AAS Type3 - System Architecture

Ahmed Ibrahim Almohamed

October 24, 2025

Abstract

This document provides a comprehensive overview of the system architecture for the AAS Type3 project. It details the system components, their interactions, and the overall design principles guiding the architecture.

Contents

1 System Overview

1.1 System Concept

After reviewing the requirements and doing the necessary research, a established simple use case was created to illustrate the system concept as in figure 1.

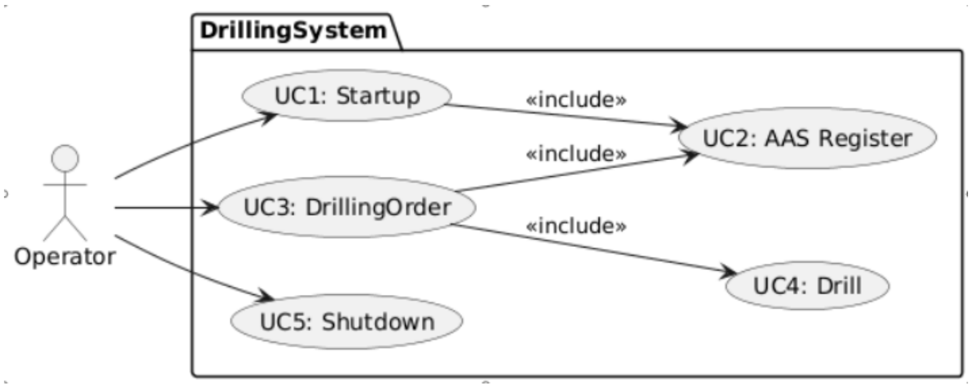


Figure 1: System Use Case

From the use case and the requirements, the system concept is established where it holds two main components:

- **An Asset Administration Shell (AAS)**
- **A Run time Environment (RTE) with Multiagent System (MAS)**

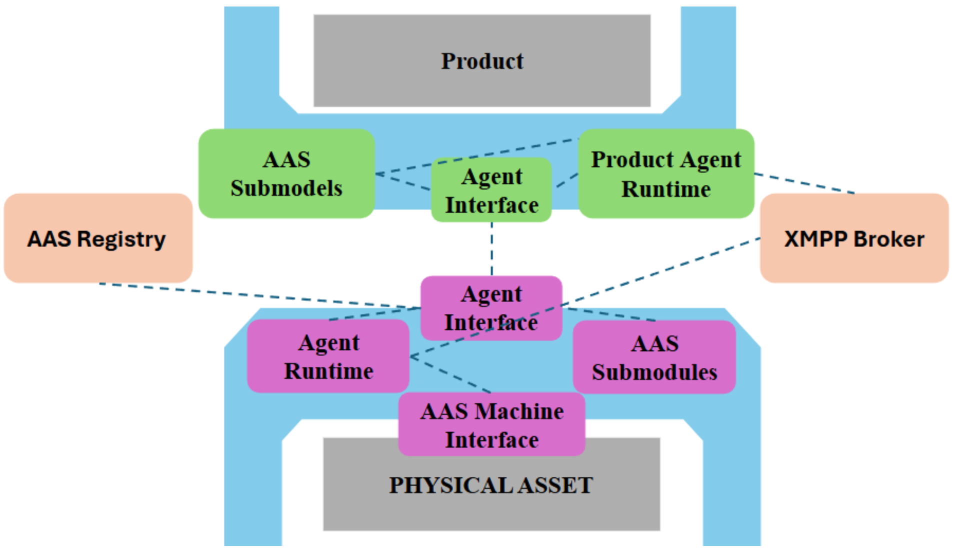


Figure 2: System Concept

The Concept illustrated in figure 2 shows the two main parts , the Product or Production AAS and the Resource AAS.

1.2 Resource AAS - Overview

The Resource AAS is the component that describes a resource in the System, where a Resource in the system can vary from a simple machine to a worker. The Resource AAS contains the following components:

- **AAS Submodels** that describe the resource properties and capabilities.
- **Agent Runtime** that contains the agent that represents the resource in the system.
- **AgentInterface** that connects the resource Agent to other agents in the system.
- **Agent Machine Interface** that connects the resource Agent to the machine or worker it represents.

The Submodels of the Resource AAS contains the following components as in table 1.

Submodel	**Description**
Operation State Submodel	Contains the current operational state of the resource and History.
Knowledge Submodel	Contains the knowledge of the resource constraints and the environmental conditions.
Capabilities Submodel	Describes the skills the asset can perform and the services it can provide.
Interaction and scheduling Submodel	Contains the Communication endpoints and the collaboration mechanisms.

Table 1: Resource Agent : Submodels

The submodels then could provide the necessary information for the Resource Agent to perform its tasks in the system. The Resource Agent then connects to other agents in the system through the Agent Interface and to the machine. As shown in figure 3. the Belifes of the agent are updated from the operational state submodel and the knowledge submodel where they provided the necessary state and constraints for the agent. The Desires of the agent are established from the capabilities submodel where they provide the skills and services. Finaly the agent can act upon the desiers and excute its intention which the scheduling and interaction submodel can provide the necessary information for the agent to interact and schedule its tasks.

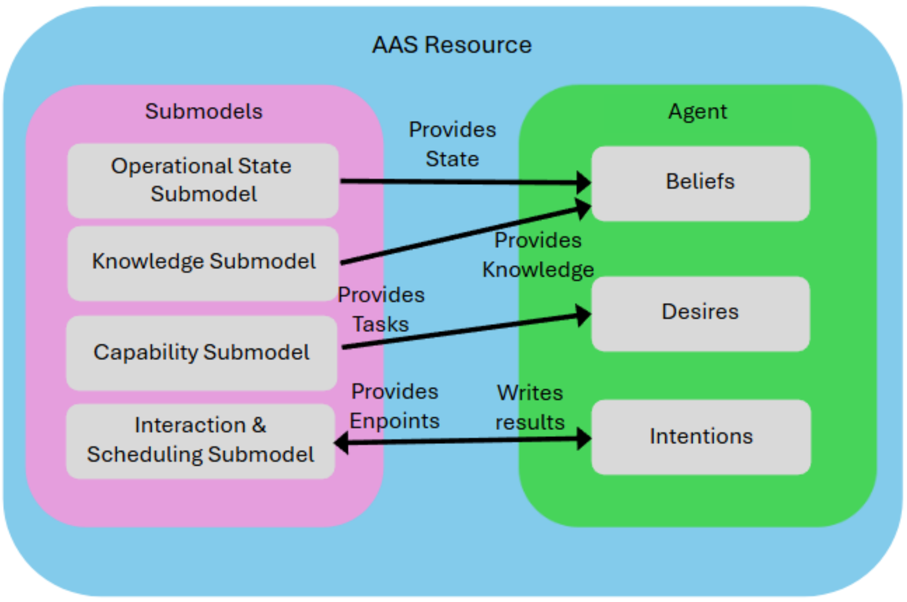


Figure 3: Resource AAS Overview

The Resource AAS can interact with other AASs (production or resource). It also capable of publishing events and subscribing to other AAS events. As shown in figure 4, the Resource AAS can interact with other AASs through via XMPP protocol which is a communication protocol for MAS frame work for python **SPADE**. XMPP can also be used as a broker for the AAS to publish and subscribe to events. Finaly the Resource AAS can interact with the Asset via a custom Interface (OPC UA, REST API ...etc). The communication interfaces and their usage are summarized in table 2.

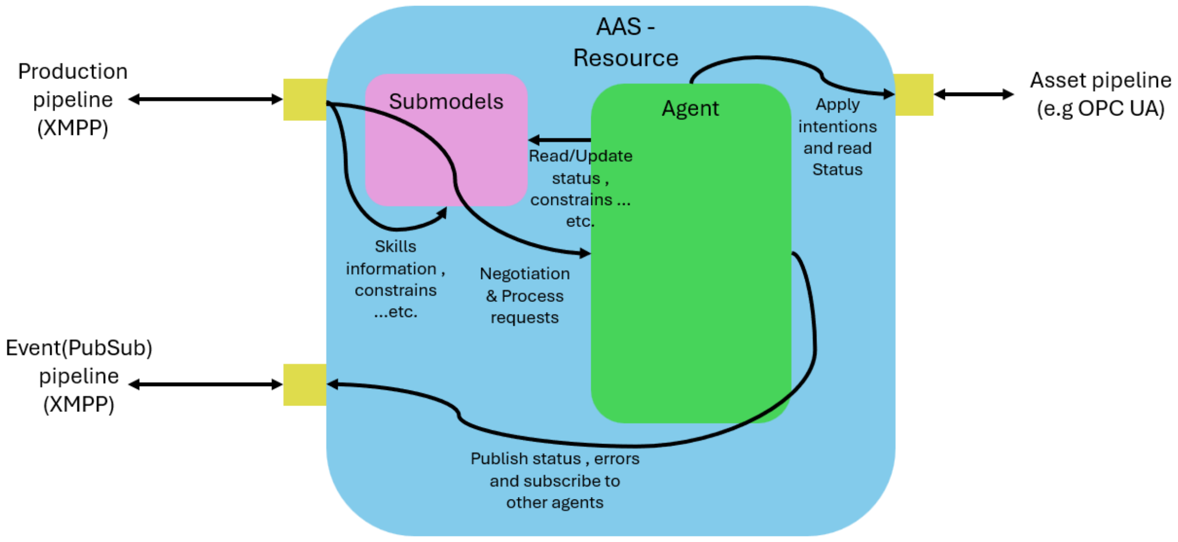

Production pipeline (XMPP)
Event(PubSub) pipeline (XMPP)
AAS - Resource
Submodels
Agent
Asset pipeline (e.g OPC UA)
Apply intentions and read Status
Read/Update status , constrains ... etc.
Skills information , constrains ...etc.
Negotiation & Process requests
Publish status , errors and subscribe to other agents

Figure 4: Resource AAS Interactions - Overview

Communication Interface	**Usage**
XMPP	Agent-to-agent communication within the MAS framework.
XMPP - Events	Publishing and subscribing to events between AASs.
Asset Interface (e.g OPC UA)	Interfacing with industrial assets and machines for data exchange and control.

Table 2: Communication Interfaces and Their Usage

Internally the Submodels are connected to the agent where they could be read and updated. The Submodels can also provide the production agents with the available services and skills the resource can provide and the state of the resource. The Agent runtime can connect to the other agents in two ways:

- **Direct Communication** where the agents can communicate directly with each other and hold negotiations.
- **Event-based Communication** where the agents can publish and subscribe to events.

Also it handles the execution of the tasks on the machine or worker it represents.

1.2.1 Resource AAS - Architecture Overview

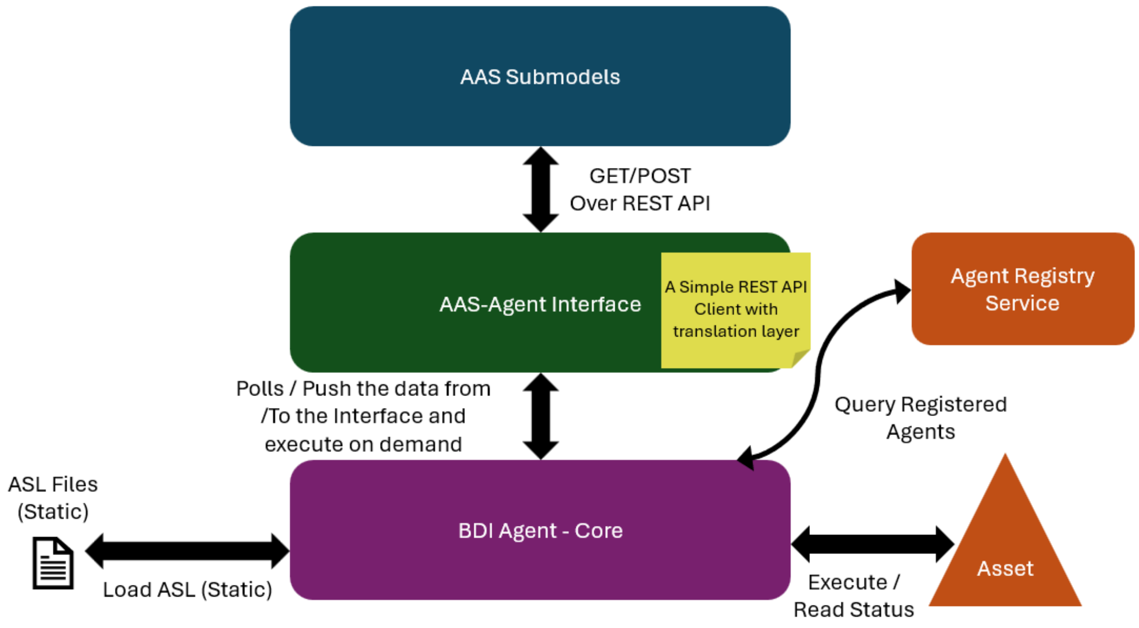


Figure 5: Resource AAS Architecture Overview

From an Architecture point of view the Resource AAS can be illustrated as in figure 5. The overview of the architecture shows three main layers, The **AAS Submodels** , **The AAS - Agebnt Interface** and Finaly **BDI Agent Core**. The AAS Submodels layer are created via **Basyx** framework which is an open source framework for creating AASs. This framework provides the necessary tools to create and manage AASs and their submodels. It also can provide a REST API for the AAS to be accessed and managed. The AAS - Agent Interface layer is responsible for connecting the AAS to the Agent Core. It mainly work as a middleman between the AAS and the Agent Core where it translate the data types of from and to the AAS and the Agent Core. Finaly The BDI Agent Core is the main component of the Resource AAS where it contains the BDI agent that represents the resource in the system. The BDI agent is created via **SPADE** framework which is an open source framework for creating Multi-agent systems in python. The BDI agent is responsible for managing the beliefs, desires and intentions of the agent. It also handles the communication with other agents in the system via XMPP protocol. The BDI agent can also execute the tasks on the machine or worker it represents. The BDI agent also can register itself with a Agent Registry service that the production Agents use for discovering the available resources in the system.

1.2.1.1 Submodels with REST API and server

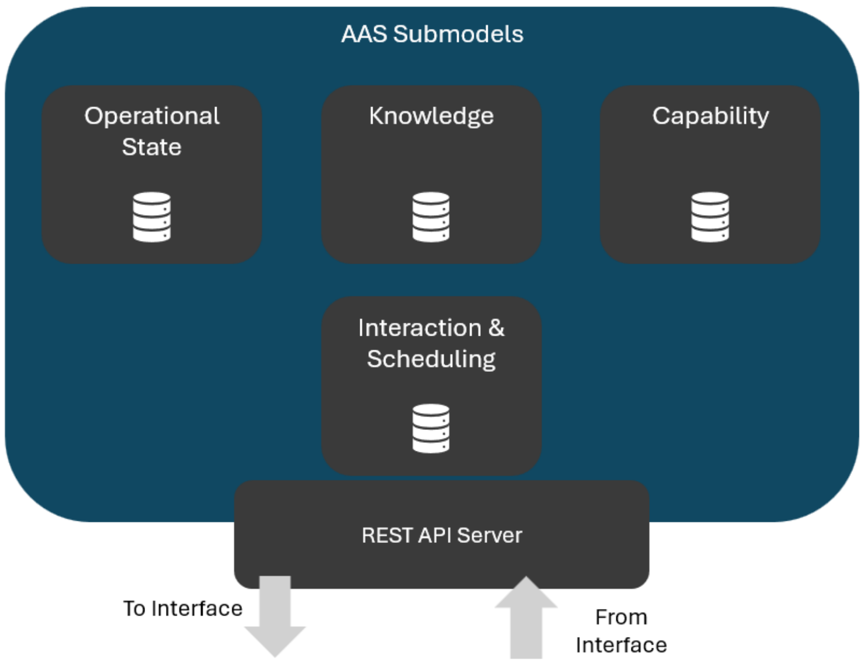


Figure 6: Resource AAS Submodels with REST API and Server

The Submodels layer as shown in figure 6 contains the AAS Submodels and a REST API server. The AAS Submodels are created via **Basyx** framework which is an open source framework for creating AASs. This framework provides the necessary tools to create and manage AASs and their submodels. It also can provide a REST API for the AAS to be accessed and managed. The Submodels connects to the AAS-Agent Interface via REST API calls where the AAS-Agent Interface can read and update the submodels.

1.2.1.2 AAS-Agent Interface

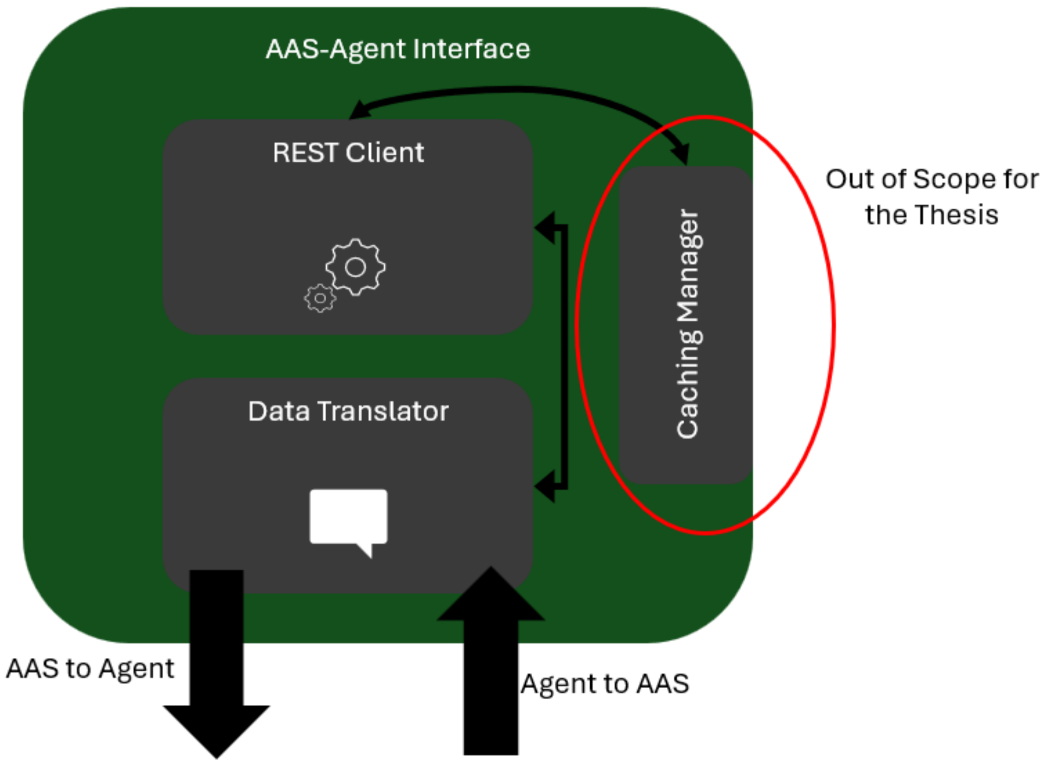


Figure 7: Resource AAS-Agent Interface

As in figure 7 the AAS-Agent Interface is responsible for connecting the AAS to the Agent Core. It mainly work as a middleman between the AAS and the Agent Core where it translate the data types of from and to the AAS and the Agent Core. The AAS-Agent Interface contains the following components:

- **REST Client** that connects to the AAS via REST API calls.
- **Data Translator** that translates the data types from and to the AAS and the Agent Core.

1.2.1.3 BDI Agent Core

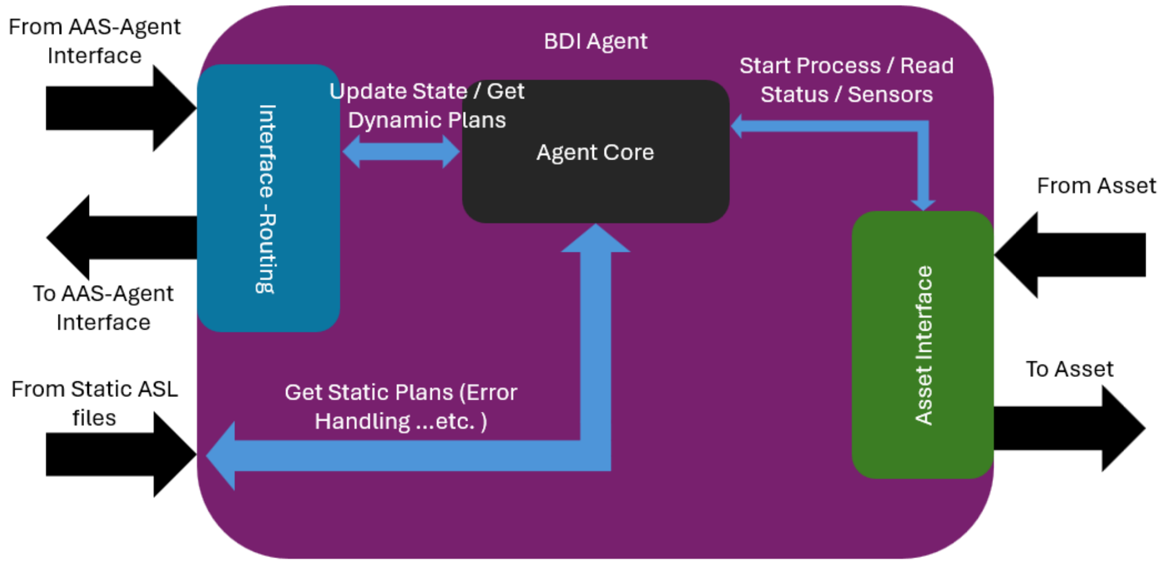


Figure 8: Resource AAS - BDI Agent Core

As in figure 8 the BDI Agent Core is the main component of the Resource AAS where it contains the BDI agent that represents the resource in the system. The BDI agent is created via **SPADE** framework which is an open source framework for creating Multi-agent systems in python. The BDI agent is responsible for managing the beliefs, desires and intentions of the agent. It also handles the communication with other agents in the system via XMPP protocol. The BDI agent can also execute the tasks on the machine or worker it represents. The BDI agent also can register itself with a Agent Registry service that the production Agents use for discovering the available resources in the system. The BDI agent contains the following components:

- **Agent Core** that contains the BDI agent and its components.
- **Asset Interface** that connects the agent to the machine or worker it represents.

The Agent Core connects with the AAS-Agent Interface so that it can read and update the AAS Submodels. The Agent Core also connects to other agents in the system via XMPP protocol. The Agent Core can delegate tasks to the Asset Interface so that it can execute the tasks on the machine or worker it represents. The Asset Interface connects to the machine or worker via a custom interface (e.g OPC UA, REST API ...etc).

1.3 Production AAS - Overview

The Production AAS is the component that describes a product in the System, where a Product in the system can vary from a simple product to a complex assembly. The Product AAS contains the following components:

- **AAS Submodels** that describe the product properties and capabilities
- **Agent Runtime** that contains the agent that represents the product in the system.
- **AgentInterface** that connects the product Agent to other agents in

The Submodels of the Product AAS contains the following components as in table 3.

Submodel	**Description**
Identification Submodel	Contains information about the product ID , version , Owner and Product context
Process plan	Contains information about the following : • Nodes (as task to be performed on the product) • Edges (as the flow between the tasks) • Entry and Exit points • Required Inputs and produced Outputs • Required Capabilities and Skills to perform the tasks • Pre and post conditions for each task
Services Catalog	Contains information about the services offered by the product, including their capabilities and requirements.
Interface and Endpoints	Contains information about the communication endpoints and protocols used by the product.
Execution state and Tracking	Contains information about the current state of the product and its history.

Table 3: Product AAS Submodels

The submodels then could provide the necessary information for the Product Agent to perform its tasks in the system. The Product Agent contains an Agent Runtime element that is divied into two cores :

- **The Orchestration Core** that is responsible for managing the overall production process and coordinating the resources.
- **The Negotiation Core** that is responsible for negotiating with the Resource Agents to allocate the necessary resources for the production process.

As shown in figure 9. The Submodels provide the necessary information for the Orchestration Core to manage the production process. The Orchestration Core gets the ID ,process plan and the Excution state from the Submodels. It then uses this information to manage the production process and coordinate the resources. The Orchestration Core then uses the Negotiation Core to negotiate with the Resource Agents to allocate the necessary resources for the production process. The Negotiation Core then uses the Agent Interface to communicate with the Resource Agents and other Product Agents in the system. The Submodels also provide the necessary information for the Negotiation Core to negotiate with the Resource Agents. The Negotiation Core gets the Services Catalog and the Interface , Endpoints from the Submodels and the ID.

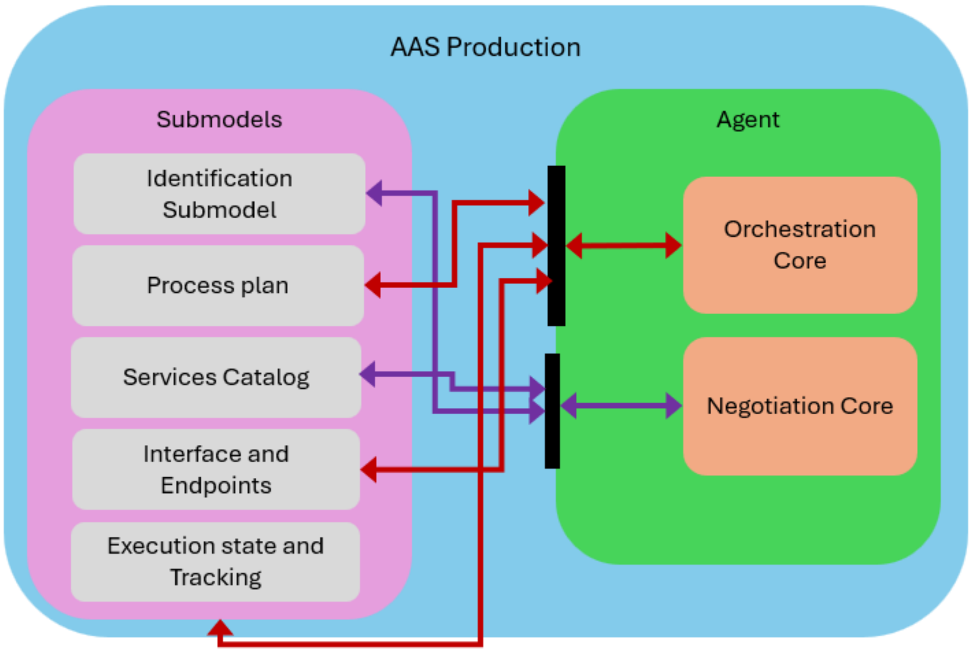


Figure 9: Product AAS Overview

The Product AAS can interact with other AASs (production or resource). It also capable of publishing events and subscribing to other AAS events. As shown in figure 10, the Product AAS can interact with other AASs through via XMPP protocol which is a communication protocol for MAS frame work for python **SPADE**. XMPP can also be used as a broker for the AAS to publish and subscribe to events. The communication

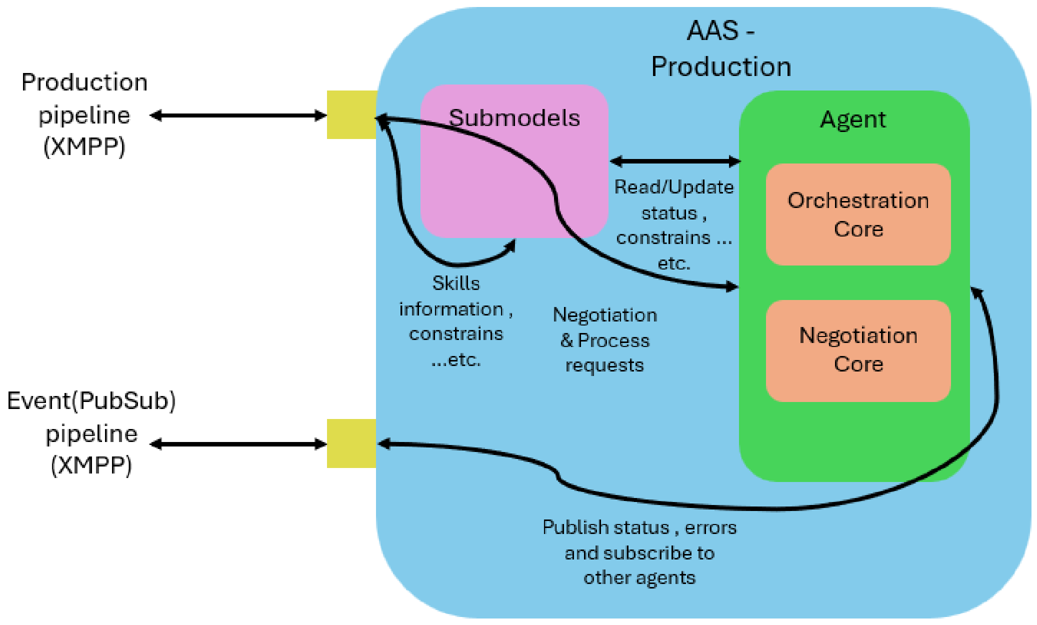

AAS - Production
Production pipeline (XMPP)
Submodels
Agent
Read/Update status , constrains ... etc.
Orchestration Core
Skills information , constrains ...etc.
Negotiation & Process requests
Negotiation Core
Event(PubSub) pipeline (XMPP)
Publish status , errors and subscribe to other agents

Figure 10: Product AAS Interactions - Overview

interfaces and their usage are summarized in table 4.

Communication Interface	**Usage**
XMPP	Agent-to-agent communication within the MAS framework.
XMPP - Events	Publishing and subscribing to events between AASs.

Table 4: Communication Interfaces and Their Usage - Product AAS

Internally the Submodels are connected to the agent where they could be read and updated. The Submodels can also provide the production agents with the process plan and the required services and skills to perform the production process. The Orchestration Core can manage the production process and coordinate the resources. The Negotiation Core can communicate with other agents in the system via XMPP protocol. The Negotiation Core can also register itself with a Agent Registry service that the Resource Agents use for discovering the available products in the system.

1.3.1 Product AAS - Architecture Overview

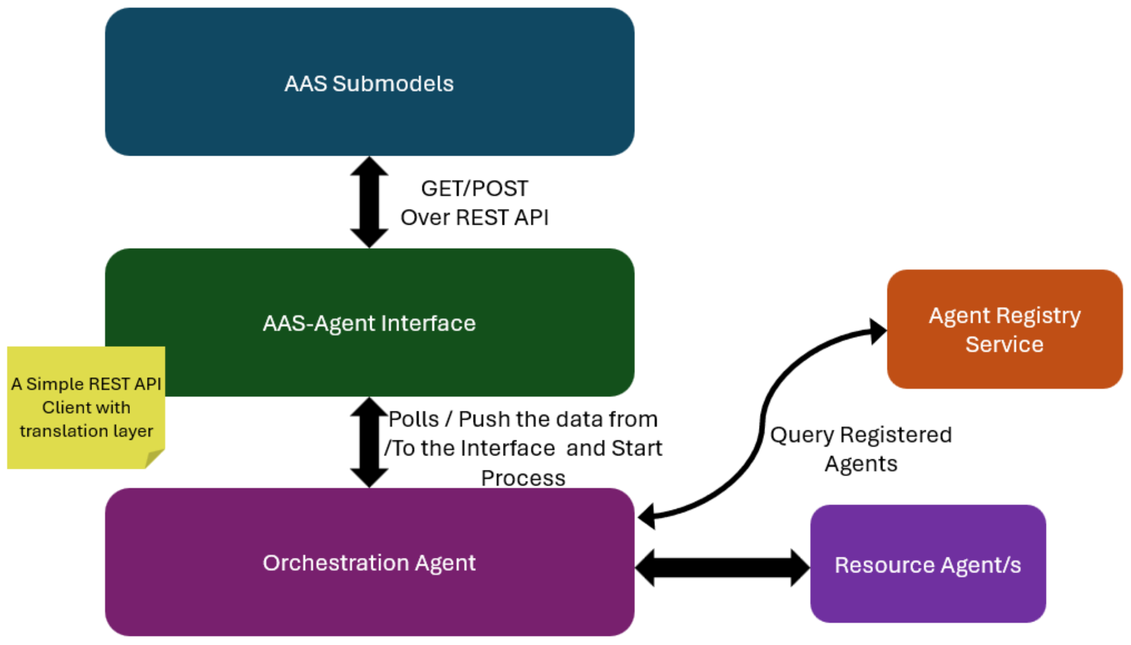


Figure 11: Product AAS Architecture Overview

From an Architecture point of view the Product AAS can be illustrated as in figure 11. The overview of the architecture shows three main layers, The **AAS Submodels** , **The AAS - Agent Interface** and Finally **Orchestration Agent**. The AAS Submodels layer are created via **Basyx** framework which is an open source framework for creating AASs. This framework provides the necessary tools to create and manage AASs and their submodels. It also can provide a REST API for the AAS to be accessed and managed. The AAS - Agent Interface layer is responsible for connecting the AAS to the Orchestration Agent. It mainly work as a middleman between the AAS and the Orchestration Agent where it translate the data types of from and to the AAS and the Orchestration Agent. Finally The Orchestration Agent is the main component of the Product AAS where it contains the Orchestration Core and the Negotiation Core that represents the product in the system. The Orchestration Core and the Negotiation Core are created via **SPADE** framework which is an open source framework for creating Multi-agent systems in python. The Orchestration Core is responsible for managing the overall production process and coordinating the resources. The Negotiation Core is responsible for negotiating with the Resource Agents to allocate the necessary resources for the production process. The Orchestration Core and the Negotiation Core can also communicate with other agents in the system via XMPP protocol. The Orchestration Core and the Negotiation Core also can register itself with a Agent Registry service that the Resource Agents use for discovering the available products in the system.

1.3.1.1 Submodels with REST API and server

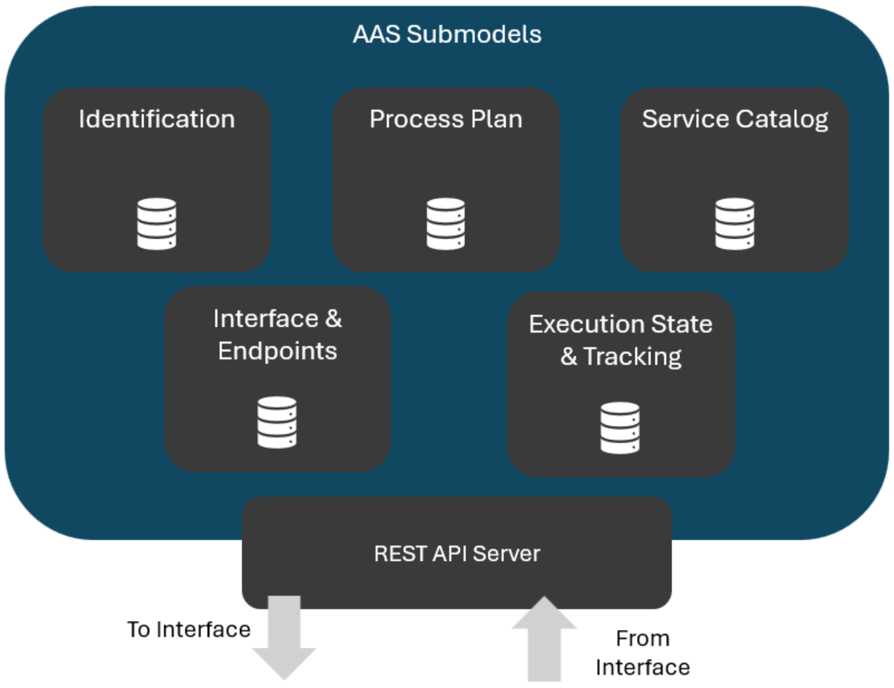


Figure 12: Product AAS Submodels with REST API and Server

Similar to the Resource AAS Submodels layer as shown in figure 12 contains the AAS Submodels and a REST API server. The AAS Submodels are created via **Basyx** framework which is an open source framework for creating AASs. This framework provides the necessary tools to create and manage AASs and their submodels. It also can provide a REST API for the AAS to be accessed and managed. The Submodels connects to the AAS-Agent Interface via REST API calls where the AAS-Agent Interface can read and update the submodels.

1.3.1.2 AAS-Agent Interface

Similar to the Resource AAS-Agent Interface as in figure 7 the Product AAS-Agent Interface is responsible for connecting the AAS to the Orchestration Agent. It mainly work as a middleman between the AAS and the Orchestration Agent where it translate the data types of from and to the AAS and the Orchestration Agent. The main diffrence to the Resource AAS-Agent Interface is that it connects to the Orchestration Agent instead of the BDI Agent Core which means the data model to translate is diffrent.

1.3.1.3 Orchestration Agent and Negotiation Core

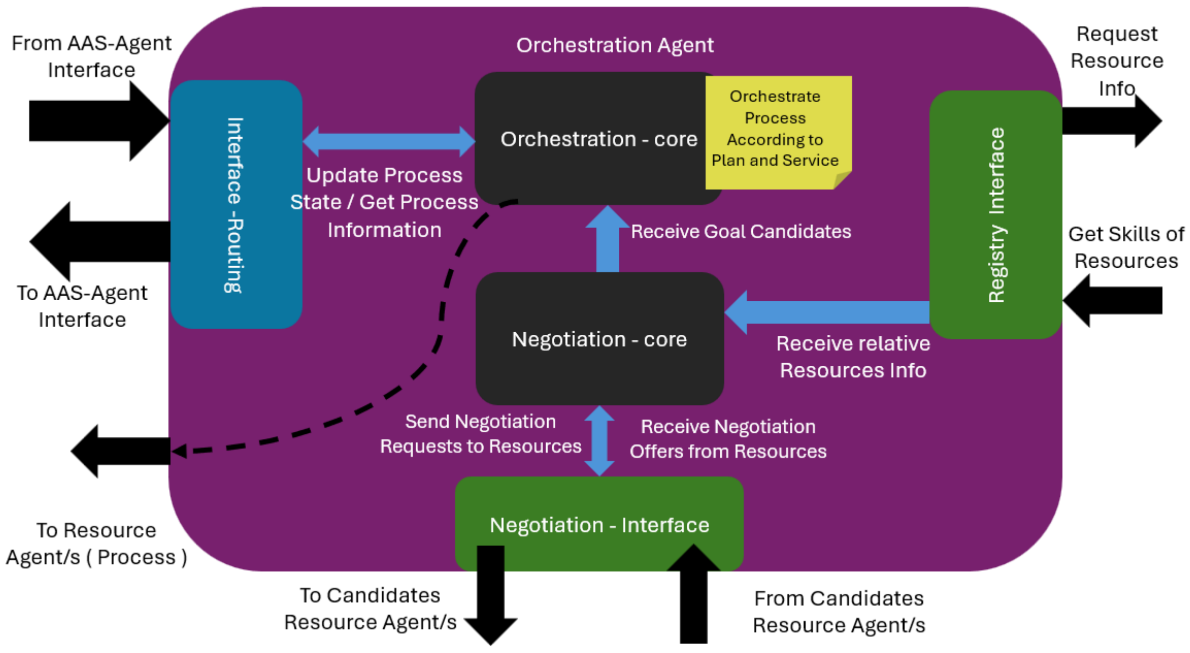

From AAS-Agent Interface
Orchestration Agent
Request Resource Info
Interface -Routing
Orchestration - core
Orchestrate Process According to Plan and Service
Registry Interface
Update Process State / Get Process Information
Receive Goal Candidates
Get Skills of Resources
To AAS-Agent Interface
Negotiation - core
Receive relative Resources Info
Send Negotiation Requests to Resources
Receive Negotiation Offers from Resources
To Resource Agent/s (Process)
Negotiation - Interface
To Candidates Resource Agent/s
From Candidates Resource Agent/s

Figure 13: Product AAS - Orchestration Agent and Negotiation Core

As in figure 13 the Orchestration Agent is the main component of the Product AAS where it contains the Orchestration Core and the Negotiation Core that represents the product in the system. The Orchestration Core and the Negotiation Core are created via **SPADE** framework which is an open source framework for creating Multi-agent systems in python. The Orchestration Core is responsible for managing the overall production process and coordinating the resources. The Negotiation Core is responsible for negotiating with the Resource Agents to allocate the necessary resources for the production process. The Orchestration Core and the Negotiation Core can also communicate with other agents in the system via XMPP protocol.

After landing a deal with the Resource Agents the Negotiation Core informs the Orchestration Core about the allocated resources. The Orchestration Core then manages the production process using the allocated resources. The Orchestration Core connects with the AAS-Agent Interface so that it can read and update the AAS Submodels. The

Orchestration Core also connects to other agents in the system via XMPP protocol. The Orchestration Core can delegate tasks to the Negotiation Core so that it can focus on managing the production process.

The Negotiation Core also connects to the Registry Service so that it can discover the available Resource Agents in the system. The Registry Service contains information about the available Resource Agents in the system including their capabilities and skills. The Negotiation Core uses this information to negotiate with the Resource Agents to allocate the necessary resources for the production process.

2 System Components

In this section, we delve into the various components that make up the AAS Type3 system architecture. Each component is described in terms of its functionality, interactions with other components, and its role within the overall system.

2.1 Resource Agent Components

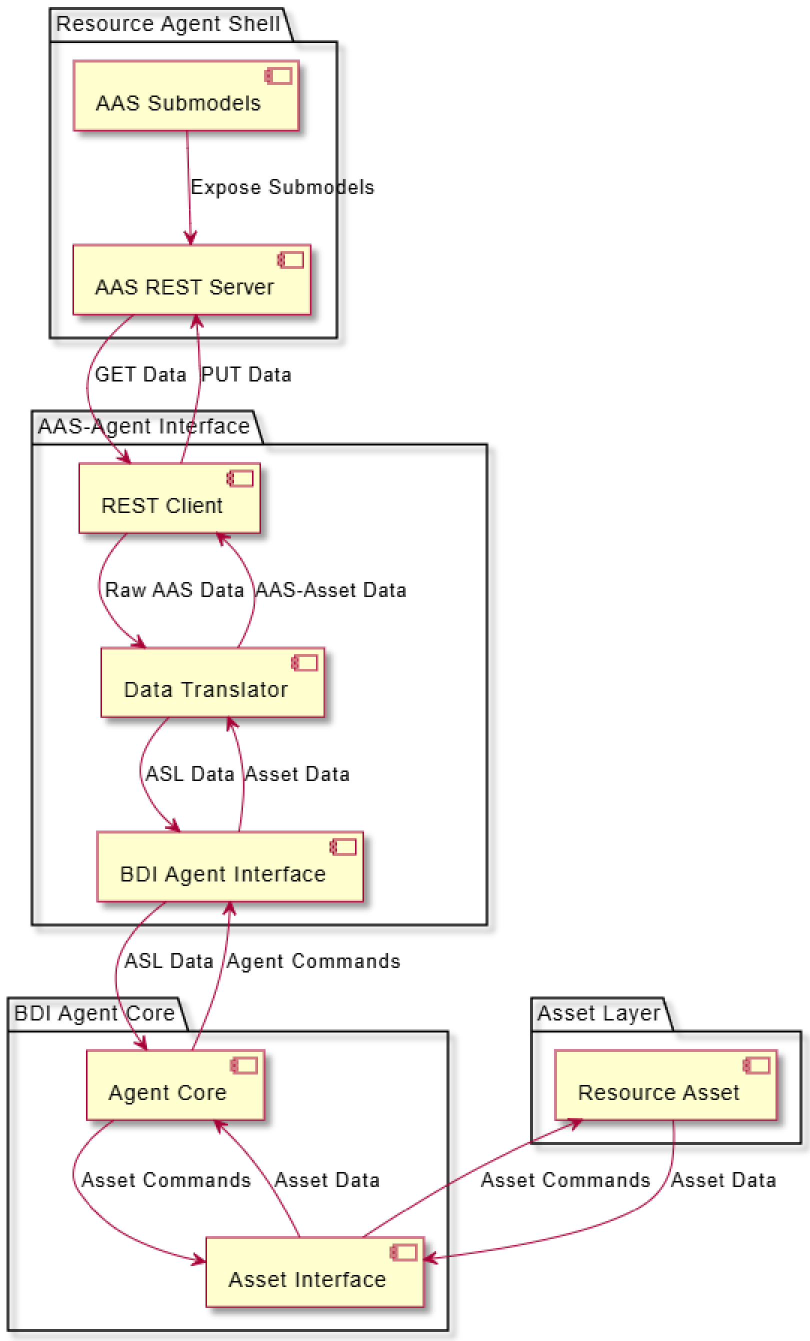


Figure 14: Resource Agent Components Diagram

As illustrated in Figure 14, the Resource Agent is composed of several key components over the layers that was discussed in the previous section. In the first layer that is the

Resource Agent Shell, it contains the following components:

- **AAS Submodels** : those are the submodels of the AAS that describe the resource properties and capabilities. in which a Java server is used to host the AAS and provide access to its submodels. The server stores the AAS data into a database MongoDB for efficient retrieval and management. The AAS are mainly Json files that follow the AAS specification.

- **REST API server** : this component provides a RESTful interface for communication between the AAS and other system components. The Basyx framework provides this REST API server that allows clients to interact with the AAS submodels. The REST API server is a part of the Java server that hosts the AAS. It also opens MQTT communication channels for pub/sub messaging for real-time data exchange on update , create and delete operations on the AAS submodels.

In the second layer that is the **AAS-Agent Interface layer**. This layer as discussed before is responsible for the communication between the AAS and the Agent. The layer contains three main components:

- **REST Client** : This is the main layer for communication between the Agent and the AAS. Which is responsible for sending requests to the AAS REST API server and receiving responses. This client shall allow the Agent to read , write and deleted information about the asset, submodels and the submodels elements.

- **Data Translator** : This is the core component of the AAS-Agent interface layer. It is responsible for translating the data between the AAS format and the Agent's internal data structures. Where it takes raw data from the AAS and converts them into a format that the Agent can understand and vice versa.

- **BDI Agent Interface** : This component acts as a bridge between the Data Translator and the BDI Agent. It ensures that the translated data is properly integrated into the Agent's belief, desire, and intention structures. It also facilitates the communication of the Agent's decisions and actions back to the AAS through the Data Translator.

The last layer is the **BDI Agent Core** layer. This layer contains the main components of the BDI Agent architecture:

- **Agent Core** : This is the central component of the BDI Agent. It manages the Agent's beliefs, desires, and intentions. It processes incoming data from the AAS and makes decisions based on its internal state and goals. It also procees data coming from the Resource Asset it is assigned to manage. The main functionality is update beliefs based on the data received from the AAS and the Resource Asset. Then it generates desires and intentions based on the updated beliefs and the Agent's objectives.

- **Asset Interface** : This component is responsible for communication between the BDI Agent and the Resource Asset. It contains the necessary protocols and methods to interact with the asset, retrieve data, and send commands.

2.2 Production Agent Components

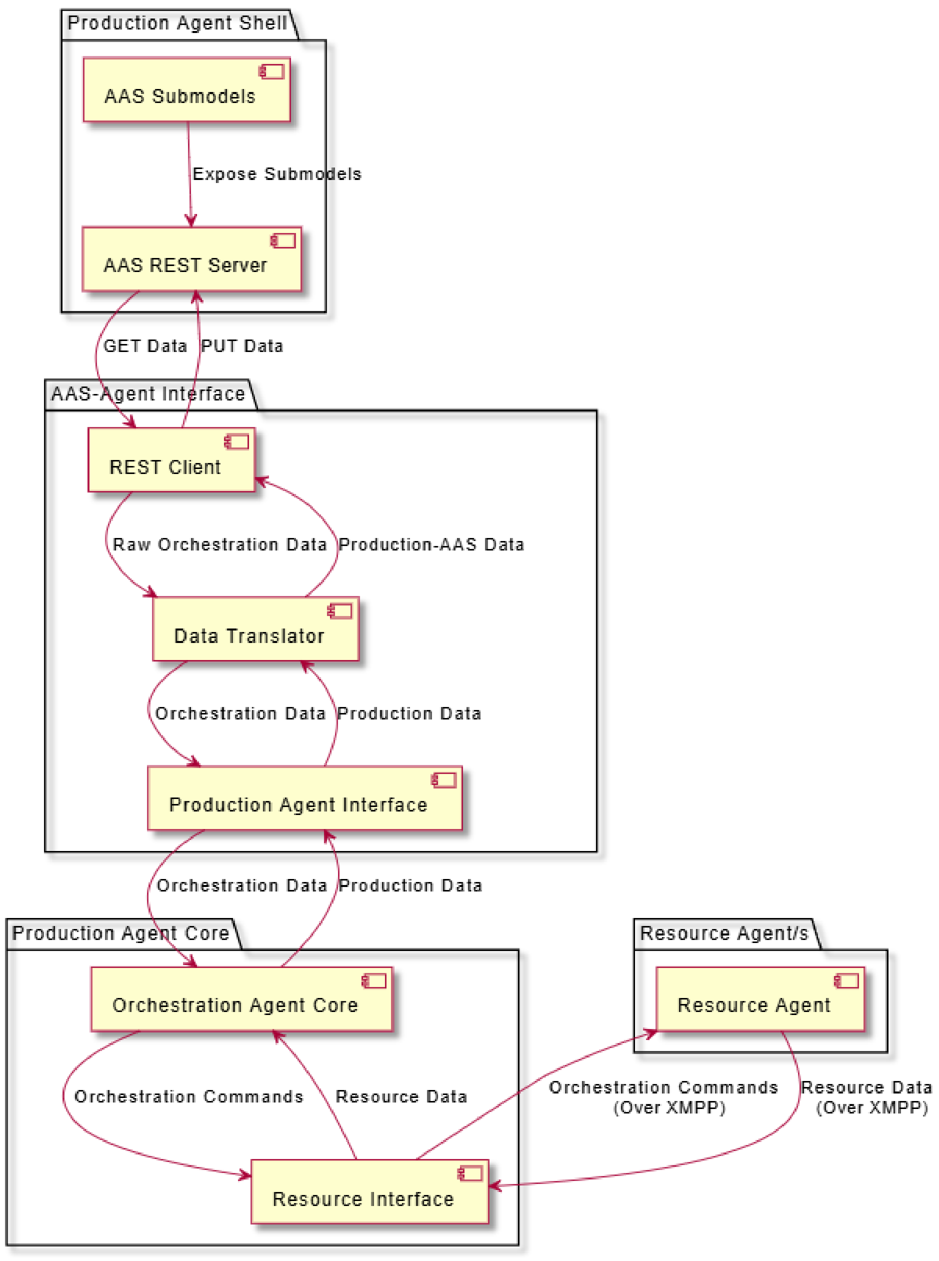


Figure 15: Production Agent Components Diagram

The Production Agent components are illustrated in Figure 15. It has similar structure to the Resource Agent components with some differences in the Agent Core layer and the AAS-Agent Interface layer. In the **AAS-Agent Interface layer**, the components are similar to the Resource Agent with the data translator being adapted to handle production-specific data from the AAS submodels. This includes translating production schedules, work orders, and status updates between the AAS format and the Agent's internal structures. The AAS-Agent Interface gets the Raw orchestration commands from the Production AAS where it then translates them into an orchestration command that the orchestration core can understand and act upon. At the same time the data coming from the Production Agent core about the production status and progress is translated back into the AAS format and updated into the Production AAS submodels.

In the main layer that is the **Production Agent core** layer the diffrences are more significant. The Production Agent core contains the **orchestration agent core** component instead of the BDI agent core. This component is responsible for managing the orchestration of production tasks and workflows. It processes orchestration commands received from the AAS-Agent interface layer and coordinates the execution of these tasks across multiple Resource Agents and assets. It also monitors the progress of production activities and updates the AAS with status information through the AAS-Agent interface layer. The **Resource Interface** component in this layer is also adapted to handle communication with production-related assets, such as manufacturing equipment and assembly lines. It ensures that the Production Agent can effectively manage and control these assets as part of the overall production process.

9

3 Classes and Objects

In this section a breakdown of the main classes and objects used in the system architecture is provided. Each class is described in terms of its attributes, methods, and relationships with other classes.

3.1 Resource AAS classes

In this subsection a detailed breakdown of the classes representing the AAS-Agent interface , the BDI agent Layer and the defnition of the Submodels .

3.1.1 Submodels definition

In this subsection is the definition of the submodels of the Resource AAS. Figure 16 shows the class diagram representing the Resource Submodel, while Table 5 provides an overview of each submodel along with its elements and definitions.

Table 5: Overview of Resource Submodels and their elements

Submodel	**Elements**	**Definition / Purpose**
Operational State	Current State , Historical Data	Represents the current and historical operational states of the resource.
Knowledge	Resource Constraints, Environmental Conditions	Contains knowledge about resource constraints and environmental conditions affecting operations.
Capability	Drill , MoveXY ,Skill	Defines the capabilities and skills of the resource for task execution.
Interaction and Scheduling	Endpoints	Manages interaction points and scheduling information for resource coordination.

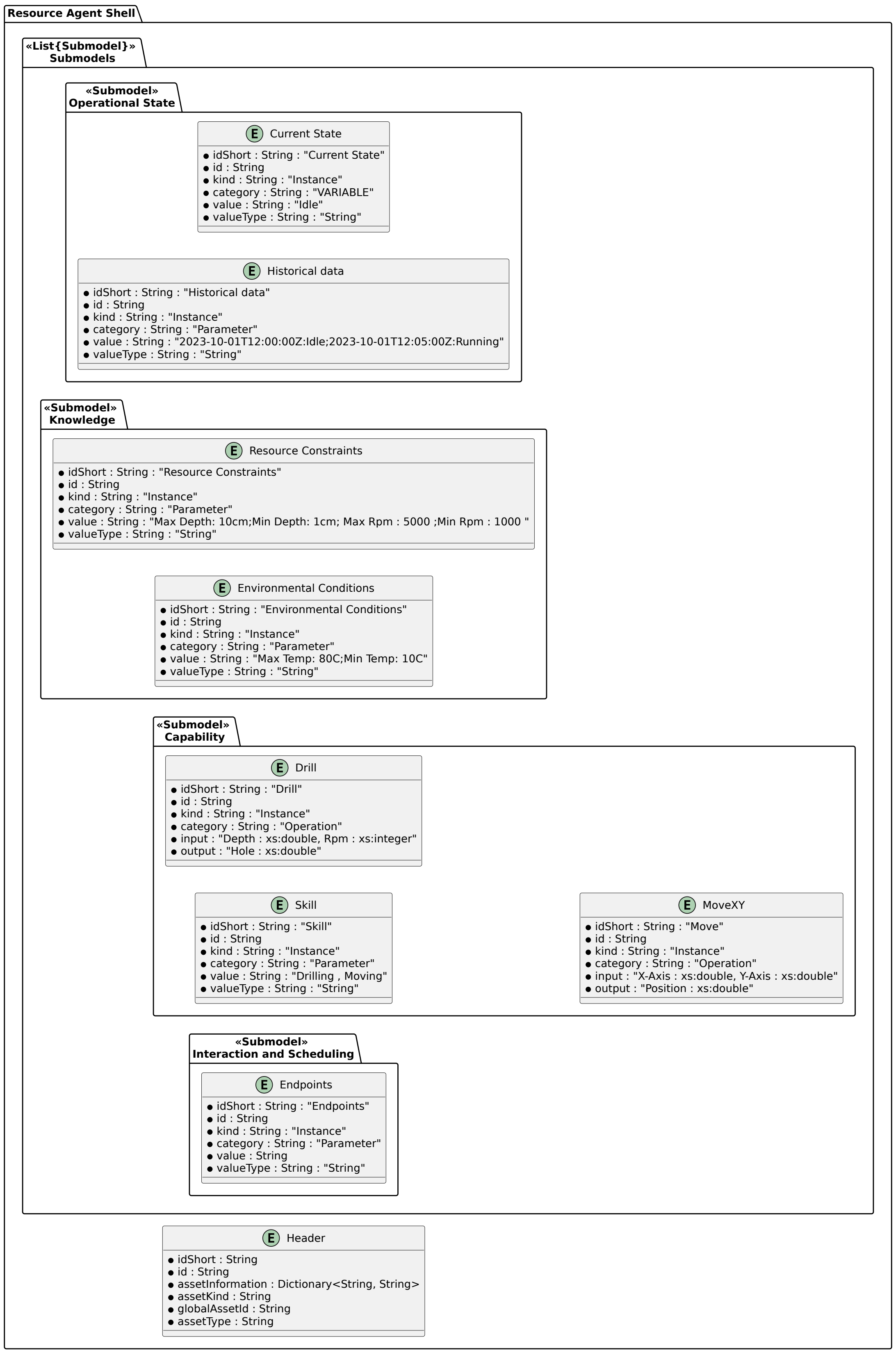

Resource Agent Shell
«List{Submodel}» Submodels
«Submodel» Operational State
E Current State
● idShort : String : "Current State"
● id : String
● kind : String : "Instance"
● category : String : "VARIABLE"
● value : String : "Idle"
● valueType : String : "String"
E Historical data
● idShort : String : "Historical data"
● id : String
● kind : String : "Instance"
● category : String : "Parameter"
● value : String : "2023-10-01T12:00:00Z:Idle;2023-10-01T12:05:00Z:Running"
● valueType : String : "String"
«Submodel» Knowledge
E Resource Constraints
● idShort : String : "Resource Constraints"
● id : String
● kind : String : "Instance"
● category : String : "Parameter"
● value : String : "Max Depth: 10cm;Min Depth: 1cm; Max Rpm : 5000 ;Min Rpm : 1000 "
● valueType : String : "String"
E Environmental Conditions
● idShort : String : "Environmental Conditions"
● id : String
● kind : String : "Instance"
● category : String : "Parameter"
● value : String : "Max Temp: 80C;Min Temp: 10C"
● valueType : String : "String"
«Submodel» Capability
E Drill
● idShort : String : "Drill"
● id : String
● kind : String : "Instance"
● category : String : "Operation"
● input : "Depth : xs:double, Rpm : xs:integer"
● output : "Hole : xs:double"
E Skill
● idShort : String : "Skill"
● id : String
● kind : String : "Instance"
● category : String : "Parameter"
● value : String : "Drilling , Moving"
● valueType : String : "String"
E MoveXY
● idShort : String : "Move"
● id : String
● kind : String : "Instance"
● category : String : "Operation"
● input : "X-Axis : xs:double, Y-Axis : xs:double"
● output : "Position : xs:double"
«Submodel» Interaction and Scheduling
E Endpoints
● idShort : String : "Endpoints"
● id : String
● kind : String : "Instance"
● category : String : "Parameter"
● value : String
● valueType : String : "String"
E Header
● idShort : String
● id : String
● assetInformation : Dictionary<String, String>
● assetKind : String
● globalAssetId : String
● assetType : String

Figure 16: Class diagram representing the Resource Submodel.

3.1.2 REST Client

This subsection provides a detailed breakdown of the classes representing the REST Client used for communication with the AAS. Figure 17 shows the class diagram representing the REST Client.

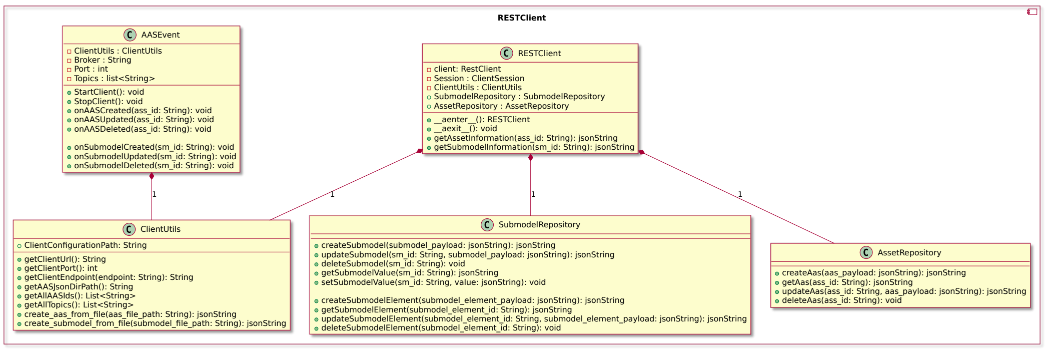

RESTClient
AASEvent
ClientUtils : ClientUtils
Broker : String
Port : int
Topics : list<String>
StartClient(): void
StopClient(): void
onAASCreated(ass_id: String): void
onAASUpdated(ass_id: String): void
onAASDeleted(ass_id: String): void
onSubmodelCreated(sm_id: String): void
onSubmodelUpdated(sm_id: String): void
onSubmodelDeleted(sm_id: String): void
RESTClient
client: RestClient
Session : ClientSession
ClientUtils : ClientUtils
SubmodelRepository : SubmodelRepository
AssetRepository : AssetRepository
__aenter__(): RESTClient
__aexit__(): void
getAssetInformation(ass_id: String): jsonString
getSubmodelInformation(sm_id: String): jsonString
1
1
1
1
ClientUtils
ClientConfigurationPath: String
getClientUrl(): String
getClientPort(): int
getClientEndpoint(endpoint: String): String
getAASJsonDirPath(): String
getAllAASIds(): List<String>
getAllTopics(): List<String>
create_aas_from_file(aas_file_path: String): jsonString
create_submodel_from_file(submodel_file_path: String): jsonString
SubmodelRepository
createSubmodel(submodel_payload: jsonString): jsonString
updateSubmodel(sm_id: String, submodel_payload: jsonString): jsonString
deleteSubmodel(sm_id: String): void
getSubmodelValue(sm_id: String): jsonString
setSubmodelValue(sm_id: String, value: jsonString): void
createSubmodelElement(submodel_element_payload: jsonString): jsonString
getSubmodelElement(submodel_element_id: String): jsonString
updateSubmodelElement(submodel_element_id: String, submodel_element_payload: jsonString): jsonString
deleteSubmodelElement(submodel_element_id: String): void
AssetRepository
createAas(aas_payload: jsonString): jsonString
getAas(ass_id: String): jsonString
updateAas(ass_id: String, aas_payload: jsonString): jsonString
deleteAas(ass_id: String): void

Figure 17: Class diagram representing the REST Client.

The main design of this component is the two repository classes, the *AAS Repository* class that is responsible for all the communication with the AAS over REST API calls, and the *Submodel Repository* class that is responsible for all the communication with a specific submodel of the AAS over REST API calls. Both classes use the *HTTP Client* class that is a wrapper over the HTTP library used to perform the REST API calls. Those then are called into the main class *REST Client* that creates, starts, and stops a client session. Also the class provides a simple MQTT client to subscribe to AAS events over MQTT protocol. This client is represented by the *AASEvent* class.

3.1.3 AAS-Agent Interface

This subsection provides a detailed breakdown of the classes representing the AAS-Agent interface. Figure 18 shows the class diagram representing the AAS-Agent Interface.

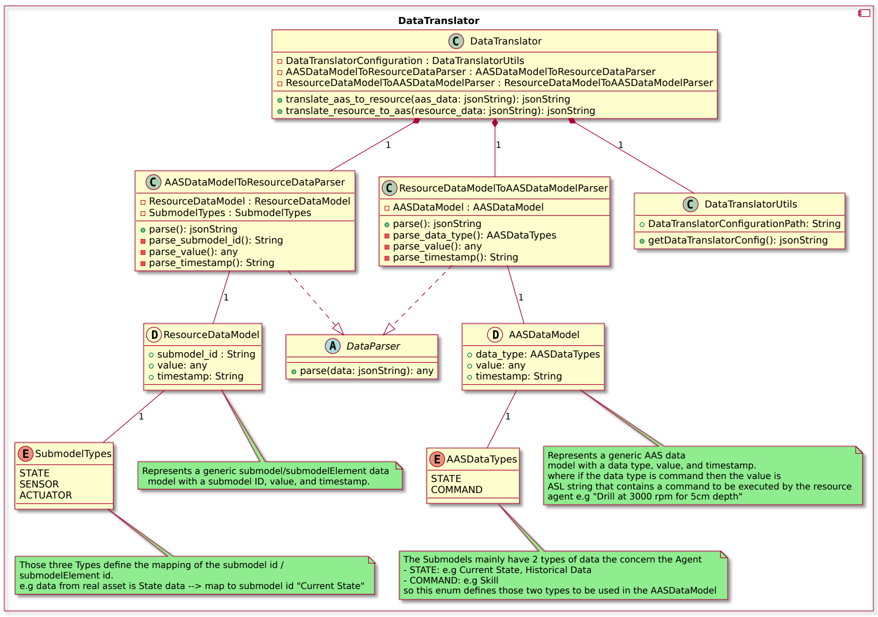
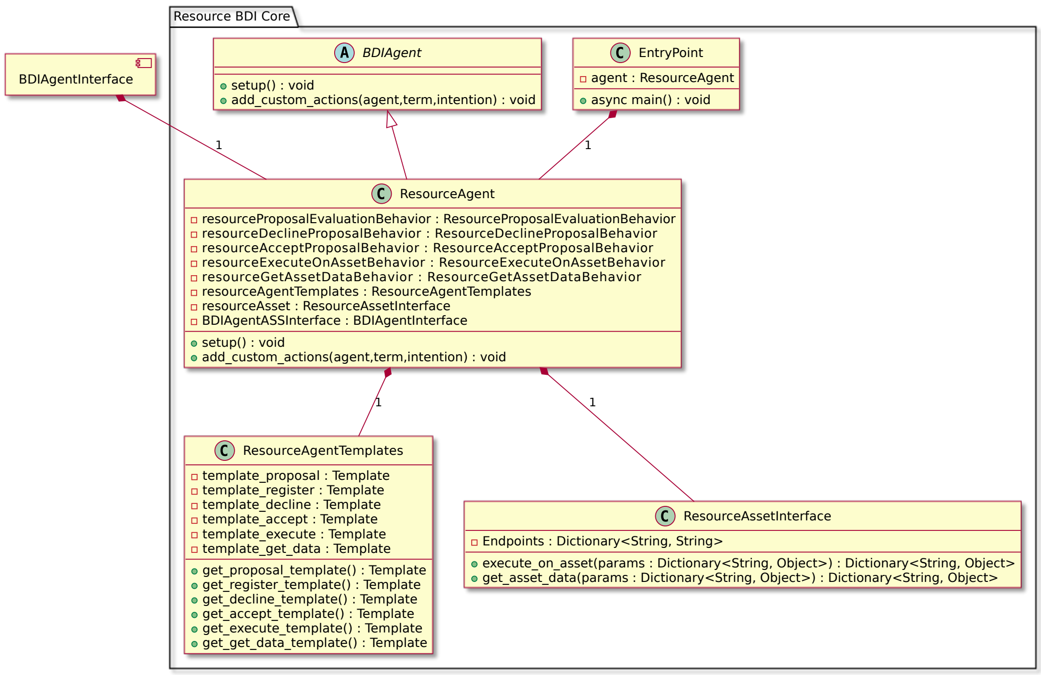

DataTranslator
C DataTranslator
DataTranslatorConfiguration : DataTranslatorUtils
AASDataModelToResourceDataParser : AASDataModelToResourceDataParser
ResourceDataModelToAASDataModelParser : ResourceDataModelToAASDataModelParser
translate_aas_to_resource(aas_data: jsonString): jsonString
translate_resource_to_aas(resource_data: jsonString): jsonString
1
1
1
C AASDataModelToResourceDataParser
ResourceDataModel : ResourceDataModel
SubmodelTypes : SubmodelTypes
parse(): jsonString
parse_submodel_id(): String
parse_value(): any
parse_timestamp(): String
C ResourceDataModelToAASDataModelParser
AASDataModel : AASDataModel
parse(): jsonString
parse_data_type(): AASDataTypes
parse_value(): any
parse_timestamp(): String
C DataTranslatorUtils
DataTranslatorConfigurationPath: String
getDataTranslatorConfig(): jsonString
1
1
D ResourceDataModel
submodel_id : String
value: any
timestamp: String
A DataParser
parse(data: jsonString): any
D AASDataModel
data_type: AASDataTypes
value: any
timestamp: String
1
1
E SubmodelTypes
STATE
SENSOR
ACTUATOR
Represents a generic submodel/submodelElement data model with a submodel ID, value, and timestamp.
E AASDataTypes
STATE
COMMAND
Represents a generic AAS data model with a data type, value, and timestamp. where if the data type is command then the value is ASL string that contains a command to be executed by the resource agent e.g "Drill at 3000 rpm for 5cm depth"
Those three Types define the mapping of the submodel id / submodelElement id. e.g data from real asset is State data --> map to submodel id "Current State"
The Submodels mainly have 2 types of data the concern the Agent - STATE: e.g Current State, Historical Data - COMMAND: e.g Skill so this enum defines those two types to be used in the AASDataModel

Figure 18: Class diagram representing the AAS-Agent Interface.

This interface provides the needed datatypes to switch between the data coming from the AAS and the data coming from the Resource Asset over the agent. The datatype that represent the data coming from the AAS to the agent is the *AAS Data Model* class, while the datatype that represent the data coming from the Resource Asset to the agent is the *Resource Data Model* class. This conversion is done through the *DataParser* class that is an abstract class that contains the methods needed for the conversion between the two datatypes. The *AAS to Resource Data Parser* class is a concrete implementation of the *DataParser* class that implements the conversion from the *AAS Data Model* to the *Resource Data Model*, while the *Resource to AAS Data Parser* class is a concrete implementation of the *DataParser* class that implements the conversion from the *Resource Data Model* to the *AAS Data Model*. Finally the *Data Translator* is the main level that is callable through entry points from the BDI agent layer to perform the needed conversion between the two datatypes. In the lowest level there is two enums the *AAS Data Types* enum that contains the data types used in the AAS Data Model and the *Resource Data Types* enum that contains the data types used in the Resource Data Model.

3.1.4 BDI Agent Core

This subsection provides a detailed breakdown of the classes representing the BDI Agent Core. Figure 19 shows the class diagram representing the BDI Agent Core.



Resource BDI Core
BDIAgentInterface
BDIAgent
setup() : void
add_custom_actions(agent,term,intention) : void
EntryPoint
agent : ResourceAgent
async main() : void
1
1
ResourceAgent
resourceProposalEvaluationBehavior : ResourceProposalEvaluationBehavior
resourceDeclineProposalBehavior : ResourceDeclineProposalBehavior
resourceAcceptProposalBehavior : ResourceAcceptProposalBehavior
resourceExecuteOnAssetBehavior : ResourceExecuteOnAssetBehavior
resourceGetAssetDataBehavior : ResourceGetAssetDataBehavior
resourceAgentTemplates : ResourceAgentTemplates
resourceAsset : ResourceAssetInterface
BDIAgentASSInterface : BDIAgentInterface
setup() : void
add_custom_actions(agent,term,intention) : void
1
1
ResourceAgentTemplates
template_proposal : Template
template_register : Template
template_decline : Template
template_accept : Template
template_execute : Template
template_get_data : Template
get_proposal_template() : Template
get_register_template() : Template
get_decline_template() : Template
get_accept_template() : Template
get_execute_template() : Template
get_get_data_template() : Template
ResourceAssetInterface
Endpoints : Dictionary<String, String>
execute_on_asset(params : Dictionary<String, Object>) : Dictionary<String, Object>
get_asset_data(params : Dictionary<String, Object>) : Dictionary<String, Object>

Figure 19: Class diagram representing the BDI Agent Core.

By using **SPADE** framework , there is an architecture that represents the BDI agent core. Mainly each agent core contians an Agent core where the BDI Agent lives and Behaviours that represent the different functionalities of the agent. In the figure 19 is the Agent Core represented ,while in figure 20 are the different behaviours of the agent represented.

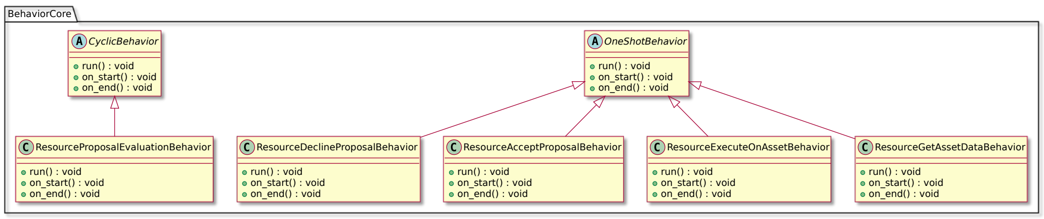


Figure 20: Class diagram representing the BDI Agent Behaviours.

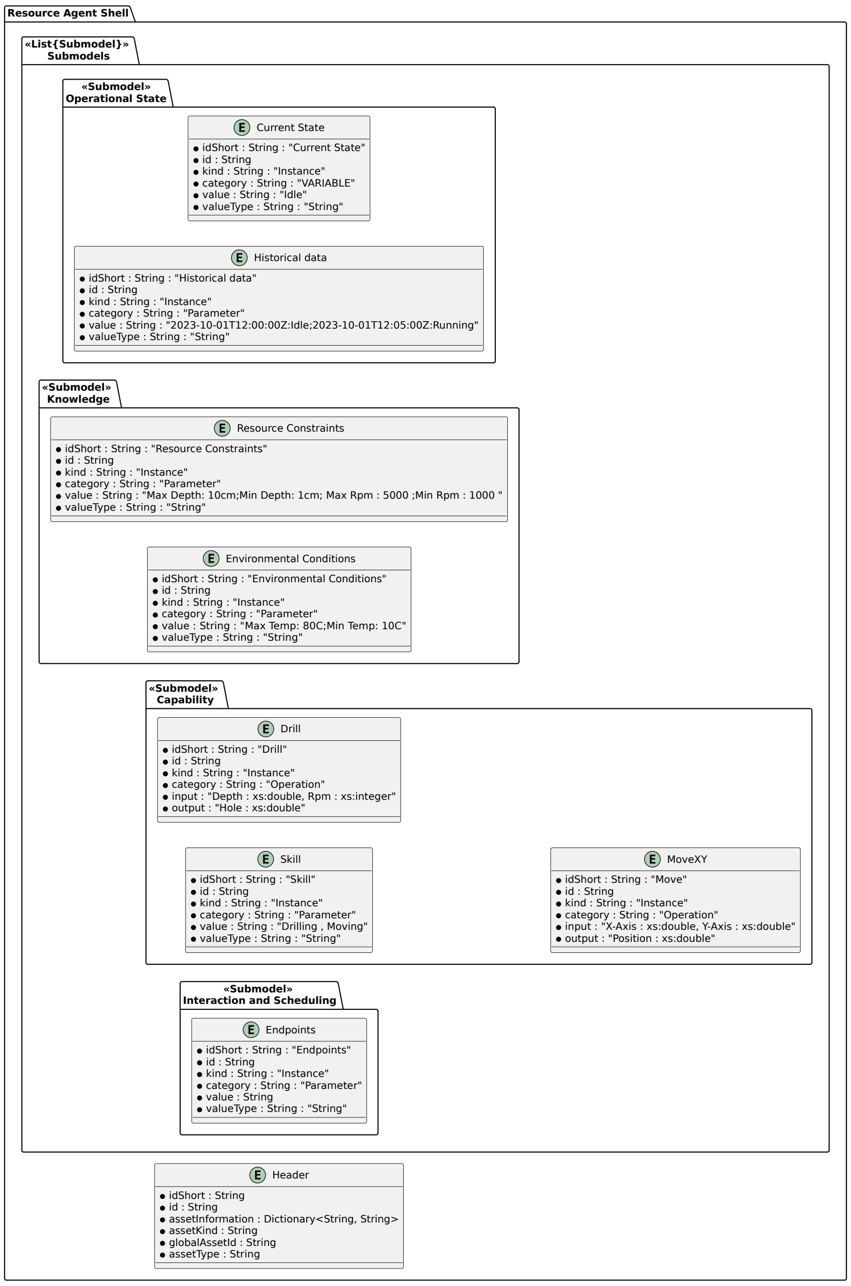
The *ResourceAgent* class is the main class that represents the BDI agent. Its mainly an implementation of the **SPADE** *BDIAgent* class. The needed methods to be implemented are the *setup* method that is called when the agent is started and the *add custom actions* that let the agent define action or desires that is represtented in the behaviours. The Data exchange between the agents in the **SPADE** framework is done mainly throgh **Templates** which are represented by the *Resource Agent Templates* class. The Behaviours of the agent is represented by two abstract classes of the **SPADE** framework: *OneShotBehaviour* and *CyclicBehaviour*. Those then are implemented into different concrete classes like the *Proposal Process Behaviour* class that is responsible for handling the proposals coming from the Orchestrator agent, the *Task Execution Behaviour* class that is responsible for executing the tasks assigned to the resource, and the *Status Update Behaviour* class that is responsible for updating the status of the resource in the AAS at regular intervals.

3.2 Production AAS classes

This subsection provides a detailed breakdown of the classes representing the Production AAS-Agent interface , the Orchestrator Layer and the definition of the Submodels.

3.2.1 Submodels definition

In this subsection is the definition of the submodels of the Production AAS. Figure 21 shows the class diagram representing the Production Sub model, while Table 6 provides an overview of each submodel along with its elements and definitions.



Resource Agent Shell
«List{Submodel}» Submodels
«Submodel» Operational State
E Current State
idShort : String : "Current State"
id : String
kind : String : "Instance"
category : String : "VARIABLE"
value : String : "Idle"
valueType : String : "String"
E Historical data
idShort : String : "Historical data"
id : String
kind : String : "Instance"
category : String : "Parameter"
value : String : "2023-10-01T12:00:00Z:Idle;2023-10-01T12:05:00Z:Running"
valueType : String : "String"
«Submodel» Knowledge
E Resource Constraints
idShort : String : "Resource Constraints"
id : String
kind : String : "Instance"
category : String : "Parameter"
value : String : "Max Depth: 10cm;Min Depth: 1cm; Max Rpm : 5000 ;Min Rpm : 1000 "
valueType : String : "String"
E Environmental Conditions
idShort : String : "Environmental Conditions"
id : String
kind : String : "Instance"
category : String : "Parameter"
value : String : "Max Temp: 80C;Min Temp: 10C"
valueType : String : "String"
«Submodel» Capability
E Drill
idShort : String : "Drill"
id : String
kind : String : "Instance"
category : String : "Operation"
input : "Depth : xs:double, Rpm : xs:integer"
output : "Hole : xs:double"
E Skill
idShort : String : "Skill"
id : String
kind : String : "Instance"
category : String : "Parameter"
value : String : "Drilling , Moving"
valueType : String : "String"
E MoveXY
idShort : String : "Move"
id : String
kind : String : "Instance"
category : String : "Operation"
input : "X-Axis : xs:double, Y-Axis : xs:double"
output : "Position : xs:double"
«Submodel» Interaction and Scheduling
E Endpoints
idShort : String : "Endpoints"
id : String
kind : String : "Instance"
category : String : "Parameter"
value : String
valueType : String : "String"
E Header
idShort : String
id : String
assetInformation : Dictionary<String, String>
assetKind : String
globalAssetId : String
assetType : String

Figure 21: Class diagram representing the Production Submodel.

Table 6: Overview of Production Submodels and their elements

Submodel	**Elements**	**Definition / Purpose**
Process Plan	Entry & Exit; Nodes; Edges; Preconditions; Postconditions; Required Input & Output; Required Capabilities.	Defines the procedural/workflow model for a production task. Nodes represent steps, edges represent transitions; pre/postconditions control applicability; inputs/outputs and required capabilities describe data and resources needed by each step. Used for orchestration and execution planning.
Services Catalog	Services (service id, signature, parameters, category).	Registry of callable services / functional capabilities exposed by the production component. Describes available service operations, required parameters and invocation metadata for integration and discovery.
Execution State & Tracking	Timestamps; Token position; Current node; Step status; Measured parameters.	Runtime information about process execution and monitoring: timestamps and token positions for traceability, current node and step status for progress reporting, measured parameters for quality/telemetry. Enables tracking, auditing and recovery.
Interface and Endpoints	SkillCall Endpoints (URIs / endpoint descriptors).	Communication and integration endpoints used to invoke skills or services remotely (e.g., orchestrator calls, skill invocation). Contains addressing and protocol metadata for external interaction.
Header / Metadata	idShort, id, administration/asset information, semantic identifiers, versioning.	Global AAS header and identification information that links the submodel to the asset and provides administrative metadata (identifiers, semantic references and versioning). Used for discovery and semantic alignment.

3.2.2 REST Client

Similar to the Resource AAS REST client, this subsection provides a detailed breakdown of the classes representing the REST Client used for communication with the Production AAS. Figure 22 shows the class diagram representing the REST Client. The design of this

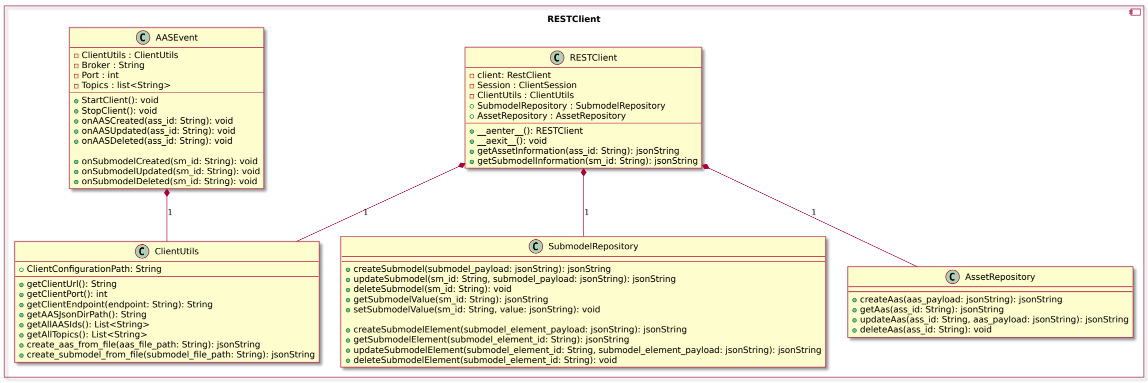

RESTClient
AASEvent
ClientUtils : ClientUtils
Broker : String
Port : int
Topics : list<String>
StartClient(): void
StopClient(): void
onAASCreated(ass_id: String): void
onAASUpdated(ass_id: String): void
onAASDeleted(ass_id: String): void
onSubmodelCreated(sm_id: String): void
onSubmodelUpdated(sm_id: String): void
onSubmodelDeleted(sm_id: String): void
RESTClient
client: RestClient
Session : ClientSession
ClientUtils : ClientUtils
SubmodelRepository : SubmodelRepository
AssetRepository : AssetRepository
__aenter__(): RESTClient
__aexit__(): void
getAssetInformation(ass_id: String): jsonString
getSubmodelInformation(sm_id: String): jsonString
1
1
1
1
ClientUtils
ClientConfigurationPath: String
getClientUrl(): String
getClientPort(): int
getClientEndpoint(endpoint: String): String
getAASJsonDirPath(): String
getAllAASIds(): List<String>
getAllTopics(): List<String>
create_aas_from_file(aas_file_path: String): jsonString
create_submodel_from_file(submodel_file_path: String): jsonString
SubmodelRepository
createSubmodel(submodel_payload: jsonString): jsonString
updateSubmodel(sm_id: String, submodel_payload: jsonString): jsonString
deleteSubmodel(sm_id: String): void
getSubmodelValue(sm_id: String): jsonString
setSubmodelValue(sm_id: String, value: jsonString): void
createSubmodelElement(submodel_element_payload: jsonString): jsonString
getSubmodelElement(submodel_element_id: String): jsonString
updateSubmodelElement(submodel_element_id: String, submodel_element_payload: jsonString): jsonString
deleteSubmodelElement(submodel_element_id: String): void
AssetRepository
createAas(aas_payload: jsonString): jsonString
getAas(ass_id: String): jsonString
updateAas(ass_id: String, aas_payload: jsonString): jsonString
deleteAas(ass_id: String): void

Figure 22: Class diagram representing the REST Client.

component is similar to the Resource AAS REST client, with two repository classes: the *Production AAS Repository* class for communication with the Production AAS over REST API calls, and the *Production Submodel Repository* class for communication with specific submodels of the Production AAS over REST API calls. Both classes utilize the *HTTP Client* class for performing REST API calls. The main class *REST Client* creates, starts, and stops a client session, and also provides an MQTT client to subscribe to Production AAS events over the MQTT protocol, represented by the *ProductionAASEvent* class.

3.2.3 AAS-Agent Interface

This subsection provides a detailed breakdown of the classes representing the Production AAS-Agent interface. Figure 23 shows the class diagram representing the Production AAS-Agent Interface.

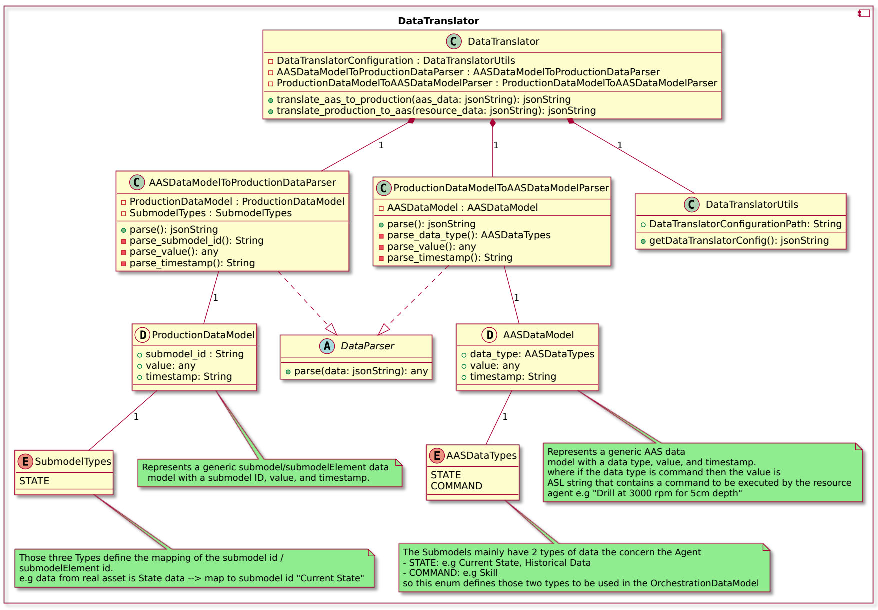

DataTranslator
DataTranslator
DataTranslatorConfiguration : DataTranslatorUtils
AASDataModelToProductionDataParser : AASDataModelToProductionDataParser
ProductionDataModelToAASDataModelParser : ProductionDataModelToAASDataModelParser
translate_aas_to_production(aas_data: jsonString): jsonString
translate_production_to_aas(resource_data: jsonString): jsonString
1
1
1
AASDataModelToProductionDataParser
ProductionDataModel : ProductionDataModel
SubmodelTypes : SubmodelTypes
parse(): jsonString
parse_submodel_id(): String
parse_value(): any
parse_timestamp(): String
ProductionDataModelToAASDataModelParser
AASDataModel : AASDataModel
parse(): jsonString
parse_data_type(): AASDataTypes
parse_value(): any
parse_timestamp(): String
DataTranslatorUtils
DataTranslatorConfigurationPath: String
getDataTranslatorConfig(): jsonString
1
1
ProductionDataModel
submodel_id : String
value: any
timestamp: String
DataParser
parse(data: jsonString): any
AASDataModel
data_type: AASDataTypes
value: any
timestamp: String
1
1
SubmodelTypes
STATE
Represents a generic submodel/submodelElement data model with a submodel ID, value, and timestamp.
AASDataTypes
STATE
COMMAND
Represents a generic AAS data model with a data type, value, and timestamp. where if the data type is command then the value is ASL string that contains a command to be executed by the resource agent e.g "Drill at 3000 rpm for 5cm depth"
Those three Types define the mapping of the submodel id / submodelElement id. e.g data from real asset is State data --> map to submodel id "Current State"
The Submodels mainly have 2 types of data the concern the Agent - STATE: e.g Current State, Historical Data - COMMAND: e.g Skill so this enum defines those two types to be used in the OrchestrationDataModel

Figure 23: Class diagram representing the Production AAS-Agent Interface.

The main diffrerence between this interface and the Resource AAS-Agent interface is the datatypes used. The datatype that represent the data coming from the Production AAS to the Orchestrator agent is the *Production AAS Data Model* class, while the datatype that represent the data used internally by the Orchestrator agent is the *Production Data Model* class. This conversion is done through the *Production DataParser* class that is an abstract class that contains the methods needed for the conversion between the two datatypes. The *Production AAS to Production Data Parser* class is a concrete implementation of the *Production DataParser* class that implements the conversion from the *Production AAS Data Model* class to the *Production Data Model* class. The Interface is showen in figure 24.

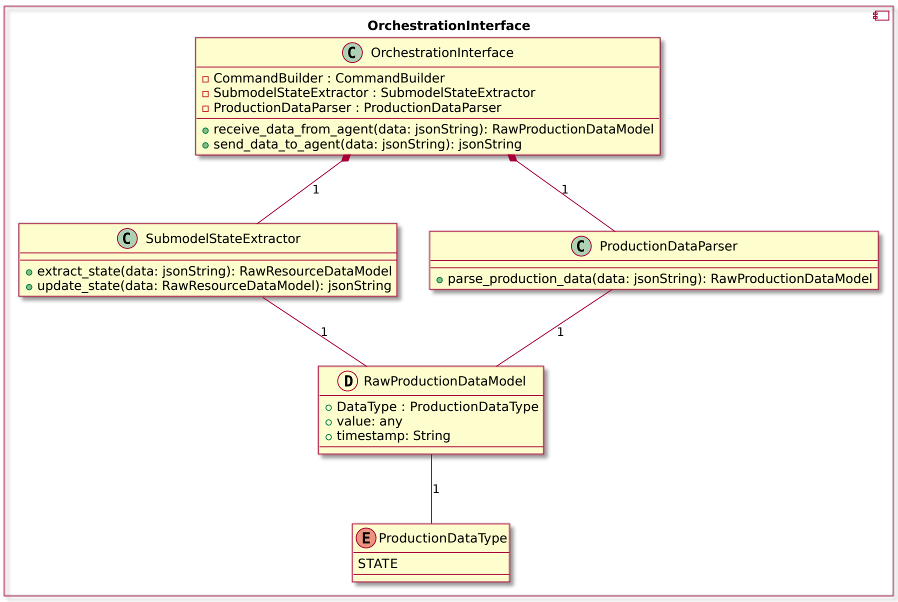

OrchestrationInterface
OrchestrationInterface
CommandBuilder : CommandBuilder
SubmodelStateExtractor : SubmodelStateExtractor
ProductionDataParser : ProductionDataParser
receive_data_from_agent(data: jsonString): RawProductionDataModel
send_data_to_agent(data: jsonString): jsonString
1
1
SubmodelStateExtractor
extract_state(data: jsonString): RawResourceDataModel
update_state(data: RawResourceDataModel): jsonString
ProductionDataParser
parse_production_data(data: jsonString): RawProductionDataModel
1
1
RawProductionDataModel
DataType : ProductionDataType
value: any
timestamp: String
1
ProductionDataType
STATE

Figure 24: Class diagram representing the Production AAS-Agent Interface.

3.2.4 Orchestration and Negotiation Core

This subsection provides a detailed breakdown of the classes representing the Orchestration and Negotiation Core. The orchestration Core is represented in figure 25. The Orchstrator is the main class that represents the Orchestrator agent. Its mainly an implementation of the **SPADE** *Agent* class. The needed methods to be implemented are the *setup* method that is called when the agent is started and the *add custom actions* that let the agent define action or desires that is represtented in the behaviours. The Data exchange between the agents in the **SPADE** framework is done mainly throgh **Templates** which are represented by the *Orchestrator Agent Templates* class. The Behaviours of the agent is represented by an abstract classe of the **SPADE** framework: *OneShotBehaviour*. Those then are implemented into different concrete classes like the *Allocate Skill To Resource Behaviour* class that is responsible for allocating skills to resources based on their capabilities, and the *Negotiation Behaviour* class that is responsible for handling the negotiation process with Resource agents to finalize task assignments.

The other core which is the Negotiation core is represented in figure 26. This is also represented by and Agent class in the **SPADE** framework. It also have defined behaviours that are represented in the figure. 26.

The two cores work together to manage the orchestration of production tasks and the negotiation with Resource agents to ensure optimal task assignments based on resource capabilities and time availability.

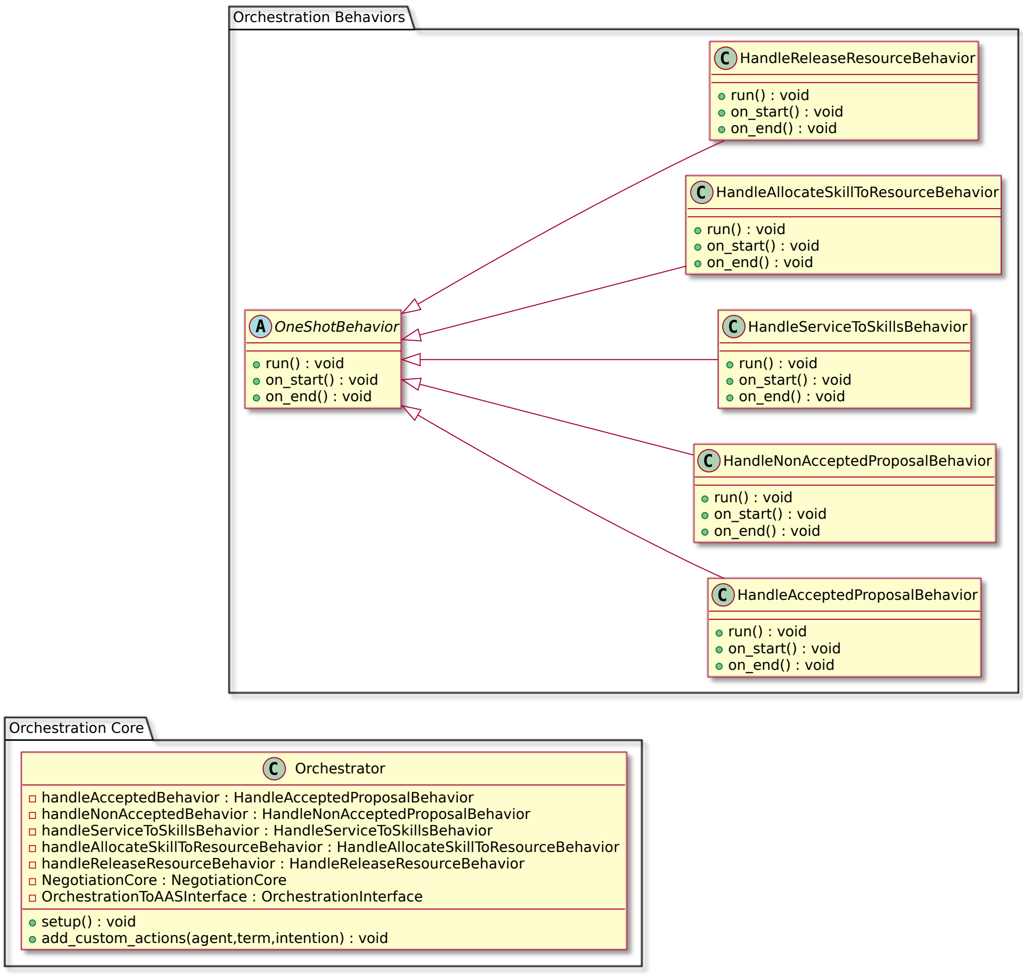

Orchestration Behaviors
HandleReleaseResourceBehavior
run() : void
on_start() : void
on_end() : void
HandleAllocateSkillToResourceBehavior
run() : void
on_start() : void
on_end() : void
OneShotBehavior
run() : void
on_start() : void
on_end() : void
HandleServiceToSkillsBehavior
run() : void
on_start() : void
on_end() : void
HandleNonAcceptedProposalBehavior
run() : void
on_start() : void
on_end() : void
HandleAcceptedProposalBehavior
run() : void
on_start() : void
on_end() : void
Orchestration Core
Orchestrator
handleAcceptedBehavior : HandleAcceptedProposalBehavior
handleNonAcceptedBehavior : HandleNonAcceptedProposalBehavior
handleServiceToSkillsBehavior : HandleServiceToSkillsBehavior
handleAllocateSkillToResourceBehavior : HandleAllocateSkillToResourceBehavior
handleReleaseResourceBehavior : HandleReleaseResourceBehavior
NegotiationCore : NegotiationCore
OrchestrationToAASInterface : OrchestrationInterface
setup() : void
add_custom_actions(agent,term,intention) : void

Figure 25: Class diagram representing the Orchestration Core.

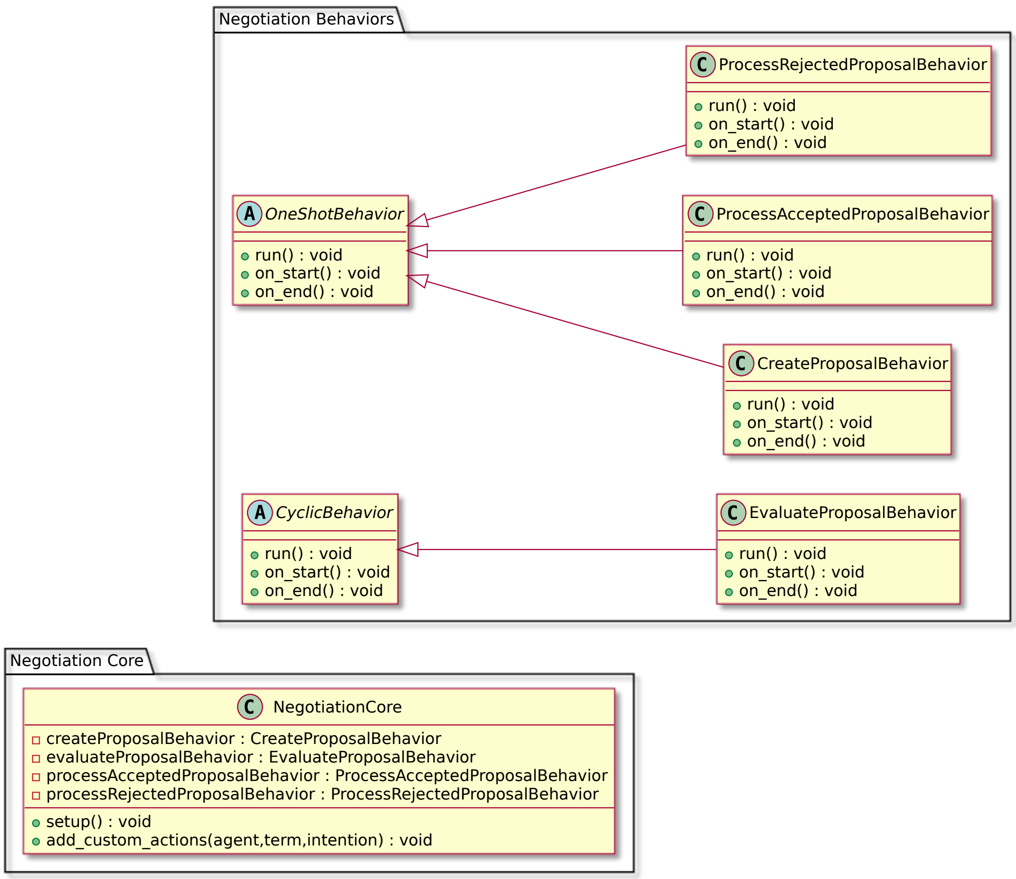


Figure 26: Class diagram representing the Negotiation Core.

4 Flows and State Machines

TBD .

References