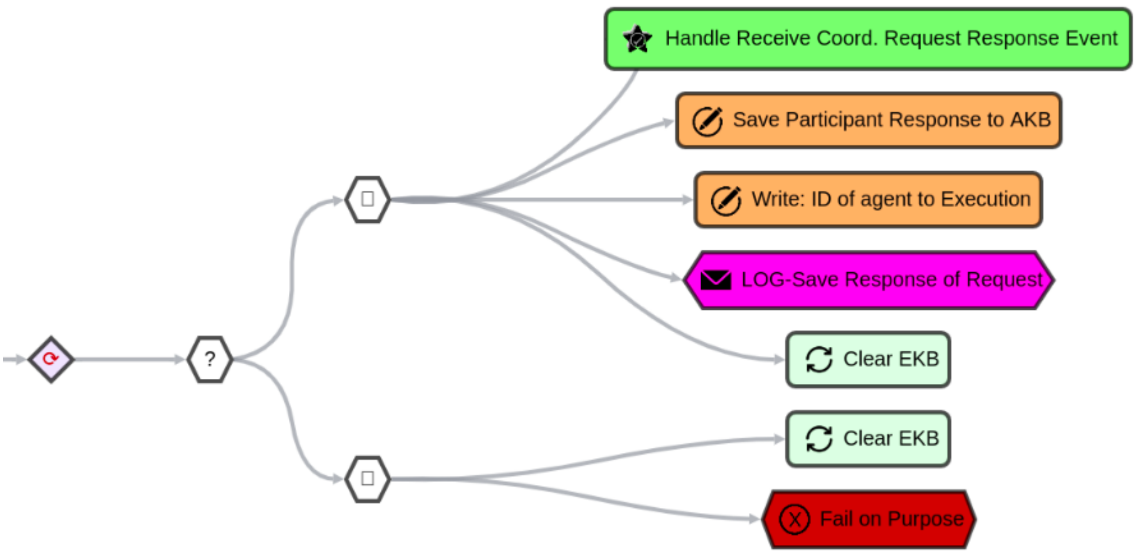


Figure 29. Receive Coord. Request Response Temp. BT

CSGP-Coordination-Protocol BTs

Coalition Structure Generation Problem

The **Coalition Structure Generation Problem (CSGP)** *(Rahwan & Michalak, 2015)* for multiagent systems focuses on **partitioning** a given set of **agents into** mutually disjoint **coalitions** (i.e. groups) so that the sum of the coalition values (also called *social welfare*) in the resulting **coalition structure** (i.e. grouping) is maximized. For this purpose, there are several centralized exact and heuristic CSGP solving algorithms available. However, approaches to distributed CSGP solving don't exist in the literature. A centralized CSGP solver algorithm works as following:

Step 1. Passing coalitions and their values as input to the algorithm

Step 2. The algorithm computes Coalition Structure values by summing coalition values.

Step 3 (final). The algorithm compares Coalition Structures by their values and computes a ranked list of them from highest to lowest value.

Coordination Protocol of Coalition Structure Generation Problem

The **goal** in CSGP-Coordination-Protocol is to form groups of agents by solving CSGP. Since each agent represents a user (i.e. person), agents have profile and preferences information. The BTs for this protocol are designed by extending **Request Protocol BTs**. The steps of this protocol are described below:

Step 1. One of the participant agents receive a **signal** to **start CSGP-Coordination-Protocol** and this agent becomes **Dedicated agent** since a CSGP solver algorithm must be executed centrally by one of the agents. Refer to Coalition Structure Generation Problem section for more info.

Step 2. The Dedicated agent **requests profile information** from all participant agents. And it waits until **a certain timeout or quorum** is reached.

Step 3. Participant agents **return a response back to Dedicated agent**. This response can be an **Agreement** (if the participant agrees to the request)**, Refusal** (if the participant refuses the request) **or Result** (if the participant sends the result of request) of the request.

Step 4. The Dedicated agent receives the **responses,** and **generates all possible coalitions** if the quorum is reached. Then it requests participants to compute and **return** their **Utility values** for the generated coalitions.

Step 5. Each agent determines its **Utility value of being member of a coalition**, for each coalition in the coalition structure, based on the preferences of its user. The Utility value of a coalition for an agent is computed as the **degree to which the preferences of the user are satisfied** in the considered coalition (i.e. agent group). All Utility values of an agent are **returned** to the **Dedicated agent**.

Step 6. The Dedicated agent **collects Utility values**, computes coalition values and **starts CSGP solver** algorithm (i.e. **BOSS** in this protocol) by passing coalitions and their values.

Step 7 (final). CSGP solver algorithm (i.e. BOSS) **produces a rank list of solutions** and the Dedicated agent **broadcasts the solutions** to participant agents.

Explanation of BTs

There are **seven** BTs to run **CSGP-Coordination-protocol** as listed below. These seven BTs implement the steps of CSGP-Coordiantion-Protocol which are described above. Please, open the **Behaviors** section of AJAN Editor and write the names of BTs in search section to find and analyze them in detail.

1. Name: **SendCsgpCoordRequestTempBt**, Label: "*Send CSGP Coord. Request Temp. BT*"

This BT is a customized version of "*Send Coord. Request Temp. BT*". The difference in this BT is that the agent requests **Agent Profile Information** from other agents, since this info is required to **solve CSGP**. Once required info is **collected** or **timeout** reached, the agent produces "Compute Csgp Coalitions Event" (*figure 32*) which triggers "*Compute CSGP Coalitions Templ. BT*". In order to request Agent Profile Info, the agent inserts "agentProfileInfoRequest" (*figure 30, 31*) as the capability of participant agents, and this capability triggers "*Handle Agent Profile Info Request Temp. BT*".

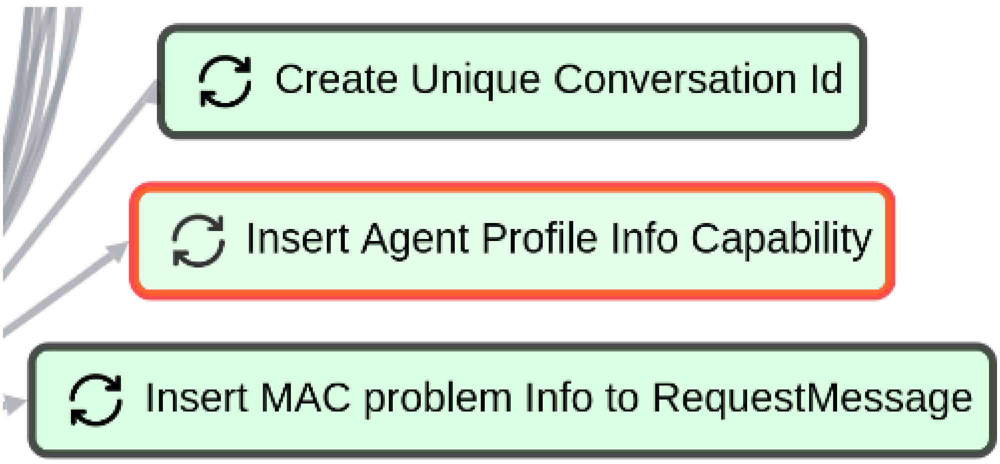


Figure 30. Insert Agent Profile Info to Conversation node

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

DELETE {
  ?subject  mac:hasReceiverCapability   ?existingCapability .
}
INSERT{
  ?subject  mac:hasReceiverCapability 'agentProfileInfoRequest' .
}
WHERE{
?subject  rdf:type  mac:Conversation .

OPTIONAL {
  ?subject mac:hasReceiverCapability  ?existingCapability .
}
}
```

Figure 31. Insert "agentProfileInfoRequest" capability of participants to request Agent Profile Info

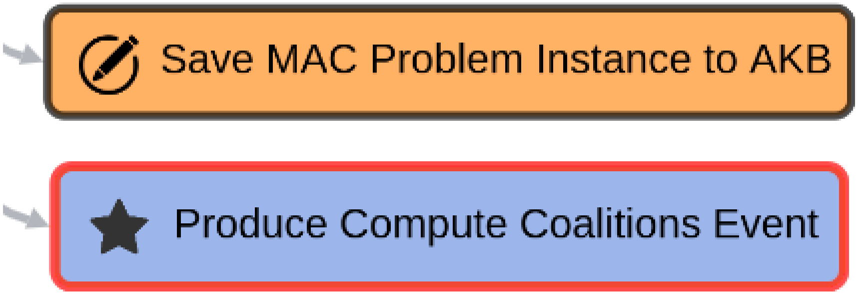


Figure 32. Produce Compute Coalitions Event

2. Name: **HandleAgentProfileInfoRequestTempBT**, Label: "*Handle Agent Profile Info Request Temp. BT*"

This BT is a customized version of "*Handle Coord. Request Templ. BT*". In this BT, the agent attaches its **Profile Information** to the Conversation (*figure 33-a, 33-b*) and the BT automatically **sends** this info to the **Dedicated agent**.

Figure 33-a. BT node to add profile info to current conversation in AJAN Editor

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX domain:
<http://localhost:8090/rdf4j/repositories/domain_specific_ontology#>

CONSTRUCT {
?bnode  rdf:type  mac:Conversation ;
        mac:hasContent ?personalInfoNode .

?personalInfoNode   rdf:type  mac:AgentProfileInfo ;
                    mac:belongsTo   ?thisAgentId ;
                    domain:hasGender  ?gender ;
                    domain:hasNationality   ?nation ;
                    domain:hasLanguage  ?lang .
}
WHERE{
?bnode  rdf:type  mac:Conversation .

 ?agProf  rdf:type  domain:DomainUser ;
          domain:hasGender  ?gender ;
          domain:hasNationality   ?nation ;
          domain:hasLanguage  ?lang .

?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
            ajan:agentId  ?thisAgentId .
{
  BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
  BIND( IRI(CONCAT(STR(mac:AgentProfileInfo), STR(?uniqueId))) AS
?personalInfoNode )
} }
```

Figure 33-b. SPARQL query to add profile information of agent to current conversation

3. Name: **ComputeCsgpCoalitionsTemplBT**, Label: "*Compute CSGP Coalitions Templ. BT*"

This BT is designed to **compute coalitions** given **constraints**, **broadcast them** to other agents to collect their **utility values**, and **compute coalition values** before producing "Solve CSGP Coordination Event" which triggers "*Solve CSGP Template BT*". Firstly, the agent computes the **cannot link connections** which are later passed to the **Coalition Generator** node as input (*figure 34*). Once coalitions are computed, the agent **broadcasts** computed coalitions and profile information of all agents to participants such that they can compute their **utility values** for computed coalitions (*figure 35*). After receiving utility values, the agent **computes coalition values** and produces "Solve CSGP Coordination Event" (*figure 36*).

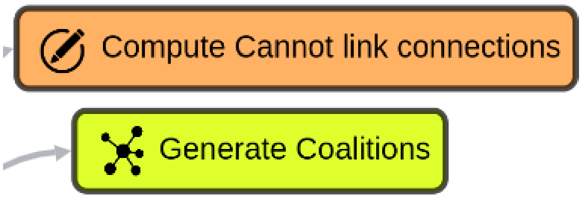


Figure 34. Compute Cannot link connections for given use case and Generate coalitions given cannot link connections

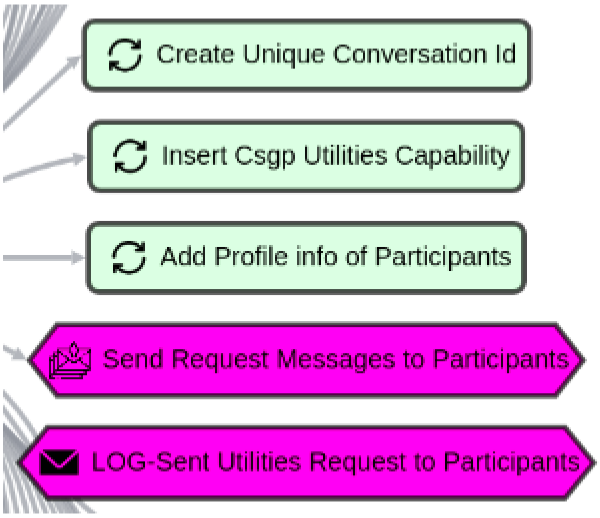


Figure 35. Broadcast coalitions and agent profile info to collect utility values

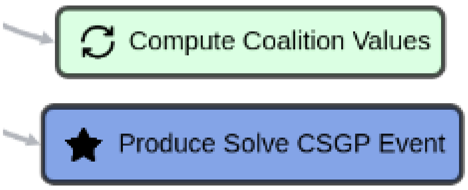


Figure 36. Compute coalition values and trigger solving CSGP BT

4. Name: **HandleCsgpUtilitiesRequestTemplBT**, Label: "*Handle Csgp Utilities Request Templ. BT*"

This BT is designed to **compute utility values** upon request and **return them** back to the dedicated agent (*figure 37-a*). SPARQL query *in figure 37-b* is used to compute utility values **to solve CSGP**.

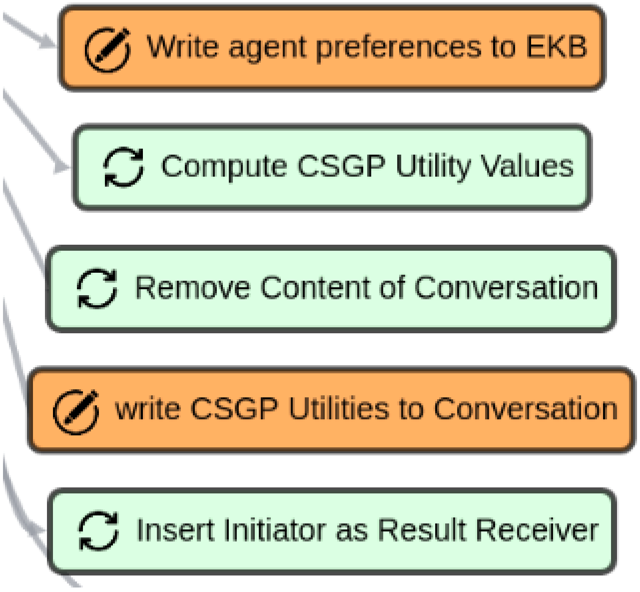


Figure 37-a. Compute Utility values after fetching preferences of this agent and finally attach the utility values to the conversation content to be sent back to the dedicated agent.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#
PREFIX domain:
<http://localhost:8090/rdf4j/repositories/domain_specific_ontology#>

INSERT {
?feasibleCoalitionNode  mac:hasUtilityValue ?uVbnode .
?uVbnode     rdf:type    mac:UtilityValue ;
            mac:isComputedBy   ?thisAgentId ;
            mac:hasValue        ?ttlV .
}
WHERE {
  {
    SELECT DISTINCT ?feasibleCoalitionNode ?thisAgentId (SUM(?uValue) AS ?ttlV)
    WHERE {
        # get This agent
        ?agent  rdf:type  ajan:Agent, ajan:ThisAgent ;
```

```
            ajan:agentId   ?thisAgentId .

        # get Feasible Coalitions which are sent by Dedicated agent
        ?macInstance  rdf:type   mac:MACProblemInstance ;
                      mac:hasId ?macId ;
                      mac:hasFeasibleCoalitions  ?feasibleCoalitionNode .

        # get the coalition information
        ?feasibleCoalitionNode  mac:hasCommonGender ?commonGender ;
                                mac:hasCommonNation ?commonNation .

        # make sure that coalition contains This agent
        FILTER EXISTS { ?feasibleCoalitionNode  mac:hasMembers  ?thisAgentId . }

        # get preferences of This agent
        ?prefsSbj rdf:type  mac:AgentPreferences ;
                  domain:hasGenderPreference  ?genderPref ;
                  domain:hasNationPreference  ?nationPref ;
                  domain:hasGenderPrefWeight  ?genPrefWeight ;
                  domain:hasNationPrefWeight  ?natPrefWeight .

        # get gender and nation of This agent
        ?resultSbj  rdf:type  mac:Conversation ;
                    mac:hasMacProblemId   ?macId ;
                    mac:hasContent  ?resultContent .

        ?resultContent  rdf:type  mac:AgentProfileInfo ;
                        mac:belongsTo   ?thisAgentId ;
                        domain:hasGender  ?gender ;
                        domain:hasNationality   ?nation .

        BIND(IF(LCASE(?genderPref) = "dont mind" || (LCASE(?genderPref) = "same"
&& ?gender = ?commonGender) || (LCASE(?genderPref) = "mixed" &&
LCASE(?commonGender) = "mixed"), xsd:float(?genPrefWeight),
-xsd:float(?genPrefWeight)) AS ?genderUValue)
              BIND(IF(LCASE(?nationPref) = "dont mind" || (LCASE(?nationPref) =
"same" && ?nation = ?commonNation) || (LCASE(?nationPref) = "mixed" &&
LCASE(?commonNation) = "mixed"), xsd:float(?natPrefWeight),
-xsd:float(?natPrefWeight)) AS ?nationUValue)

        BIND((?genderUValue + ?nationUValue) AS ?uValue)

        } GROUP BY ?feasibleCoalitionNode ?thisAgentId
    }
    BIND(BNODE() AS ?uVbnode)
}
```

Figure 37-b. SPARQL Query to compute Utility values in CSGP Protocol

5. Name: **SolveCsgpTemplBT**, Label: "*Solve CSGP Template BT*"

This BT's **goal** is to **solve CSGP** by using the **BOSS** node (*figure 38*). Users can replace the **BOSS** node with any other **CSGP solver** node without any problem since the BT is generic.

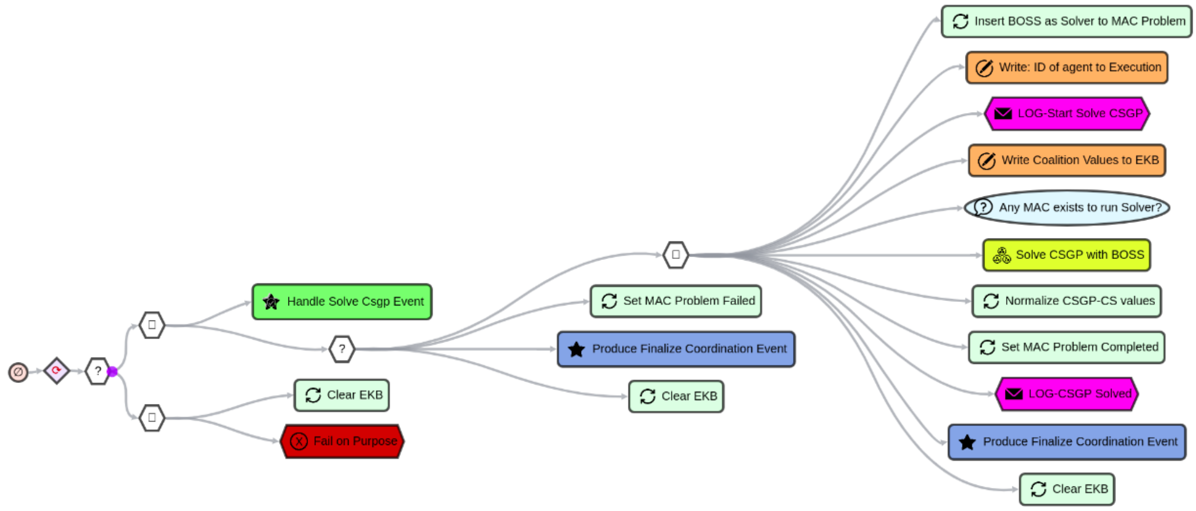


Figure 38. BT to solve CSGP with BOSS algorithm and start finalizing the coordination process.

Additionally, users can normalize the values of Coalition Structures (with **Normalize CSGP-CS values** node, *figure 39*) in case they are **negative** but this is **optional**. Finally, the agent produces "Finalize Coordination Event" which triggers "*Finalize Coordination to Groups BT*".

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

DELETE{
?macSolution  mac:hasValue  ?csValue .
}
INSERT {
?macSolution  mac:hasValue  ?normalizedCsValue .
}
WHERE{
?macInstance  rdf:type  mac:MACProblemInstance .

OPTIONAL{
?macInstance  mac:hasSolution   ?macSolution ;
              mac:hasMinCsValue   ?minCsValue .
FILTER(?minCsValue <= 0)
```

```
?macSolution  rdf:type  mac:CSGP-CoalitionStructure ;
              mac:hasValue  ?csValue .

BIND((?csValue + 5 - ?minCsValue) AS ?normalizedCsValue)
}
}
```

Figure 39. SPARQL query to normalize Coalition Structure values

6. Name: **FinalizeCoordinationToGroupsBT**, Label: "*Finalize Coordination to Groups BT*"

This BT, as described in *figure 40*, is designed to **finalize any coordination process**. It is enough to produce the required event successfully. In this BT, the agent computes the **runtime** of the coordination process and **broadcasts** the result of the coordination process to participant agents. By using "*Just Success Node*", the agent ensures that broadcasting the result to all participants is **optional** since some agents **might be inactive**, but this shouldn't cause the coordination **process to fail**. Nevertheless, users can remove the **Success node** and make sure that it is necessary for all participant agents to receive coordination results.

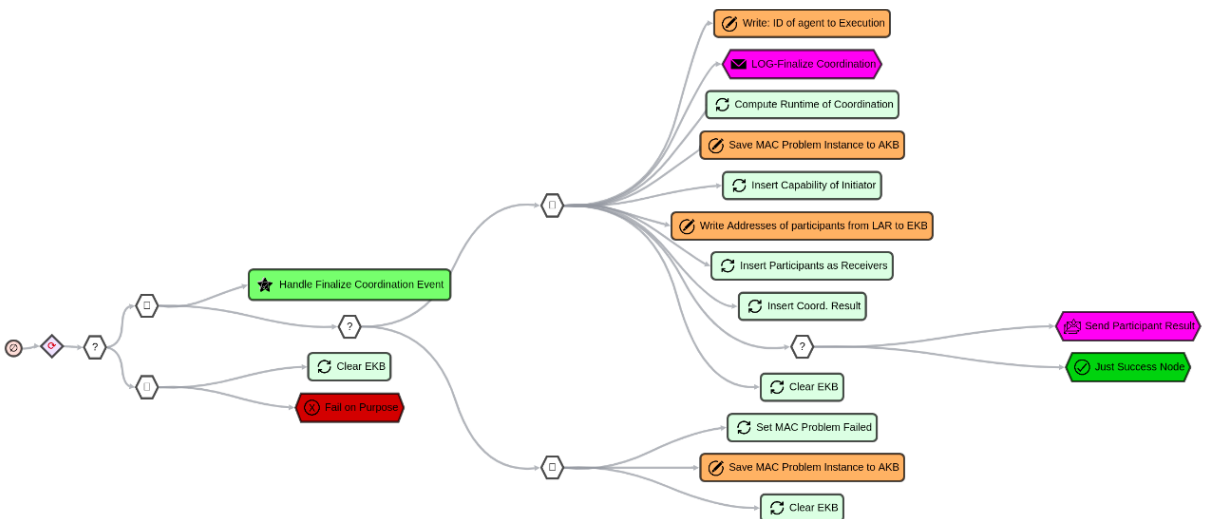


Figure 40. ***FinalizeCoordinationToGroupsBT*** to finalize coordination processes

7. Name: **ReceiveCoordRequestResponseTempBT**, Label: "*Receive Coord. Request Response Temp. BT*"

In order to **save the responses** in coordination protocols, agents use the **same** BT in **CSGP protocol as** they use in the **Request protocol**. Check **Request Protocol** section for more info about this BT.

Clustering-Coordination-Protocol BTs

Clustering Problem

The objective of **Clustering problem (Hartigan, 1975)** is to partition objects in a way that the objects in the same groups are more similar to each other than objects in other groups. A clustering is a grouping that consists of clusters and each cluster consists of objects (in this case, agents). The process of solving Clustering problem is described below:

Step 1. **Distance scores** between agents need to be computed.

Step 2. Distance scores are shared and collected by one agent.

Step 3. The agent, who collects distance scores, **computes reciprocal scores**. That is because, each agent computes a distance score based on its own preferences and consequently, there becomes 2 distance score between two agents. But clustering solver algorithms require only one distance score between two agents.

Step 4. The agent, who computes reciprocal scores, **starts clustering solver** algorithm by passing reciprocal scores as input.

Step 5 (final). Clustering solver algorithm **produces a clustering solution**, and it is broadcasted to participant agents.

Coordination Protocol of Clustering Problem

The **goal** in Clustering-Coordination-Protocol is to form groups of agents by solving Clustering problem. The BTs for this protocol are designed by extending **Request Protocol BTs**. The steps of this protocol are described below:

Step 1. One of the participant agents receive "start coordination" signal and this agent becomes **Dedicated agent**.

Step 2. The Dedicated agent sends a request message to **request profile information** from participant agents. Then it waits until **Timeout or Quorum reached**.

Step 3. Participant agents **send** back either an **Agreed or Refused** message.

Step 4. The Dedicated agent receives responses and acts accordingly (i.e. it aborts if there is not enough Agreed message and waits for results, otherwise)

Step 5. Participant agents compute the requested **result (i.e. agent profile information)** and **send** it to the **Dedicated** agent.

Step 6. The Dedicated agent receives the **results** and if quorum reached, it requests participants to compute and **return** their **Distance scores**.

Step 7. The Dedicated agent **collects Distance scores**, computes reciprocal scores and **starts Clustering solver** algorithm (in this case, **HDBSCAN**) by passing reciprocal scores as input.

Step 8 (final). The Clustering solver algorithm (i.e. **HDBSCAN**) **produces a solution** and the Dedicated agent broadcasts the solution to participant agents.

Explanation of BTs

There are **seven BTs** to run **Clustering-Coordination-protocol** as listed below. These seven BTs implement the steps of Clustering-Coordiantion-Protocol which are described above. Please, open the Behaviors section of AJAN Editor and write the names of BTs in search section to analyze them.

1. Name: **SendClusteringCoordRequestTempBt**, Label: "*Send Clustering Coord. Request Templ BT*"

This BT (*figure 41*) is designed to **start the coordination process** to solve a **clustering problem** to **form groups** of agents. It is based on "*Send Coord. Request Temp. BT*" just like "*Send CSGP Coord. Request Temp. BT*". Since **profile info** of agents is required for clustering problems as well, agent requests them just like in "*Send CSGP Coord. Request Temp. BT*". The only difference between "*Send Clustering Coord. Request Templ BT*" and "*Send CSGP Coord. Request Templ BT*" BTs is the event that is produced at the end of BT. In clustering, the agent produces "Collect Clustering Distances Event" which triggers "*Collect Clustering Distances Templ. BT*".

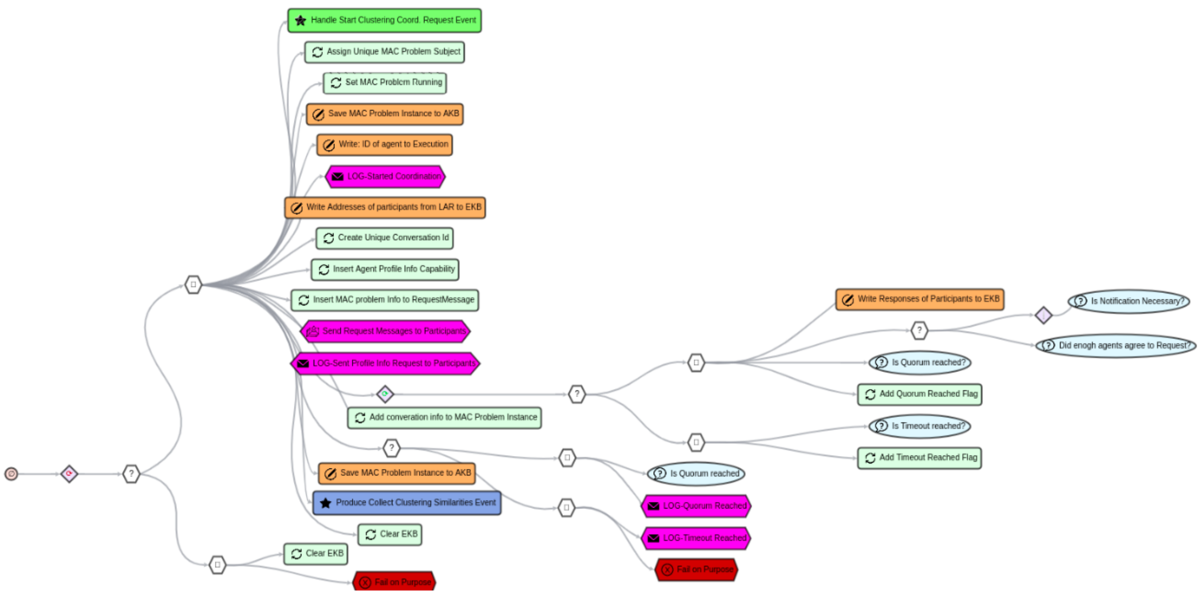


Figure 41. Send Clustering Coord. Request Templ BT in AJAN Editor

2. Name: **HandleAgentProfileInfoRequestTempBT**, Label: "*Handle Agent Profile Info Request Temp. BT*"

In order to **collect profile information** from participants, agents in clustering protocol use the **same BT** that they use in CSGP protocol. Refer to CSGP Protocol BTs section for more info about this BT.

3. Name: **CollectClusteringDistancesTemplBt**, Label: "*Collect Clustering Distances Templ. BT*"

In this BT (*figure 42, 43*), the dedicated agent **broadcasts** the profile info of all agents to participants such that they can compute **distance scores** and **return them** to the dedicated agent. Upon receiving this request, the **participant** agent executes "*Handle Clustering Distances Request Templ. BT*". After collecting distance scores, the dedicated agent produces “Solve Clustering Event” which triggers "*Solve Clustering Templ. BT*".

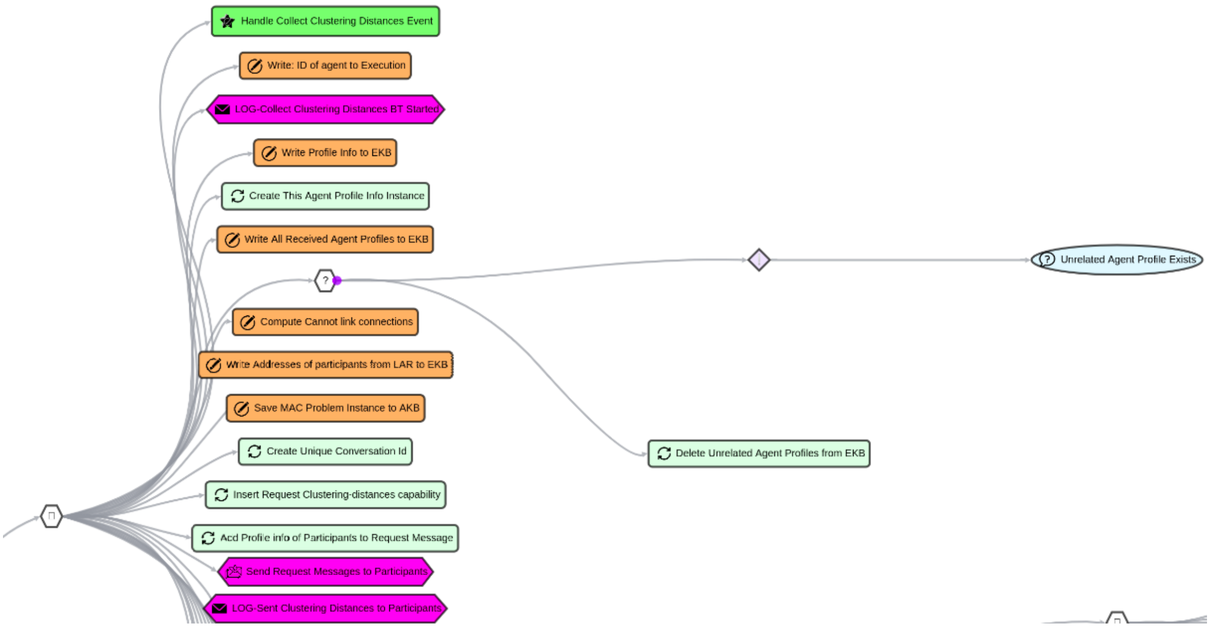


Figure 42. PART 1 of Collect Clustering Distances Templ. BT

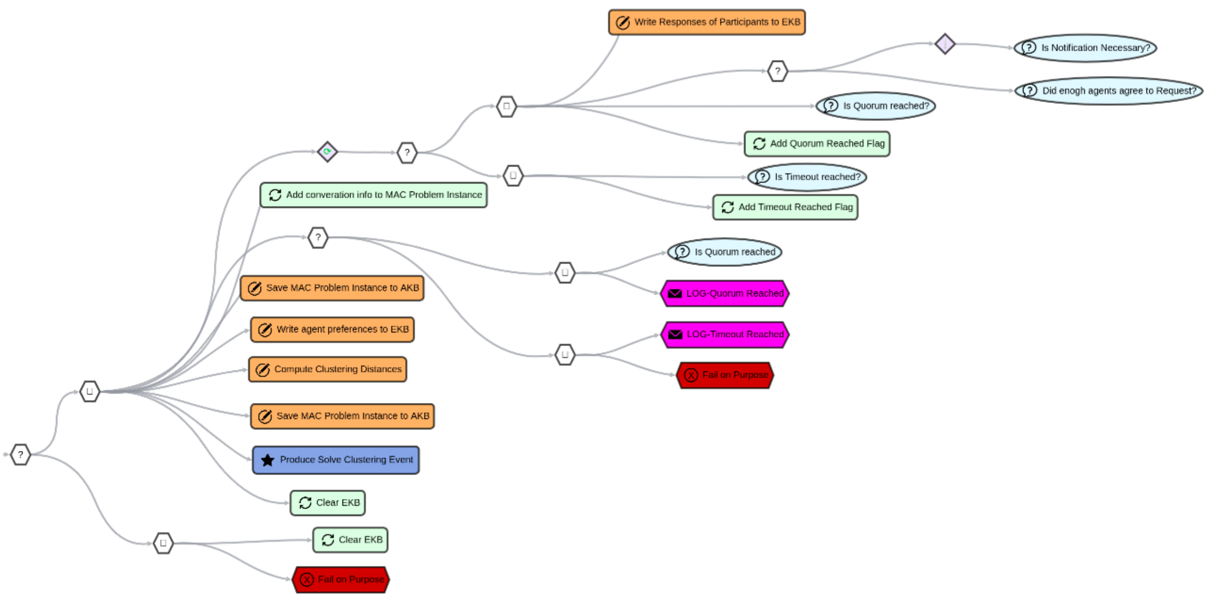


Figure 43. PART 2 of Collect Clustering Distances Templ. BT

4. Name: **HandleClusteringDistancesRequestTemplBT**, Label: "*Handle Clustering Distances Request Templ. BT*"

This BT (*figure 44-a*) is executed when participant agents are requested to compute their **distance scores** in a clustering problem. Agents firstly **reply** with an **Agreement or Refusal**. Afterward, agents **fetch** their **preferences** from AKB and compute **distance scores** by comparing the profile information of every single participant agent with their individual preferences. Once they compute distance scores, they send scores back to the dedicated agent.

Figure 44-b describes the **SPARQL query** of "**Compute Clustering Distances**" node to compute distance scores.

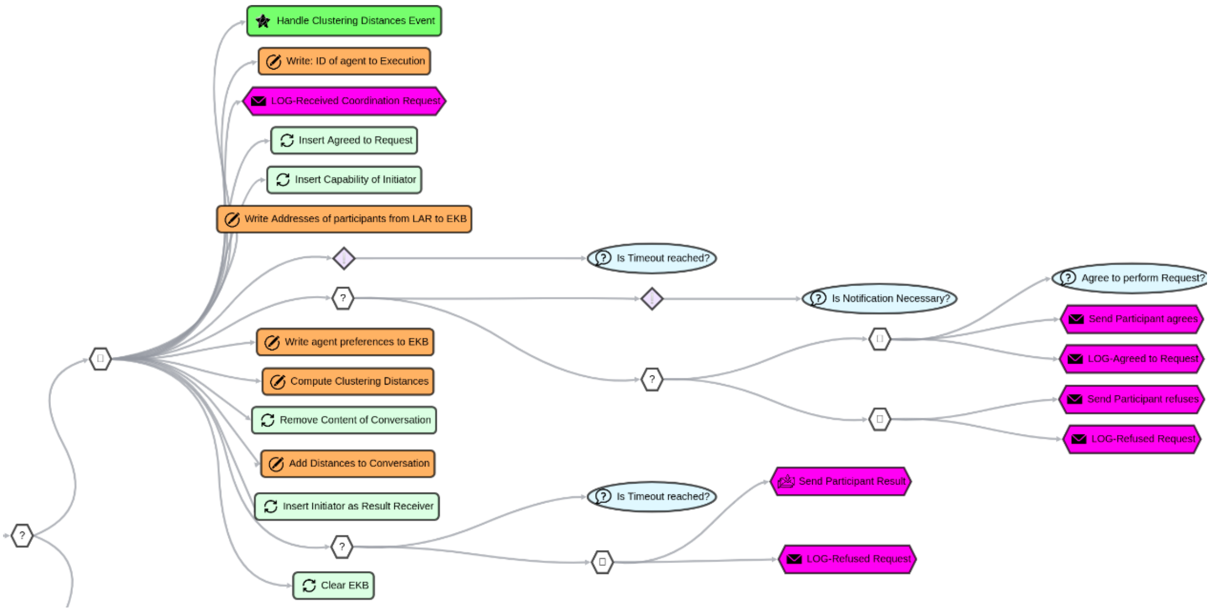


Figure 44-a. Handle Clustering Distances Request Templ. BT in AJAN Editor

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX domain:
<http://localhost:8090/rdf4j/repositories/domain_specific_ontology#>

CONSTRUCT  {
          ?macInstance  mac:hasDistanceScore  ?distanceScoreIri .
          ?distanceScoreIri   rdf:type  mac:DistanceScore ;
                    mac:hasValue  ?minTotalDistance ;
                    mac:isComputedBy  ?thisAgentId ;
                    mac:isComputedAgainst ?participantId ;
                    mac:isComputedFor   ?macId .
}
```

```
WHERE {
{
      SELECT ?macInstance ?thisAgentId ?participantId ?macId
(MIN(?totalDistance) AS ?minTotalDistance)
      WHERE {    {
              SELECT ?macInstance ?thisAgentId ?participantId  ?totalDistance
?macId
              WHERE {
                ### retrieve this agent
                ?thisAgentIRI   rdf:type  ajan:Agent, ajan:ThisAgent ;
                        ajan:agentId  ?thisAgentId .

                ### retrieve Preferences of this agent
                ?thisAgentPrefs   rdf:type  mac:AgentPreferences ;
                          domain:hasGenderPreference  ?thisGenderPref ;
                          domain:hasNationPreference  ?thisNationPref ;
                          domain:hasGenderPrefWeight  ?thisGenPrefWeight ;
                          domain:hasNationPrefWeight  ?thisNatPrefWeight .

                ### retrieve participant agent id
                ?macInstance  rdf:type  mac:MACProblemInstance ;
                        mac:hasId   ?macId ;
                        mac:hasParticipants   ?participantId .

                ### rule out agent computing similarity with itself
                FILTER(?thisAgentId != ?participantId)

                ### retrieve Personal Info of this agent
                ?conversation   rdf:type  mac:Conversation .
                ?conversation   mac:hasContent  ?thisAgentProfile,
?partAgentProfile.
                ?thisAgentProfile   rdf:type  mac:AgentProfileInfo ;
                          mac:belongsTo   ?thisAgentId ;
                          domain:hasGender  ?thisGender ;
                          domain:hasNationality   ?thisNation .

                ### retrieve Profile Info of the participant agent(s)
                ?partAgentProfile   rdf:type  mac:AgentProfileInfo ;
                          mac:belongsTo   ?participantId ;
                          domain:hasGender  ?participantGender ;
                          domain:hasNationality   ?participantNation .

                ### set config values
                BIND(0 AS ?matchScore)                  BIND(2 AS ?unmatchScore)

                ### compute distance value between this and participant agents
                ### Gender preference
                BIND(IF(LCASE(?thisGenderPref) = "dont mind" ||
LCASE(?thisGenderPref) = "don't mind"
|| (LCASE(?thisGenderPref) = "same" && ?thisGender = ?participantGender) ||
(LCASE(?thisGenderPref) = "mixed" && ?thisGender != ?participantGender),
```

```
?matchScore, ?unmatchScore) AS ?genderDistance)

                ### Nationality preference
                BIND(IF(LCASE(?thisNationPref) = "dont mind" ||
LCASE(?thisNationPref) = "don't mind"
|| (LCASE(?thisNationPref) = "same" && ?thisNation = ?participantNation) ||
(LCASE(?thisNationPref) = "mixed" && ?thisNation != ?participantNation),
?matchScore, ?unmatchScore)
AS ?nationDistance)

                ### total distance
                BIND((?genderDistance + ?nationDistance) AS ?totalDistance)
              } GROUP BY ?macInstance ?participantId  ?totalDistance
?thisAgentId ?macId }
          } GROUP BY ?macInstance ?participantId ?thisAgentId ?macId }
        BIND(BNODE() AS ?distanceScoreIri) }
```

Figure 44-b. SPARQL query of "**Compute Clustering Distances**" node

5. Name: **SolveClusteringTemplBT**, Label: "*Solve Clustering Templ. BT*"

The objective of this BT is to **solve a clustering problem with the HDBSCAN** algorithm (Campello & Moulavi, 2015). Just like in "*Solve CSGP Template BT*", the **solver** (i.e. HDBSCAN) can be replaced with any other clustering solver algorithm. In this BT, once the dedicated agent computes the **solution for the clustering problem** with the HDBSCAN algorithm, it produces "Finalize Coordination Event" which triggers "*Finalize Coordination to Groups BT*".

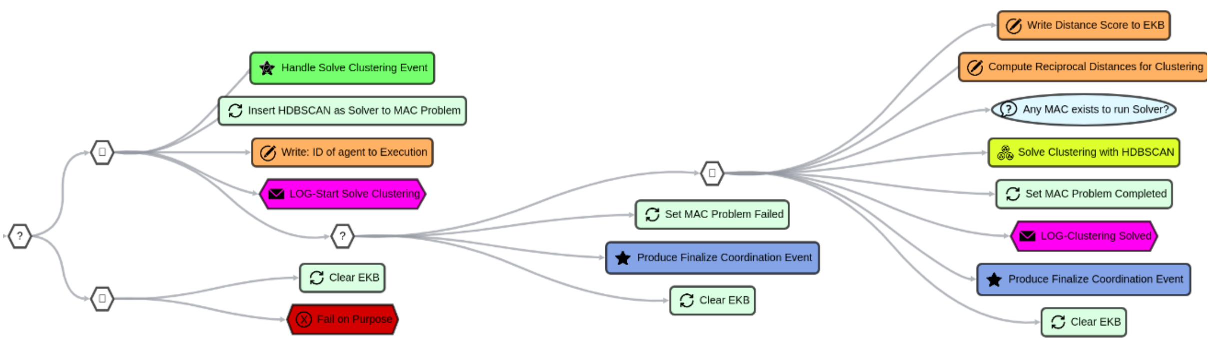


Figure 45. Solve Clustering Templ. BT in AJAN Editor

6. Name: **FinalizeCoordinationToGroupsBT**, Label: "*Finalize Coordination to Groups BT*"

In order to finalize a coordination process, agents in clustering protocol use the same BT that they use in CSGP protocol. Refer to CSGP Protocol BTs section for more info about this BT.

7. Name: **ReceiveCoordRequestResponseTempBT**, Label: “*Receive Coord. Request Response Temp. BT*”

In order to save the responses in coordination protocols, agents use the same BT in CSGP protocol as they use in the Request protocol. Refer to the Request Protocol section for more info about this BT.

MAJAN Postman Collections

Postman is a platform with lots of features including sending HTTP requests such as **Get**, **Post**, **Put**, **Delete**, etc. AJAN agents can be **created**, and **executed** by sending Post requests to AJAN Service. In order to support users to be able to create multiple agents, populate their agent repositories and start coordination protocols with one click, **MAJAN provides postman collections** which are explained in the following sections.

Create Agents

Create with Postman

AJAN Editor doesn't support **creating multiple agents at the same time**. Therefore, MAJAN provides a Postman Collection, called "MAJAN - Create Agents.postman_collection.json" consisting of **ten Postman Requests** to create ten agents with one click. This collection is provided in the "MAJAN/Postman Collections/Create Agents" folder in MAJAN repository in GitHub.

Step 1. Firstly, this collection must be **imported into Postman**. To do so, first open Postman, click on the **Collections tab** and then click on **Import button** in the top right corner as shown *in figure 46-a*.

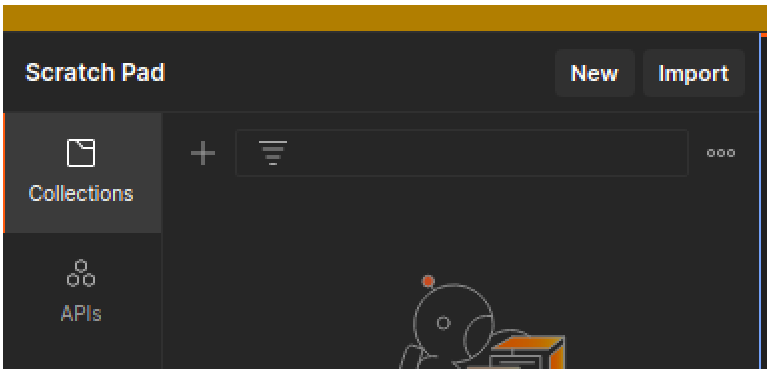


Figure 46-a. Import collection into Postman

Step 2. Once the **Import button** is clicked, *figure 46-b* will appear where it is asked to **upload** the collection. Find and upload the correct collection in "MAJAN/Postman Collections" folder.

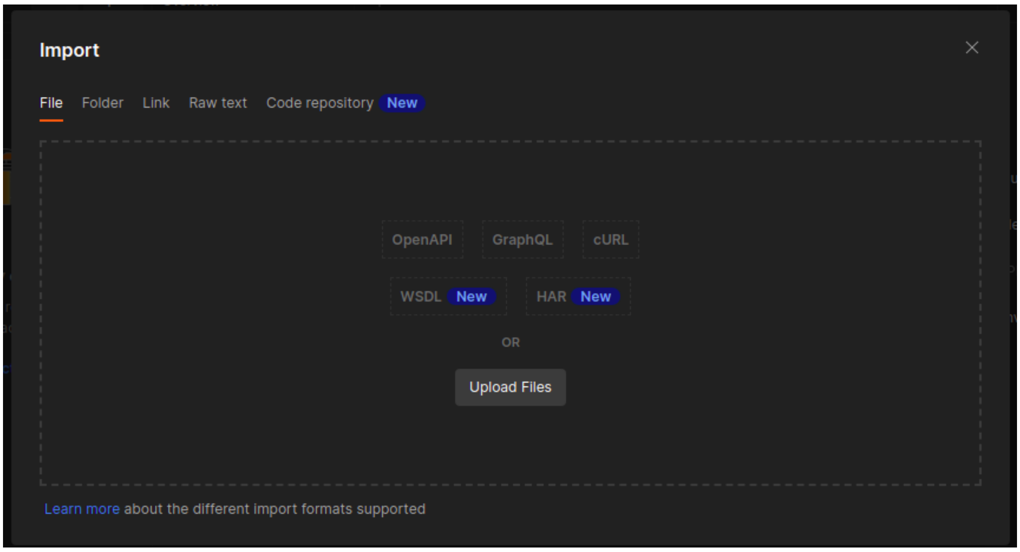


Figure 46-b. Upload collection to Postman

Step 3. After uploading the collection, we should see it in the **Collections** section as shown *in figure 46-c*.

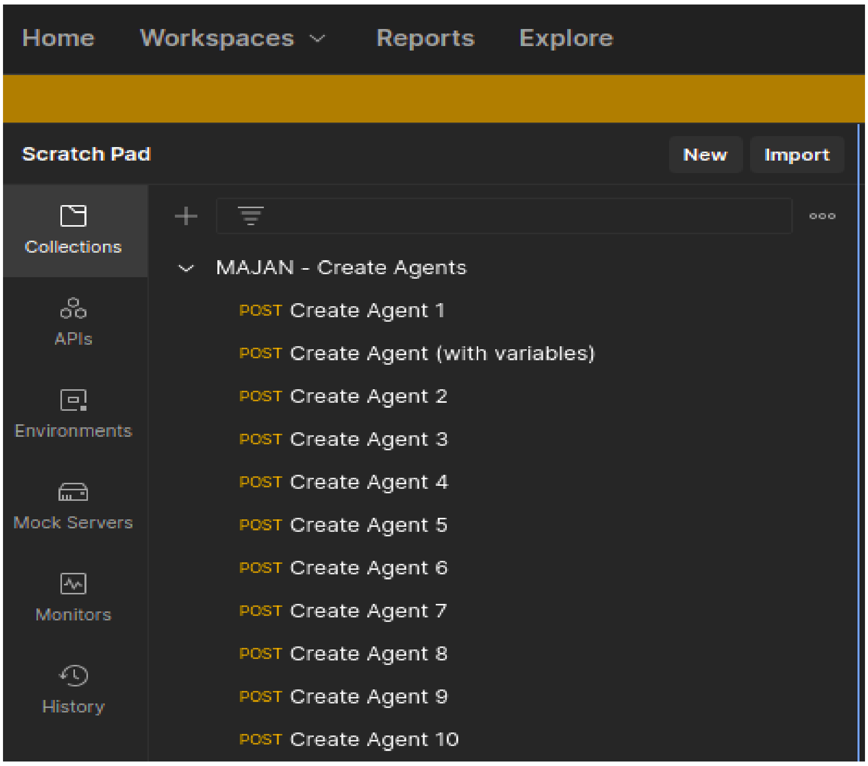


Figure 46-c. Collection is uploaded and available in Postman

Step 4. Since the collection is available in postman, we can run it now. To do so, click on the **three dots** *(in figure 46-d)*. Then click on the "**Run collection**" button in the opened list.

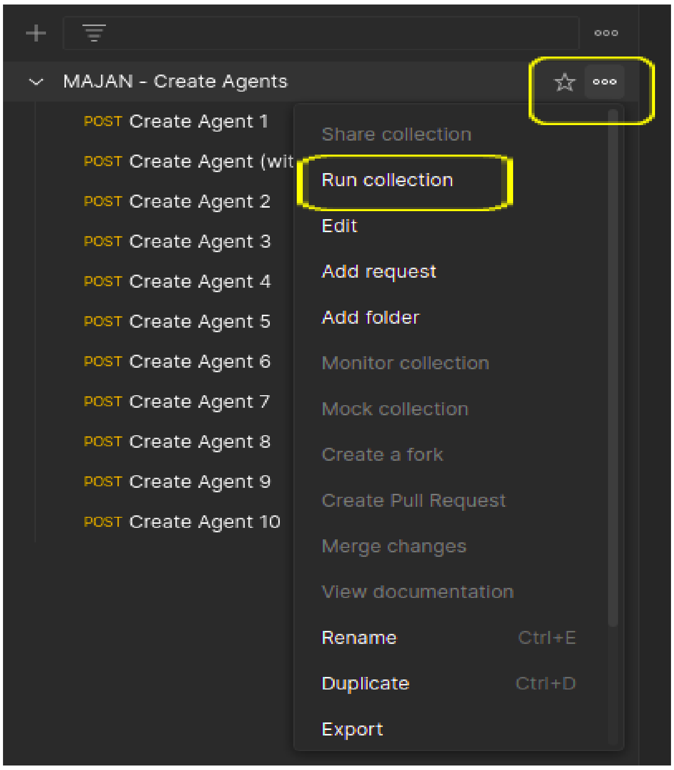


Figure 46-d. Running the collection

Step 5. Select the Postman requests (i.e. *Create Agent x*) that we want to create as shown in *figure 46-e*.

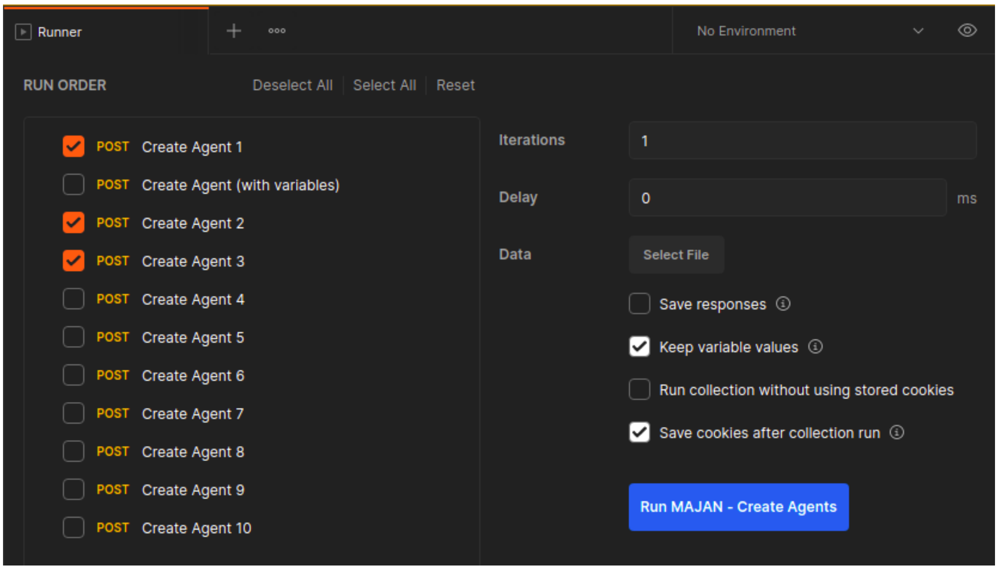


Figure 46-e. Selection of 3 requests to create 3 agents

Step 6. We need to select a **configuration file** in the "**Data → Select File**" section in Postman *(figure 46-f)*. The configuration file, **named as "IP-PORT.json"**, is provided in the **same folder** as the collection file, and it consists of values for **IP and Port of AJAN Service** to create respective agents.

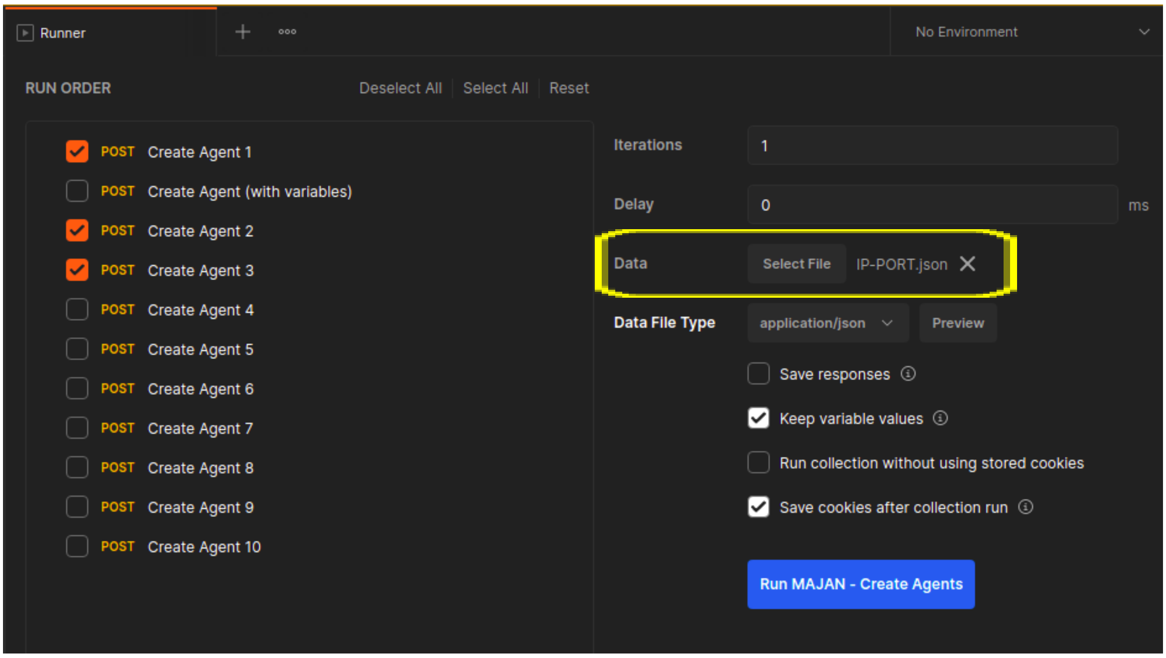


Figure 46-f. Selection of configuration file

Step 7 (final). Click the **Run MAJAN – Create Agents button** in blue to create the selected agents. *Figure 46-g* shows the **result of creation of selected agents**.

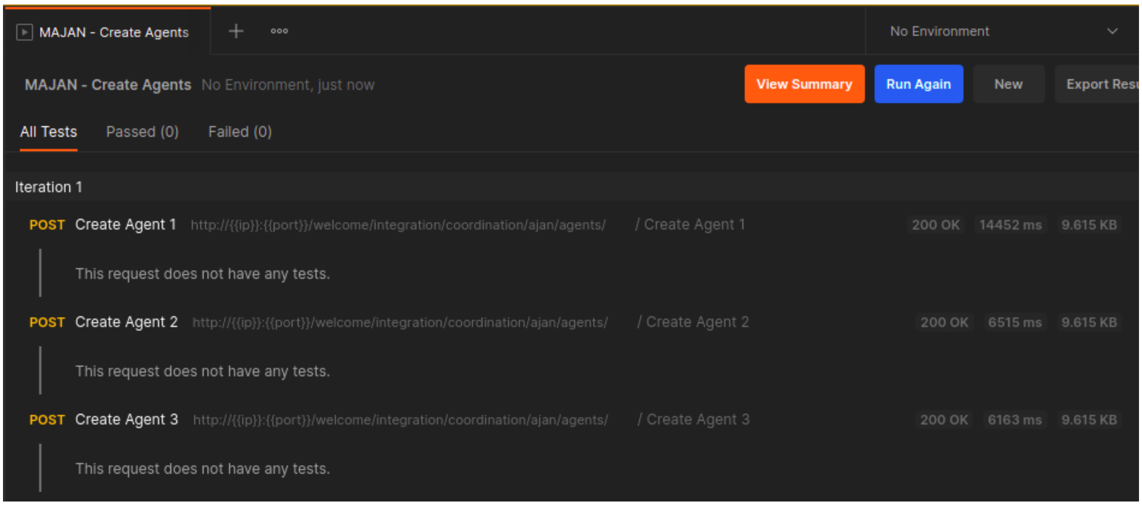


Figure 46-g. Three agents are created

Create in AJAN Editor

Additionally, AJAN Editor supports creating agents one by one in **Instances** subsection of **Agents** section as shown *in figure 47-a and figure 47-b*. However, there are no predefined templates to create agents in AJAN Editor , and it is not possible to create multiple agents at once unlike Postman.

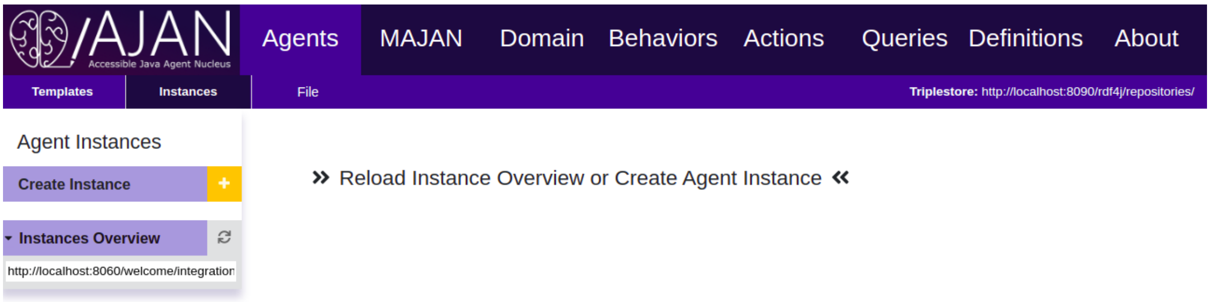


Figure 47-a. **Create agents** in *Instances* subsection in AJAN Editor

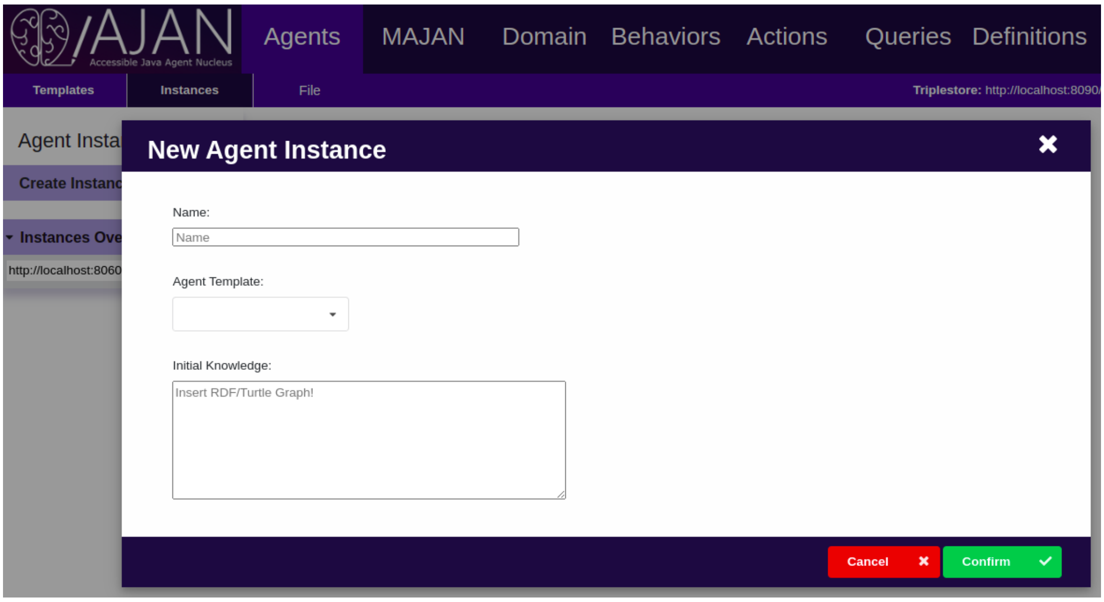


Figure 47-b. Form to enter *agent name*, *template* and *initial knowledge*.

Local Agents Repository

Overview

In the multiagent coordination, agents **need to communicate** with each other and therefore, they need to **know** how they can **contact** other agents. To do so, agents make use of their **Local Agents Repository (LAR)** where they store the **contact information of other agents**. However, it is necessary to populate the LAR of agents before they start coordination. Thus, MAJAN provides a collection called “MAJAN - Populate LAR.postman_collection.json” which consists of **eight Postman requests** with different content in terms of **agent name and amount**. These requests contain agent **URI, ID, and address in the payload**, and they are sent to “populateLAR” capabilities of agents. Moreover, as shown *in figure 48-a*, these requests have 3 variables: **ip, port, and agentId** which are given in the configuration file we select to run collection.

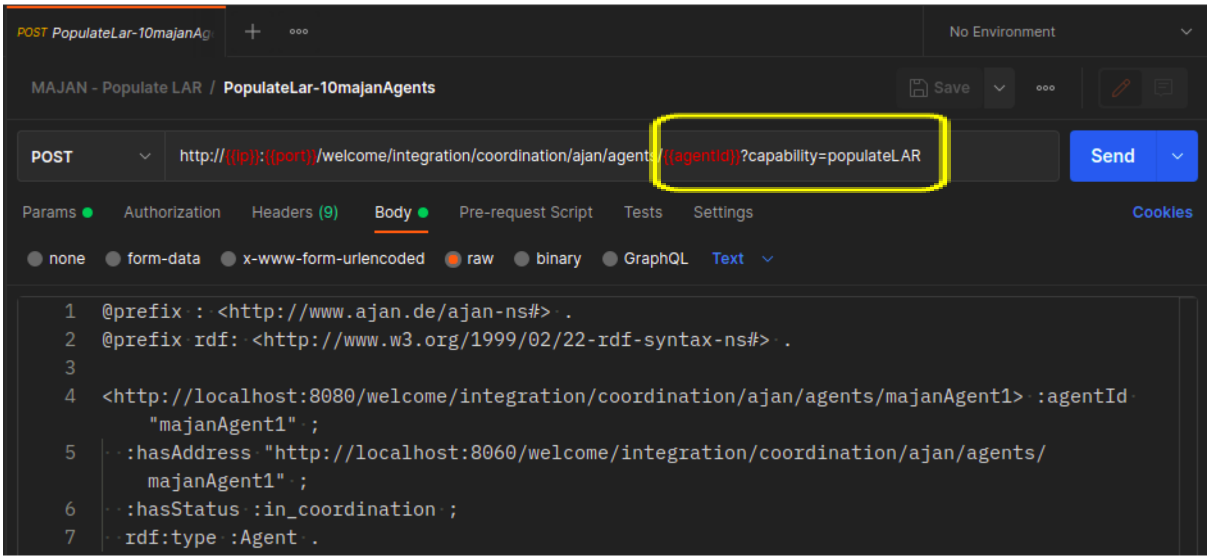


Figure 48-a. The Request to populate LAR , and it requires three values for ip, port and agentID variables

Populate Local Agents Repository

Step 1. **Import** Postman collection: "MAJAN - Populate LAR.postman_collection.json". It is provided in **"MAJAN/Postman Collections/Populate LAR" folder**.

Step 2. Click on "**Run Collection**" just like it was done in step 2 of Create Agents.

Step 3. **Select "PopulateLar-10majanAgents" request**. In the body of this request, URLs (i.e. contact info) of 10 agents are specified. This body is sent to the agents who are specified in the configuration file (next step). It doesn't hurt to have more info in repository. Therefore, we created this request that contains info about 10 agents instead of creating 10 different requests that contain info about 1, 2, 3, … 10 agents one by one.

Step 4. **Select configuration file: "5-IP-PORT-MajanAgentId.json"** by using "**Data -> Select File**" as shown *in figure 48-b*. In this JSON file, 5 MajanAgents are specified, and the selected request will be sent to all 5 of them. Since we have created 3 agents, only those 3 will receive this request. The requests to other 2 agents will not be successful and this will not cause any error during the execution of the collection. This JSON configuration file is in "MAJAN/Postman Collections/Populate LAR" folder.

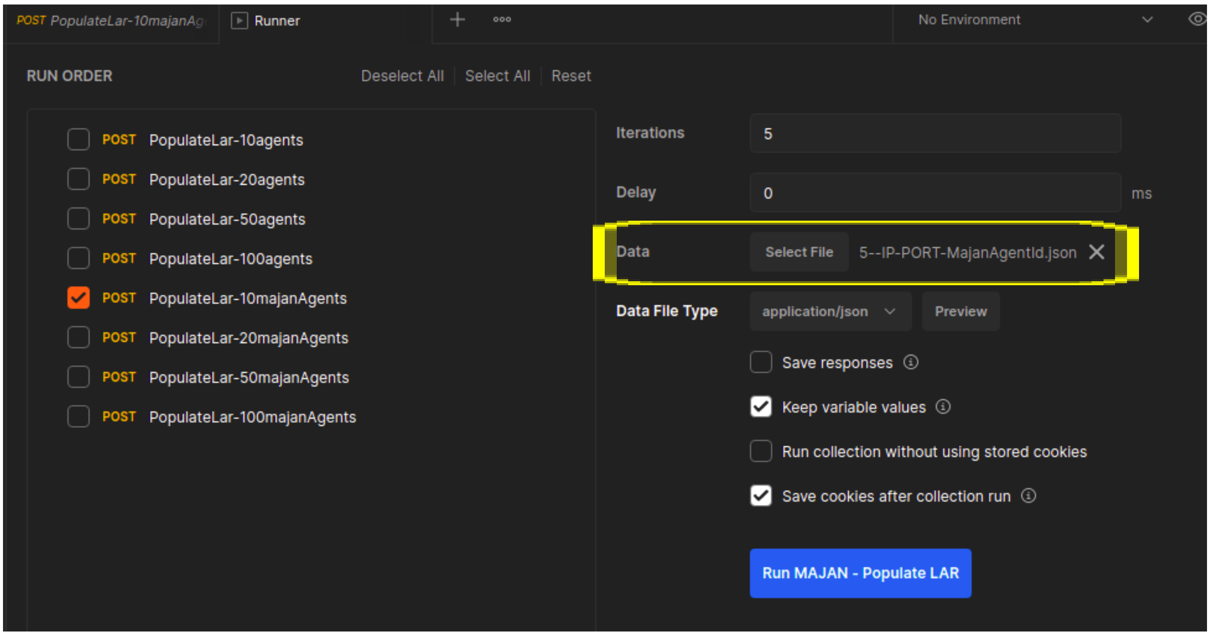
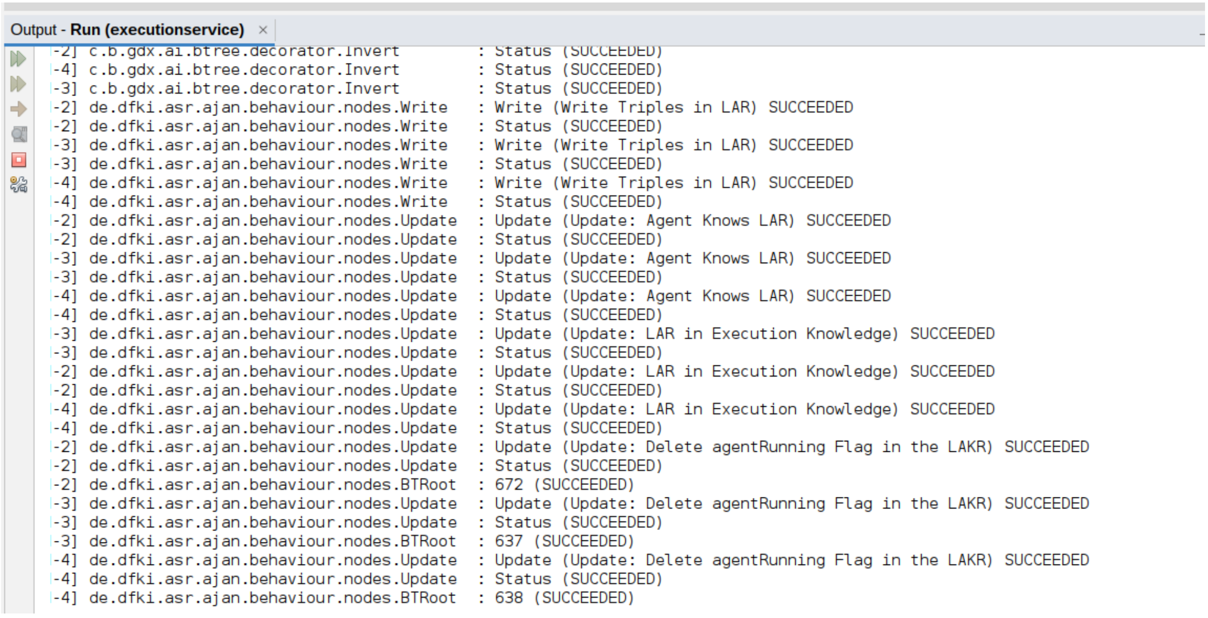


Figure 48-b. Selection of PopulateLar-10majanAgents and respective configuration file.

Step 5 (final). Click on **Run MAJAN-Populate LAR button** in blue.

After the selected request is sent to agents, we will see the logs as shown *in figure 48-c* which represents that agents **successfully populated their LAR**.




```
-2] c.b.gdx.ai.btree.decorator.Invert        : Status (SUCCEEDED)
-4] c.b.gdx.ai.btree.decorator.Invert        : Status (SUCCEEDED)
-3] c.b.gdx.ai.btree.decorator.Invert        : Status (SUCCEEDED)
-2] de.dfki.asr.ajan.behaviour.nodes.Write   : Write (Write Triples in LAR) SUCCEEDED
-2] de.dfki.asr.ajan.behaviour.nodes.Write   : Status (SUCCEEDED)
-3] de.dfki.asr.ajan.behaviour.nodes.Write   : Write (Write Triples in LAR) SUCCEEDED
-3] de.dfki.asr.ajan.behaviour.nodes.Write   : Status (SUCCEEDED)
-4] de.dfki.asr.ajan.behaviour.nodes.Write   : Write (Write Triples in LAR) SUCCEEDED
-4] de.dfki.asr.ajan.behaviour.nodes.Write   : Status (SUCCEEDED)
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Agent Knows LAR) SUCCEEDED
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Agent Knows LAR) SUCCEEDED
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Agent Knows LAR) SUCCEEDED
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: LAR in Execution Knowledge) SUCCEEDED
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: LAR in Execution Knowledge) SUCCEEDED
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: LAR in Execution Knowledge) SUCCEEDED
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Delete agentRunning Flag in the LAKR) SUCCEEDED
-2] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-2] de.dfki.asr.ajan.behaviour.nodes.BTRoot  : 672 (SUCCEEDED)
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Delete agentRunning Flag in the LAKR) SUCCEEDED
-3] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-3] de.dfki.asr.ajan.behaviour.nodes.BTRoot  : 637 (SUCCEEDED)
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Update (Update: Delete agentRunning Flag in the LAKR) SUCCEEDED
-4] de.dfki.asr.ajan.behaviour.nodes.Update  : Status (SUCCEEDED)
-4] de.dfki.asr.ajan.behaviour.nodes.BTRoot  : 638 (SUCCEEDED)
```

Figure 48-c. Logs in Netbeans for Population of LAR of 3 agents.

Multi Agent Coordination

Overview

In order to **start coordination**, agents need to receive the necessary **signal** to their respective endpoints. To do so, **MAJAN provides** a collection consisting of **three requests** to **start *Request, CSGP, and Clustering coordination protocols***. Each of these requests contains the necessary message, including:

- [ ] **Use Case**: title of the use case
- [ ] **Participants**: name of participating agents
- [ ] **Notification Necessary**: whether it is necessary to return a notification back during coordination
- [ ] **Timeout**: expiration date of the coordination process. Always make sure that the Timeout date is not expired (i.e. *don't set a date that is PAST*)
- [ ] **Quorum**: amount of agents that should actively participate in the coordination.

Refer to MAJAN Ontology to get more detailed information about the attributes listed above. Additionally, the name of the collection is "MAJAN - Start Coordination.postman_collection.json" and it is provided in "MAJAN/Postman Collections/Start Coordination" folder in MAJAN GitHub repository.

Start Multi Agent Coordination

Step 1. **Import** Postman collection: "**MAJAN/Postman Collections/Start Coordination/MAJAN - Start Coordination.postman_collection.json**".

Step 2. **Select one** of the these three requests by **clicking on one of them**:

1. "Start Request Coordination Protocol": to start Request coordination protocol
2. "Start CSGP Coordination Protocol": to start CSGP coordination protocol, "sendCsgpCoordRequest" capability (*figure 49-a)* is requested.
3. "Start Clustering Coordination Protocol": to start Clustering coordination protocol

   As an example, "Start CSGP Coordination Protocol - 3 agents" is selected as shown in *figure 49-a*. This request starts coordination for 3 agents.

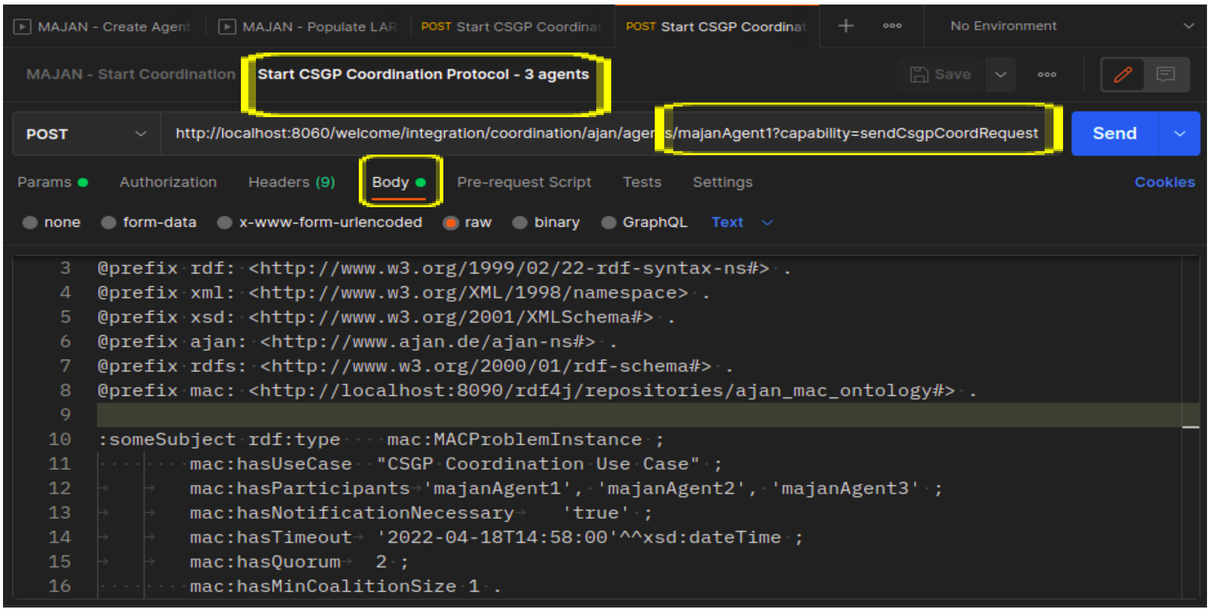


Figure 49-a. “Start CSGP Coordination Protocol” request is selected that informs agent to start CSGP protocol.

Step 3. Make sure that message body is correct. Check the followings:

- ➢ Ensure that a past date is not set to **Timeout**.
- ➢ Ensure that the value of **Quorum** is not greater than the amount of participants.

Step 4 (final). **Click** on “**Send**” in blue (*figure 49-a)*

Details

Furthermore, it is enough to send this message to **only one agent** which will act as a **dedicated agent**, and it will **inform** other participant agents respectively. That is why there is no need to upload any configuration file to send this request, unlike “Create Agents” and “Populate LAR” sections. *In figure 49-a*, we send this request to majanAgent1 as you can see from the URL. Once the request is sent, agents will log the execution of their BTs into NetBeans or CMD, depending on the execution method. Additionally, agents will log their activities to **Monitoring panel** in MAJAN. Refer to the next section (i.e. **Monitoring Activities of Coordinating Agents**) to get more information.

MAJAN Web features in AJAN Editor

In order to provide features required for multiagent coordination, AJAN Editor has been extended appropriately with MAJAN. The **first feature** is the **monitoring panel** where users can monitor the *activities of coordinating agents*. Moreover, **MAJAN allows users** to view the **result of multiagent coordination (into groups)** in a user-friendly way rather than RDF triples in RDF4J repositories. Furthermore, users can execute a **centrally running grouping algorithm** and **view its result** appropriately in MAJAN. Finally, users can compare *MAC and Central solutions* in the **Compare Solutions section**.

Monitoring the Activities of Coordinating Agents

All the execution of all **BT nodes** is normally **logged** to Netbeans or CMD depending on the execution method of AJAN Service. This logging technique is good when there is one agent running in AJAN Service. However, it gets **difficult very quickly** to **debug** when multiple agents are **running in parallel** and all of their BT nodes are **logged to the same place** since these logs don't contain the ID of the agent and usually there are lots of BT nodes that create a huge amount of logs in Netbeans. Therefore, the **Monitoring Panel** in MAJAN is developed to overcome these issues and allow users to be able to **monitor the coordination process**. The UI of this panel is shown *in figure 50-a*.

In order to log any activity, agents need to explicitly **send a message to MAJAN Logging Service**. The message can be constructed with the **SPARQL query** shown in *figure 50-b* and all log messages should have a **common structure** such that MAJAN can understand them. It is important to note that the **Activity field** can be specified **flexibly**. In figure 50-b, the Activity field is built by using **Concat function of SPARQL**.

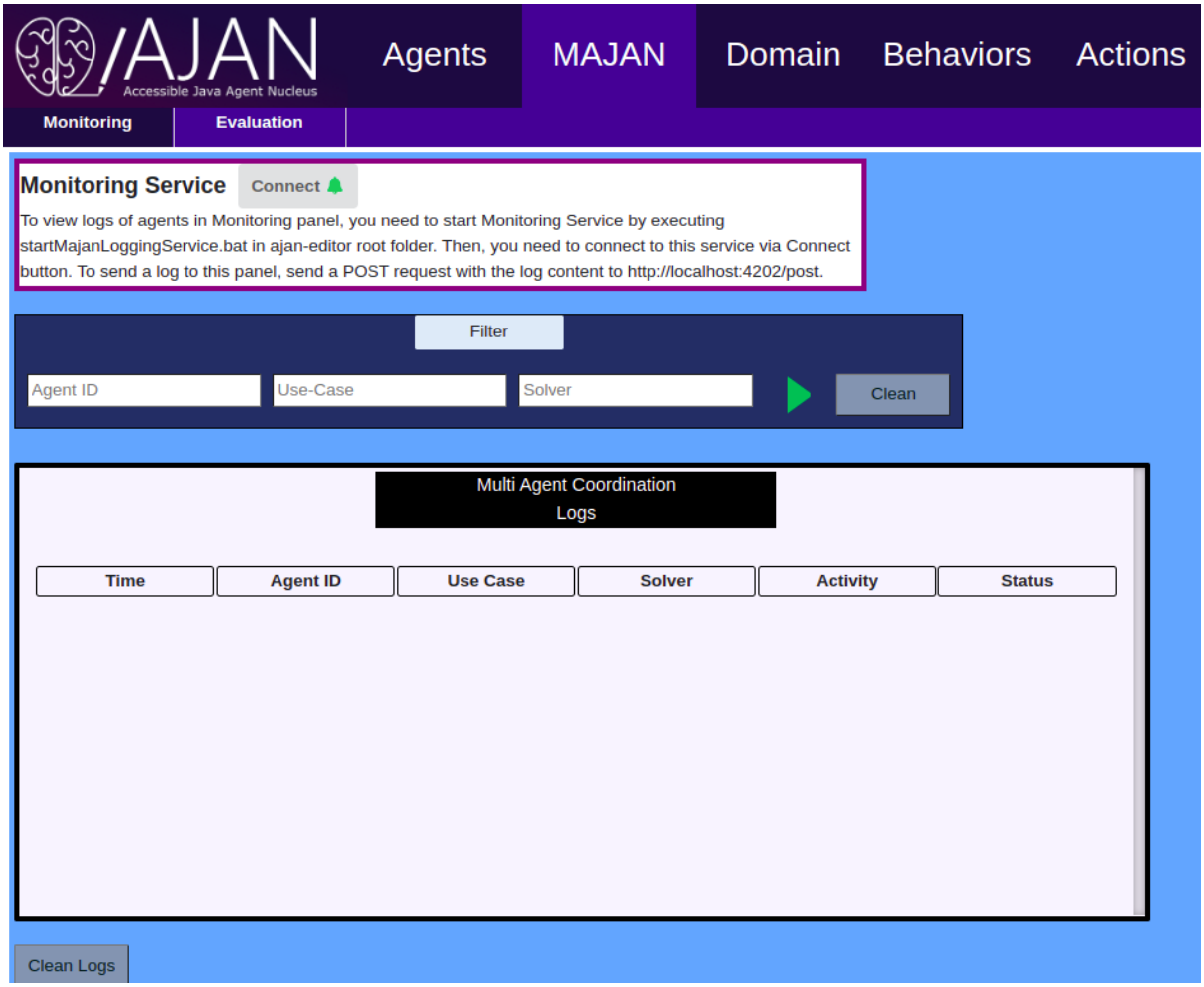


Figure 50-a. Monitoring Panel UI in AJAN Editor

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

CONSTRUCT {
?newBnode rdf:type  mac:Log ;
          mac:hasAgentId  ?thisAgentId ;
          mac:hasUseCase  ?useCase ;
          mac:hasSolver   ?solver ;
          mac:hasActivity   ?activity ;
          mac:hasActivityStatus   ?activityStatus .
}
WHERE {
?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
            ajan:agentId  ?thisAgentId .

?macInstance  rdf:type  mac:MACProblemInstance ;
              mac:hasId   ?macId ;
```

```
              mac:hasUseCase  ?useCase .
OPTIONAL {
?macInstance  rdf:type  mac:MACProblemInstance ;
              mac:hasSolver   ?solver .
}
OPTIONAL{
?conversation   rdf:type  mac:Conversation ;
                mac:hasId   ?convId ;
                mac:hasTimeout  ?timeout ;
                mac:hasNotificationNecessary  ?notificationNecessary ;
                mac:hasMacProblemId   ?macId .
}
{
SELECT (GROUP_CONCAT(?participantId ; separator=", ") AS ?participants)
WHERE{
?conversation   rdf:type  mac:Conversation ;
                mac:hasParticipants   ?participantId .
}}
{
SELECT (GROUP_CONCAT(?contentIri ; separator=", ") AS ?contentIris)
WHERE{
?conversation   rdf:type  mac:Conversation ;
                mac:hasContent  ?contentIri .
}}
BIND(CONCAT("Starting to Solve Clustering\\nConversation ID: ", STR(?convId),
"\\nConversation Participants: ",STR(?participants), "\\nTimeout:
",STR(?timeout), "\\nNotification Necessary: ", ?notificationNecessary, "\\nMAC
ID: ", ?macId, "\\nContent IRIs: ", ?contentIris) AS ?activity)
BIND("Succes" AS ?activityStatus)
{
BIND(BNODE() as ?newBnode)
}
}
```

Figure 50-b. SPARQL query to construct a log message for MAJAN Monitoring Panel.

Activate Monitoring Feature

In order to be able to **receive logs** and **display them in UI**, the **first step** is to start **MAJAN Logging Service** such that the logs that are sent by agents can be **read, parsed and displayed by MAJAN**. To start logging service, find "startMajanLoggingService.bat" in the **AJAN Editor folder** and just **run** it. If you are using **Linux**, this file might not be executable. In that case, you can run "*node majanLoggingService.js*" command in terminal in AJAN Editor root folder (i.e. MAJAN/AJAN_Editor_w_MAJAN). Once it is running, go to the **Monitoring UI** in MAJAN and click on the **Connect** button. Now it is activated.

Test Monitoring Feature

In case you would like to test whether log messages are received and displayed correctly before starting a MAC process, you can use the Postman request provided in "MAJAN – Logging.postman_collection.json" which is in "MAJAN/Postman Collections/MAJAN Logging" folder in MAJAN GitHub repository.

We can view the logs for our example of creating three agents and running **CSGP protocol** *in figure 50-c*.

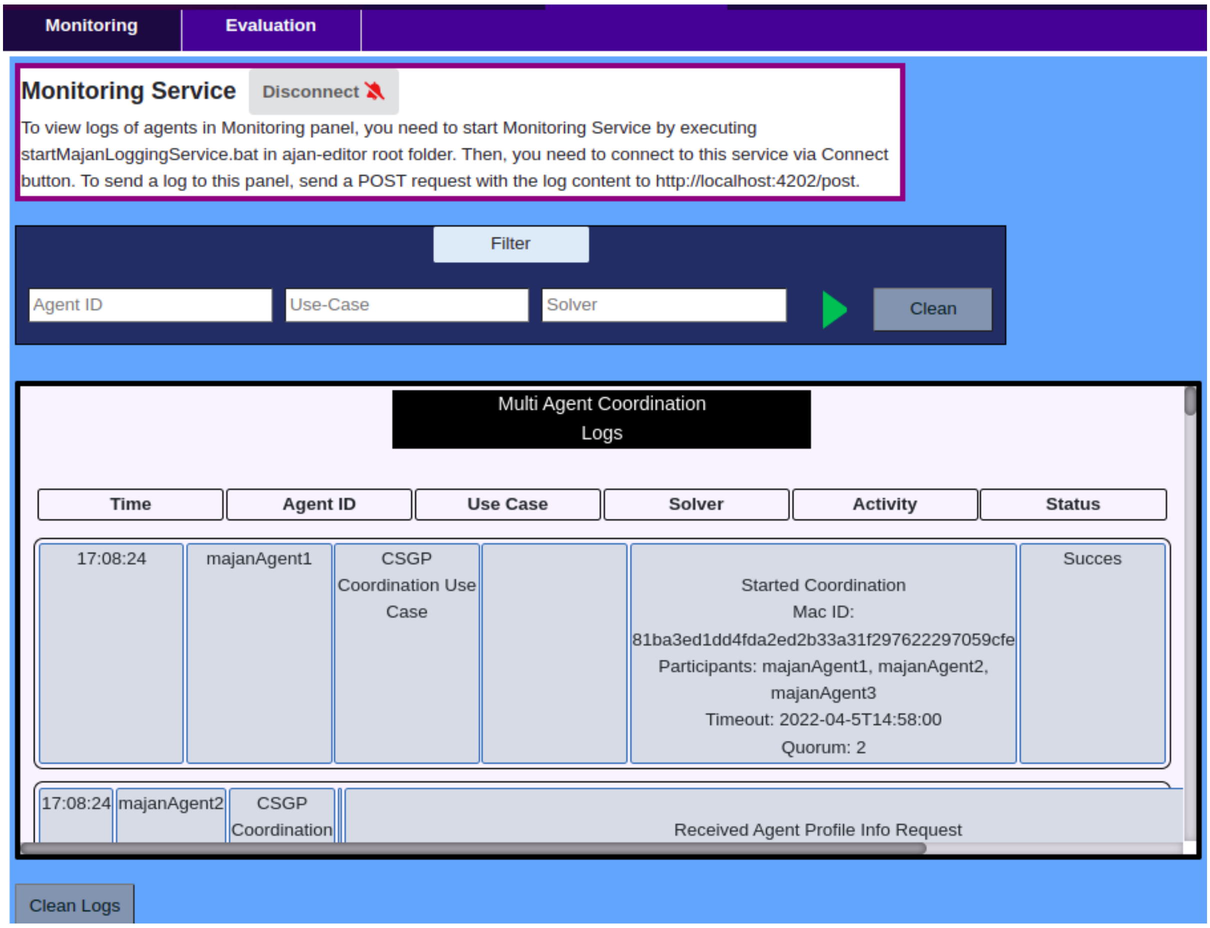


Figure 50-c. Logs for CSGP Coordination Protocol

Evaluating Results of Coordination into Groups

MAJAN provides an **Evaluation panel** (*in figure 51*) where users can display **grouping results** of MAC and centrally running algorithms and compare those results.

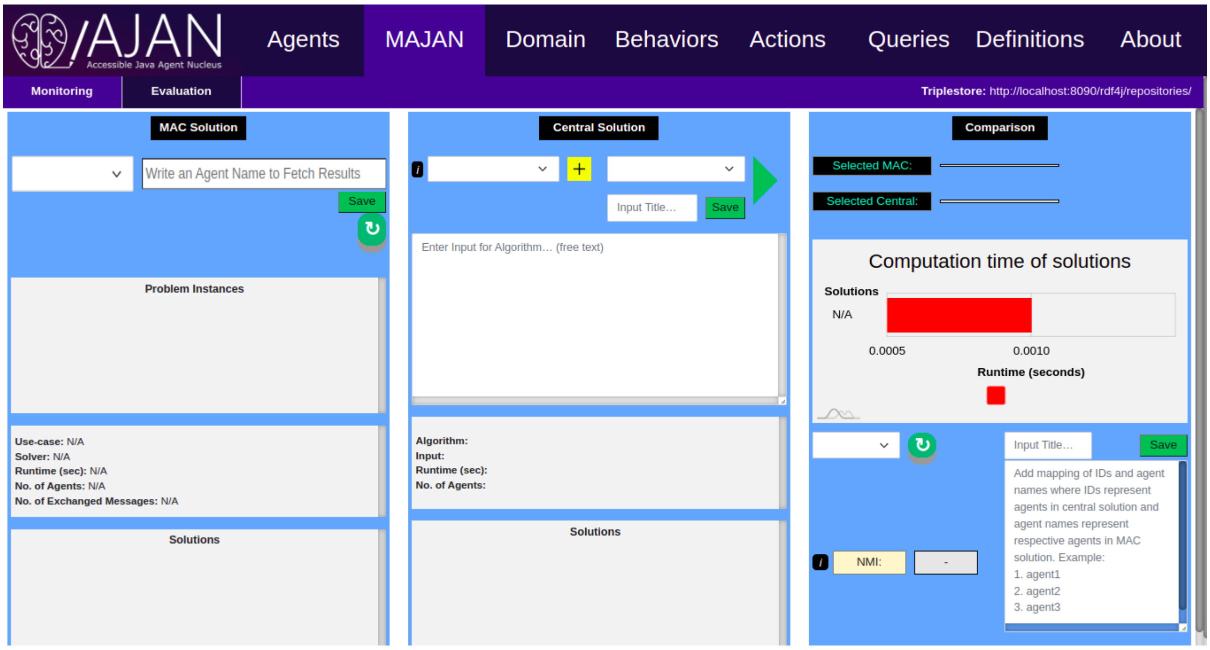


Figure 51. Evaluation panel of MAJAN in AJAN Editor

MAC Solution

Overview

MAC Solution section, as shown *in figure 51*, allows users to display **grouping results of MAC** processes.

- ➢ **Problem Instances** section displays all the problem instances stored in the selected repository, whether they have a solution or not.
- ➢ Once a problem instance is selected, its solutions (if they exist) are displayed in the **Solutions** section.

Step 1. **Write an agent name** (which has the results) in the "*Write an Agent Name to Fetch Results*" field and **click on Save button**.

Since we created majanAgent1, majanAgent2, and majanAgent3, we can write any of **their names (e.g. majanAgent1)** in the "*Write an Agent Name to Fetch Results*" field *(figure 52-a)*.

Step 2 (final). Click on **refresh (↻)** button. MAJAN **reads** the results (if they exist) from the selected repository and **displays** them accordingly.

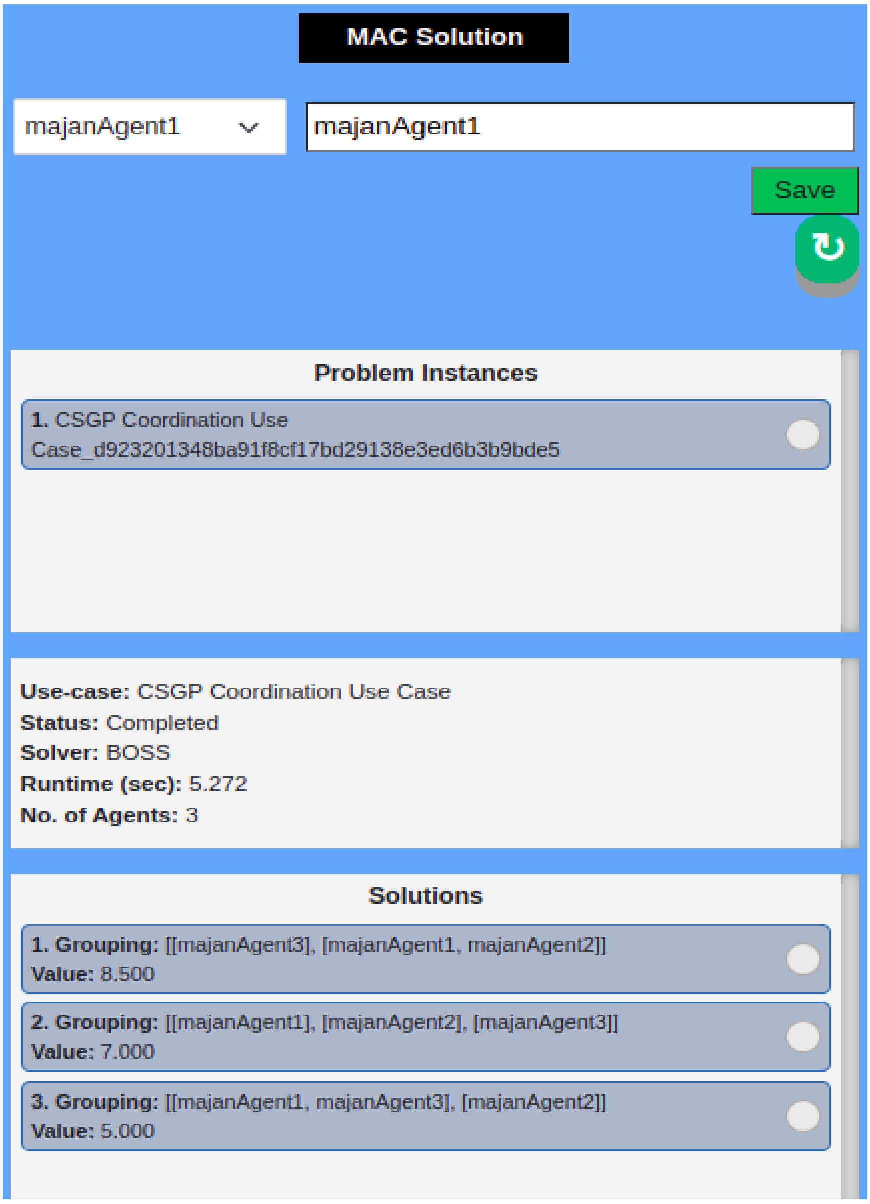


Figure 52-a. Results of running CSGP protocol displayed in MAC Solution panel

Central Solution

Overview

This section (*figure 53)* provides the features listed below:

- ➢ **Upload** a centrally running algorithm that is written in JAVA by uploading its **JSON configuration file**.
- ➢ **View** the **configuration info** of the uploaded algorithm by using the *tooltip (i)*.
- ➢ **Add and Save input** that can be accepted by uploaded algorithm.
- ➢ **Run the selected algorithm** with the selected input.
- ➢ **View** the result.

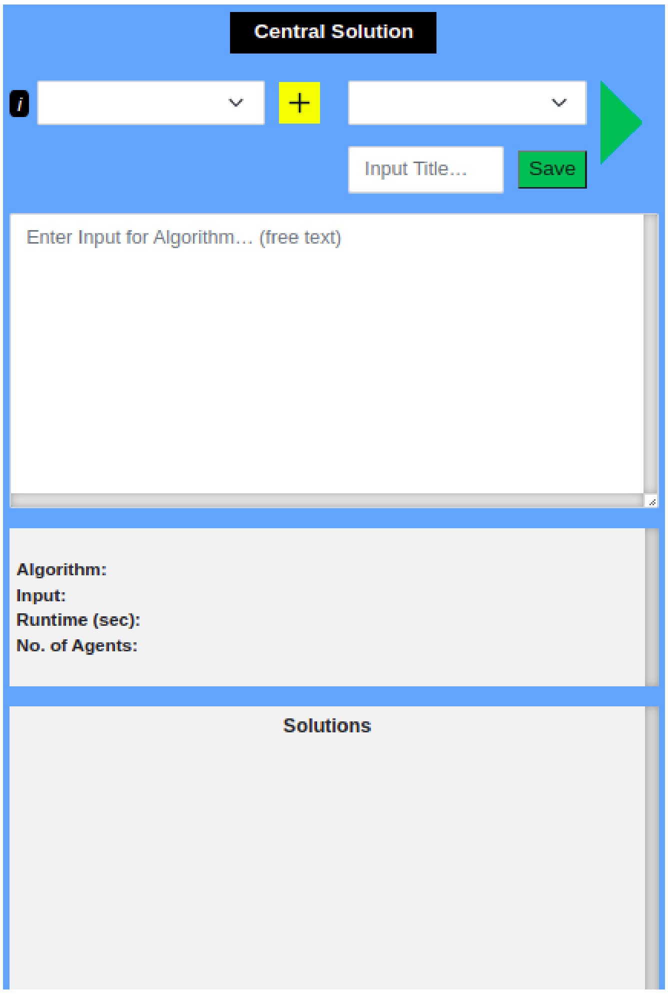


Figure 53. Central Solution section of MAJAN in AJAN Editor

Using Central Solution

Here we use the BOSS **(Changder & Aknine, 2021)** algorithm to explain the features one by one.

Step 1. **Click on** yellow plus button in Central Solution section (*figure 53*)

Step 2. **Select** “BOSS_config.json” file that is provided in the “MAJAN/Grouping Algorithms/BOSS” folder. For more detailed information about the configuration file, check the **Configuration File Structure section** below.

Step 3. **Click on *i*** button to display information about the uploaded and selected BOSS algorithm as a tooltip (*figure 54-a)*.

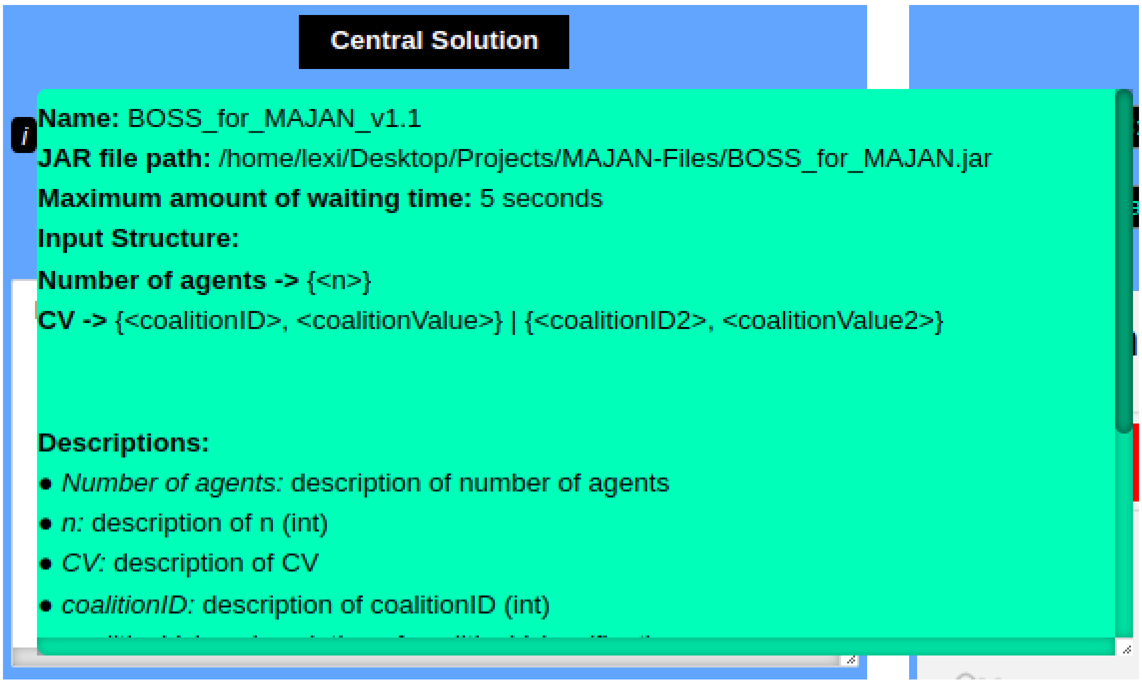


Figure 54-a. Representation of detailed information about BOSS algorithm in tooltip

Step 4. **Add input** that can be passed to the selected BOSS algorithm. To do so, **open** “*3-agents-input-for-BOSS.txt*” text file in “MAJAN/Grouping Algorithms/BOSS” folder. **Copy** its content and **paste** to “*Enter Input for Algorithm… (free text)*” field in Central Solution section (*figure 54-b*). This input contains coalitions and their values for 3 agents, since BOSS algorithm expects them. Refer to here for more info about computing coalitions and their values.

Step 5. **Add a title** in the “Input Title…” field to be able to save the input. And **click on Save** button in Central Solution (*figure 54-b)*.

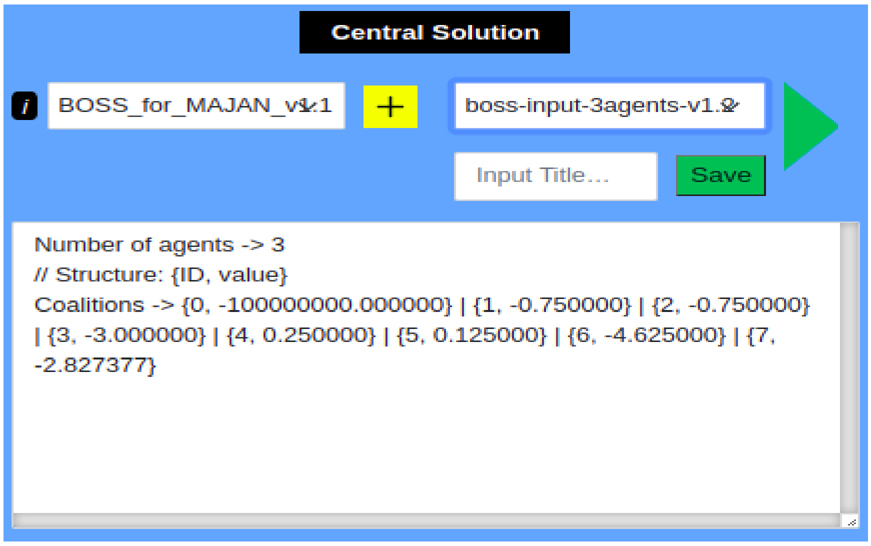


Figure 54-b. Example input added and selected to run BOSS algorithm

Step 6 (final). Click on the Run button to run the selected **BOSS algorithm** with the selected BOSS input. **The result** of the execution is shown in *figure 54-c*. To get more detailed information about running a JAR file in MAJAN, refer to Running JAR Files section.

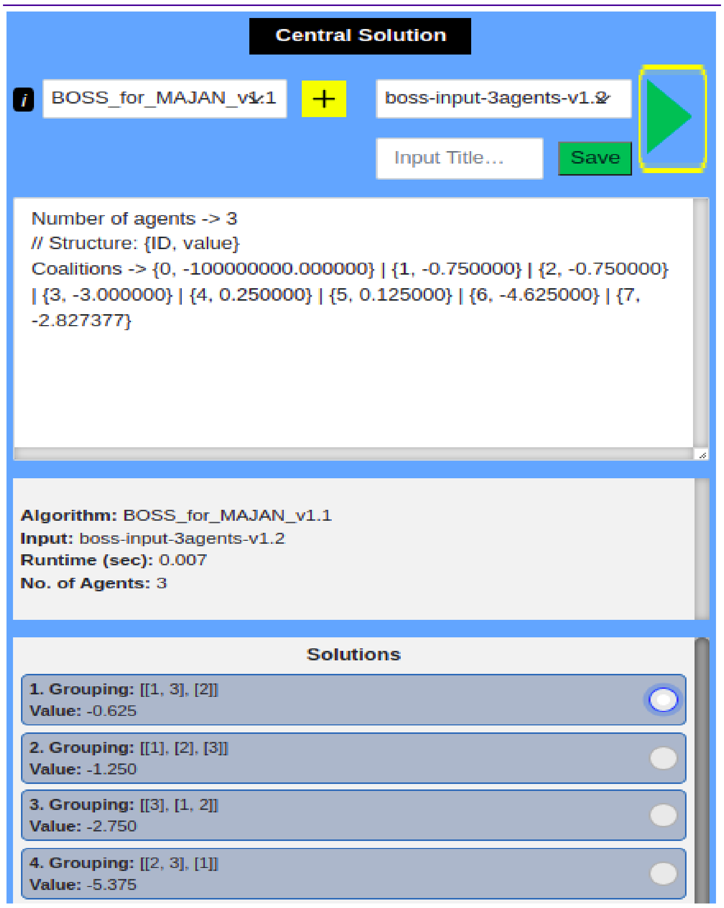


Figure 54-c. Grouping result of execution of BOSS algorithm

Centrally Running Grouping Algorithms

There are **four predefined** algorithms provided by MAJAN to run in **Central Solution section**:

1. **BOSS**: this algorithm solves CSGP by finding an exact solution. In "MAJAN/Grouping Algorithm/BOSS" folder in MAJAN GitHub repository, there are 3 files:
    1. **BOSS_config.json**: JSON configuration file for the algorithm
    2. **BOSS_for_MAJAN.jar**: jar file to execute the algorithm
    3. **BOSS-input-5agents.txt**: an example input for the algorithm
2. **HDBSCAN**: this algorithm solves clustering problems. In "MAJAN/Grouping Algorithm/HDBSCAN" folder, there are 3 files:
    1. **HDBSCAN_config**: JSON configuration file for HDBSCAN algorithm
    2. **HDBSCAN_for_MAJAN.jar**: jar file to execute the algorithm
    3. **hdbscan_majan_input.txt**: an example input for the algorithm
3. **LCC_BOSS**: LCC is a use case developed and solved with MAJAN. In order to find a central solution for LCC, it is being integrated into the original BOSS algorithm. The LCC_BOSS algorithm expects use-case related information and then passes the required information to the BOSS algorithm automatically and finally finds a solution for LCC. In "MAJAN/Grouping Algorithm/LCC_BOSS" folder, there are 3 files:
    1. **LCC_BOSS_config.json**: JSON configuration file for LCC_BOSS algorithm
    2. **LCC_BOSS_for_MAJAN.jar**: jar file to execute the algorithm
    3. **lcc-boss-input-5agents.txt**: an example input for the algorithm
4. **CHC_HDBSCAN**: CHC is a use case developed and solved with MAJAN. In order to find a central solution for CHC, it is being integrated into the original HDBSCAN algorithm. CHC_HDBSCAN algorithm expects use-case related information and then passes the required information to HDBSCAN algorithm automatically and finally finds a solution for CHC. In "MAJAN/Grouping Algorithm/CHC_HDBSCAN" folder, there are 3 files:
    1. **CHC_HDBSCAN_config.json**: JSON configuration file for CHC_HDBSCAN algorithm
    2. **CHC_HDBSCAN_for_MAJAN.jar**: jar file to execute the algorithm

3. **chc-hdbscan-input.txt**: an example input for the algorithm

To summarize, there are **3 steps** to **run** any of the provided algorithms:

1. Upload *JSON configuration file*
2. **Add input** that can be accepted for the selected algorithm
3. Click **Run button** and view results

Configuration File Structure

```
{
  "algorithm": "Example",
  "pathToJarFile": "/path/to/jar/file/Example.jar",
  "timeout": "5",
  "keywords": [
    {
      "keyword": {
        "type": "Number of agents",
        "description": "Represents the amount of agents given in the input",
        "element": {
          "item": {
            "name": "n",
            "example_value": "10",
            "datatype": "int",
            "description": "Represents the value for number of agents"
          }
        }
      }
    },
    {
      "keyword": {
        "type": "CoalitionValue",
        "description": "Represents the values for coalitions",
        "elements": [
          {
            "element": {
              "items": [
                {
                  "item": {
                    "name": "coalitionID",
                    "example_value": "0",
                    "datatype": "int",
                    "description": "ID of coalitions. Starting from 0"
```

```
                }
              },
              {
                "item": {
                  "name": "coalitionValue",
                  "example_value": "1.4",
                  "datatype": "float",
                  "description": "Value of a coalition"
                }
              }
            ]
          }
        },
        {
          "element": {
            "items": [
              {
                "item": {
                  "name": "coalitionID2",
                  "example_value": "1",
                  "datatype": "int",
                  "description": "ID of coalitions. Starting from 0"
                }
              },
              {
                "item": {
                  "name": "coalitionValue2",
                  "example_value": "1",
                  "datatype": "float",
                  "description": "Value of a coalition"
                }
              }
            ]
          }
        }
      ]
    }
  }
 ]
}
```

Figure 55-a. Example configuration file

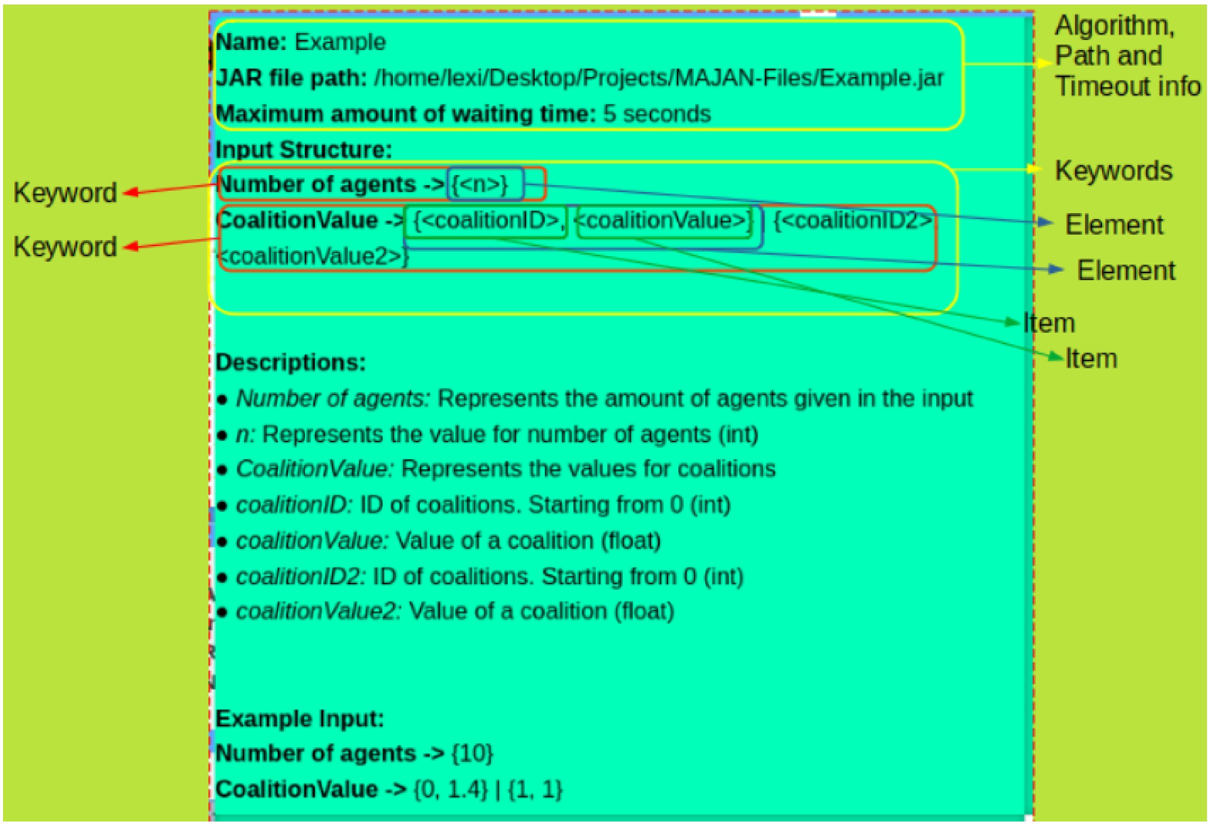


Figure 55-b. Representation of example configuration file

Configuration file is a way of **describing** an algorithm to MAJAN so that necessary information about algorithms can be **read and displayed** by MAJAN. To do so, the file should include values for "*algorithm*", *"pathToJarFile", "timeout" and "keywords*".

KEY in Json file	Description
algorithm	Describes the **name** of algorithm
pathToJarFile	Describes the **path** to the jar file of the algorithm
timeout	Describes the **amount** of **time** for MAJAN to wait until stopping the execution of algorithm since algorithm might get stuck in a loop
keywords, keyword, element, and item	Describe the **structure of input** that algorithm expects. An example

	configuration file is shown *in figure 55-a* which includes the json fields listed below. For a visual explanation, see *figure 55-b* where corresponding parts for each of the following json fields are shown clearly.
keywords	The starting point to describe the whole structure of input for the algorithm. It is named this way since it contains "keyword"s
keyword	As shown in *figure 55-b*, represents a single input line. An input line consists of "*type*", "*description*", "*element*" or "*elements*" as shown in *figure 55-a*
type	Represents the name for the type of input. For example, **number of agents**, **minimum group size, coalition value,** etc.
description	Contains the description of the json field it belongs to. The value for "*description*" fields are used in the **Descriptions** section as shown in figure 55-a.
element	Describes **values** to be given to the algorithm. It has "*item*" or "*items*" which are used to give more details about the input values expected by the algorithm.
items	Represents the array of "item"s.
item	Describes the single value to be given to the algorithm. It has "name", "*example_value*", "*datatype*", and "*description*".
name	This is a Placeholder
example_value	This is used to build an example input automatically
datatype	Describes the **datatype** of input that is expected by algorithm

Running JAR Files

AJAN Editor is developed with *Ember JS*, which is a **JavaScript framework**. Since Ember JS doesn't support **running jar files** or executing any commands, MAJAN executes the jar files via **Java**. To do so, the Executionservice module in AJAN Service provides a class name **MajanService** which has an endpoint (*ajan_service_base_path+"/majanService/runJar"*) that accepts the *input for the algorithm*, the *path of the jar file, and timeout*. Then, the **jar file** in the given path is executed with **Java** code. More specifically, this endpoint consumes **text/plain** and produces **application/json**. Moreover, it requires two headers called "*jarPath*" and "*timeout*" for the reasons mentioned above. Finally, the payload that is sent to the endpoint is passed as the input to the algorithm.

It is important to note that central algorithms must produce a **JSON** text describing a **grouping**. In order to produce such a JSON for a grouping result, **MAJAN provides** the necessary java module in "MAJAN/Grouping Algorithms/GroupingResultAsJson" folder in MAJAN GitHub repository.

Once MAJAN receives the **grouping result** as a JSON, it **parses** and **displays** the result in a user-friendly format, as shown in figure 54-d.

Compare Solutions

The **objective** of **Compare Solutions** panel (*figure 56*) in MAJAN is to allow users to *compare selected MAC and Central solutions* in terms of their **runtime** and **similarity of groups**.

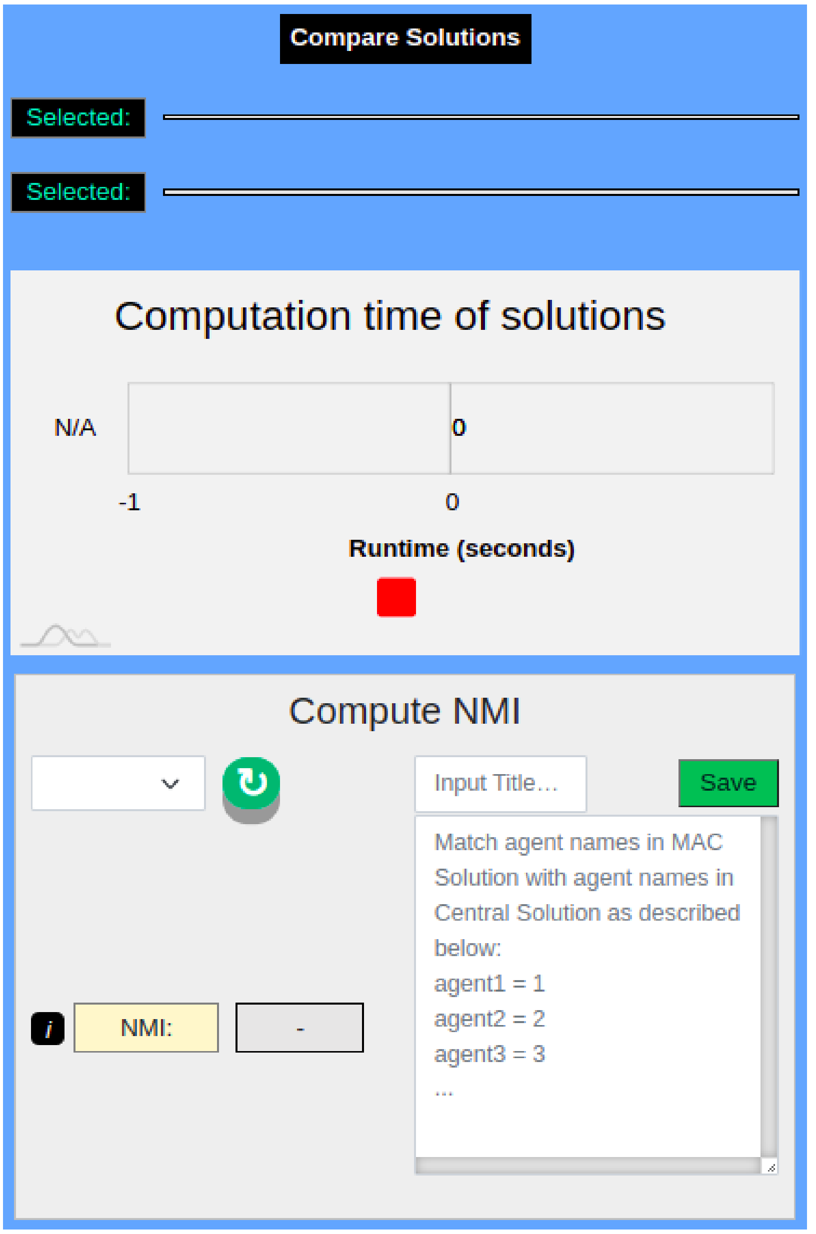


Figure 56. Compare Solutions panel in MAJAN

To do so, a MAC and Central solution should be selected. Let’s select the **first solutions of MAC and Central** that we executed in previous sections (*figure 57-a)*. Additionally, the **runtime** of selected solutions are in the chart shown *in figure 57-a*.

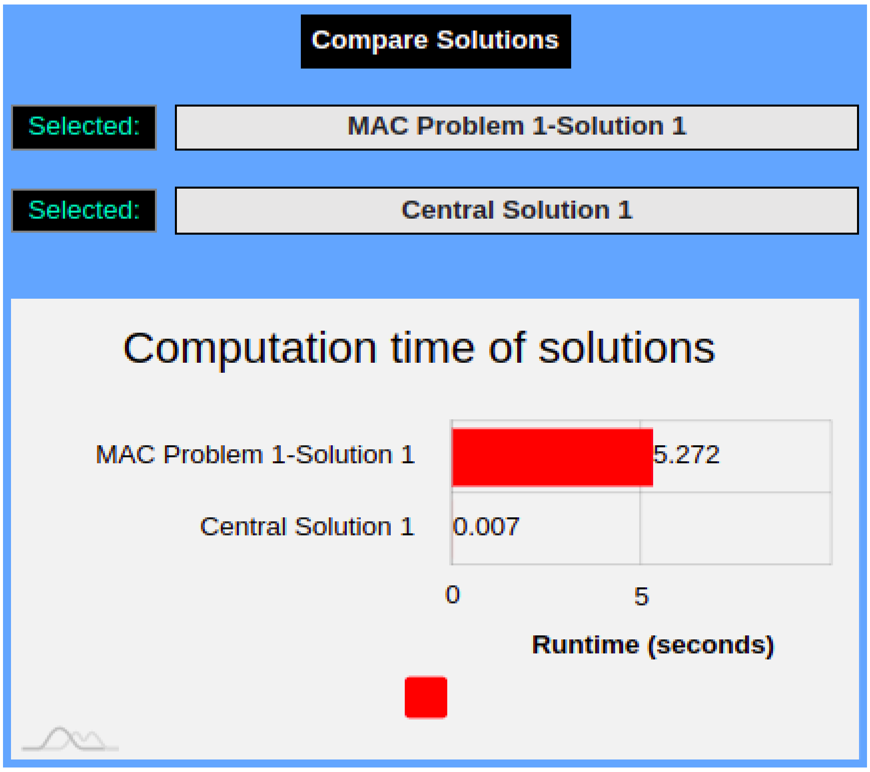


Figure 57-a. Visualization of selected solutions in Compare Solutions section

NMI Score

Overview

In order to compute the similarity of selected grouping solutions, MAJAN supports computing Normalized Mutual Information (NMI) scores. NMI is a standard clustering evaluation metric that allows computing the **similarity between two clustering results**. NMI score ranges between 0 (**no similarity**) and 1 (**maximum similarity**).

Compute NMI

Step 1. Since agents can take various names in MAC, it is necessary to provide a **list** that describes the **correct mapping** between **agent names for MAC** and **Central solutions.** Add a list as shown in *figure 57-b*.

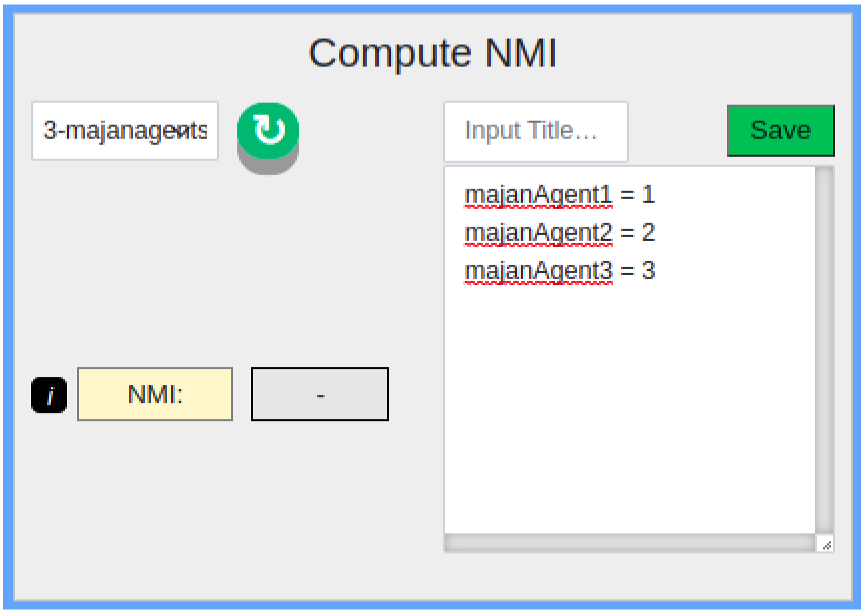


Figure 57-b. List of names of MAC agents corresponding to Central agents

For example, we created 3 agents with the names majanAgent1, majanAgent2, and majanAgent3 to run the MAC process. In the **Central solution**, agents are named 1, 2, and 3. It is necessary to let MAJAN know which agent in the MAC solution corresponds to which agent in the Central solution. Only then, MAJAN can compare the grouping solutions. Therefore, MAJAN requires a list to be able to map MAC agents to Central agents. An example is given below:

<Agent Name in MAC solution> = <Agent Name in Central Solution>

```
majanAgent1 = 1
majanAgent2 = 2
majanAgent3 = 3
```

Step 2 (final). Add a title in the "*Input Title"* field. And click on **Save** button (*figure 57-b*).

Examples

Since we have seen how to **run** the **MAC process**, **view its result** in MAJAN, **run a Central** algorithm, **view its result**, and **compare the selected solutions**, let's take a look at the comparison of some solutions in the figures below.

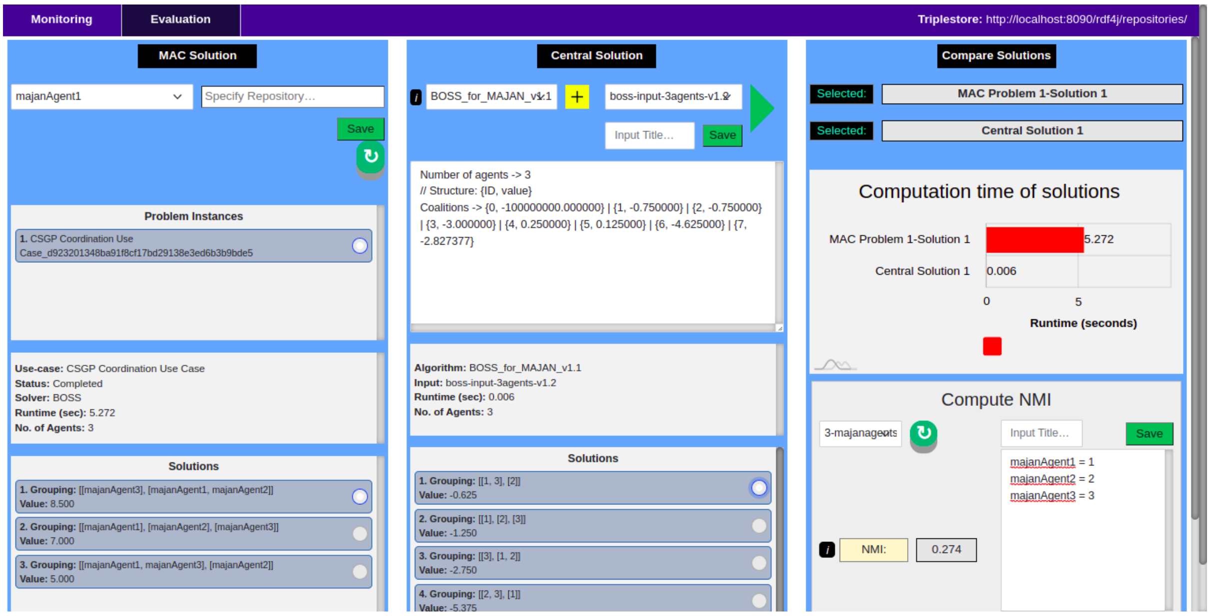


Figure 58-a. <u>First solutions</u> in MAC and Central Solution sections are selected and their NMI score is 0.274.

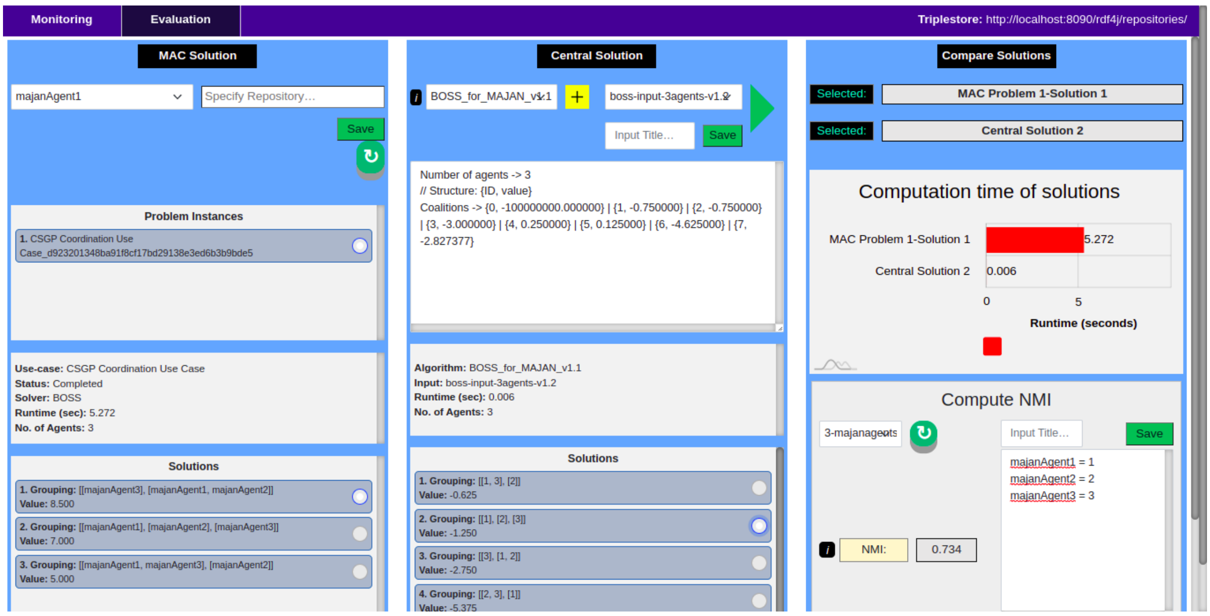


Figure 58-b. <u>First solution</u> of MAC and <u>second solution</u> of Central are selected and their **NMI score** is 0.734.

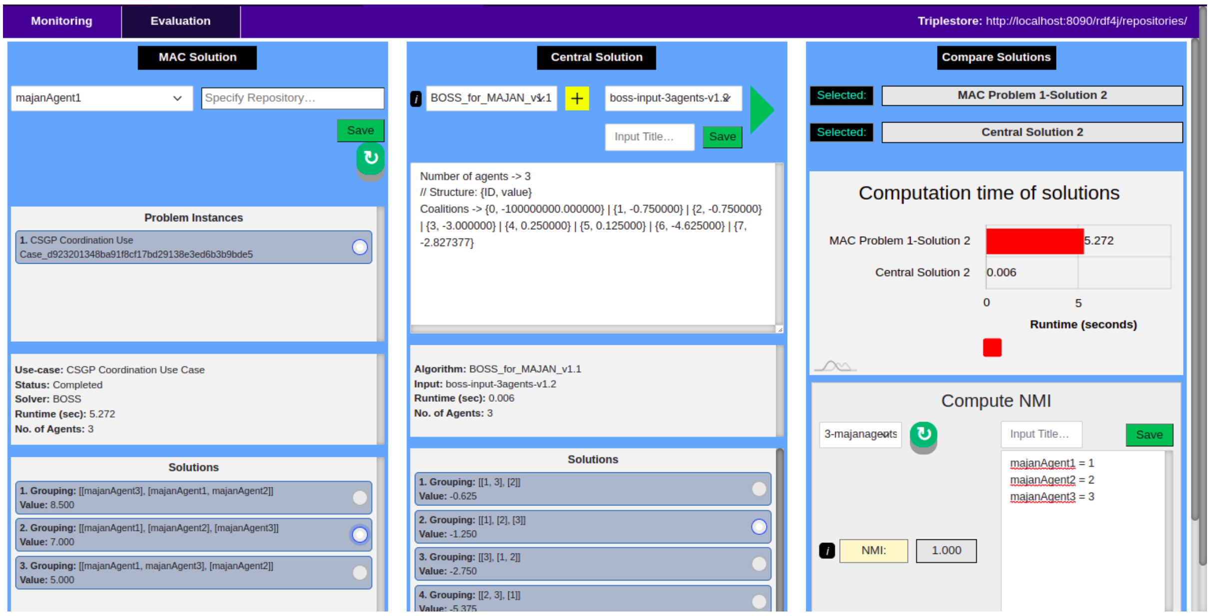


Figure 58-c. <u>Second solution</u> of MAC and <u>second solution</u> of Central are selected and their **NMI score** is 1.000.

MAJAN Extra

In this section, we provide additional information based on our experience which might be useful for users when using MAJAN, designing MAC behaviors of agents, or running MAC processes.

Useful Tips

1. **SPARQL Validator**

When designing behaviors of agents, sometimes SPARQL queries might get large, and therefore, it could be difficult to easily check the validity of queries with eyes. For this reason, you can use this website (https://sparql-playground.sib.swiss/) to validate SPARQL queries.

2. **RDF Validator**

Moreover, you can validate RDF triples by using this website (http://rdfvalidator.mybluemix.net/).

3. **Large SPARQL Queries**

Sometimes it is necessary to run a SPARQL query in a repository to check if it works correctly before starting to run agents. To do so, RDF4J provides a nice UI (figure 59-a) to run queries, add/remove triples to/from repositories, etc. However, RDF4J Workbench cannot run large queries, as shown in figure 59-b. In such a case, it is necessary to send queries to the SPARQL endpoint of a repository. To do so, you can use Postman. The Postman collection including some large SPARQL queries that have been executed is provided in "MAJAN/Postman Collections/RDF4J Sparql Endpoint/MAJAN - Large Sparql Queries.postman_collection.json".

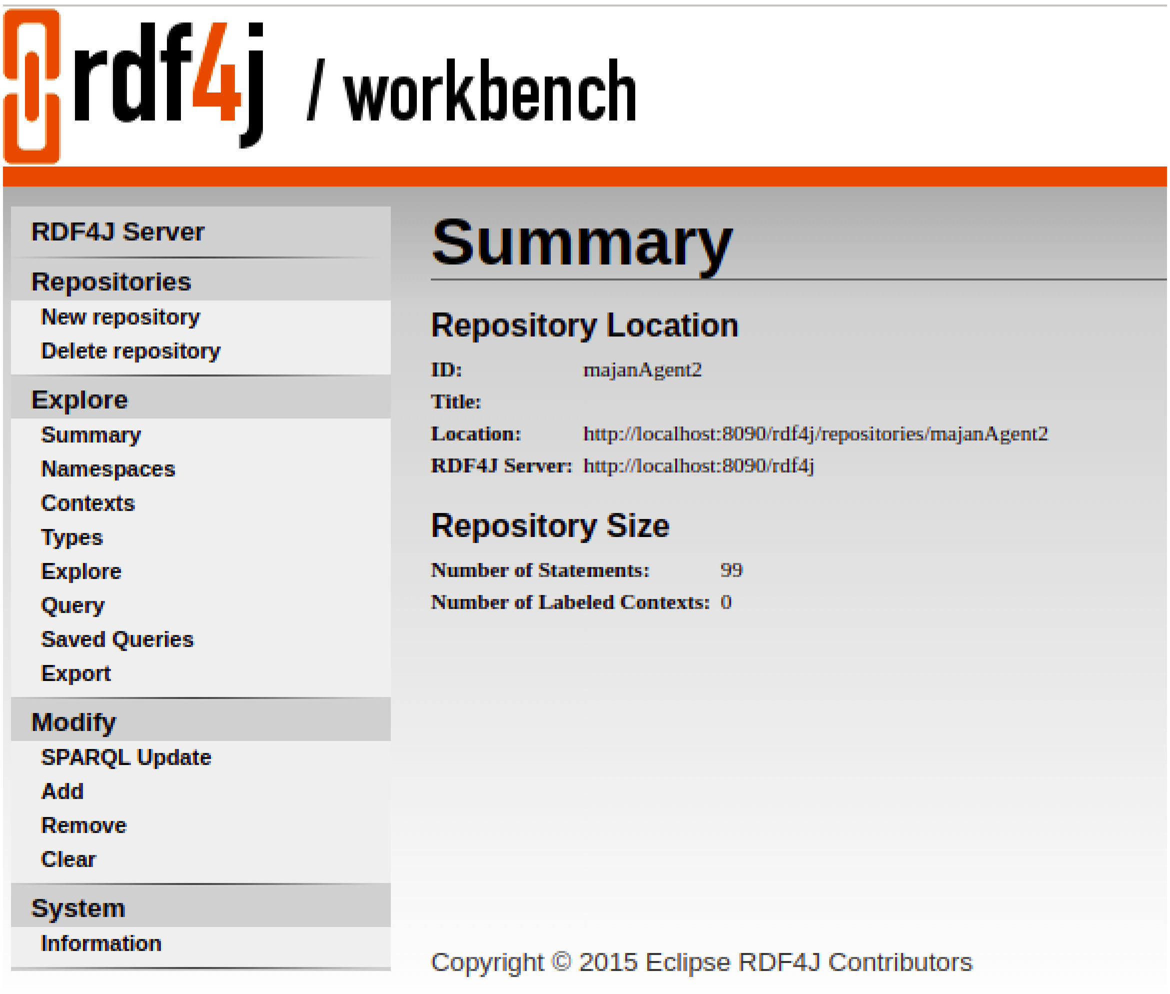


Figure 59-a. RDF4J Workbench UI with features on the left side.

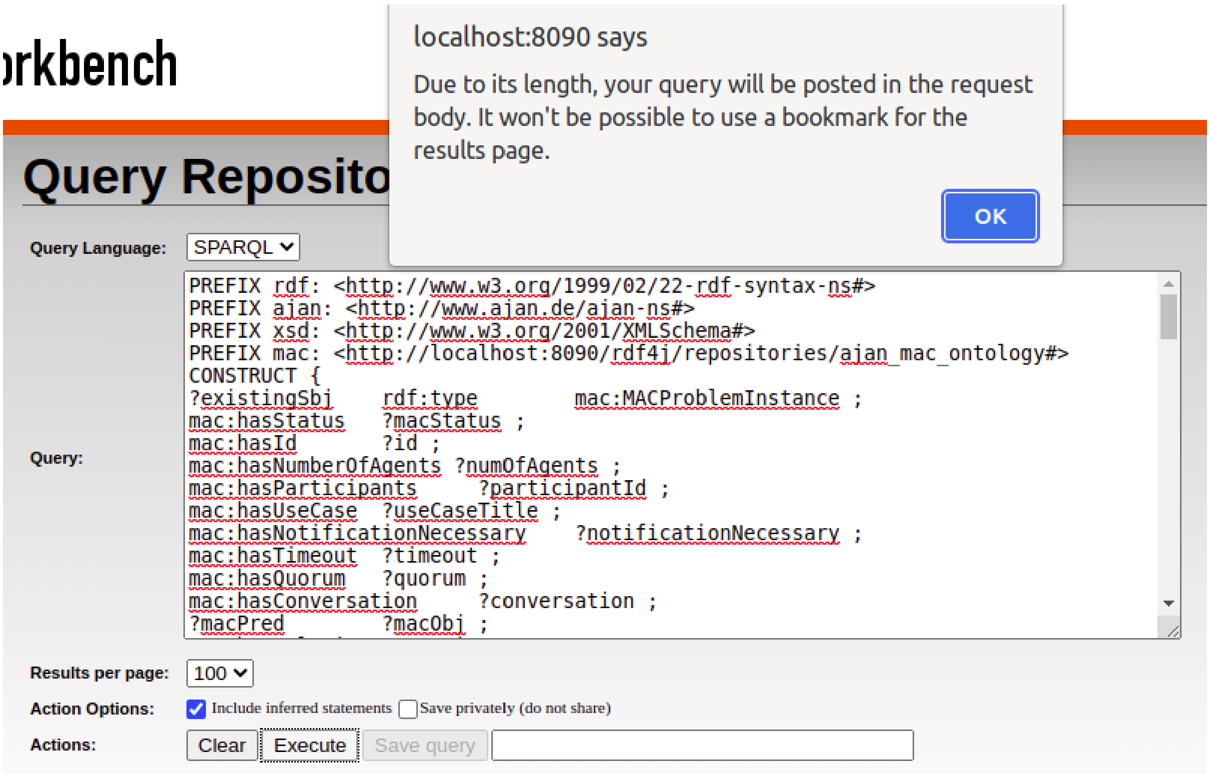


Figure 59-b. Error thrown when SPARQL query is large

4. Check triples in EKB

Furthermore, sometimes the result of the execution of a SPARQL query is unexpected. If the query is executed in Agent Knowledge (AKB), then we can simply execute the query manually to find out the problem. However, this is not the case when the query is executed in Execution Knowledge (EKB) since there is no physical repository in the RDF4J workbench for EKB. Therefore, we cannot know what is stored in EKB at the time of execution of the SPARQL query. To overcome this issue, you can write everything in EKB to any empty repository in RDF4J Workbench at any step of BT to be able to manually run the SPARQL. To do so, you can use **WriteEverythingToLSR** BT node as shown *in figure 59-c* as an example.

```
:WriteEverythingToLSR
        a bt:Write ;
        rdfs:label "Write everything to LSR" ;
        bt:query [
                a bt:ConstructQuery ;
                bt:originBase ajan:ExecutionKnowledge ;
                bt:targetBase ajan:LocalServicesKnowledge ;
                bt:sparql """
                                        CONSTRUCT {
```

```
                                    ?s ?p ?o .
                                }
                                WHERE {
                                    ?s ?p ?o .
                                }
            """^^xsd:string ;
    ] .
```

Grouping Algorithms Source Code

As mentioned before, MAJAN provides algorithms to be executed in the **Central Solution** section of MAJAN in AJAN Editor. In this section, we provide the location of the source code of those and other algorithms.

1. **BOSS** algorithm: source code is provided in "MAJAN/Grouping Algorithms/BOSS/BOSS_for_MAJAN(source_code)" folder in MAJAN GitHub repository.

2. **HDBSCAN** algorithm: source code is provided in "MAJAN/Grouping Algorithms/HDBSCAN/HDBSCAN_for_MAJAN(source_code)" folder.

3. **LCC_BOSS** algorithm: source code is provided in "MAJAN/Grouping Algorithms/LCC_BOSS/LCC_BOSS_for_MAJAN(source_code)" folder.

4. **CHC_HDBSCAN** algorithm: source code is provided in "MAJAN/Grouping Algorithms/CHC_HDBSCAN/CHC_HDBSCAN_for_MAJAN(source_code)" folder.

5. **CoalitionGenerator** algorithm to generate coalitions for CSGP solvers: source code is provided in "MAJAN/Grouping Algorithms/CoalitionGenerator(source_code)" folder. This algorithm is necessary to generate coalitions and compute coalition values such that they can be passed to BOSS or any other CSGP solver algorithm. That is because CSGP solver algorithms are generic, use-case independent, and they expect only the coalitions and their values. They don't compute coalition values. Because computation of coalition values are completely use-case dependent. That is why, coalition generator algorithm must be modified and adopted for the use-case at hand. And then, Coalition Generator algorithm can compute coalitions and their values which can be passed to CSGP solver algorithms. In multiagent coordination protocols, agents compute use-case dependent coalition values with SPARQL.

6. **GroupingResultAsJson** module to describe Grouping Solutions in JSON format: source code is provided in "MAJAN/Grouping Algorithms/GroupingResultAsJson(source_code)" folder.

Possible Errors and Their Solutions

1. **QueueEvent instead of Event**

When designing behaviors for MAC, it is very important to pay attention to the type of events. In most cases, agents should be able to handle the same events multiple times in parallel. For example, agents should be able to run multiple MAC processes in parallel. In order to make BTs be able to handle multiple events, they should use QueueEvent instead of Event. Otherwise, agents will not be able to run correctly when a certain event tries to trigger its BT multiple times in parallel.

2. **Blank nodes**

Moreover, sometimes behaviors of agents don't work correctly when using blank nodes created with the BNODE() function of SPARQL. In such a case, you can create new IRIs by using the lines as shown *in figure 59-d*.

```
{ BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
  BIND( IRI(CONCAT(STR(mac:AgentPreferences), STR(?uniqueId))) AS ?newNode )}
```

Figure 59-d. SPARQL to create new IRI instead of using BNODE().

Justifications

1. **Why MAC BTs use EKB instead of AKB?**

Because agents should be able to run multiple MAC processes in parallel. If all MAC Problem Instances would be handled completely in AKB, then it wouldn't be possible to differentiate them. Because there would be multiple MAC instances running at the same time in AKB and any BT that needs to manipulate a running MAC problem instance, it couldn't know which MAC problem it should manipulate. However, by using EKB, this problem is resolved. Because there can be only one MAC problem instance in each EKB that is running. This issue can apply to other info or use cases. Thus, this is a solution to overcome the issue explained above.

2. **Why not use variables for everything in Postman collections?**

Because it is not possible to dynamically specify the values in JSON. E.g. 1 variable specifies a single value. Let's say an agent has a "**canSpeak**" predicate, which stores the language that the agent can speak. Different agents can speak different amounts of languages, and it is not possible to know and manage this dynamically beforehand. If we specified 3 language variables in postman and if the agent can speak 2 languages, then the last variable will be sent as well even though it is an empty string. And if the agent can speak more than 3 languages, there won't be any way to specify all of them in a JSON configuration file dynamically.

References

Campello, R., & Moulavi, D. (2015). Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. *ACM Transactions on Knowledge Discovery from Data*, *10*(1), 1-51. https://dl.acm.org/doi/10.1145/2733381

Changder, N., & Aknine, S. (2021). BOSS: A Bi-directional Search Technique for Optimal Coalition Structure Generation with Minimal Overlapping (Student Abstract). *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(Vol. 35 No. 18: AAAI-21 Student Papers and Demonstrations), 0-100. https://ojs.aaai.org/index.php/AAAI/article/view/17879

Hartigan, J. (1975). *Clustering Algorithms*. The University of Michigan. https://books.google.az/books/about/Clustering_Algorithms.html?id=cDnvAAAAMAAJ&redir_esc=y

Prabhakar, S. (2017). *Reciprocal Recommender System for Learners in Massive Open Online Courses (MOOCs)*. researchgate. http://dx.doi.org/10.1007/978-3-319-66733-1_17

Rahwan, T., & Michalak, T. (2015). Coalition structure generation: A survey. *Artificial Intelligence*, *229*(ISSN 0004-3702), 139-174. https://doi.org/10.1016/j.artint.2015.08.004.