AOS – ADD Document

Michel Mosa, Majd Atwan, Moamen Mawasi, Natalie Ahmad

Software Engineering

Faculty of Engineering

Ben Gurion University

Contents

1. Use Cases

1.1 User Profiles – The Actors

AOS Developers: These are skilled professionals who specialize in setting up and programming robots. They focus on using metadata.

System Administrators: These are experts who take care of managing and fixing the system. They handle tasks like setting up the middleware (the software that connects different parts of the system) and solving any technical problems that come up.

External Systems are essentially components of a system that operate autonomously, meaning they function automatically without direct human intervention. These components serve a vital role in providing essential data and control functions to the main system. (Such as postman automates the process of interacting with external APIs, providing the main system with the necessary information it needs to function effectively).

The main difference between these groups is what they do:

- AOS developers will have the following responsibilities: expanding the system with additional features.

- Administrators manage the whole system.

- External Systems provide data and functions for the system to use.

1.2 Use – Cases

1.2.1 Configure Robot Capabilities

- *Description:* an autonomous robot developer or system administrator configures the system by providing detailed metadata about the robot's capabilities, including tasks and relevant details
- *Actors:* autonomous robot developer, system

administrator

- *Pre-Condition:* middleware should be activated
- *Post-Condition:* robot capabilities are accurately configured and securely stored in the system.
- *Main Success Scenario:*
  1. The user accesses the configuration interface, which provides clear guidance and options.
  2. Inputs detailed metadata describing the robot's capabilities, with prompts and validation to ensure accuracy.
  3. The system efficiently processes and securely stores the configuration data, with real-time feedback to the user.
  4. The system confirms successful configuration, providing confirmation messages and access to configuration logs for verification.

1.2.2 Activate Robot Capability

- *Description:* developer initiates the activation of a specific robot capability, enabling the robot to perform designated tasks efficiently and effectively. Activation can be performed through the intuitive graphical interface or via RESTful commands for seamless integration.

- *Actors:* AOS developer

- *Pre-Condition:* robot capabilities must be accurately configured and validated.

- *Post-Condition:* the specified robot capability is successfully activated and ready for immediate use.

- *Main Success Scenario:*
  1. The user navigates to the activation section within the interface, which clearly displays available capabilities and their status
  2. Selects the desired robot capability for activation, with visual indicators for clarity.
  3. The system promptly translates the activation request into executable tasks for the robot, with error handling and fallback mechanisms in place.
  4. Confirms successful activation, providing real-time status updates and integration logs for verification.

1.2.3 Monitor Real-Time Robot Behavior

- **Description:** Developers monitor and analyze the real-time behavior and status of autonomous robots through the Gazebo simulator. This includes comprehensive insights into sensor data, task execution progress, and immediate alerts for timely intervention.

- **Actors:** AOS Developer

- **Pre-Condition:**

    - Robot capabilities must be successfully activated and operational.
    - Gazebo simulator is set up and integrated with the robot's ROS2 environment.

- **Post-Condition:** Developers gain actionable insights into the robot's behavior, enabling informed decision-making and optimization.

- **Main Success Scenario:**

    1. **Access Gazebo Simulation:**
       The developer launches the Gazebo simulator to visualize the robot in its environment.
    2. **Monitor Real-Time Data:**
        - The developer observes real-time sensor data and task execution progress through Gazebo's interface.
        - The system continuously publishes relevant data from the robot to ROS2 topics, which Gazebo subscribes to and visualizes.
    3. **Receive Immediate Alerts:**
       Gazebo provides dynamic visualizations and notifications for critical events, such as the robot reaching its goal or encountering obstacles.
    4. **Dynamic Updates:**
       The system ensures that Gazebo's interface is continuously updated with real-time information, maintaining accuracy and relevance.
    5. **Gain Insights and Take Action:**
        - The developer gains valuable insights into the robot's behavior, with the ability to make informed decisions.

- Options for immediate action or further analysis are available based on the insights gained.

1.2.4 Send RESTful Command

- **Description:** External systems or users interact with the middleware through RESTful commands using Postman, enabling seamless integration and interoperability. Commands can include queries for robot parameters, initiation of specific actions, or retrieval of system information for enhanced functionality.

- **Actors:** External Systems

- **Pre-Condition:**

  - Middleware must be operational and accessible.

  - Secure authentication and authorization mechanisms must be in place.

  - Postman is set up and configured to interact with the middleware.

- **Post-Condition:** The system promptly processes and responds to the RESTful command, providing accurate data or executing requested actions.

- **Main Success Scenario:**

  1. **Construct RESTful Command:**
     The external system or user constructs a valid RESTful command using Postman, following the documented API specifications.
  2. **Submit Command via Postman:**
     The user submits the command to the middleware through Postman, including proper authentication credentials and error handling mechanisms.
  3. **Process Command:**
     The middleware efficiently processes the command, retrieving relevant data or initiating the specified actions.
  4. **Send Response:**

The middleware sends a clear and structured response back to Postman, including appropriate status codes and data payloads for seamless integration.

1.2.5 Initialize Project Request

- **Description**: The user can request the AOS server to build the project by integrating the project's skills and environment files through a RESTful command sent using Postman. There are several modes of Initialize Project Request: code generation only, inner simulation (without activating the robot), sequence of actions to run, robot activation, and robot activation/inner simulation without rebuilding the solver engine. The user can choose the integration mode and add additional parameters relevant to it (e.g., in robot activation mode, the user can choose the time interval between robot actions).

- **Actors**: Autonomous Robot Developer, System Administrator, and External Systems.

- **Pre-Condition:**

    - The AOS server is operational.

    - At least one project exists in the system.

    - Postman is set up and configured to interact with the AOS server.

- **Post-Condition**: Initialize Project Request ends successfully.

- **Main Success Scenario:**

    1. Construct RESTful Request:

        The user constructs a valid RESTful Initialize Project Request using Postman, following the documented API specifications.
    2. Submit Request via Postman:
        The user submits the Initialize Project Request to the AOS server through Postman, including proper authentication credentials and error handling mechanisms.

3. Inquire Integration Mode and Parameters:
   The system prompts the user for the integration mode and any relevant parameters for the chosen mode.
4. Process Initialize Project Request:
   - The system sends the Initialize Project Request to the AOS server.
   - The AOS server processes the request and performs the integration to build the project's solver engine code (unless the chosen mode is code generation/start without rebuilding).
5. Send Response:
   The system notifies the user via Postman that the Initialize Project Request has ended successfully.

1.2.6 Configure Solver Parameters

- **Description:** The user configures solver parameters to optimize the robot's decision-making processes during simulations and operational runs.

- **Actors:** Autonomous Robot Developer, System Administrator

- **Pre-Condition:** The AOS server is operational and the project exists in the system.

- **Post-Condition:** Solver parameters are configured and saved, optimizing the robot's performance.

- **Main Success Scenario:**

  1. Access Solver Configuration Interface:
     The user accesses the solver configuration interface or constructs RESTful commands using Postman.
  2. Define Solver Parameters:
     The user defines solver parameters such as the number of particles, planning time per move, and verbosity level.
  3. Submit Configuration:
     The user submits the configuration to the AOS server.

4. Confirm Configuration:
   The system confirms the successful configuration of solver parameters and provides logs for verification.

1.2.7 Manual Control Activation

- **Description:** The user activates manual control mode to directly control the robot's movements and actions during a simulation or operational run.

- **Actors:** Autonomous Robot Developer, System Administrator

- **Pre-Condition:** The AOS server is operational and the project exists in the system.

- **Post-Condition:** Manual control mode is activated, allowing the user to control the robot directly.

- **Main Success Scenario:**

  1. Construct RESTful Request:
     The user constructs a valid RESTful request using Postman to activate manual control mode.
  2. Submit Request via Postman:
     The user submits the request to the AOS server with the necessary parameters, including the manual control flag.
  3. Process Request:
     The AOS server processes the request and activates manual control mode.
  4. Direct Robot Control:
     The user gains direct control of the robot, able to issue commands and control movements in real-time.

1.2.8 Generate Project Code

- **Description:** The AOS system generates project code based on the provided skills and environment files, enabling seamless integration with the ROS2 framework.

- **Actors:** Autonomous Robot Developer, System Administrator.

- **Pre-Condition:** The AOS server is operational and the project exists in the system.

- **Post-Condition:** The generated project code is ready for deployment and further development.

- **Main Success Scenario:**

    1. Construct RESTful Request:
       The user constructs a valid RESTful request using Postman to generate project code.
    2. Submit Request via Postman:
       The user submits the request to the AOS server with the required parameters, including PLPs directory path and ROS target configuration.
    3. Process Request:
       The AOS server processes the request, generating the necessary project code.
    4. Confirm Code Generation:
       The system confirms the successful generation of the project code and provides logs for verification.

1.2.9 Perform Internal Simulation

- **Description:** The user initiates an internal simulation to test the robot's behavior and decision-making processes within a controlled environment.

- **Actors:** Autonomous Robot Developer, System Administrator.

- **Pre-Condition:** The AOS server is operational and the project exists in the system.

- **Post-Condition:** The internal simulation runs successfully, providing insights into the robot's performance.

- **Main Success Scenario:**

    1. Construct RESTful Request:
       The user constructs a valid RESTful request using Postman to perform an internal simulation.
    2. Submit Request via Postman:

The user submits the request to the AOS server with the necessary parameters, including the internal simulation flag.

3. Process Request:
   The AOS server processes the request and performs the internal simulation.
4. Analyze Simulation Results:
   The system provides the results of the internal simulation, allowing the user to analyze the robot's behavior and decision-making processes.

1.2.10 Consistent Data Handling for Inter-Skill Communication in ROS2

- **Description:** The developer implements a feature allowing skills to choose their data source (Database, ROS, or False), ensuring that other skills can access and use a local variable consistently. This enables seamless and reliable inter-skill communication and data handling within the robotic system.
- **Actors:** AOS developer
- **Pre-Condition:** The system must be correctly configured with ROS2, and connections to the database (e.g., MongoDB) should be validated.
- **Post-Condition:** The specified data source is successfully selected, and the data is accessible to all relevant skills in a consistent manner.
- **Main Success Scenario:**

1. Initialize System and Nodes:
   The user initializes the ROS2 environment, ensuring that all necessary nodes (AOS_TopicListenerServer, SharedState1, ListenToMongoDbCommands) are initialized and operational.

2. Set Up Data Source Command:
   The developer configures the system to allow data source selection commands via the AM File.

3. Write Data Source in AM File:
   The developer writes the desired data source (DB, ROS, or False) in the (AM) before starting the system.

4. Parse and Apply Activation Message:
   The node applies the command to set the data source accordingly:
   - **Database:** Retrieves the data from the MongoDB database.
   - **ROS:** Subscribe to the relevant ROS topic to obtain the data.
   - **False:** Sets the local variable to a default or null value.
5. Fetch Data from Selected Source:
   - The ListenToMongoDbCommands node fetches the latest data from the selected source.
6. Update Shared State:
   - The node updates the SharedState with the fetched data, ensuring that other skills can access the data as needed.
   - Visual indicators and logs confirm the successful update and current data source.
7. Skills Access Shared Data:
   - Other skill nodes (e.g., Navigation, Manipulating) access the SharedState to retrieve the required data.
   - Skills perform their designated tasks using the consistent data provided.
   -

2. System Architecture

Our system architecture is structured into three main layers: the presentation layer, the service layer, and the domain layer. Each layer serves distinct purposes and encapsulates specific functionalities:

2.1. Service Layer

- *Description*: The service layer contains the business logic and core functionalities of the system. It handles requests from the presentation layer, processes data, and orchestrates interactions between different components.

- *Components*: Planning Engine Integration, External Device Interface.
- *Deployment*: Deployed on dedicated servers
- *Functionality*: Manages communication between the presentation and domain layers, handles user requests, performs data processing, and integrates with external devices and services.

- ***The system uses a service of external server – the AOS server***

2.2. Domain Layer

- *Description*: the domain layer encapsulates the system's domain-specific logic and data models. It represents the core concepts and entities within the system, such as robot configurations, tasks and performance metrics.
- *Components*: database, middleware, business logic components.
- *Deployment*: the software components of the system are installed and run on servers that are specifically allocated and reserved for hosting the system.
- *Functionality*: stores and manages persistent data related to robot configurations, historical performance, user profiles and system settings. Implements domain-specific business logic and ensures data integrity and consistency.

3. Data Model

3.1. Description of Data Objects

In the following section we are going to introduce the main data objects in the system. Most of these objects are sections from the documentation files – JSON sections. We include these files to perform documentation file correction checks and other logical checks.
Therefore, the meaning of their fields relates to the inner logic of the AOS server. We provide a short explanation of each field for completeness of the explanation.

- **InitializeProject:**
  This component represents the initialization process of a project within the system. It contains configurations and settings necessary to set up and prepare a project for execution. It

includes properties such as PLPsDirectoryPath, RunWithoutRebuild, OnlyGenerateCode and references to other configurations like RosTarget, SolverConfiguration and MiddlewareConfiguration.

- **RosTargetProject:**
  This component holds configurations specific to a target project within the Robot Operating System (ROS) ecosystem.

- **SolverConfiguration:**
  This component encapsulates configurations related to a solver component within the project.

- **MiddlewareConfiguration:**
  This component contains configurations related to the middleware used within the project.

Each component plays a specific role in defining how the system operates and interacts with external dependencies like ROS and middleware services.

InitializeProject
PLPsDirectoryPath : string
RunWithoutRebuild : bool
OnlyGenerateCode : bool
RosTarget : RosTargetProject
SolverConfiguration: SolverConfiguration
MiddlewareConfiguration:MiddlewareConfiguration

- **GenerateRos2Middleware:**
  Is responsible forgenerating ROS middleware files based on provided project configurations and data. It interacts with the file system to create directories and write necessary files for ROS middleware packages.
  Additionally, it utilizes a configuration object to retrieve settings relevant to the middleware generation process.

GenerateRos2Middleware
ROS_MIDDLEWARE_PACKAGE_NAME: string
conf : Configuration

- **Ros2MiddlewareFileTemplate:**
  as an object with methods for generating ROS middleware files and managing dependencies and configurations. Its properties would include various settings and configurations required for middleware generation and ROS package management.

- **JsonTextModel:**
  serves as a bridge between the internal data structures of our application and their JSON representations.

  1. PlpMain:
     - Project: the name of the project associated with the PLP.
     - Name: the name of the PLP.
     - Type: the type of the PLP.
     - Version: the version number of PLP.
  2. ModuleResponse:
     - FromStringLocalVariable: a string representing the local variable from which the module response is derived.
     - ResponseRules: an array of response rules defining different responses for the module.
  3. ResponseRule:
     - Response: the response name.
     - ConditionCodeWithLocalVariables: the condition code with references to local variables.
  4. ModuleActivation:
     - RosService: an instance of RosService class representing ROS service activation details.
  5. RosService:
     - ImportCode: an array of import codes for the ROS service.
     - ServiceName: the name of the ROS service.
     - ServicePath: the path of the ROS service.
     - ServiceParameters: a list of services parameters.
  6. ServiceParameter:
     - ServiceFieldName: the name of the service field.
     - AssignServiceFieldCode: the code for assigning a value to the service field.
  7. ImportCode:
     - From: the source from which the import is done.

- Import: an array of strings representing the imported items.

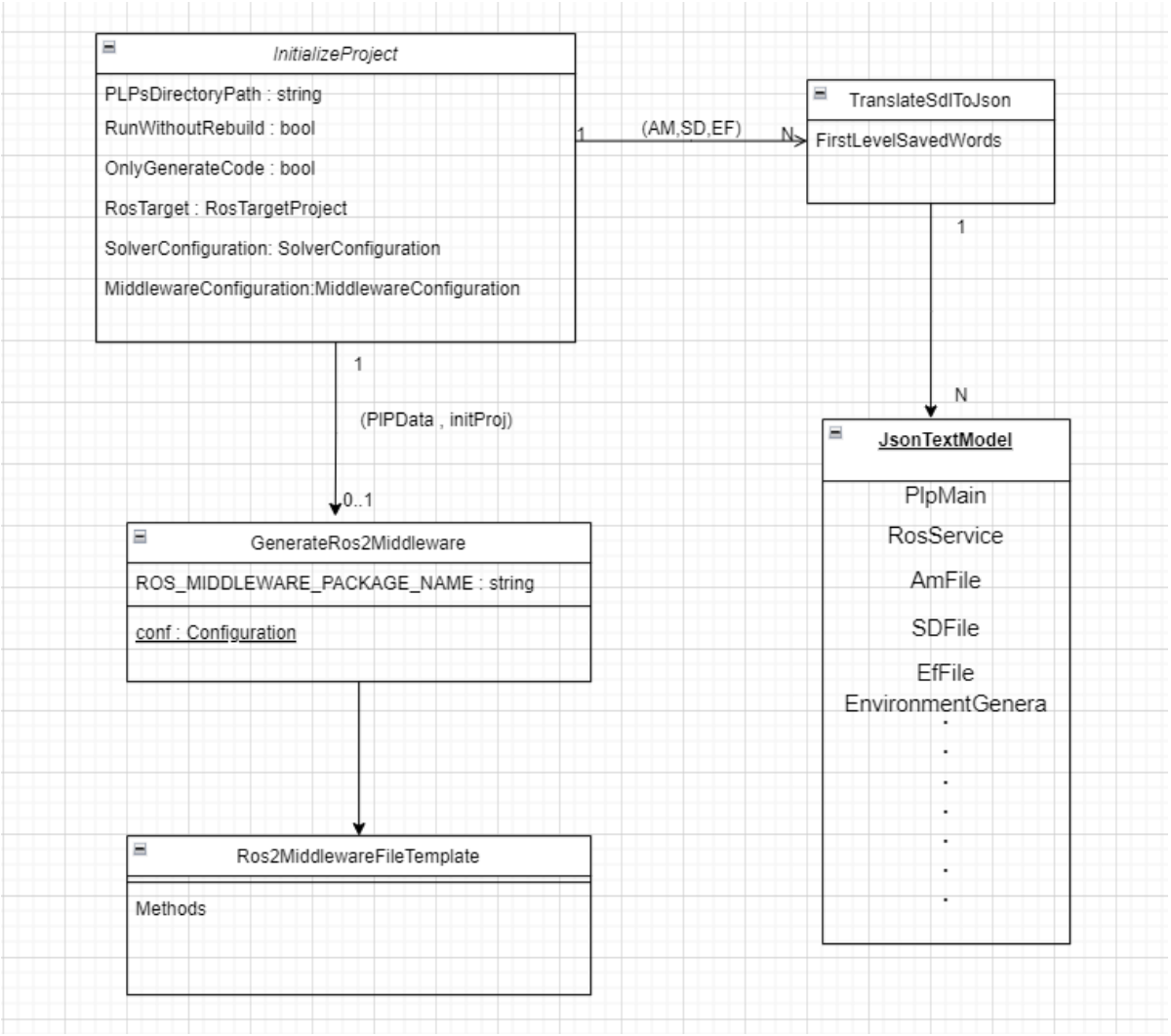
8. LocalVariableInitialization:
   - Various properties representing details about local variable initialization, including its source, name, type, topic, message type, initial value, etc.
9. AM-File:
   - PlpMain: an instance of PlpMain class representing PLP main details.
   - GlueFramework: a string representing the glue framework.
   - ModuleResponse: an instance of ModuleRespone class representing module response details.
   - ModuleActivation: an instance of ModuleActivation class representing module activation details.
   - LocalVariablesInitialization: a list of LocalVariableInitialization instances representing local variable initialization details.
10. EnvironmentGeneral:
    - Horizon: an integer representing the horizon.
    - Discount: a float representing the discount factor.

11. GlobalVariableType:
    - TypeName: the name of the global variable type.
    - Type: the type of the global variable.
    - Variables: an array of GlobalCompoundTypeVaribale instances representing compound type variables.
    - EnumValues: an array of strings representing enum values.
12. GlobalCompoundTypeVariable:
    Various properties representing details about global compound type variables.
13. GlobalVariableDeclaration:
    Various properties representing details about global variable declarations.
14. CodeAssignment:
    AssignmentCode: an array of strings representing code assignments.
15. SpecialStateCode:
    StateFunctionCode: an array of CodeAssignment instances representing state function code.

16. EF-File:
    Various properties representing different aspects of the EF file, including PLP main details, environment general settings, global variable types, global variable declarations, initial belief state assignments, special state codes and extrinsic changes in the dynamic model.
17. GlobalVariableModuleParameter:
    - Name: the name of the global variable module parameter.
    - Type: the type of the global variable module parameter.
18. Preconditions:
    - GlobalVariablePreconditionAssignments: an array of CodeAssignment instances representing global variable precondition assignments.
    - PlannerAssistancePreconditionsAssignments: an array of CodeAssignment instances representing planner assistance precondion assignments.
    - ViolatingPreconditionPenalty: an integer representing the penalty for violating preconditions.
19. DynamicModel:
    NextStateAssignments: an array of CodeAssignment instances representing next state assignments.
20. SD-File:
    Various properties representing different aspects of the SD file, including PLP main details, possible parameters value, global value, global variable module parameters, preconditions and dynamic model.

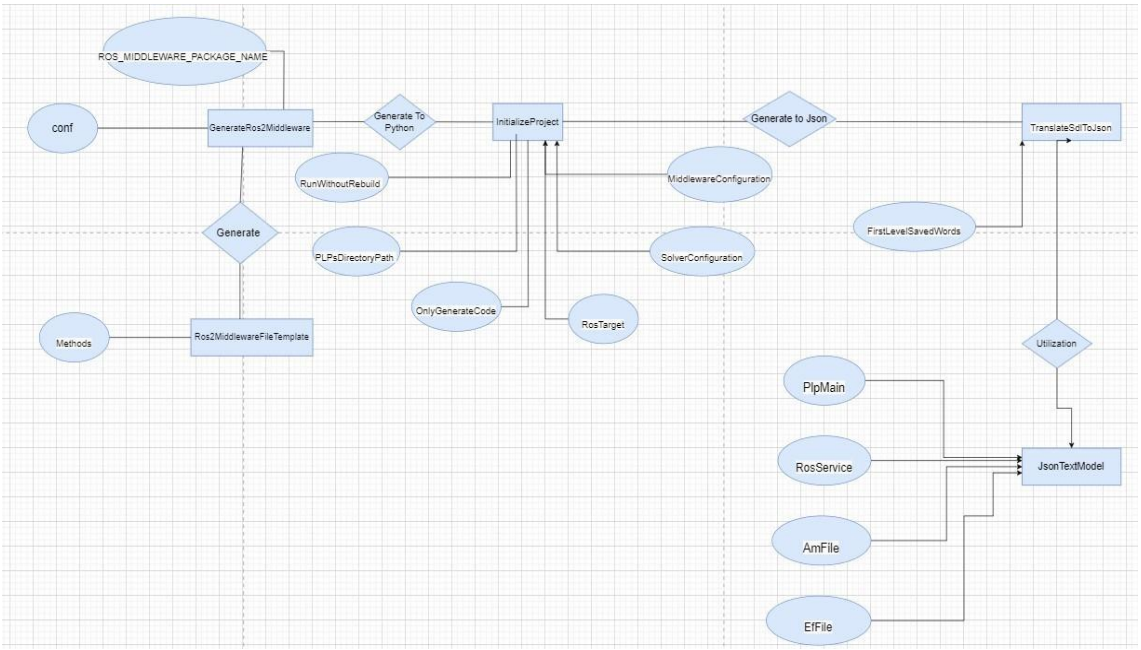
- **TranslateSdlToJson:**
  Is responsible for translating SDL (Skill Description Language) files into JSON format. It contains methods to translate SDL files representing different types of skills (SD, AM, EF) into their corresponding JSON representations.

3.2. Data Objects Relationships



3.3. Databases

ERD:



Main tables and their structures:

- **GenerateRos2Middleware:**
  Structure:

  This class likely represents functionality related to generating ROS middleware**.**

  Fields:

  ROS2_MIDDLEWARE_PACKAGE_NAME (string): Represents the name of the ROS middleware package.

  Conf (Configuration): references a configuration object.

- **Ros2MiddlewareFileTemplate:**
  Structure:

  This class defines a template for ROS middleware files**.**

  Fields:

  no fields just methods.

- **InitializeProject:**
  Structure:

  Manages the initialization process for a project**.**

  Fields:

  PLPsDirectoryPath (string) : Path to the directory containing PLP files.

RunWithoutRebuild (bool): Boolean flag indicating whether to run the project without rebuilding.

OnlyGenerateCode (bool): Boolean flag indicating whether to only generate code without executing it.

RosTarget (RosTargetProject): Configuration related to ROS target project.

SolverConfiguration (SolverConfiguration): Configuration related to the solver.

MiddlewareConfiguration (MiddlewareConfiguration): Configuration related to middleware.

- **JsontextModule:**
  Structure:

  Represents a module related to JSON text processing**.**

  Classes in JsontextModule (there are over 15 classes we mentioned above):

  PlpMain: Represents the main structure of a PLP file.

  RosService: Represents a ROS service.

  AmFile: Represents an AM file.

  SdFile: Represents an SD file.

  EfFile: Represents an EF file.

  EnvironmentGeneral: Represents general environment settings.

- **TranslateSdlToJson:**
  Structure:

  Provides functionality to translate SDL (System Description Language) files to JSON format**.**

  Fields:

  FirstLevelSavedWord: An array containing the first-level saved words used in the translation process.

The main transactions:

- **Initialization Transaction:**
  Purpose:

  initializes a new project**.**

  InitializeProject table:

Fields such as PLPsDirectoryPath, RunWithoutRebuild, OnlyGenerateCode , RosTarget, SolverConfiguration, and MiddlewareConfiguration may be updated during initialization.

Actions:

The transaction sets up the project environment by configuring paths, flags for rebuilding, code generation, ROS target, solver configuration, and middleware configuration.

- **Translation Transaction:**
Purpose:

Translates SDL files to JSON format**.**

TranslateSdlToJson table:

The FirstLevelSavedWord field may be updated during the translation process.

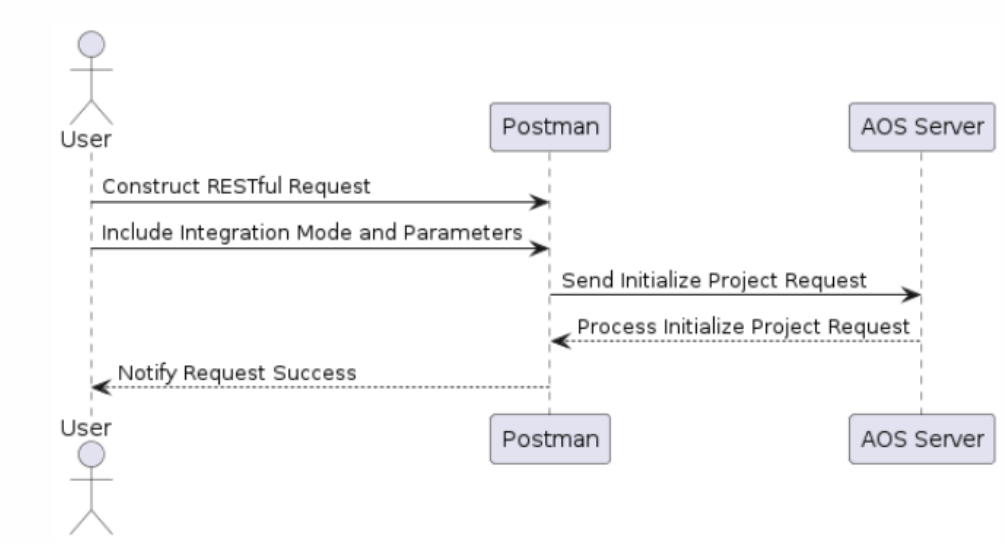
Actions:

The transaction reads SDL files, processes them, and converts them into JSON format. It may update metadata related to the translation process.

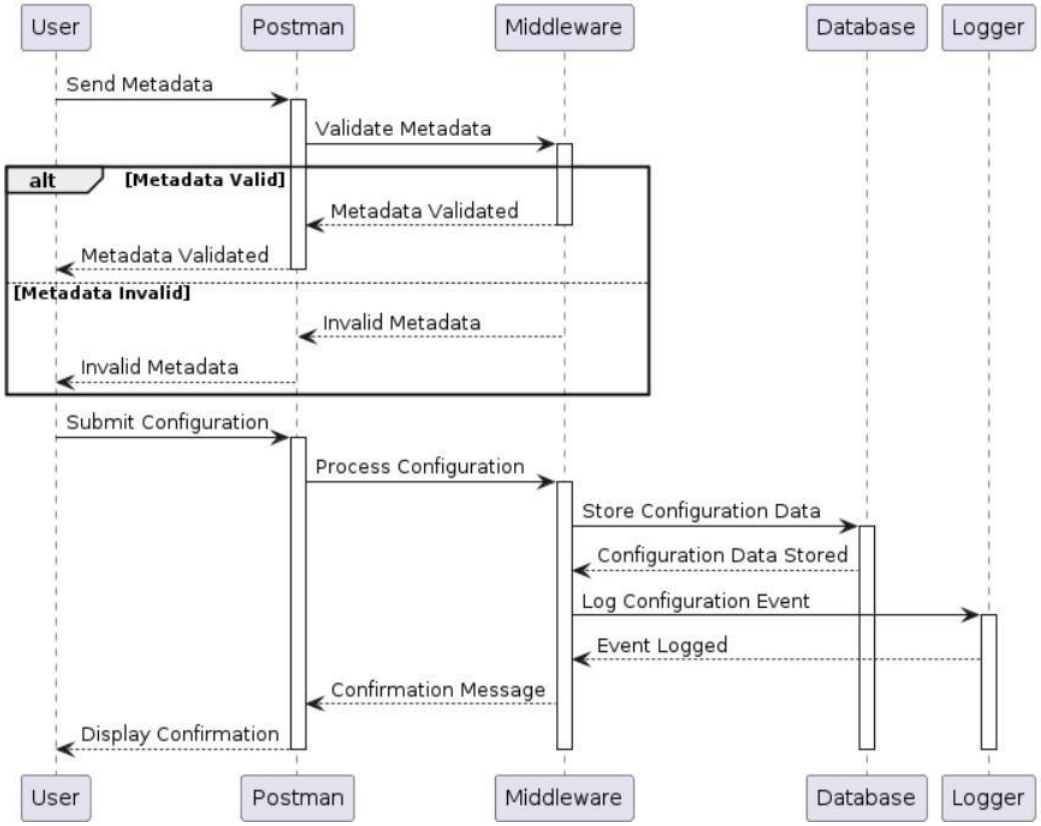
4. Analysis

4.1. Sequence Diagrams

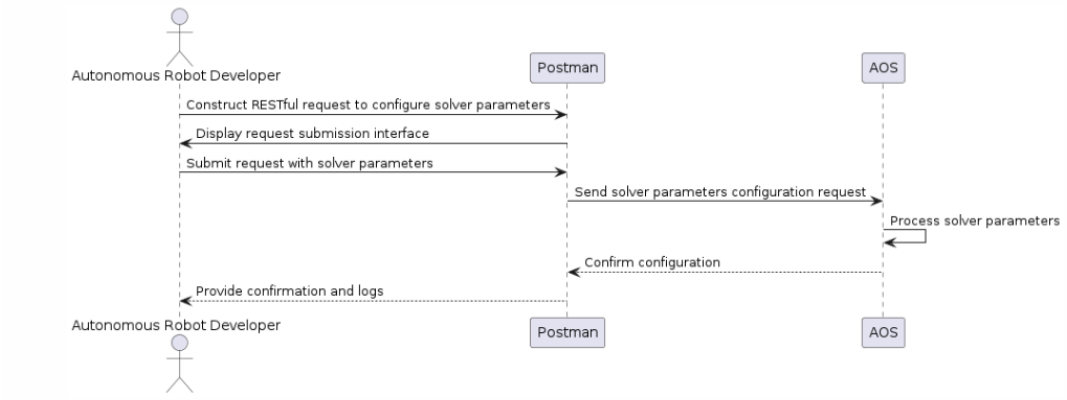
4.1.1. Initialize Project Request



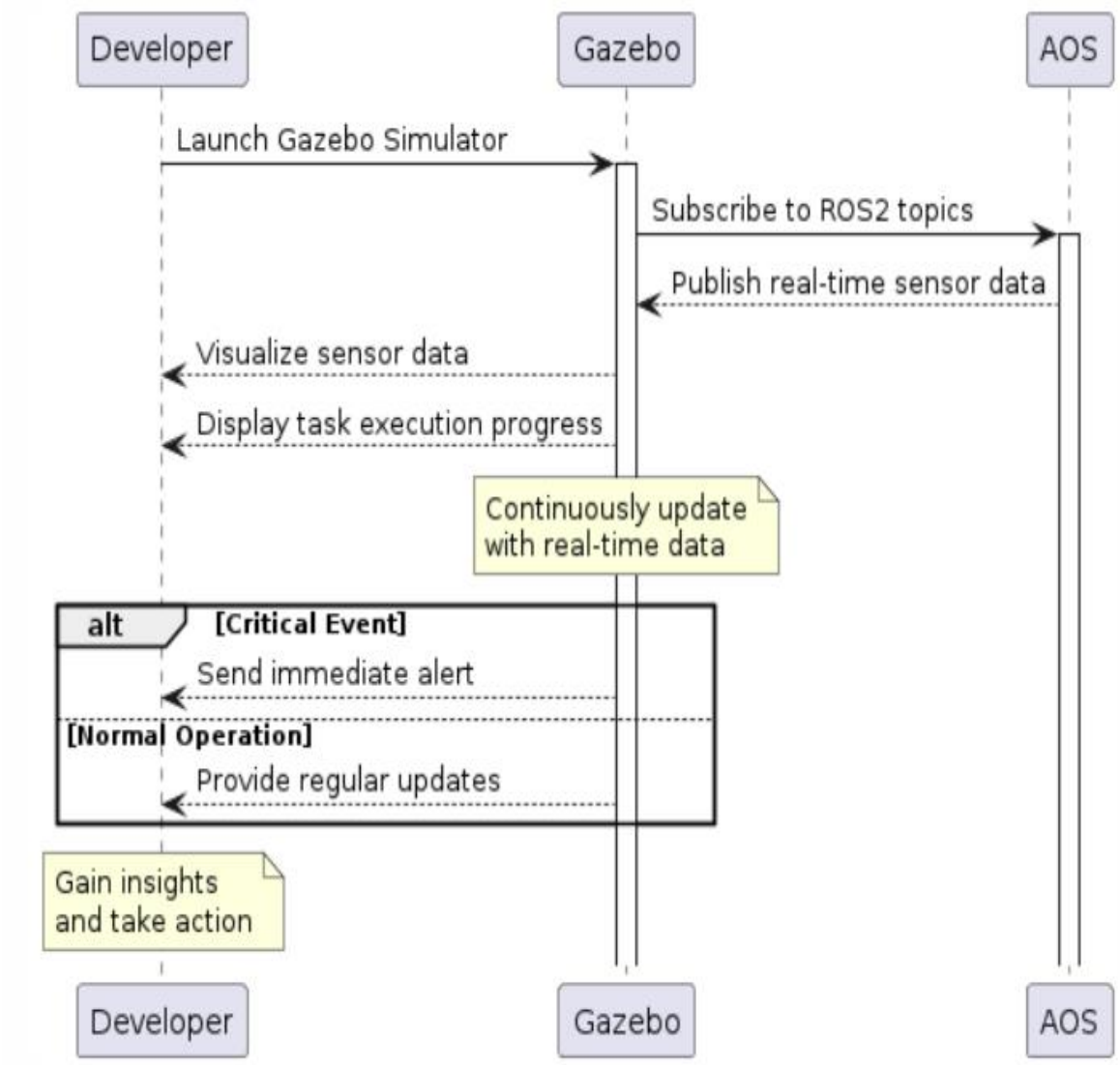
4.1.2. Configure Robot Capabilities



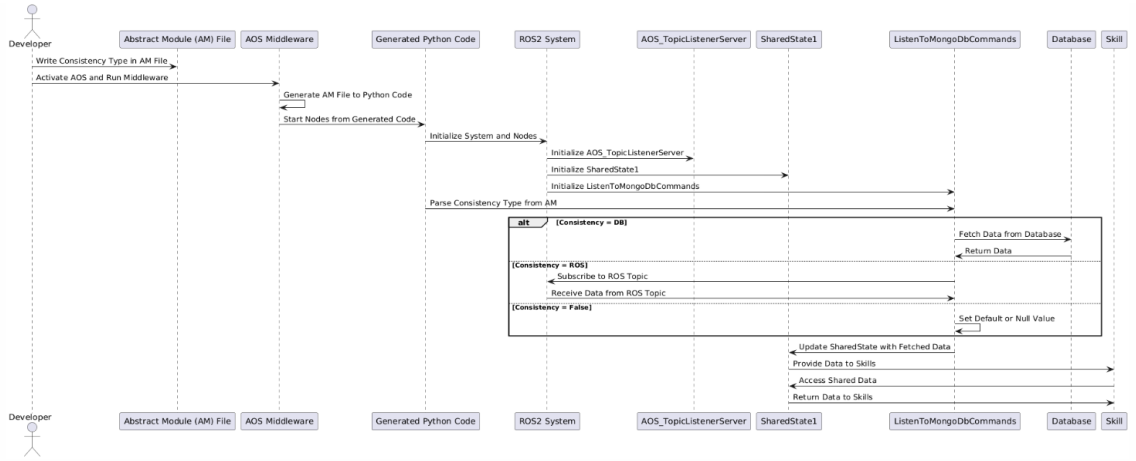
4.1.3. Configure Solver Parameters



4.1.4. Monitor Real-Time Robot Behavior

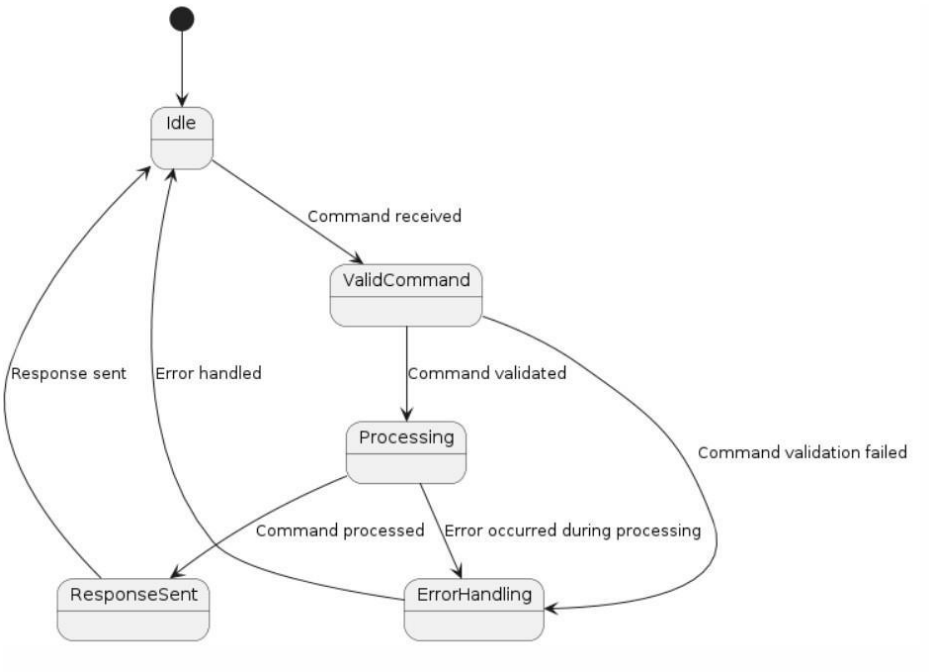


4.1.5. Data Consistency Selection and Initialization in AOS.

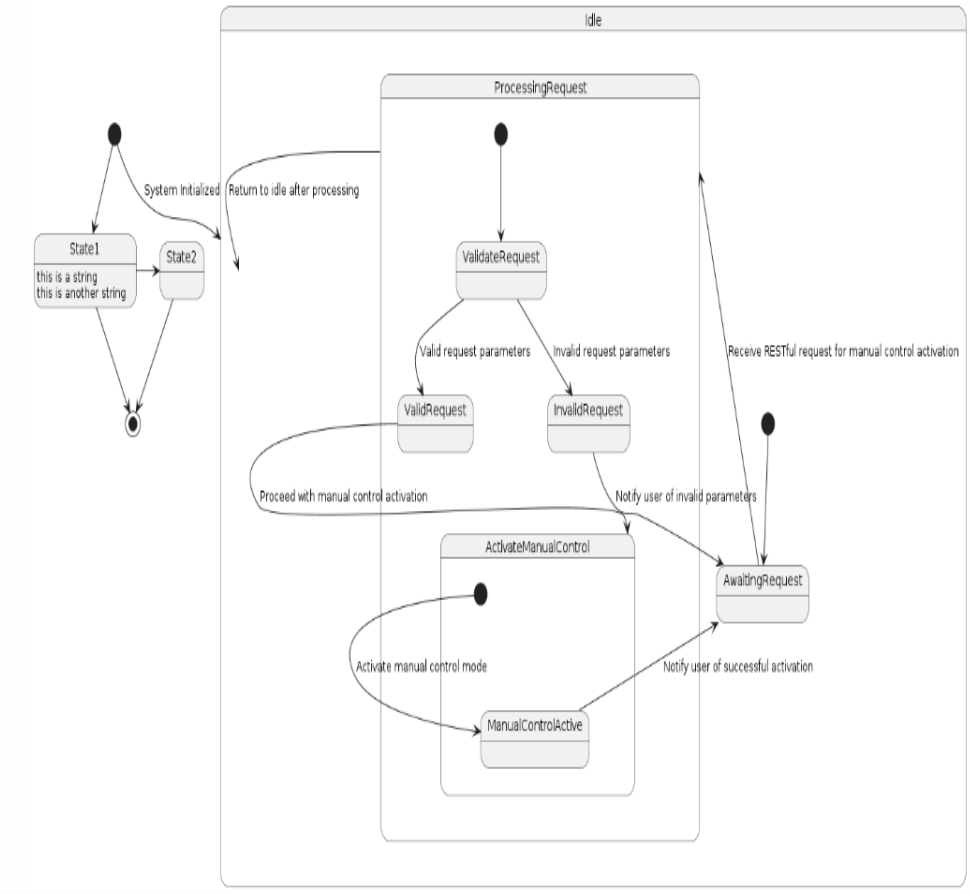


4.2. States

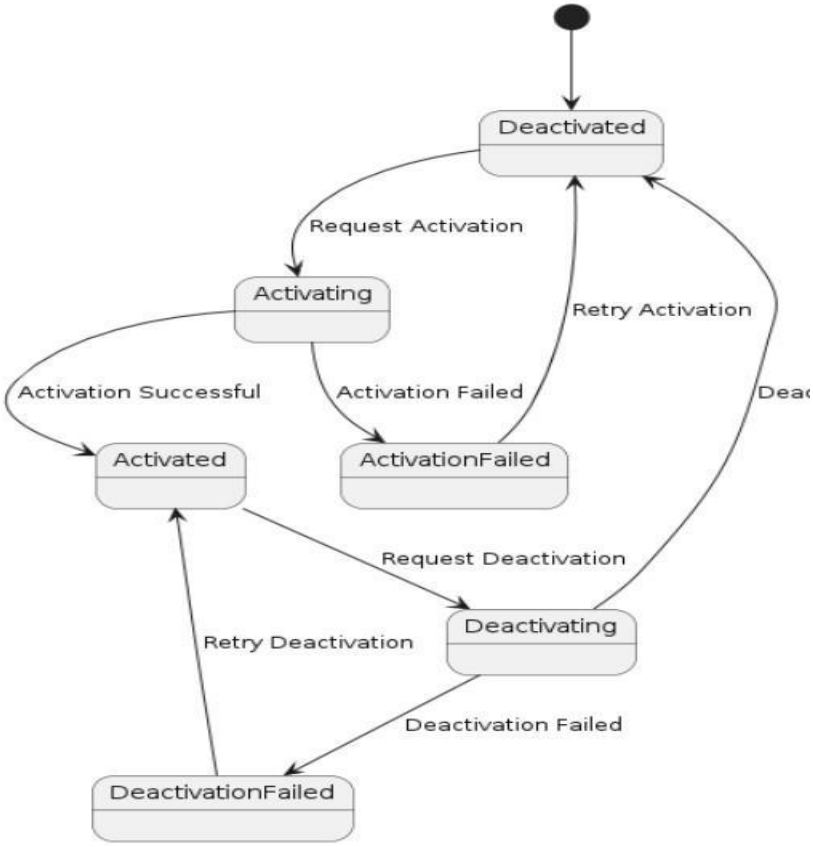
4.2.1. Send RESTful Command



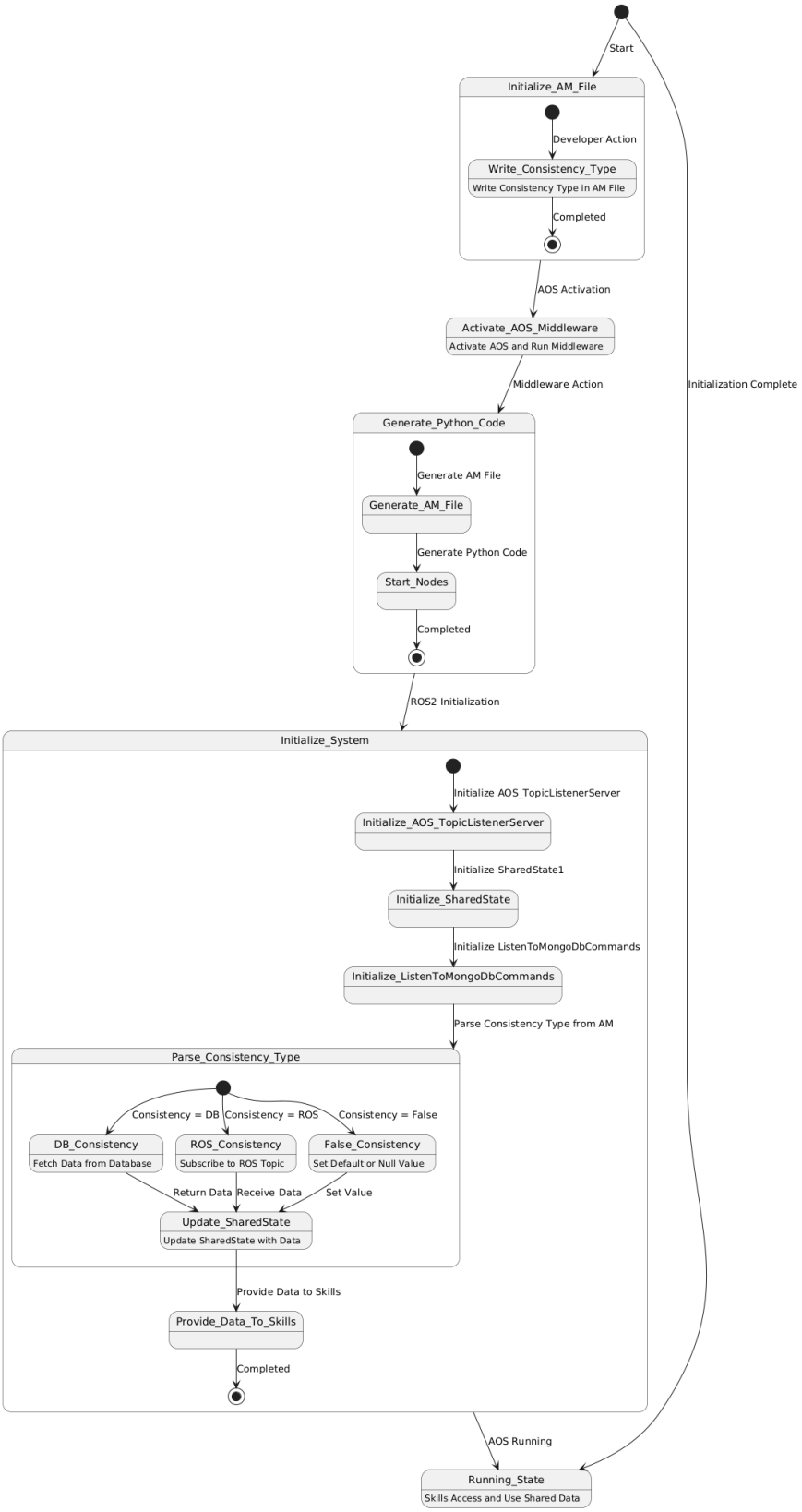
4.2.2. Manual Control Activation



4.2.3. Activate Robot Capability

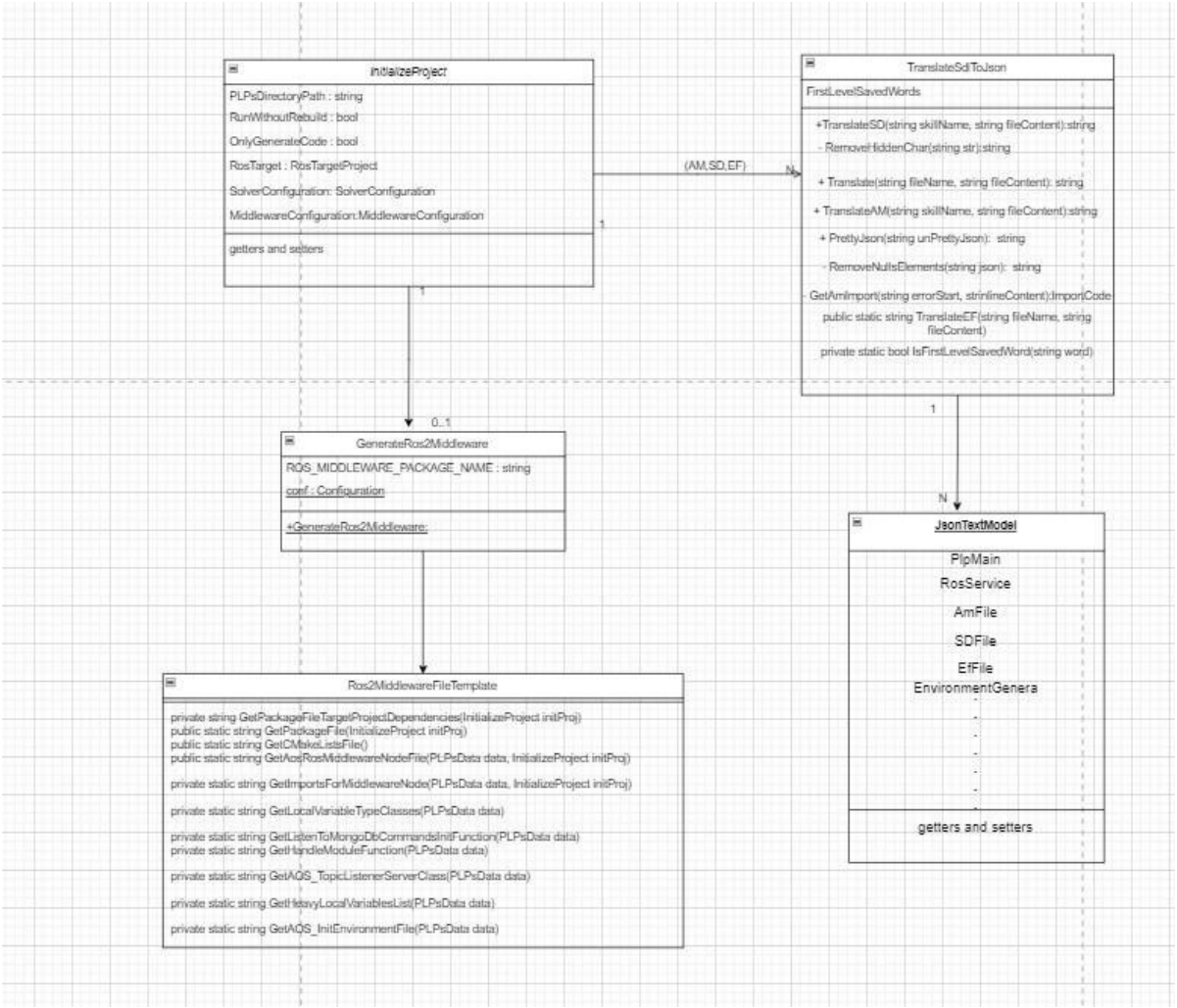


4.2.4. Data Consistency Selection and Initialization in AOS



5. Object-Oriented Analysis

5.1. Class Diagrams



5.2. Class Description

- **InitializeProject:**
  Responsibilities:

  Manages the initialization of a new project**.**

  Stores and retrieves project configuration settings.

  Methods:

  Getters and Setters: Responsible for getting and setting various project configuration settings such as PLPsDirectoryPath, RunWithoutRebuild,OnlyGenerateCode,RosTarget, SolverConfiguration, and MiddlewareConfiguration.

  (inner classes in InitializeProject).

  Pre-conditions/post-conditions:

  No.

- **GenerateRos2Middleware:**
  Responsibilities:

  Generates ROS 2 middleware files based on provided configurations**.**

  Handles the creation of middleware package files, CMakeLists files, ROS middleware node files, etc.

  Methods:

  GetPackageFileTargetProjectDependencies (InitializeProject initProj): retrieves package file target project dependencies based on the provided initialization project.

  GetPackageFile(InitializeProject initProj): generates a package file based on the provided initialization project.

  GetCMakeListsFile(): generates a CMakeLists file.

  GetAosRosMiddlewareNodeFile(PLPsData data, InitializeProject initProj): generates a ROS middleware node file based on PLP data and the provided initialization project.

  Pre-conditions/post-conditions:

  These methods have pre-conditions related to valid input parameters. Post-conditions might involve ensuring successful file generation or the absence of errors.

- **Ros2MiddlewareFileTemplate:**
  Responsibilities:

    Provides templates for ROS 2 middleware files**.**

    Handles the formatting and structure of middleware files

  Methods:

    The provided methods are responsible for generating specific sections or components of middleware files, such as imports, local variable types, function definitions…

  Pre-conditions/post-conditions:

    Similar so GenerateRos2Middleware.

- **JsonTextModel:**
  Responsibilities:

    Represents a JSON text model**.**

    Provides methods for accessing and modifying JSON data.

  Methods:

    Includes standard getters and setters for interacting with JSON data.

  Pre-conditions:

    Involve valid input parameters for setters,

  Post-conditions:

    Could involve ensuring successful retrieval or modification of JSON data.

- **TranslateSdlToJson:**
  Responsibilities:

    Translates SDL files to JSON format.

  Methods:

    RemoveHiddenChar(string str):string: Removes hidden characters from the input string.

    TranslateSD(string skillName, string fileContent):string: Translates SDL skill data to JSON format.

    TranslateAM(string skillName, string fileContent):string: Translates SDL AM data to JSON format.

Translate(string fileName, string fileContent): string : Translates SDL files to JSON format based on the provided file name and content.

IsFirstLevelSavedWord(string word): Checks if a word is a first-level saved word.

TranslateEF(string fileName, string fileContent): Translates SDL EF data to JSON format.

RemoveNullsElements(string json): Removes null elements from JSON data. GetAmImport(string errorStart, strinlineContent): ImportCode: Generates AM import code.

Pre-conditions:

Pre-conditions related to valid input parameters,

Post-conditions:

Might involve ensuring successful translation or formatting of SDL to JSON data.

5.3. Packages

- **GenerateCodeFiles**
  This package contains classes responsible for generating middleware files for ROS 2 middleware and contains classes that provide templates for ROS 2 middleware files. such as (Ros2MiddlewareFileTemplate, GenerateRos2Middleware).
- **Controllers**
  This package contains classes related to project initialization and configuration settings.
- **TranslateSDL**
  This package contains classes responsible for translating SDL files to JSON format.such as (TranslateSdlToJson ,JsonTextModel).

5.4. Unit Testing

- **InitializeProject**
  Test Getters and Setters: Ensure that each getter returns the correct value after setting it with a known input.
- **GenerateRos2Middleware**
  Mock or stub dependencies and test that the generated middleware files meet expected criteria.

Test that the generated files contain necessary configurations and content based on various input scenarios.
Test edge cases such as empty inputs or extreme values to ensure robustness.

- **Ros2MiddlewareFileTemplate**
  Test each method individually, mocking any dependencies or inputs.
  Verify that the generated templates meet expected standards and configurations.
  Test edge cases and boundary conditions to ensure correctness and robustness.

- **JsonTextModel**
  Getters and Setters:

  Ensure that getters return expected values after setting them with known inputs.

  Test that setters correctly update the state of the object.

  Verify that invalid inputs result in appropriate errors or exceptions.

- **TranslateSdlToJson**
  Test Translate Methods: Test each translation method with various input SDL files, ensuring that the output JSON meets expected standards.
  Test edge cases such as empty files, files with unusual formats, or files with extreme sizes.
  Mock or stub dependencies to isolate the unit under test.

6. GUI - our system does not have a GUI.

7. Testing

7.1. Unit Testing

- **InitializeProject**
  - *Test Getters and Setters:*
    Ensure that each getter returns the correct value after setting it with a known input. For example: test that setting the 'PLPsDirectoryPath' to a specific value results in the getter returning that same value.
  - *Test Initialization Process*: Testing the initialization process by setting up different project configurations

and verifying that the initialization occurs as expected.

- **GenerateRos2Middleware**
    - *Testing Edge Cases:* Testing edge cases such as empty inputs or invalid input to ensure robustness.
    - *Dependencies*: Testing that the generated middleware files meet expected criteria. and ensuring that the necessary configurations and content are present in the generated files.

7.2. Integration Testing

- **InitializeProject-GenerateRos2Middleware Integration**
  Testing the interaction between InitializeProject and GenerateRos2Middleware to ensure that project initialization properly triggers middleware integration and that the generated middleware files reflect the project configuration.
- **TranslateSdlToJson**
  Verify that TranslateSdlToJson interacts correctly with JsonTextModel to perform SDL to JSON translation. Test scenario where TranslateSdlToJson utilizes various JsonTextModel methods and ensure seamless integration between the two components.

7.3. System Testing

- **End-to_End Testing**
    - **Test Project initialization:** Perform end-to-end tests to ensure that the entire project initialization process works as expected, from setting up project configurations to generating middleware files.
    - **Test Translation process:** Validate the SDL to JSON translation process by providing sample SDL files and verifying that the resulting JSON files match expected outputs.
    - **Edge Case Testing:** Include edge cases and boundary conditions in end-to-end testing to evaluate system behavior under various scenarios.

- **User Interface Testing**
    - **Functional Testing:** Test the functionality of user interface elements such as buttons, input fields, and navigation. Ensure that users can interact with the interface as intended.

- **Usability Testing:** Evaluate the user interface for usability, clarity, and ease of use.

7.4. Acceptance Testing

- **Client/User Acceptance Testing**
  - **Client/User Feedback:** Present the system to clients or end-users for acceptance testing. Gather feedback on system functionality, usability, and performance to ensure that it meets their requirements and expectations.
  - **Addressing Issues:** Address any issues or concerns raised during acceptance testing and make necessary adjustments to the system to meet client/user needs.